JEITA

EDAアニュアルレポート　2008

Annual Report on Electronic Design Automation

－　65nmから45nmテクノロジ世代のEDA技術の進展に向けて　－

2009年5月発行

作　　成

EDA技術専門委員会

EDA Technical Committee

発　　行

社団法人　電子情報技術産業協会

Japan Electronics and Information Technology Industries Association

【巻頭言】

「65nmから45nmテクノロジ世代のEDA技術の進展に向けて」

ＥＤＡ技術専門委員会　2008年度　委員長　　山田節

現在および近い将来のユビキタス社会を構築していく上で、基幹となる半導体産業への期待が高まっている。なかでも、半導体の設計に関する技術は、LSI の高密度化、高集積化、高性能化、低消費電力化の要求を支える重要な役割を担っており、より一層の技術向上を図る必要がある。

半導体加工の超微細化の進展に伴い、設計上流では超大規模システム LSI の機能・論理の設計・検証問題、設計下流では SI(Signal Integrity）や DFM(Design For Manufacturing)問題、そしてこれをつなぐインプリメンテーションの難易度の飛躍的増大への対応である。

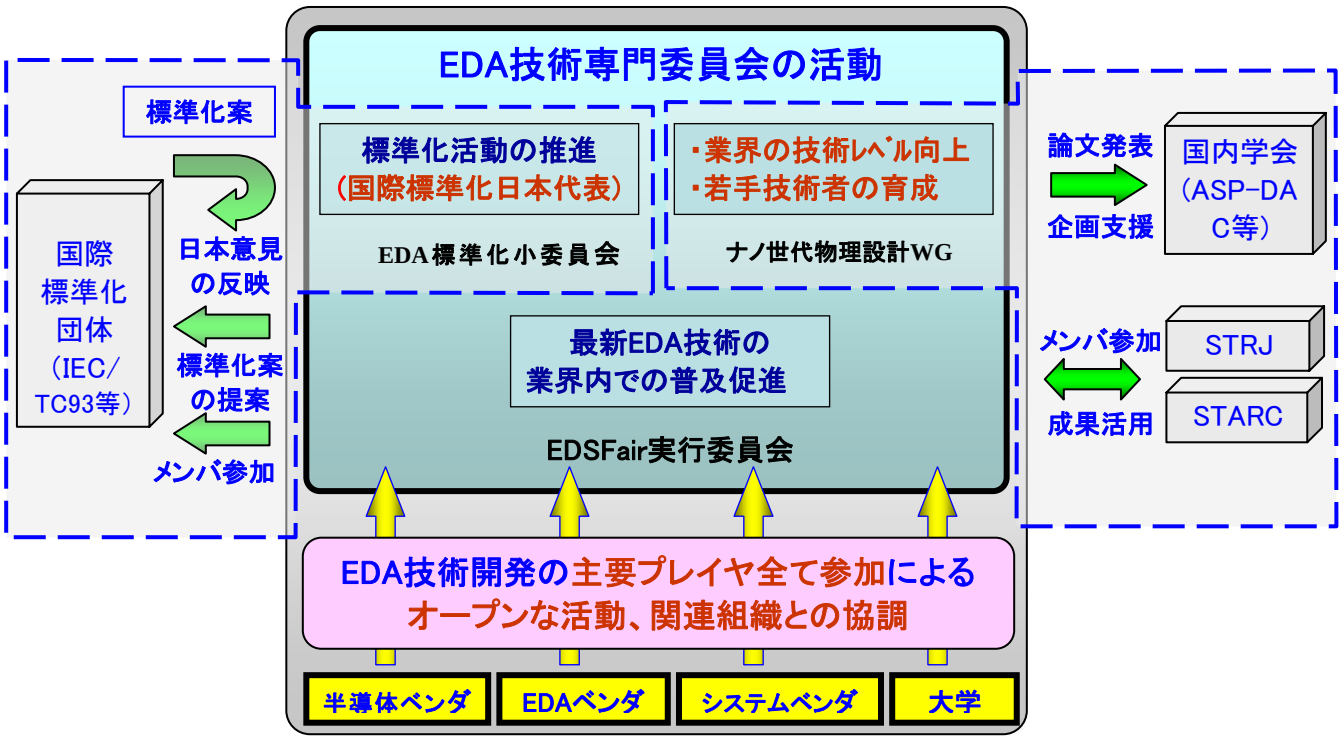
EDA 技術専門委員会は、電子情報技術産業協会(JEITA)における業界活動組織の一つとして、電子機器の設計自動化(EDA：Electronic Design Automation)に関わる様々な活動を行っている。特に、電子機器の機能・性能を決定するシステム LSI 設計技術に関して、以下の 3 つの領域を推進している。

第 1 の活動の領域は、システム LSI 設計技術に関する調査・検討と課題解決手法の提案である。本年度は、製造ばらつきに起因するリーク電流変動の低減法などのプロセス微細化に伴う設計課題とその課題解決手法を、ナノ世代物理設計ワーキンググループが検討を行った。

第 2 の活動領域は、EDA 技術に関する標準化活動と関連機関、団体への協力と貢献である。IEC、IEEE 等の国際的な標準化活動に対し EDA 標準化小委員会を中心に、標準化提案の検証、技術提案・交流会議などを行なっている。設計記述言語 SystemC および SystemVerilog の IEEE 標準化では、本委員会の各ワーキンググループが、2005 年末の標準化に貢献し、その後も両標準の普及推進活動と、追加標仕様の検討、提案を行っている。二つの標準化グループが存在し混迷している Power Format は、PowerFormat ワーキンググループが業界としての検討を行い、IEEE.p1801 への投票を行った。また、EDA 標準化小委員会は電子情報通信学会内の IEC/TC93 国内委員会/WG2(ハードウェア設計記述言語)としての標準化活動も行っており、2008 年 11 月には IEC/TC93 国際会議(シンガポール)に参加し、国際標準化の課題を議論してきた。

最後に、第 3 の活動領域は、EDA 技術および標準化の普及、推進のためのイベント開催、支援である。本年度も、2009 年 1 月にパシフィコ横浜にて EDSFair2009（Electronic Design and Solution Fair 2009）を開催した。EDSFair は、電子機器の設計技術に関する、国内唯一の総合的な展示会であり、時開催の FPGA コンファレンスおよびシステム・デザイン・フォーラム 2009 と合わせ、特設ステージでの当委員会企画イベントも含めて積極的・精力的に、企画、出展者誘致、来場者誘致活動を行った。その結果、最新の EDA 技術と関連情報との出会いの場として、LSI 設計者・電子機器設計者をはじめとした 1 万人に近い方々に参加頂いた。

次の図でこれらの委員会活動と関係する団体の関係を示す。



IEC/TC93 ： 国際電気標準会議/設計自動化

ASP-DAC：Asia South Pacific-Design Automation Conference

STRJ ： 半導体技術ロードマップ委員会

STARC ： 半導体理工学研究センター

図-１　EDA技術専門委員会と関連組織との関係

EDA 技術専門委員会は、上図の関連組織・団体との密な連携のもと、技術検討、標準化、そしてそれらの普及促進という 3 つの活動領域の活動を通じ、65nm から 45nm テクノロジー世代の「システム・オン・チップ時代」の電子情報機器業界に対し、さらには地球温暖化などの世界規模での課題に対し、設計技術の面で貢献し、「システム・オン・チップ時代」がもたらす産業界の変革を乗り越えて、日本の電子情報機器業界の発展に寄与すべく、本年度 21 社約 60 名の業界各社・有志メンバーの参画で運営してきた。

「システム・オン・チップ時代」に入り、半導体および電子機器が切り拓く素晴らしい未来が今後も広がることを確信しつつ、2009年度も積極的な活動を展開していきたい。

本冊子「ＥＤＡアニュアルレポート2008」は、EDA技術専門委員会の2008年度年次報告として、上記３つの活動領域について、活動成果をまとめたものである。
Webにも各種報告を掲載しているので、ご覧いただきたい。
(http://eda.ics.es.osaka-u.ac.jp/jeita/eda/index-jp.html)

2008 年度 JEITA/EDA 技術専門委員会 委員一覧

委員長	山田　節	三洋電機㈱	デジタル技術研究所 担当課長（ＬＳＩ設計技術担当）
副委員長	太田　光保	パナソニック㈱	セミコンダクター社 システムＬＳＩ事業本部 商品開発センター 設計第三開発グループ　参事
副委員長	江田　努	ローム㈱	ＹＴＣ開発システムユニット ＬＳＩ企画推進Ｇ　次席技術員
監事	齋藤　茂美	ソニー㈱	半導体事業本部　設計基盤技術部門 企画部第１課　統括課長
幹事	吉田　正昭	ＮＥＣエレクトロニクス㈱	基盤技術開発本部 設計技術開発部 シニアプロフェッショナル
幹事	西本　猛史	シャープ㈱	電子デバイス事業本部 ＮＢ事業化推進センター ＮＢプロジェクトチーム　副参事
幹事	熊谷　敬	セイコーエプソン㈱	半導体事業部　ＩＣ基盤技術部 部長
幹事	南　文裕	㈱東芝	セミコンダクター社 システムＬＳＩ設計技術部 設計メソドロジ技術開発　担当主査
幹事	河村　薫	富士通マイクロエレクトロニクス㈱	テクノロジ開発統括部 主席部長
幹事	秋山　俊恭	㈱ルネサステクノロジ	製品技術本部　設計技術統括部 副統括部長

委員	長野　義史	㈱エッチ・ディー・ラボ	代表取締役社長
委員	川原　常盛	コーウェア㈱	取締役　営業技術本部長
委員	下出　隆文	三洋半導体㈱	ＳＳ事業本部　ＳＳ事業推進統括部 設計技術部　課長
委員	山城　治	㈱ジーダット	取締役
委員	野坂　啓介	㈱図研	ＳｏＣ事業部イノベーション営業部
委員	平尾　栄二	凸版印刷㈱	半導体ソリューション事業本部 事業企画部　部長
委員	安井　孝史	日本ケイデンス・デザイン・システムズ社	ＴＦＯ　Ｄｉｒｅｃｔｏｒ
委員	飯島　一彦	日本シノプシス㈱	技術本部　本部長
委員	牧　克眞	マグマ・デザイン・オートメーション㈱	営業ディレクタ
委員	瀬谷　和宏	丸紅情報システムズ㈱	製造ソリューション事業本部 事業本部長補佐
委員	三橋明城男	メンター・グラフィックス・ジャパン㈱	テクニカルセールス部 テクニカルディレクター
委員	前野　酉治	㈱リコー	電子デバイスカンパニー 画像ＬＳＩ開発センター ＣＡＤ技術室シニアスペシャリスト
委員	山本　一郎	ローム㈱	ＯＫＩセミコンダクタ㈱ 開発本部　設計技術サポートチーム チームリーダー
特別委員	黒川　敦	三洋半導体㈱	ＬＳＩ事業本部　設計統括部 設計技術部　課長

特別委員	小島　智	ＮＥＣシステムテクノロジー㈱	ＰＦ統括本部　ＣＷＢ事業推進室 テクニカルマーケティング ディレクター
特別委員	浜口加寿美	パナソニック㈱	セミコンダクター社 システムＬＳＩ事業本部 商品開発センター 設計第三開発グループ チームリーダー
特別委員	中森　勉	富士通マイクロエレクトロニクス㈱	テクノロジ開発統括部 第一設計技術部 プロジェクト課長
特別委員	長谷川　隆	富士通マイクロエレクトロニクス㈱	共通技術本部　設計共通技術統括部 第１設計部　部長
特別委員	金本　俊幾	㈱ルネサステクノロジ	製品技術本部　設計技術統括部 ＳＩＰ・アナログＥＤＡ技術開発部 物理検証技術開発グループ主任技師
客員	神戸　尚志	近畿大学	理工学部　電気電子工学科 教授
客員	今井　正治	大阪大学	大学院　情報科学研究科 情報システム工学専攻　教授
客員	岡村　芳雄	㈱半導体理工学研究センター	執行役員・開発第２部長
客員	若林　一敏	日本電気㈱	中央研究所 ビジネスイノベーションセンター ＥＤＡ開発センター研究部長

略 語 一 覧

［１］団体・組織の名称

Accellera	VIとOVIを統合した、設計記述言語の標準化に関連する活動機関
ANSI	American National Standards Institute 米国の標準化国家機関
ASP‐DAC	Asia and South Pacific Design Automation Conference アジア･太平洋地域でのEDA関連の国際学会（1995年に始まる）
CENELEC	European Committee for Electrotechnical Standardization EC（欧州委員会）の電気電子分野に関する標準化機関
DAC	Design Automation Conference 米国で行われるEDA関連の国際学会
DASC	Design Automation Standardization Committee IEEEの下部組織で設計自動化に関する標準化委員会
ECSI	European Electronic Chips & Systems design Initiative 欧州の設計自動化に関する標準化機関
EDIF Div.	Electronic Design Interchange Format Division EIAの下部組織で電子系の惜報データ交換規格の検討機関
EIA	Electronic Industries Alliance 米国の電子業界団体(AssociationをAllianceに改称)
JEITA	Japan Electronics and Information Technology Industries Association 社団法人電子情報技術産業協会（電子業界団体）
ICCAD	International Conference on Computer Aided Design CADに関する国際学会
IEC	International Electrotechnical Commission 電気電子分野に関する国際標準化機関
IEEE	Institute of Electrical and Electronics Engineers,Inc. 米国の電気電子分野の国際的な学会組織
IPC	Institute for Interconnecting and Packaging Electronic Circuits Industry Association 米国のプリント回路に関する業界組織
ISO	International Organization for Standardization 国際標準化機関

IVC	International Verilog Conference OVIが主催するVerilog HDL国際学会
JPCA	Japan Printed Circuit Association 社団法人日本プリント回路工業会
OSCI	Open SystemC Initiative SystemC の標準化団体
OVI	Open Verilog International Verilog - HDLに関連する技術の標準化と普及推進組織
SEMATECH	Semiconductor Manufacturing Technology Initiative （Consortium） 半導体技術を向上するために始まった米国の官民プロジェクト
Si2	Silicon Integration Initiative 設計環境の整備促進を支援する米国の非営利法人（旧CFI）
VASG	VHDL Analysis and Standards Group DASC傘下のVHDL標準化に関するワーキンググループ
VITAL	VHDL Initiative Toward ASIC Libraries VHDLライブラリ標準化団体
VSIA	Virtual Socket Interface Alliance LSIの機能ブロックのI/F標準化を目指している業界団体

［２］標準化・規格に関する技術用語

ALF	Advanced Library Format OVIで検討されたIPをも含むASICライブラリのフォーマット
ALR	ASIC Library Representation ASICライブラリ表現
CALS	Computer Aided Logistics Support（1985） Commerce At Light Speed（1995）
CHDS	Chip Hierarchical Design System SEMATECHが要求仕様を作成した0.25-0.18um世代設計システム
CHDStd	Chip Hierarchical Design System technical data CHDSで使用するデータモデルの標準化

DCL	Delay Calculation Language 遅延計算のための記述言語
DPCS	Delay and Power Calculation System IEEE1481として標準化推進されている遅延と消費電力の計算機構仕様
EDI	Electronic Data Interchange 電子データ交換
EDIF	Electronic Design Interchange Format EIAの下部組織で検討されている電子系の情報データ交換規格
ESPUT	European Strategic Program for Research and Development in Information Technology 欧州情報技術研究開発戦略計画
HDL	Hardware Description Language ハードウェア記述言語
IP	Intellectual Property 流通/再利用可能なLSI設計資産（本来は知的財産権の意）
JIS	Japanese Industrial Standard 日本工業規格
SDF	Standard Delay Format 遅延時間を表記するフォーマット
SLDL	System Level Design Language システム仕様記述言語
STEP	Standard for the Exchange of Product Model Data CADの製品データ交換のための国際規格
VHDL	VHSIC（Very High Speed Integrated Circuit) Hardware Description Language IEEE1076仕様に基づくハードウェア記述言語
VHDL-AMS	VHDL-Analog and Mixed-Signal (Extensions) DASCの中で進められているVHDLのアナログ及びミックストシグナルシステムへの拡張

１．ＥＤＡ技術専門委員会の活動

1.1　2008 年度　JEITA/EDA 技術専門委員会　事業計画

委員会の名称　EDA 技術専門委員会（Electronic Design Automation Technical Committee）

委員会の目的　EDA に関連する技術およびその標準化の動向を調査し、その発展、推進を図り、もって国内外の関係業界の発展に寄与する

委員会の構成

会員会社/委員	21 社/24 名	
特別委員	6 名	
客員	4 名	

委員会の役員

委員長：	三洋電機	山田　節
副委員長（正）：		パナソニック
副委員長（代行）：	ローム	江田　努
監事：	ソニー	齋藤　茂美

下部組織の役員

EDA 標準化小委員会	主査	パナソニック	太田　光保
SystemC　WG	主査	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	長谷川　隆
SystemVerilogWG	主査	パナソニック	浜口　加寿美
Power Format WG	主査	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	中森　勉
ナノ世代物理設計 WG	主査	ルネサステクノロジ	金本　俊幾
EDSFair 実行委員会	委員長	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	河村　薫
EDSFair 企画　WG	主査	ソニー	齋藤　茂美
SDF WG	主査	ローム	江田　努

<方針>
1.情報収集力の強化(PowerFormat 取組みの反省)
　IEEE-SA の加入継続
　IEEE/DASC への加入検討
　国際会議(TC93, DASC)への積極的参加と国内開催（WG2）
　標準化団体との定期的な会議開催
2.委員会活動の活性化
　委員会を主体にした運営 EDSFair に係　る　JESA,ASP-DAC,FPGA/PLD Conf.
　などとの連携を再構築とイベント運営の見直し
　HP を活用したドキュメントの電子化（情報共有強化）
　ペーパーレス化の推進
3.委員会の継続的発展
　新公益法人制度への対応
　予算の最適運用

EDA 技術専門委員会メンバと担当　（敬称略）

委員長： 三洋電機	山田　節	EDSFair/ASP-DAC 小委員会 主査、ASP-DAC 支援
副委員長：パナソニック	太田　光保	EDA 標準化小委員会 主査（IEC/TC93 WG2 国内 主査）
		国際標準化対応支援委員会 委員（IEC/TC93 担当）
副委員長：ローム	江田　努	SDF WG 主査、ＥＤＡ標準化小委員会 副主査、内規改訂、EDSFair 実行委員、ASP-DAC2009 Designer’s Forum

委員

監事：ソニー	齋藤　茂美	EDSFair 企画 WG 主査、EDSFair 実行委員
幹事：NEC エレクトロニクス	吉田　正昭	ホームページ、メールシステム、予算実績管理
幹事：シャープ	西本　猛史	ASP-DAC2009 Designer’s Forum 委員
		EDSFair 企画 WG 委員
幹事：セイコーエプソン	熊谷　敬	アニュアルレポート、EDSFair 企画 WG 委員
		システム･デザイン･フォーラム WG 委員
幹事：東芝	南　文裕	EDSFair 企画 WG 委員
幹事：富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	河村　薫	EDSFair 実行委員長、SDF WG 委員
幹事：ルネサステクノロジ	秋山　俊恭	EDSFair 実行委員、広報パンフレット
委員：エッチ･ディー･ラボ		長野　義史
委員：コーウェア		川原　常盛
委員：三洋半導体		下出　隆文
委員：ジーダット		山城　治
委員：図研		野坂　啓介
委員：凸版印刷		平尾　栄二
委員：日本ｹｲﾃﾞﾝｽ・ﾃﾞｻﾞｲﾝ・ｼｽﾃﾑｽﾞ社		安井　孝史
委員：日本シノプシス		飯島　一彦
委員：ﾏｸﾞﾏ･ﾃﾞｻﾞｲﾝ･ｵｰﾄﾒｰｼｮﾝ		船津　英世
委員：ﾏｸﾞﾏ･ﾃﾞｻﾞｲﾝ･ｵｰﾄﾒｰｼｮﾝ		牧　克眞

委員：丸紅情報システムズ	瀬谷　和宏	
委員：ﾒﾝﾀｰ・ｸﾞﾗﾌｨｯｸｽ・ｼﾞｬﾊﾟﾝ	三橋　明城男	EDSFair 企画 WG 委員
委員：リコー	前野　酉治	
委員：ローム	山本　一郎	
特別委員：三洋半導体	黒川　敦	EDA 用語辞典担当
特別委員：NEC ｼｽﾃﾑﾃｸﾉﾛｼﾞｰ	小島　智	IEC/TC93/WG2 コンベナ、EDA 標準化小委員会 副主査 国際標準化担当
特別委員：パナソニック	浜口　加寿美	SystemVerilog WG 主査
特別委員：富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	中森　勉	Power Format WG 主査
特別委員：富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	長谷川　隆	SystemC WG 主査
特別委員：ルネサステクノロジ	金本　俊幾	ナノ世代物理設計 WG 主査
客員：　近畿大学	神戸　尚志	IEC/TC93 国内委員長、元委員長
客員：　大阪大学	今井　正治	上流設計識者、ASP-DAC リエゾン
客員：　STARC	岡村　芳雄	元委員長
客員：　日本電気	若林　一敏	ASP-DAC リエゾン
事務局：JEITA	古川　昇	
事務局：JEITA	岩渕　幸吾	

活動計画の概要　＜別紙-1 参照＞

委員会の予算　会費　230,000 円 * 22 社 =5,060,000 円

委員会の開催　年 6 回程度（予定日： 別紙-2 参照）
必要に応じて幹事会を開催する

担当事務局　JEITA/電子デバイス部

＜別紙 - 1＞

活動計画の概要

1. EDA 技術の動向＆関連情報の調査検討、課題解決への提案

(1) 小委員会及び WG による技術動向とニーズ調査
- ・最先端テクノロジ ： ナノ世代物理設計 WG
- ・設計言語 ： EDA 標準化小委員会

(2) 関連機関、団体、キーパーソン等との合同会議、意見交換、交流
- ・STARC, STRJ 等

(3) 国内外の学会,研究会,イベントへの参加と連携
- ・ASP-DAC2009, DAC2008, IEICE, 軽井沢ワークショップ

2. EDA に関する標準化活動への貢献と関連機関、団体への対応

(1) EDA 設計言語およびモデル標準化のための技術的検討と提案
- ・SystemC, SystemVerilog, PowerFormat を継続
- ・Verilog-HDL,VHDL,A-HDL などは、EDA 標準化小委員会で必要に応じて対応

(2) 国際的な関連機関、団体への参画・連携と標準化活動への協力
- ・IEC/TC93 国際会議(11 月@シンガポール)への参加
- ・WG2 会議を開催(6 月@DAC(US), 1 月@EDSFair(Yokohama), 2 月@DVcon(US))
- ・IEEE/DASC,IEEE-SA, OSCI, Accellera, Si2 等との連携を強化
- ・IEEE/DASC への加入を検討

3. EDA 技術および標準化の普及推進のためのイベント実施、支援

(1) 「EDSFair2009」（横浜）
- ・日本エレクトロニクスショー協会へ運営委託
- ・半導体部会/技術委員会を活用した各社への依頼とアナウンス
- ・ASP-DAC, FPGA/PLD Conference との連携と新企画の提案
- ・特設ステージの継続実施

(2) 各種ワークショップ、講演会の開催
- ・「システム・デザイン・フォーラム 2009」を EDSFair2009 と同時開催

(3) 「ASP-DAC2009」（Yokohama）
- ・Designer’s Forum との連携

4. 委員会活動の広報

(1) 広報パンフレットの配布@EDSFair2009

(2) アニュアルレポートの発行(下記 HP でも公開)

(3) WWW ホームページの公開

(4) 活動成果の発表
- ・システムデザインフォーラム ： 標準化活動
- ・学会での公演/学術論文 ： ナノ世代

＜別紙 - ２＞

2008 年度　　JEITA/EDA 技術専門委員会 会合予定

年/月	技術専門委員会	懇親会	関連イベント
2008/4	4/25(金) (東京地区)　議事録　メンター ・08 年度事業計画説明、承認 ・08 年度小委員会/WG 計画説明、承認 ・07 年度会計収支と 08 年度会計予算説明、承認 ・07 年度版アニュアルレポート作成状況報告	○	・DATE2007（4/16-4/19）　@ Nice ・軽井沢 WS (4/21-22)　　@軽井沢
2008/5			
2008/6	6/20(金) (東京地区)　議事録　図研 ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・委員名簿更新内容確認 ・予算消費状況		・DAC2008　(6/8-6/13) @Anaheim, California
2008/7			・STARC Forum&Symp. (7/16-17) @新横浜
2008/8			・DA シンポジウム 2008 (8/26-27) @浜松
2008/9	9/19(金)　(関西地区) 議事録　コーウェア ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・予算消費状況		・TC93 国際会議　(11 月) @シンガポール
2008/10			
2008/11	11/14(金) (東京地区)　議事録　エッチ･ディ･･ラボ ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・予算消費状況 ・EDSFair 用パンフレット作成手順説明		・ICCAD2008 (11/10-13) @San Jose, California ・デザインガイア (11/17-19) @北九州
2008/12			
2009/1	1/14(水)　(東京地区)　議事録　マグマ ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・09 年度体制協議・EDSFair 用パンフレット内容確認 ・アニュアルレポート作成分担 ・手順説明・予算消費状況		・ASP-DAC2009 (1/19-21) @Yokohama ・EDSFair2009 (1/22-23)　@横浜
2009/2			
2009/3	3/19(木)　(関西地区)　議事録　沖 ・半導体部会/半導体技術委員会報告内容説明 ・08 年度小委員会/WG の年間活動報告 ・08 年度予算消費状況 ・09 年度事業計画説明 ･09 年度プロジェクト/研究会の年間活動計画説明	○	・SASIMI2009 (3/9-10) @沖縄 ・DATE2009 (4/20-24)@ Nice, 仏

1.2　2008 年度 JEITA/EDA 技術専門委員会ホームページ

1.2.1　目的

電子情報技術産業協会（JEITA）の EDA 技術専門委員会の活動状況を公開し、EDA 技術の標準化や技術調査に関するご理解とご協力をいただくことを目的とする。

1.2.2　ホームページの詳細

2006 年度よりホームページを一新し、よりわかりやすく、また、欲しい情報に簡易にアクセスできるような構成に変更を行った。ホームページは日本語版の他、英語版も用意し海外からの利用者の利便性を考慮している。日本語版、英語版の切り替えは簡単にできるように構成されている。

(1) URL: http://eda.ics.es.osaka-u.ac.jp/jeita/eda/

大阪大学のご協力を頂き、大阪大学のサーバーにホームページを設置させていただいている。またデータの更新など、メンテナンスについてもご協力を頂いている。

(2) エントリーページの構成

日本語版、英語版はそれぞれ次のエントリーで構成されている。

日本語版：	英語版：
委員会の紹介	Introduction of a committee
委員会活動	Committee activity
公開資料ライブラリ	Open data library
イベント・関連機関	Event・A related organization
お問い合わせ	Inquiry
サイトマップ	Site map

(3) 委員会の紹介/ Introduction of a committee

委員長挨拶、活動と成果、メンバーをサブエントリとする。本委員会の概要、前年度の活動内容・成果、本年度の活動計画、委員会メンバーを紹介している。

(4) 委員会活動/ Committee activity

下記の研究会・小委員会等の活動状況を紹介している。

- EDA 標準化小委員会（EDA 標準化小委員会の他、SystemC WG、SystemVerilog WG、PowerFormat WG の活動が紹介されている）
- ナノ世代物理設計 WG
- EDSFair 実行委員会

(5) 公開資料ライブラリ/ Open data library

「公開資料ライブラリ」のページでは、EDA 技術専門委員会内の各委員会の活動報告や各委員からの発表資料等を適宜掲載している。主な掲載資料を以下に示す。なお、英語版も日本語版と同一の日本語資料を掲載している。

- EDA 技術専門委員会（過去のアニュアルレポート）
- EDA 標準化小委員会（IEEE-SA セミナー、Power Format 比較表など）
- ナノ世代物理設計 WG（過去の資料）

- EDSFair 実行委員会（システムデザインフォーラムの紹介）
- システムレベル設計研究会（旧サイトへのリンク）
- その他（過去の委員会活動報告）

(6) イベント・関連機関/ Event・A related organization

関連の会議としては、次の関係の深い EDA 関連技術委員会の紹介が行われている。

- IEEE/DASC（電気電子学会/設計自動化標準化委員会）
- IEC/TC93（国際電気標準化会議/デザインオートメーション標準化技術委員会）

また、関連機関として本委員会に関連のある 17 機関の紹介があり、さらにそれぞれのホームページへのリンクが行われている。

1.3　2008 年度 JEITA/EDA 技術専門委員会 年間実績・予定表

月	EDA 技術専門委員会	
	幹事会	委員会
2008 年 4 月		4/25（金）　14:00-17:00　JEITA　402 会議室
5 月		
6 月		6/20（金）　14:00-17:00　JEITA　506 会議室
7 月		
8 月		
9 月	9/19（金）　11:00-13:00	9/19（金）　14:00-17:00　JEITA 関西支部　第２会議室
10 月		
11 月	11/14（金）　10:00-13:00	11/14（金）　14:00-17:00　JEITA　506 会議室
12 月		
2009 年 1 月	1/14（水）　11:00-13:30	1/14（水）　14:00-17:00　JEITA　502 会議室
2 月		
3 月	3/19（金）　10:00-12:30	3/19（金）　13:00-17:00　ｷｬﾝﾊﾟｽﾌﾟﾗｻﾞ京都　第２会議室

月	EDA 標準化小委員会関連
2008 年 4 月	4/22(火) 13:00-17:00　第 1 回 SystemC-WG　JEITA　406 会議室
5 月	5/30(金) 10:00-12:00　第 1 回 EDA 標準化小委員会　機械振興会館　会議室 5/16(金) 13:00-17:00　第 2 回 SystemC-WG　龍名館　竹の間
6 月	6/20(金) 13:00-17:00　第 3 回 SystemC-WG　JEITA　406 会議室
7 月	7/25(金) 11:00-13:00　第 2 回 EDA 標準化小委員会　機械振興会館　会議室 7/18(金) 13:00-17:00　第 4 回 SystemC-WG　JEITA　405 会議室
8 月	
9 月	9/19(金) 13:00-17:00　第 5 回 SystemC-WG　日本ケイデンス社　新横浜本社内会議室
10 月	10/ 2(金) 11:00-13:00　第 3 回 EDA 標準化小委員会　機械振興会館　会議室 10/17(金) 13:00-17:00　第 6 回 SystemC-WG　JEITA　関西支部　第２会議室
11 月	11/26(水) 10:00-17:00　第 1 回 Power FormatWG　龍名館　龍の間 11/21(金)-11/22(土)　第 7 回 SystemC-WG(集中審議) 下呂ロイヤルホテル　雅亭 11/27(木) 11:00-13:30　第 4 回 EDA 標準化小委員会　機械振興会館　会議室
12 月	12/19(金) 14:00-17:30　第 2 回 SystemVerilog-WG　日本ケイデンス社　新横浜本社内会議室 12/19(金) 13:00-17:00　第 8 回 SystemC-WG　JEITA　404 会議室
2009 年 1 月	1/16(金) 13:00-17:00　第 9 回 SystemC-WG　龍名館　龍の間
2 月	2/20(金) 13:00-17:00　第 10 回 SystemC-WG　JEITA　501 会議室
3 月	3/26(木) 11:00-13:30　第 5 回 EDA 標準化小委員会　機械振興会館　会議室 3/19(金) 13:00-17:00　第 11 回 SystemC-WG　九段会館 4F　あおいの間

月	ナノ世代物理設計 WG 関連
2008 年 4 月	4/18(金) 13:00-17:00　第 1 回ナノ世代物理設計 WG　日本教育会館　902 会議室
5 月	5/23(金) 10:00-17:00　第 2 回ナノ世代物理設計 WG　龍名館　竹の間
6 月	6/27(金) 13:00-17:00　第 3 回ナノ世代物理設計 WG　JEITA　502 会議室
7 月	7/25(金) 13:00-17:00　第 4 回ナノ世代物理設計 WG　JEITA　関西支部　電子会館 8 階会議室
8 月	8/29(金) 13:00-17:00　第 5 回ナノ世代物理設計 WG　JEITA　507 会議室
9 月	9/26(金) 10:00-17:00　第 6 回ナノ世代物理設計 WG　JEITA　505、507 会議室
10 月	10/24(金) 11:00-17:00　第 7 回ナノ世代物理設計 WG　JEITA　関西支部　第一会議室
11 月	11/28(金)-11/29(土)（集中審議）第 8 回ナノ世代物理設計 WG　ヴィラ伊豆（富士通保養所）
12 月	12/19(金) 13:00-17:00　第 9 回ナノ世代物理設計 WG　JEITA　403 会議室
2009 年 1 月	1/16(金) 10:00-17:00　第 10 回ナノ世代物理設計 WG　龍名館　竹の間
2 月	2/20(金) 10:00-17:00　第 11 回ナノ世代物理設計 WG　JEITA　503 会議室
3 月	3/27(金) 10:00-17:00　第 12 回ナノ世代物理設計 WG　JEITA　405 会議室

月	EDS Fair 実行委員会関係
2008 年 4 月	4/ 1(火) 13:00～17:00 第 1 回実行委員会 JESA 会議室 4/25(金) 10:00～13:50 第 2 回実行委員会 JEITA5 階「501」会議室
5 月	5/30(金) 13:00～16:50 第 1 回企画 WG JESA 会議室
6 月	6/20(金) 11:00～14:00 第 2 回企画 WG　JEITA 403⇒506 会議室 6/27(金) 14:00～18:00 第 3 回実行委員会 JESA 会議室
7 月	7/25(金) 13:00～16:30 第 3 回企画 WG　JEITA 会議室
8 月	8/28(木) 14:00～18:30 第 4 回実行委員会 JESA 会議室
9 月	9/11(木) 10:00～13:30 第 4 回企画 WG　JESA 会議室
10 月	10/ 9(木) 13:30～17:00 第 5 回実行委員会 JESA 会議室 10/23(木) 14:00～17:30 第 5 回企画 WG　JESA 会議室
11 月	11/ 6(木) 14:00～17:30 委員長会議 JESA 会議室 11/19(木) 14:00～17:30 第 6 回企画 WG　JESA 会議室 11/28(金) 13:30～17:30 第 6 回実行委員会 JESA 会議室
12 月	12/12(金) 12:00～15:00 第 7 回企画 WG　JESA 会議室
2009 年 1 月	
2 月	2/ 6(金) 10:00～12:00 幹部会議(次期体制) JESA 会議室 2/19(木) 11:00～13:00 幹部会議(次期体制) JESA 会議室 2/19(木) 13:00～15:00 第 8 回企画 WG　JESA 会議室 2/27(金) 13:30～17:00 第 7 回実行委員会 JESA 会議室
3 月	

月	システム・デザイン・フォーラム 2009WG 関連
2008 年 4 月	
5 月	5/12(月) 15:00-18:00　第 1 回システム・デザイン・フォーラム 2009WG　JESA 会議室
6 月	6/20(金) 10:00-12:00　第 2 回システム・デザイン・フォーラム 2009WG　JEITA　502 会議室
7 月	7/23(水) 14:00-17:00　第 3 回システム・デザイン・フォーラム 2009WG　JESA 会議室
8 月	
9 月	9/11(木) 14:00-17:00　第 4 回システム・デザイン・フォーラム 2009WG　JESA 会議室
10 月	10/17(金) 10:00-12:00　第 5 回システム・デザイン・フォーラム 2009WG　JESA 会議室
11 月	
12 月	12/12(金) 15:00-17:00　第 6 回システム・デザイン・フォーラム 2009WG　JESA 会議室
2009 年 1 月	1/ 8(木) 15:00-17:00　第 7 回システム・デザイン・フォーラム 2009WG　JESA 会議室
2 月	2/19(木) 15:00-17:00　第 8 回システム・デザイン・フォーラム 2009WG　JESA 会議室
3 月	

月	関連行事
2008年 4月	
5月	5/14(水) 9:30-18:00 IEEE SAセミナー 「Global Standards, Local Benefits」 ソニー（株）本社 2階 大会議場
6月	6/10(火) IEC/TC93/WG2 会議 アナハイム
7月	
8月	
9月	
10月	10/28(火)-10/30(木) IEC/TC93国際会議 IEC Asia-Pacific Regional Centre (IEC-APRC)
11月	
12月	
2009年1月	1/22(木) IEC/TC93 Secretary、JPN NC 意見交換会 パシフィコ横浜 会議室 1/23(金) IEEE DASC会議 パシフィコ横浜 会議室
2月	2/25(水) IEC/TC93/WG2 会議 サンノゼ
3月	

２．各技術委員会の活動報告

2.1　ナノ世代物理設計ワーキンググループ(Nano Scale Physical Design Working Group)

2.1.1　目的

半導体デバイス・配線テクノロジの進化に伴い、新たな設計上の課題があらわれてきている。また、これらの課題を解決するため各社が開発した手法やライブラリが、そのテクノロジが一般化した後も標準化されず、設計環境の開発・サポートコスト低減の障害となる事例や、半導体ベンダと顧客との情報授受がスムーズに行えない事例が増えてきている。

上記課題を背景として、本ワーキンググループでは、次のような調査及び標準化を実施することにより、より効率的な設計環境の実現に貢献することを目的として活動を行っている。

- 次世代(45 ナノメータ)以降のテクノロジ･ノードにおける、LSI の物理設計・検証に関する課題の抽出
- 半導体ベンダとその顧客との間でやり取りするライブラリや設計情報等を規定する、設計ルール・ガイドラインの作成
- LSI の物理設計、検証手法の精度、互換性や効率を向上できるライブラリの標準化
- 各種ライブラリを用いて行う検証が十分な精度で行えるかを判定するための標準ベンチマークデータの作成

2.1.2　活動内容

2007 年 5 月から活動を開始し、今年度は、ばらつきに関わる下記のテーマを取り上げて調査や検討を行った。

- 製造ばらつきに起因するリーク電流変動の低減法
- 32nm プロセスにおける配線自己発熱の信号伝播遅延に対するインパクト調査
- 配線ばらつきの表現手段の調査、検討

今年度の活動の成果として、以下の２項目について調査を行い、有用な知見を得た。

(1) 製造ばらつきに起因するリーク電流変動の低減アプローチ

リーク電流は LSI 製品の商品価値を大きく左右する重要な因子である。プロセスばらつきによるリーク電流ばらつきは、チップ毎にリーク電流が異なる結果となり、リーク電流ばらつきを抑えることは LSI の品質を確保する上で重要である。本活動では、プロセスばらつきが遅延とリーク電流ばらつきへ与える影響に着目した。はじめにプロセスばらつきが遅延とリーク電流へ与える影響を評価するための解析式を導出し、遅延が一定でもリーク電流ばらつきが変動する場合を議論し、リーク電流ばらつきを低減する方法を提案した。提案するリーク電流ばらつき低減方法は、実験により、何も考慮しない従来の一般設計と比較して、歩留まり 90%とした場合のリーク電流値に対して 50%の低減効果があることを示した。

(2) 32nm プロセスにおける配線自己発熱の信号伝播遅延に対するインパクト調査

現実的な物理的なパラメータを使用して、32nm プロセスにおける信号伝播遅延に対する配線自己発熱のインパクト評価を行った。配線自己発熱が顕著になると思われる回路構成の 1 つとして 64 ビットのデータ伝送モデルを評価した。解析の結果、周囲の温度からの最大温度上昇は 5.49℃、配線抵抗の増加は 4.82%となり、それによる遅延時間増加は 2.72%となった。想定した 32nm プロセスに対しては、遅延マージンとして 2.72%分に相当するインパクトを生じうることがわかった。また、実験結果は、配線自己発熱が大きな負荷を短距離配線で駆動する場合に顕著であることも示した。

これらの活動で得られた成果は、次のような形態により無償で一般に公開する。

- アニュアルレポート
- JEITA のホームページ
- 関連学会の研究会・学会における発表や論文誌への投稿

成果の詳細は本アニュアルレポートの付録に掲載した。また、今年度の成果の一部を以下の学会、にて発表した。

[1] 「製造ばらつきに起因するリーク電流変動の低減アプローチ」, 回路とシステム軽井沢ワークショップ, 2009 年 4 月.

また、本年度は EDS フェア/システムデザインフォーラムにおいて、プロセス微細化による製造ばらつきの問題に対して、「最先端統計から見た 32nm ばらつき予測と設計法」をテーマとした、「ナノ世代物理設計フォーラム」を開催。

さらに、EDS フェア会期中に開催された IEEE DASC 会議に参加、配線ばらつきを表現する Sensitivity SPEF フォーマットの策定に対応する DPC-WG との連絡を開始した。

2.1.3 関連機関の動向

米国に本部を置き、世界各国に多数の会員を持つ国際的な学会であり、電気および電子関連の標準化活動を長年にわたり実施している IEEE。その下部組織として、エレクトロニクス産業における設計自動化関連の標準化活動を行っている DASC (Design Automation Standards Committee がある。その中のワーキンググループのひとつに、P1481 Circuit Delay and Power Calculation (DPC) Working Group があり、本ワーキングのテーマのひとつである配線ばらつきの表現フォーマット(Sensitivity SPEF)策定を行っている。

一方日本では、半導体MIRAI(Millennium Research for Advanced Information Technology)プロジェクトが、ASRC（産総研次世代半導体研究センター)、ASET（技術研究組合 超先端電子技術開発機構)、Selete（株式会社 半導体先端テクノロジーズ）を中心とする産官学で共同し、半導体最先端技術の「壁」を克服する共通基盤としての役割を担っている。具体的には、半導体の微細加工において、45nmの技術世代以降の LSIの消費電力や処理速度といった基本的な性能を格段に向上させる技術を開発している。また、半導体理工学研究センターSTARC（http://www.starc.or.jp/）では、SoC設計技術等の先導開発(あすか 2)が行われている。

2.1.4　参加メンバー

主　　査	金 本 俊 幾	㈱ルネサステクノロジ
副 主 査	中 島 英 斉	ＮＥＣエレクトロニクス㈱
同	高 藤 浩 資	㈱リコー
委　　員	黒 川　　敦	三洋半導体㈱
同	蜂 屋 孝 太 郎	㈱ジーダット
同	古 川　且 洋	㈱ジーダット
同	田 中　正 和	パナソニック㈱
同	増 田 弘 生	㈱ルネサステクノロジ
同	佐　方　　剛	富士通マイクロエレクトロニクス㈱
同	奥 村 隆 昌	㈱半導体理工学研究センター
客　　員	佐 藤 高 史	東京工業大学
客　　員	橋 本 昌 宜	大阪大学

２．２　EDA 標準化小委員会

２．２．１　EDA 標準化小委員会

(1)発足の背景とミッション

JEITA/EDA 技術専門委員会の標準化活動は、1990 年の EIAJ/EDIF 研究委員会設立に始まり、当初は EDA に関するグローバルな重要課題に対して日本の業界を代表する唯一の機関として、特に設計記述言語の仕様標準化とその啓蒙等に多大な貢献を果たしてきた。近年は活動の中心が設計記述言語の普及定着と環境変化に応じて、先端的設計技術に関する調査･研究等にシフトしてきている。

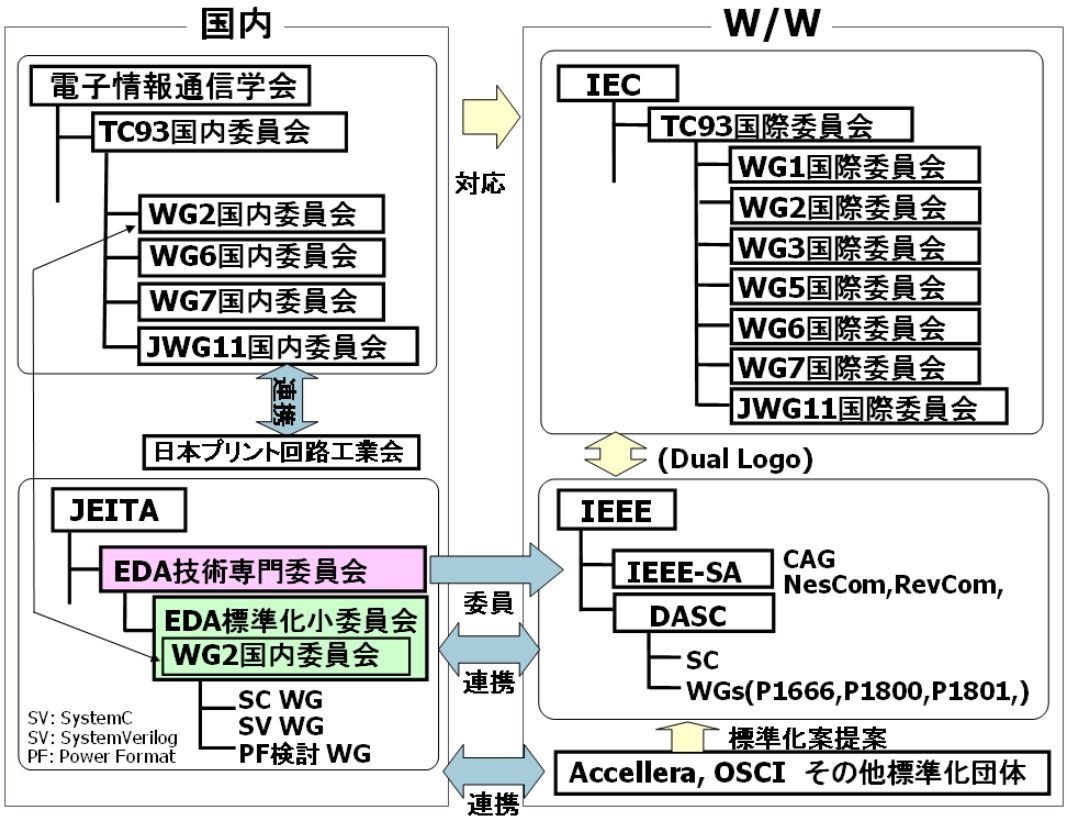
システム設計メソドロジの革新が進展する中で、設計技術言語の国際標準化は依然として重要なテーマである。そこで、標準化関連の活動をより明確に位置づけるため、2000 年 11 月に本小委員会が設立された。

世界的にみれば EDA 関連の標準は IEC(International Electrotechnical Commission)と IEEE(The Institute of Electrical and Electronics Engineers)で議論、制定されてきた。IEC ではデザインオートメーションを議論する TC(Technical Committee)93、IEEE はコンピュータソサイエティの DASC(Design Automation Standards Committee)、および SA(Standards Association)である。これまでは IEEE で定められた標準を IEC でも追認するものも多かった。2003 年より議論は IEEE の DASC/SA のワーキンググループでも、標準の制定は IEC と IEEE で同時にできるようになった(Dual Logo)。

国内では IEC の対応機関は、日本工業標準調査会（ＪＩＳＣ:Japanese Industrial Standards Committee）である。また、TC 毎に国内委員会があり、電子情報通信学会や JEITA 内に組織化されている。TC93 とハードウェア設計記述言語関連のワーキンググループ(WG2)の国内委員会は電子情報通信学会にある。

本小委員会は IEC/ＴＣ93/ＷＧ2 国内委員会を兼ねて活動するという協調体制を 2002 年度に確立した（図-1 参照)。その結果、EDA 標準化小委員会の委員が IEC/ＴＣ93/ＷＧ2 の各種標準化提案を直接審議することができるようになった。

2003 年度には、SystemC および SystemVerilog の標準化を業界として検討･推進する目的で、それぞれワーキンググループを発足させた。2007 年度には、CPF(Common Power Format)と UPF(Unified Fower Format)の二つの Power Format の標準化案の議論と統一を目的に、検討ワーキンググループを発足させた。SystemC は、ますます重要性が認識されているシステムレベルの設計言語のひとつであり、SystemVerilog は IEEE1364(VerilogHDL)の後継・検証技術の拡張である。CPF/UPF の Power Format は、主にシステム LSI の低消費電力化設計の効率化を目的とした設計言語である。これらワーキンググループは、日本の標準化組織として、海外の関連団体と連携し、言語仕様の専門的な技術検討と改善提案を通じて、標準化へ貢献すること目指して活動を行っている。



図－１　EDA 標準化小委員会と他の標準化組織との関係

以上のような発足の背景の中で、2002 年度に本小委員会は、そのミッションを以下のように規定し、活動に取り組んでいる。

目的：

EDA 標準化小委員会(以下本小委員会)は EDA(Electronic Design Automation)関連技術の標準化動向の調査、標準化の推進、標準の策定、標準案の調査、標準の保守・改定、などを推進し、もって国内外の関連業界の発展に寄与することを目的とする。

活動内容：

本小委員会は EDA および関連技術の標準化に関して、

・内外の動向を調査、検討し、
・技術および関連業界の発展に資する提案の必要性を模索し、
・必要かつ可能な場合には、関係機関に対して提案を行い、
・内外の標準化関連機関との連携・協調・協力を推進し、
・特に、デザインオートメーション/設計記述言語(TC93/WG2)WG の活動を支援し、
・また広報活動を行う。

（2）２００８年度 EDA 標準化小委員会メンバー（2009 年 3 月現在 敬称略）

主査	太田　光保	パナソニック(株)
副主査	江田　努	ローム(株)
副主査	小島　智	NEC システムテクノロジー(株) （TC93/WG2 ココンベナ）
委員	吉田　正昭	NEC エレクトロニクス(株)
委員	山田　節	三洋電機(株)
委員	西本　猛史	シャープ(株)
委員	熊谷　　敬	セイコーエプソン(株)
委員	齋藤　茂美	ソニー㈱
委員	南　　文裕	(株)東芝
委員	河村　薫	富士通マイクロエレクトロニクス(株)
委員	秋山　俊恭	(株)ルネサステクノロジ
特別委員	相京　隆	(株)半導体理工学研究センター （TC93/WG2　Member）
特別委員	長谷川　隆	富士通マイクロエレクトロニクス(株) （SystemC WG 主査、TC93/WG2　Member）
特別委員	今井　浩史	東芝(株) （SystemC WG 副主査）
特別委員	浜口加寿美	パナソニック(株) （SystemVerilog WG 主査 TC93/WG2　Member）
特別委員	中森　勉	富士通マイクロエレクトロニクス㈱ （Power Format WG 主査）
特別委員	金本　俊幾	㈱ルネサステクノロジ (ナノ世代物理設計 WG 主査)
特別委員	星野　民夫	(株)アプリスター （TC93/WG2　Member）
特別委員	石河久美子	富士通マイクロソリューションズ(株) （TC93/WG2　Member）
客員	今井　正治	大阪大学
客員	神戸　尚志	近畿大学 （TC93　国内委員長）

（3）２００８年度活動

EDA標準化小委員会としては、年5回の会合を行い、傘下のSystemCワーキンググループ、SystemVerilogワーキンググループ、Power Formatワーキンググループの活動状況の確認、標準化関連課題の議論を行った。SysyemC/SystemVerilogの両ワーキンググループは2005年末のSystemC、SystemVerilogのIEEE標準化に引き続き、新規標準化項目の調査検討、設計適用事例

の調査などの積極的な活動を継続した。具体的には、SystemCワーキンググループは、TLM2.0の標準化の推進と合成サブセットの状況調査、および、推奨設計メソドロジのガイドライン作成に取り組んだ。また、最新SystemCに関わる技術の普及促進と標準化の推進を目的として、2009年1月のシステム・デザイン・フォーラム200９でSystemCユーザ・フォーラムを開催した。

SystemVerilogワーキンググループは、IEEE1364(VerilogHDL)とIEEE1800 (SystemVerilog)の統合ドラフトに対し、国内半導体業界の意向を反映させるべくレビューを行い、IEEE-SA標準案作成に貢献し、最終案に対して賛成投票を行なった。

昨年活動を開始したPower Formatワーキンググループでは、今年は、国内半導体業界の意向を反映させるべくIEEE P1801(UPF)の標準案についての調査と検討を行い、Si2が規定しているCPFとのインターオペラビリティに関する課題の指摘・修正案の提示を行なうと共に、反対投票を行なった。

また、関連する標準化関連の組織・団体とも、下記のような活発な交流を行った。

2008 年 10 月の IEC/TC93 シンガポール国際会議には、本委員会から TC93 国内委員会委員長である神戸客員、WG2 国際コ・コンベナの小島副主査、および小委員会主査の太田の 3 名が出席した。（詳細は 2.2.3 章 IEC/TC93 の（５）項参照）

また、IEEE DASC の段階から標準の詳細についての論議が進む実態があることから、本年度より小島副主査を正規にメンバーとして登録し、深い連携がとれる体制をとった。また、2009 年 1 月には、Victor Berman 議長他を招き、DASC の月例会議を日本で開催し、本小委員会からも多数が出席した。

更に、IEEE-SA との連携・協力の一環として、IEEE-SA One-Day Seminar(Tokyo Japan)において、小島副主査と長谷川 SystemC ワーキンググループ主査が、それぞれ「日本ＥＤＡ標準化の推進状況」「The IEEE P1666 Project :日本国内における SystemC 標準化活動」というタイトルで講演し、我々の活動のアピールを行なった。

また、SystemC ワーキンググループは、TLM2.0 等の調査・検討で OSCI との連携を図っている。システム・デザイン・フォーラム 2009 では、OSCI から、後援と、TLM2.0 の最新状況に関するご講演とを頂いている。

２．２．２ IEEE/DASC(電気電子学会／設計自動化標準委員会)・IEEE-SA

（１）活動の概要

IEEE は米国に本部を置く電気、電子、情報、などの国際的な学会である。また、この分野の標準化活動を長年にわたり、しかも広範囲に実施している。DASC、SA は IEEE の下部組織として、エレクトロニクス産業における設計自動化関連の標準化活動を行っている。

活動の中心は、標準設計記述言語(HDL: Hardware Description Language)の VHDL と Verilog HDL に関連する設計と検証であり、タイミング情報、論理合成、算術関数とテストの標準化に注力している。これら設計言語に関連して、 システムレベルまで適用範囲を拡大して、Analog Mixed Signal、ソフトウェアとハードウェア協調設計等の拡張の標準化を検討している。2005 年には SystemC と SystemVerilog という高位設計技術言語、設計と検証を統合した記述言語の標準化作業が完了した。

（２）JEITA/EDA-TC との関連

これまでは EDA 技術専門委員会は IEEE/DASC のメンバーとして関連する WG に参加し、標準化案に日本の意見を反映してきた。2004 年 12 月には IEEE-SA の正式メンバーにもなり、IEEE の標準化活動に、ドラフトレビュー・標準化案の改善の提案・投票を通じて積極的に参加している。

特に、今年度は、EDSFair2009 合わせてに IEEE/DASC の月例会議を日本で開催した。(2008 年 1 月 23 日)　会議には、米国からは Victor Berman 議長、Stan Krolikoski 副議長 Dennis Brophy 氏 (Accellera Vice Chair)、本小委員会関係からは、太田主査、小島副主査、齋藤委員、山本委員、古井 TC93 国内幹事、長谷川特別委員、浜口特別委員、中森特別委員、金本特別委員、他ナノ世代物理設計ＷＧからも委員 3 名が出席した。また、国外からの電話会議出席者も 8 名あり、計 23 名が出席し、盛会であった。

会議において、EDA 技術専門委員会からは、活動状況の紹介と、SystemC TLM2.0 の標準化のマイルストーン、IEEE1801（Power Format）の balloting、Sensibity SPEF、ＡＭＳ関連言語のハーモナイゼーション等の状況確認を行なった。Sensibity SPEF については、WG の John Beatty 議長と金本特別委員がコンタクトをとることとなった。Berman 議長からは、各標準化団体間での標準の調整についての課題提起等があった。標準の調整に関する課題提起は、予ねてより、一つの分野には一つの標準を主張している JEITA の方針に合致しており、今後、重要となる案件である。

（３）これまでの成果と現在の状況

①これまでの成果

DASC/SA ではこれまでに以下の標準化作業を行っており、そのうちのいくつかは、IEC でも標準として承認されている。

(1) VHDL (Std-1076)

(2) VHDL Analog Extensions (Std-1076.1)

(3) VHDL Math Package (Std-1076.2)

(4) VHDL Synthesis Package (Std-1076.3)

(5) VHDL Timing (VITAL) (Std-1076.4)

(6) Verilog HDL (Std-1364)

(7) MVL-9 (Std-1164)

(8) Waveform and Vector Exchange (WAVES) (Std-1029.1)

(9) The RTL Synthesis Interoperability Standard (Std-1076.6)

(10) The Delay and Power Calculation Standard (Std-1481)

(11) The Open Model Fundations Standard (Std-1499)

(12) SystemVerilog (Std 1800-2005)

(13) Verilog (Std 1364-2005)

(14) SystemC 2.1 (Std 1666)

②現在の状況

2009年3月現在のDASCおよびSAとその傘下のWorking GroupとStudy Groupは以下のとおり。

- P1076 Standard VHDL Language Reference Manual (VASG)
    - VHDL-200x: the next revision
    - Issues Screening and Analysis Committee (ISAC)
    - VHDL Programming Language Interface Task Force (VHPI)
- P1076.1 Standard VHDL Analog and Mixed-Signal Extensions (VHDL-AMS)
- P1076.1.1 Standard VHDL Analog and Mixed-Signal Extensions - Packages for Multiple Energy Domain Support (StdPkgs) - this group is now part of 1076.1
- P1076.4 Standard VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification (VITAL) - This group is now part of 1076.
- P1481 Standard for Integrated Circuit (IC) Open Library Architecture (OLA) (IEEE1481R)
- P1647 Standard for the Functional Verification Language 'e' (eWG)
- P1685 SPIRIT XML Standard for IP Description (IEEE-1685)
- P1699 Rosetta System Level Design Language Standard
- P1778 ESTEREL v7 Language Standardization
- SystemVerilog Working Group
    - P1800 SystemVerilog: Unified Hardware Design, Specification and Verification Language (SV-IEEE1800) [cosponsored with IEEE-SA CAG]

- P1364 Standard for Verilog Hardware Description Language (IEEEVerilog)
- P1850 Standard for PSL: Property Specification Language (IEEE-1850)
- P1801 Standard for the Design & Verification of Low Power ICs

２．２．３ IEC/TC93（国際電気標準会議／デザインオートメーション）

（１）活動の概要

IEC は 1906 年に設立された国際標準化機関であり、本年が 102 年目ににあたる。設計自動化を取り扱う IEC/TC93 は 1992 年に設立された。TC93 の全体会議は毎年開催されており、スイス、英、仏、米、デンマーク、日、英、米、独、伊と開催されてきた。最近は、2002 年 10 月 中国・北京、2003 年 11 月・2004 年 10 月 米国・Piscataway、2005 年 9 月 日本・京都（奈良）、2006 年 9 月 ドイツ・ベルリン、2007 年 9 月米国・ガイザーズバーグでの開催があり、本年度は、10 月にシンガポールの IEC Asia-Pacific Regional Centre で開催された。

（２）TC93 の組織と参加国

2008 年 3 月現在 IEC の Web サイト(www.iec.ch)によれば､24 カ国が TC93 のメンバーとなっている。IEC のメンバー資格には、P(Participating) と O(Observer) の二種類があるが、今年度、韓国が O メンバーから P メンバーに変わったことで、P メンバーが 4 カ国、O メンバーが 20 カ国となった。韓国以外の P メンバーとしては、日本、中国、米国が登録されており、O メンバーとして、オーストラリア、ベルギー、チェコ共和国、デンマーク、エジプト、フィンランド、フランス、ドイツ、ハンガリ、インド、アイルランド、イタリア、オランダ、ロシア、セルビア、シンガポール、スペイン、スウェーデン、ウクライナ、イギリスが登録されている。幹事国は米国（国際幹事：Victor Berman 氏）が、国際会議議長は日本(唐津氏)が担当している。

（３）TC93 の組織とワーキンググループ(WG)

TC93 は７つの WG/JWG から構成されている。 特に、WG2、WG3、WG6、および WG7 は日本から提案も含め積極的な貢献をしてきた。今までの各 WG の主な活動を示す。

WG1:モデルのハーモナイゼーション： (a)STEP Electrical(ISO 規格)と EDA 標準の整合性の検討、(b)EDIF と AP-210 との整合性の検討。(c)言語間のＩｎｔｅｒｏｐｅｒａｂｉｌｉｔｙの検討

WG2:ハードウェア設計記述言語： (a)VHDL 言語仕様、Verilog HDL の整合性等の検討、システム記述言語(SLDL)も議題に取り上げられてきた。(b)IC delay&power calculation system の検討。 日本からの提案 ALR 標準化;IS(国際規格)化完。現在は

SystemC,SystemVerilog,PowerFormat が中心。

WG3:設計データ交換表現：PDX(Product data eXchange)によるマテリアルデクラレーション関連への対応の議論。

JWG11:記述の XML 化の流れへの取り組み方の議論。

WG5:規格適合性（コンフォマンス）テストの具体的事案の議論。

WG6:再利用可能部品ライブラリ、日･米･欧の各プロジェクト間の仕様整合と連携の検討、日本からは JEITA/ECALS プロジェクトの成果を提案している。IBIS も話題に取り上げられている。最近は電子カタログの流通に関する規格案が議論の中心となっている。

WG7:システムテスト記述言語、ATML(Automatic Test Markup Language)の検討。

（4）TC93 国内委員会と主要メンバー(2008 年 3 月現在。敬称略)

・ TC93 国際会議

議長: 唐津 治夢(SRI ｲﾝﾀｰﾅｼｮﾅﾙ)

・ 国内専門委員会

委員長: 神戸 尚志(近畿大学) *

幹事: 古井 芳春 (STARC)

委員: 太田 光保(パナソニック)*、柴田 明一(JPCA) 、高橋 満(日立)、山下 寛巳(SML)

・ WG2:(ハードウェア設計記述言語) 主査: 太田 光保 (パナソニック) *

国際コ・コンベナ: 小島 智(NEC システムテクノロジー) *

委員: 長谷川 隆(富士通マイクロエレクトロニクス) *、浜口 加寿美(パナソニック) *、
中森 勉(富士通マイクロエレクトロニクス)*、山田 節 (三洋) *

・ WG3:(設計データ交換表現) 主査: 神戸 尚志(近畿大学) *

・ WG6:(再利用可能部品ライブラリ) 主査: 高橋 満(日立、国際コ・コンベナ-)

・ WG７:(システムテスト記述言語) 主査: 山下 寛巳(SML)

委員: 唐津 治夢(SRI ｲﾝﾀｰﾅｼｮﾅﾙ、国際コ・コンベナ)

*印は EDA 技術専門委員会からの参加者

（5）TC93 シンガポール会議の報告

2008 年の国際会議は、10 月にシンガポールの IEC Asia-Pacific Regional Centre(IEC-APRC)で開催された。

従来、P メンバー国としては日本と米国のみの参加も多かったが、本年は、中国と、新たに P メンバー国として加わった韓国、および、O メンバー国シンガポールの計 5 カ国からの参加があり、深い議論ができた。出席者は、米国から 4 名、日本からは 7 名、韓国・中国・シンガポールからは、各 1 名の出席があった。

会議では、TC93 プレナリー会議、WG1,WG2,WG3, WG6,WG7,JWG11 の 7 会合が開催された。

WG２では､小島副主査から国際コ･コンベナとして､IEEE1076-2008/IEC61691-1-2 (VHDL) 、IEEE1800-2010/ IEC61691-6-2(Integration of System Verilog and VerilogHDL)、IEEE1801/

IEC61523-4(Power Format)、IEEE1481-1999/IEC61523-1-2(DPC)、
IEEE1497-2001/IEC61523-3-2(SDF)等の 2009 年以降に予定されているデュアルロゴ案件の報告・確認等が行なわれた。

JEITA からは EDA 技術専門委員会の活動状況や、IEEE、Accellera、OSCI 等との連携状況を紹介するとともに、Power Format に関する二つの標準化の動向に懸念を示し、インタオペラビリティの確保を求めた。

また、WG1 では、神戸国内委員長より、WG2 で標準が規定される前に、その言語やフォーマットを検討して、既存の標準とのインタオペラビリティを確認すべきであるとの提案が出され、WG1 の課題として残すこととなった。

（6）IEC 規格投票について

本年は IEC 標準に関わる投票は無かったが、2009 年度以降、次の 5 件が IEEE 標準からのデュアルロゴとして、FDIS(Final Draft International Standard)投票が行われる見込みである。

1) IEEE1076-2008(VHDL) ⇒ IEC61691-1-2
2) IEEE1800-2010(Integration of System Verilog and VerilogHDL) ⇒ IEC61691-6-2
3) IEEE1801 (Power Format) ⇒ IEC61523-4
4) IEEE1481-1999(DPC) ⇒ IEC61523-1-2
5) IEEE1497-2001(SDF) ⇒ IEC61523-3-2

２．２．４ SystemC ワーキンググループ報告

(1) 背景

ハードウェア記述言語によるシステムLSIの設計は、VHDL（IEEE 1076）やVerilog-HDL（IEEE 1364）の標準化へのJEITA(旧 EIAJ)の貢献とともに広く普及して、産業界で活用されている。一方、半導体の微細化技術は開発がさらに加速され、既に1000万ゲート規模のLSIが開発されるに至り、さらに抽象度の高いレベルからの設計が必須となってきている。1990年代半ばより複数のシステムレベル設計言語の提案が行われ、標準化推進団体が結成されたものもあった。この中で、C++言語を基本とするSystemCは広く半導体メーカ、システムメーカ、EDAベンダーの賛同を得て、Open SystemC Initiative（OSCI） が結成され、標準化のための言語仕様の策定と整備が進められてきた。

システムレベル設計言語としての要件を備えたSystemC 2.0のリファレンスシミュレータがまず2001年10月にリリースされ、その後2003年5月に言語参照マニュアル（Language Reference Manual, 以下LRM）が一般公開された。このLRMが2004年11月にOSCIよりIEEEに移管され、IEEE P1666として正式な標準化プロセスが開始された。並行してOSCIにて開発されていたSystemC 2.1の言語拡張仕様もIEEE P1666標準の一部として追加移管され、2005年12月にIEEE Std. 1666-2005としてSystemCのコア言語部分の標準化が完了した。

(2) 目的

上記のように標準化が進められたSystemCは、SoC(System on Chip)の開発のためのシステムレベル記述言語として既に設計や検証に幅広く使われるようになり、欠くことのできない言語となってきている。設計言語は設計の基本となるもので、この標準化策定に早くから関わることは、産業界にとって次世代の設計手法を構築する上で非常に重要なことである。

本ワーキンググループは2003年10月に設置され、日本国内における唯一のSystemCの標準化関連組織として、IEEE P1666で進められるSystemC標準作業に対して日本の産業界として意見を述べ、国内事情・要求事項を取り込んだ形で国際標準化に貢献していく。また、SystemCに関連した調査結果をアニュアルレポートやユーザ・フォーラム等で積極的に情報発信を行うことで、SystemCを利用した設計手法の国内普及を図り、ひいては日本の産業界の国際競争力を高めることを目指す。

(3) これまでの成果

2003年10月に発足した後、これまでに次のような成果をあげた。

① SystemC 標準化活動

- 2003年度はOSCIより2003年5月に一般公開されたLRMについてレビューを行い、問題点を62件抽出し(うち46件については2003年度の活動報告書に一覧を記載)、IEEE並びにOSCIに報告した。
- 2004年度はIEEE P1666のメンバーとして活動を行い、IEEE版のLRM(Draft)

をレビューし、43 件の問題点を IEEE に報告した。

- 2005 年度は SystemC 2.1 が追加された IEEE P1666 版 LRM についてレビューを行い、１９件の問題点を抽出し IEEE に報告した。2005 年 12 月 5 日に IEEE Std. 1666-2005 として SystemC の基本言語部分の標準化が完了した。プレスリリースも発行され、EDA-TC/JEITA としてもコメントを掲載した。
- 2006 年度は OSCI よりリリースされた TLM(トランザクションレベルモデリング) 2.0 ドラフト１、及び合成サブセットドラフトについてレビューを行い、問題点や要望事項をそれぞれ４件、５７件 OSCI に伝えた。
- 2007 年度は OSCI よりリリースされた TLM 要求仕様､用語集について分析を行い、また 11 月にリリースされた TLM 2.0 ドラフト２についてレビューを行い、問題点や要望事項を 10 件 OSCI に伝えた。
- 2008 年度は OSCI よりリリースされた TLM 2.0 正式版のユーザーズマニュアルについてレビューを行い、問題点や要望事項を 7 件 OSCI に伝えた。（詳細は「4.2.4 OSCI TLM2.0 へのフィードバック」を参照）また、ユーザーズマニュアルの抄訳を作成し、本アニュアルレポートにて公開した。

② SystemC 技術調査

- 2003 年 11 月度に集中審議を行い、本ワーキンググループ参加各社の SystemC 利用状況について紹介しあい、業界内の現状ステータスについて理解を深めた。
- 2004 年度には、過去５年間に一般に公開されている SystemC 関連の論文や発表資料等 50 件の調査を行い、報告書を作成した。
- 2005 年度は、SystemC 2.1、TLM(トランザクションレベルモデリング)、合成サブセットのテーマを定め、それぞれ分科会形式で掘り下げた調査を行った。
- 2006年度はTLMに関する動向調査を実施し、結果をSystemCユーザ・フォーラム2007にて公表。欧州ユーザと比較し、国内ユーザは低抽象度のモデルを利用する傾向が高いことを掴んだ。TLMの標準化の遅れにより再利用性があまり高くないことから、RTL設計に近いレベルでの利用に留まっていると予想される。欧州では標準化を持たずに社内でTLM標準化を行い、一丸となって利用を進めているようである。
- SystemCを用いた高位合成を効果的に行うためには記述スタイルの標準化が望ましいため、スタイルガイドの骨子を検討し、「合成スタイルガイド構成要件」としてまとめ、アニュアルレポートにて公開した。
- 2007年度はSystemCを用いた推奨設計メソドロジについて検討し、合成編について審議を完了し、アニュアルレポートにて公開した。
- 2008年度は引き続きSystemCを用いた推奨設計メソドロジについて検討し、昨年作成した合成編以外の部分について審議を完了し、本アニュアルレポートにて公開した。

③ SystemC 普及活動

- 2004 年度より、それまでの OSCI から引き継いで JEITA EDA 技術専門委員会の主催で SystemC ユーザ・フォーラムを開催している。
- 2005 年 1 月 27 日に SystemC ユーザ・フォーラム 2005 を開催。受講料は無料。定員 200 名のところ 250 名弱の聴講者が訪れ、立ち見が出るほどの盛況であった。
- 2006 年 1 月 27 日に SystemC ユーザ・フォーラム 2006 を開催。今回より、受講料を徴収することにした。(SystemC 単独の場合¥1,600、SystemVerilog と通しの場合¥2,000) 定員 200 名のところ、事前予約では満員であったが、実際に会場に訪れた聴講者は 172 名で前年比 30%減となった。 また、アンケートは 134 名の方に記入いただけた。
- 2007 年 1 月 26 日に SystemC ユーザ・フォーラム 2007 を開催。前回より値上げした影響か、聴講者は前年比 14%減の 148 名と減少した。また、アンケートは 13 2 名の方に記入いただけた。
- 2008 年 1 月 25 日に SystemC ユーザ・フォーラム 2008 を開催。今回は TLM に関するトピックを多くしたせいか、聴講者は前年比 12%増の 166 名となった。また、アンケートは 1 4 4 名の方に記入いただけた。
- 2009 年 1 月 23 日に SystemC ユーザ・フォーラム 2009 を開催。今回は聴講者が前年比 26%減の 123 名と大幅に減少した。TLM 2.0 に関しては 2008 年夏に山場を越えており、SystemC そのものにおける大きな変化がなかった点と、景気の悪化もひとつの要因と考えられる。また、アンケートは 113 名の方に記入いただけた。
- 2009 年のアンケート調査結果から、次のような事が読み取れた。(詳細については 4.2.6 を参照)

  - 今年度は SystemC の利用経験がない方の参加が多かった。
  - 過去のアンケートで、SystemC を利用しない理由としては、「HDL で十分」「言語の完成度が低い」等の回答があったが、今回のアンケートでは、これらが大幅に減少する一方で、 「効果が不明」 「検討する時間がない」といった理由が多く挙がった。これについては、本アニュアルレポートにて公開した推奨設計メソドロジが貢献できるものと期待している。
  - TLM の標準化が一段落したこともあり、標準化に対する興味よりは、適用事例に期待して参加する方が多かった。

(4) 参加メンバー

主査	長谷川　隆	富士通マイクロエレクトロニクス㈱
副主査	今井　浩史	㈱東芝
委員	中西　早苗	ＮＥＣエレクトロニクス㈱
	小島　智	ＮＥＣシステムテクノロジー㈱

	清水　靖介	ＯＫＩセミコンダクタ㈱
	長尾　文昭	三洋半導体㈱
	柿本　勝	ソニー㈱
	逢坂　孝司	日本ケイデンスデザインシステムズ社
	西園寺 修	日本シノプシス㈱
	竹村　和祥	パナソニック㈱
	牧野　潔	メンターグラフィクスジャパン㈱
	大島　良紀	㈱ルネサステクノロジ
	長谷川 裕恭	㈱エッチ・ディー・ラボ
	東島　　清宏	コーウェア㈱
客員	今井　正治	大阪大学

（計 15 名）

2.2.5 SystemVerilog ワーキンググループ

(1) 背景

昨今の LSI 設計において、一般的に使用されている Verilog HDL(ハードウエア記述言語)は、特定ツールの独自言語として開発されたが、その後、OVI（Open Verilog International）による言語仕様の公開を経て、標準化がなされた。

JEITA（旧 EIAJ）の EDA 技術専門委員会では、当時より Verilog HDL 標準化プロジェクトを設置し、継続的に言語仕様の技術検討・国際標準化に貢献してきた。

VerilogHDL は、1995 年に IEEE1364 として標準化承認がなされ、その 5 年後にはディープサブミクロン対応の機能が盛り込まれ、IEEE1364-2001 として改訂された。

その後、半導体の微細化技術はさらに進化し、1000 万ゲート規模の LSI が開発されるに至り、機能検証の網羅性と検証効率を飛躍的に向上させるために、新しいテストベンチ記述、アサーション／プロパティ記述など、いくつかの検証用言語が実用化された。

SystemVerilog は、Verilog HDL に 新たな「デザイン記述構文」の機能追加と、「検証用言語」を追加したものである。デザインにおいて、Verilog HDL に比べ「記述量の削減」や「曖昧性の排除」といったメリットがあり、設計品質の向上が期待される。また検証においては、「網羅性」や「効率」の向上が期待される。Accellera が、この言語仕様を SystemVerilog V3.1a としてまとめ、IEEE での標準化作業を経て、2005 年に IEEE1800 として標準化された。

現在は、Verilog HDL の IEEE1364 と、SystemVerilog の IEEE1800 をひとつの標準言語として統合するための標準化活動がされており、2009 年度中に IEEE1800 として標準化される計画である。本ワーキンググループは、2005 年の初の IEEE1800-2005 と IEEE1364 を統合した改訂版 IEEE1800-2009 の標準化に取組んできた。

(2) 目的

日本の半導体業界の要望に沿った形で、SystemVerilog 言語の標準化を進めることが、国内の設計現場での適用容易性を高め、「設計品質の向上」そして「国際的な競争力確保」といった結果につながる。本ワーキンググループでは、業界各社から参加したエキスパートにより、SystemVerilog 言語仕様の技術的な検討を実施し「国際標準化に貢献すること」、SystemVerilog に関する最新の情報収集と発信、「日本国内での SystemVerilog の普及推進を図ること」を目的としている。

(3) 活動内容

①グループ結成

03 年 10 月に、SystemVerilog言語仕様の技術検討・標準化を推進するためのタスクグループとして「SystemVerilogタスクグループ」を結成した。07 年度 4 月に「SystemVerilogワーキンググループ」に名前を変更し現在に至る。

②IEEE1800-2005、IEEE1800-2009 言語仕様検討とIssue Listの提出(03～07 年度)

IEEE1800-2005 の標準化に対し、04 年 8 月に改善提案を含む 32 件の Issue List をまとめ Accellera と IEEE に提出､IEEE1800-2009 の標準化に対しては､07 年 5 月に新たに 35 件の Issue List を IEEE に提出した。

各提案については、全件が採択あるいは対応を完了するまで、状況チェックとフォローを実施した。

③国際的な情報収集・標準化組織との連携と投票(03～08 年度)

04 年と 07 年には、米国で開催された国際学会 DVCon(Design & Verification Conference)にメンバを派遣し、最新技術動向の情報収集をおこなうとともに、IEEE のワーキンググループ会議参加により、日本での活動内容を報告した。

また､本ワーキンググループでの活動成果を IEEE の標準化作業に反映するため(提案の優先度をあげ投票権を得る)、IEEE の標準化機関 IEEE-SA に加入した。

そして、05 年度、08 年度の IEEE1800 投票グループに登録し、それぞれ投票を実施した。

④SystemVerilog/SystemCの対訳表の作成(05～06 年度)

SystemVerilog の専門用語に関し、多くの翻訳書や EDA ベンダが提供するユーザマニュアル類で多様な訳語が存在すると利用者の混乱を引き起こすため、標準的な訳語を定義することを目的として対訳表を作成した。SystemC ワーキンググループとも連携し、SystemC と SystemVerilog 共通の対訳表をひとつにまとめた。本対訳表を標準訳語として、出版社・EDA ベンダをはじめ、業界全体に公開し、適用されるよう推進した。対訳表は、2006 年度版アニュアルレポートに「SyetemC/SystemVerilog 専門用語対訳表」として掲載した。

⑤SystemVerilogユーザフォーラムの開催(04～06 年度)

05 年、06 年、07 年の 1 月に、EDS フェア併設の「システム・デザイン・フォーラム」のカリキュラムのひとつとして「SystemVerilog ユーザ・フォーラム」を 3 年連続開催した。

このフォーラムでは、Accellera・IEEE p1800-WG の標準化状況の説明、SystemVerilog の特徴・利点を広く周知させるための「SystemVerilog 言語チュートリアル」を 3 部に分けて実施した。

(4) 08 年度の主な活動内容

○08 年 6 月　IEEEp1800/Draft5（英語 1162 ページ)のレビュー

35 件の指摘･要望のうち、32 件が対応済みであることを確認

残り 3 件に対し、追加説明と再要求を IEEEp1800-WG に対して実施

○08 年 12 月　IEEEp1800/Draft8（英語 1162 ページ)のレビュー

35 件の指摘･要望の全件が対応済みであることを確認

○08 年 3 月　最終ドラフトを確認し投票を実施

本ワーキンググループは、この IEEE1800-2009 の投票をもって活動を終了した。

本ワーキンググループが IEEEp1800 に提出した Issue List(35 案件とその最新対応状況)を「SVTG-1　IEEEp1800-2009 Issue List」として付録に添付する。

(6) 参加メンバ

主　　査	浜　口　加寿美	松下電器産業㈱
副 主 査	明　石　貴　昭	日本シノプシス㈱
委　　員	湯　井　丈　晴	㈱沖ネットワークエルエスアイ
同	後　藤　謙　治	日本ケイデンス・デザイン・システムズ社
同	土　屋　丈　彦	㈱東芝
同	竹田津　弘　州	松下電器産業㈱
同	李　　建　道	メンター・グラフィックス・ジャパン㈱
同	高　嶺　美　夫	㈱ルネサステクノロジ
同	杉　浦　正　志	㈱図研

2.2.6 PowerFormat 検討グループ報告

(1) 背景

近年、System-LSI に対する低消費電力化の要求はますます強くなってきている。 バッテリー駆動の携帯機器用のみならず、環境保護の面から家電製品やサーバー用途に至るまで全ての領域で低消費電力化はLSI設計における最大の課題のひとつとなってきた。 一方で、性能向上の要求とプロセスの微細化によるチップあたりの回路規模の増加、さらにはリーク電流の増加が低消費電力化を困難な問題にしている。

その結果、LSI インプリ時において従来から適用されてきたクロックゲーティングやマルチ Vth といった低消費電力技術に加え、マルチ VDD・パワーゲーティング・バックゲートバイアス等の高度な技術が一般に適用される割合が飛躍的に増加している。

市販のインプリツールでもこれらの技術のサポートが進んできたが、従来の RTL では表現することが出来ない電源接続（や分離）に関する情報を記述する共通のフォーマットが設計における重要なポイントのひとつになってきた。

2006 年 6 月、CADENCE 社は 十数社の半導体ベンダー、EDA ベンダーと共に PFI(Power Forward Initiative) を組織し、電源接続やマルチ VDD,パワーゲーティングを表現可能な標準形式 CPF(Common Power Format)を策定した。 一方、SYNOPSYS と MENTOR はこの動きに対抗し、Accellera で UPF(Unified Power Format) を作り、２つの「標準」フォーマットが出来てしまった。

現在、CPF は Si2 にて Rev 1.0 が公開され、CADENCE をはじめとしたインプリツール群で 2007 年初頭からサポートされている。 UPF は Accellera にて Rev 1.0 が公開された後、IEEE に引き継がれ IEEE-P1801 として標準化作業中である。 2007 年終り頃からシノプシス、メンターのツールでのサポートも始まっている。

複数のツール間で電源表現を共通にするべく作成された両フォーマットであるが、ベンダー毎にサポートしている PF（パワーフォーマット）が異なるため、皮肉にもコンバージョンや互換性といった新たな問題を半導体ベンダーにもたらす結果となった。

(2) 目的

日本を始め多くの半導体ベンダーでは、複数の EDA ベンダーのツールを使用して LSI 設計を行っているため、２つの「標準」PF の存在は今後障害となることが予想される。 フローのある部分のツールは CPF, 別の部分のツールは UPF が必要となれば、結局フォーマットのコンバージョンが必要となり、手間が減らず互換性も問題となる。

PF を統一し、全てのツールがそれをサポートするのが理想ではあるが、既に CPF/UPF 共に仕様が固まっており、Si2, Accellera 両陣営も統一化に対しては後ろ向きであるため、現時点では非常に難しい。 ツールへの CPU/UPF インプリメントも進んでしまっている状況である為、たとえ今

すぐフォーマットが統一されたとしても全ツールがそのサポートを完了するには１～２年の時間が必要で、その間はやはり両フォーマット間のコンバージョンが必要となることが予想される。

以上の理由から、早急なフォーマットの統一は目指さず、両フォーマットのインターオペラビリティの確保を第一の目的とする。　また、IEEE P1801 へのフィードバック、投票を行う。

(3) 活動内容

ワーキンググループの結成とPF比較表の作成(08 年度 下期)

08 年 10 月に開始された IEEE P1801 (UPF) 投票に JEITAとして参加した。　具体的には 各委員によりドラフトをレビューし､特にCPFとのインターオペラビリティ保持の観点から　13 件のフィードバック（改善提案を含む）を JEITAとして合意した。　このうち 4 件が採用され、ドラフトの最新版に反映された。

主な活動は以下のミーティングとメーリングリスト上にて実施した。

2008 年 11 月 26 日　第一回ミーティング　IEEE P1801 ドラフトへの各社フィードバックすり合わせ、合意

2008 年 11 月 27 日　IEEE P1801 投票

(4) 関連機関の動向

IEEE では、P1801 の投票を　2008 年 11 月 27 日（第一回)、 2009 年 01 月 08 日（第二回）を実施、2009 年 3 月に正式に規格化見込みである。

(5) 参加メンバ

主　　査	中 森　　勉	富士通マイクロエレクトロニクス
副 主 査	南　　文 裕	東芝
委　　員	安 井 卓 也	パナソニック
同	井 上 善 雄	ルネサステクノロジ
同	中 村 正 昭	三洋半導体
同	熊 野 義 則	リコー
同	山 田 陽 一	セイコーエプソン
同	山 縣 暢 英	ソニー
同	北 原　　健	東芝
客　　員	神 戸 尚 志	IEC/TC93 国内委員長　(近畿大学)
特 別 委 員	小 島　　智	IEC/TC93/WG2 ココンベナ　(NEC システムテクノロジー)

３．各種イベント(主催／協賛)報告

3.1 Electronic Design and Solution Fair 2009 (EDSFair2009)

社団法人 電子情報技術産業協会（JEITA） 半導体部会主催により、2009 年 1 月 22 日（木）～23 日（金）の 2 日間、パシフィコ横浜にて、半導体に関連する設計、製造技術の専門性の高い情報を一堂に集めた展示会「Electronic Design and Solution Fair 2009 (略称:EDSFair2009)」を開催した。

出展社数は、143 社、来場者 9,117 人であった。

今回は、キャッチフレーズを 「未来を設計する！～先端技術を見て、聴いて、感じる 2 日間～」とし、新しいソリューションが求められる時代に対応した世界最先端技術・サービスの展示とともに、若手技術者向けのオープンセッション、国内外のベンチャー企業を集めたゾーンの設置、産学官の技術交流を深める企画、多彩なコンファレンス等を展開し、広く情報発信を行った。

キーノートスピーチでは、「より速く、より大きく、日本の創造的半導体技術」と題して、テラ・ヘルツ帯の光通信技術や、損失の少ない送電や給電を研究されてきた、首都大学東京学長の西澤潤一先生にご登壇いただき、300 名が熱心に聴講した。

また、会場内の特設ステージでは、ベンダのエグゼクティブと日本のユーザ（設計者）とのパネルディスカッションを行い、開発現場が抱えているノイズと FPGA/PLD の課題について解決案をぶつけて討論した。また、前回に引続き、『今さら聞けないことがわかる！未来がわかる！』と題した若手エンジニアや元エンジニアの方向けの講演を行った。いずれのセッションも多数の聴衆を集め、立ち見がでるほどであった。

さらに、エンジニアが注目する最新技術やトピックスに関する展示をまとめた特別ゾーンとして、SI/PI/EMC 対策をテーマに、この問題の解決策を展示紹介する、「電磁界解析・SI/PI テクノロジ・ゾーン」、FPGA 関連の出展者を集めた、「FPGA ビレッジ」、近年日本への進出が目覚ましい、インドの半導体設計技術に関連する企業が集結した、「インド・パビリオン」を新たに設置した。また、昨年から引き続き、普段接することが少ない国内外のベンチャー企業のソリューションを集めた「新興ベンダエリア」や、産学官の技術交流を深める大学の研究発表の場となる「ユニバーシティ・プラザ」、EDA 開発に携わる国内のベンチャー企業が一同に集結し、日本企業ならではの「ものづくり力」を活かした技術や製品をアピールする、「JEVeC ビレッジ」の合計 6 つの特別ゾーンを会場内に設置した。

また、日頃多忙な来場者に、効率的、効果的に EDSFair を見学いただくためのサポートサービスとして、消費電力、コスト削減、技術課題を解決するソリューションが一目でわかる案内マップを提供する「コンシェルジュサービス」を新たに企画した。また、海外出展企業を見学希望する来場者に、日本の設計技術・EDA 技術の第一人者が、ツアー・ガイドとしてブースへ同行訪問し、各社の技術紹介・質疑応答を日本語でサポートする「ガイド・ツアー」を「インド・パビリオン」まで拡大させ実施した。各ツアーとも 10～20 名の参加者がおり、訪問した新興ベンダからは非常によい企画と好評であり、参加者からも新興ベンダへアクセスするきっかけができたと大変好評であった。

3.1.1 EDSFair2009 の概要

(1) 開催期間：2009 年 1 月 22 日(木)～1 月 23 日(金)　2 日間

(2) 場所：パシフィコ横浜（展示ホール、アネックスホール）

(3)主催：社団法人電子情報技術産業協会（JEITA）

協力：Electronic Design Automation Consortium（EDAC）

後援：経済産業省、アメリカ合衆国大使館、外国系半導体商社協会（DAFS)、横浜市経済観光局（順不同）

協賛：社団法人電子情報通信学会（IEICE)、社団法人情報処理学会（IPSJ)、社団法人日本電子回路工業会（JPCA）（順不同）

運営：有限責任中間法人日本エレクトロニクスショー協会（JESA）

(4) 開催概況

① 入場者数：9,117（前年 10,431 名）

② 出展者数：143 社/317 小間（前年 169 社／339 小間）

③ 出展者セミナー：104 セッション、延べ聴講者数 2,070 名（前年 101 セッション、3,225 名）

④ スイートルーム：6 社（前年 6 社）

⑤ ユニバーシティ・プラザ：22 ブース、22 大学研究室（前年 21 ブース、21 大学研究室）

⑥ キーノートスピーチ：聴講者数 300 名（前年 355 名）

⑦併催

第 16 回 FPGA/PLD Design Conference：8 セッション、聴講者数 349 名（前年 8 セッション、515 名）

⑨同時開催

システム・デザイン・フォーラム 2009：2 セッション、聴講者数 216 名（前年 2 セッション、282 名）

ASP-DAC 2009 (Asia and South Pacific Design Automation Conference)/Designers' Forum

来場者業種内訳

	2009 年	2008 年	率
半導体・電子部品	3,747	4,736	79%
機器メーカ	2,250	2,660	85%
ツールベンダ	579	563	103%
設計関連サービス	1,071	1,095	98%
商社・営業	437	428	102%
出版・マスコミ	135	115	117%
その他	898	834	108%
計	**9,117**	10,431	87%

世界経済が低迷する中で、来場者数が昨年より約 13%減少し、2 日間合計で 9,117 名となった。

大手電機メーカーが大幅な赤字を発表しい出張規制を行うなど、大変厳しい環境の中での開催となった。

3.1.2 出展カテゴリー

(1)ハードウェア・ソリューション
システム LSI、ASIC/ASSP、MPU／MCU／DSP、FPGA/PLD デバイス、他
(2)ハードウェア開発環境（EDA)
①EDA/LSI 設計関連ツール
システムレベル設計（RTL より高位)、論理設計（RTL～ネットリスト)、論理検証、アナログ設計・検証、レイアウト、レイアウト検証・解析、LSI 信号解析、テスト設計（DFT/BIST/ATPG など)、DFM 関連（OPC/RET/PSM/LRC/TCAD など)、ASIC プロトタイピング、他
②PCB/SiP 設計関連ツール
回路図作成、アナログ設計・検証、レイアウト、SI/PI/EMC 解析、電磁界解析、熱解析、他
(3)ソフトウェア・ソリューション
組込み OS、デバイスドライバ、ファームウェア、ミドルウェア、仮想開発環境・技術、他
(4)LSI テスト、計測器
LSI テスタ、PCB テスタ、計測器、他
(5)IP コア、マクロ、セルライブラリ
(6)組込みプロセッサ開発環境
リコンフィギャラブルプロセッサ、ICE、デバッガ、マイコン CASE、コンパイラ/クロスコンパイラ、シミュレータ、ハード/ソフト協調設計環境、他
(7) 設計サービス関連（LSI/PCB)
デザインセンタ、設計サービス、設計コンサルティング、試作・製造、IP 流通サービス、他
(8)設計インフラ（WS/PC、ネットワーク）
(9)設計データ管理ツール
設計データ管理ツール、他
(10)マスクメーカ、ファウンダリメーカ
(11)大学（研究室)、コンソーシアム
(12)PR 関連
出版物、他

3.1.3 開会式

1 月 22 日(木)午前 9 時 40 分より展示会場入口において開会式を執り行った。開会式への登壇者は次のとおりである。

・ご祝辞･テープカット:

経済産業省商務情報政策局情報通信機器課長 住田孝之様

アメリカ合衆国大使館上席商務官 マーク・ワイルドマン様

横浜市経済観光局長 塚原良一様

・主催者挨拶/テープカット:

社団法人　電子情報技術産業協会　半導体部会 部会長　岡田 晴基

社団法人　電子情報技術産業協会　専務理事 半田力

・

開会式終了後、登壇者および関係者による会場一巡が行われ、本年は下記のブースを訪問し、新技術･研究開発等の成果の説明をおこなった。

（株）半導体理工学研究センター(STARC)、日本ケイデンス・デザイン・システムズ社、新興ベンダエリア、(株)図研、ユニバーシティ・プラザ（以上、見学順）

3.1.4 出展者一覧

アーム(株)
(株)アイヴィス
アットデザインリンクス(株)
　Accelicon Technologies Inc.
アトレンタ(株)
(株)ｱﾉｰﾊﾞ・ｿﾘｭｰｼｮﾝｽﾞ /ｴｰﾄｯﾌﾟﾃｯｸ(株)
アパッチデザインソリューションズ(株)
(株)アプリスター
アルデック・ジャパン(株)
E2 パブリッシング(株)
伊藤忠テクノソリューションズ(株)
　日本ヒューレット・パッカード(株)
　ネットアップ(株)
　日本アイ・ビー・エム(株)
　シーティーシー・エスピー(株)
　フォーステン・ネットワークス(株)
　(株)アノーバ・ソリューションズ
　マクニカネットワークス(株)
　アジリティ・ジャパン(株)
イノテック(株)
　Arteris Inc.
　eASIC Corporation
　Jazz Semiconductor
　Novelics LLC
　Rapid Bridge LLC
　Target Compiler Technologies N.V.
ATE サービス(株)
　Sigrity, Inc.
(株)エッチ･ディー･ラボ
エポック・マイクロエレクトロニクス
(株)沖ネットワークエルエスアイ
カーボン・デザイン・システムズ・ジャパン(株)
兼松エレクトロニクス(株)
カリプト・デザイン・システムズ(株)
コーウェア(株)
コ・フルエント デザイン
(株)コスモス・コーポレイション
CyberTec(株)
　Jasper Design Automation
サイバネットシステム(株)
サン・マイクロシステムズ(株)
シーケンスデザイン(株)
情報処理装置等電波障害自主規制協議会
(株)シルバコ・ジャパン
(株)図研
　NEC システムテクノロジー(株)
(株)スピナカー・システムズ
　ベリフィック・デザイン・オートメーション
　シリコン・デザイン・ソリューションズ
スプリングソフト(株)（旧社名；ノバフロー(株)）
スペイシャル
タナーリサーチジャパン(株)
ChipVision Design Systems
DSM ソリューションズ(株)
　Prolifi c,Inc.
　DeFacTo Technologies SA
DCG システムズ(株)
デナリソフトウエア(株)
(株)電波新聞社
日経 BP 社
日本イヴ(株)
日本ケイデンス･デザイン･システムズ社
　イノテック(株)
日本シノプシス（株）
日本ヒューレット・パッカード(株)
(株)ノア
　LogicVision, Inc.
　Nascentric, Inc.
　Javelin Design Automation
Berkeley Design Automation, Inc.
パルシックジャパンリミテッド
(株)半導体理工学研究センター(STARC)
日立情報通信エンジニアリング(株)
フォルテ・デザイン・システムズ(株)
(財)福岡県産業・科学技術振興財団
マグマ・デザイン・オートメーション(株)

三菱電機エンジニアリング(株)
MunEDA GmbH
メンター・グラフィックス・ジャパン(株)
(株)モーデック
リード・ビジネス・インフォメーション(株)
OneSpin Solutions Japan K.K.
電磁界解析・SI/PI テクノロジ・ゾーン
アジレント・テクノロジー(株)
アンソフト･ジャパン(株)
(株)エーイーティー
日本ケイデンス･デザイン･システムズ社
PHYSWARE INC
HELIC, Inc.
FPGA ﾋﾞﾚｯｼﾞ/特定非営利活動法人 FPGA ｺﾝｿｰｼｱﾑ
(株)アルティマ
特定非営利活動法人 FPGA コンソーシアム
匠ソリューションズ(株)
立野電脳(株)
東京エレクトロン デバイス(株)
(株)トーメンエレクトロニクス
特殊電子回路(株)
(有) ヒューマンデータ
富士エレクトロニクス(株)
プロトタイピング・ジャパン(株)
三菱電機マイコン機器ソフトウエア(株)
菱洋エレクトロ(株)
インドパビリオン
泉ネットワーク(有)
Wipro Limited
Kitatec KK.
コスミック・サーキット
Zensar technologies Ltd. & Zensar Advanced technologies Co.
ナレッジアイティ(株)
ﾊﾟﾄｳﾆ・ｺﾝﾋﾟｭｰﾀｰ・ｼｽﾃﾑｽﾞ・ﾘﾐﾃｯﾄﾞ
IC サービス（株）

新興ベンダエリア

Analog Rails
ALTOS DESIGN AUTOMATION
Incentia Design Systems, Inc.
EDXACT SA
nSys Design Systems Pvt Ltd
ENTASYS DESIGN, INC.
ENVIS CORPORATION
CONCEPT ENGINEERING GMBH
Certicom
Sidense Corp.
CLK DESIGN AUTOMATION, INC.
シリコン・フロントライン・テクノロジー (株)
シンテスト・ジャパン
SYNFORA
Z Circuit Automation
SOLIDO DESIGN AUTOMATION
DUOLOG TECHNOLOGIES
Dorado Design Automation, Inc.
Nangate
Pextra Corporation
POLYTEDA Software Corporation
Micrologic Design Automation Inc.
メガシス(株)
Rapid Bridge LLC
リアルインテント社
JEVeC ビレッジ
(株)アストロン
(株)礎デザインオートメーション
エイシップ・ソリューションズ(株)
ギガヘルツテクノロジー(株)
ケイレックス・テクノロジー(株)
(株)ジーダット
(株)ジェム・デザイン・テクノロジーズ
(株)システム・ジェイディー
(株)数理システム
TOOL(株)
日本 EDA ベンチャー連絡会(JEVeC)

50 音順 （※一字下げは共同出展）

3.1.5 出展傾向

EDSFair2009 は、出展者数が 143 社となった。

	出展者数	小間数
2009 年	143 社	317 小間
2008 年	169 社	339 小間
2007 年	154 社	348 小間
2006 年	148 社	343 小間
2005 年	119 社	336 小間
2004 年	104 社	306 小間
2003 年	99 社	320 小間

	2009 年		2008 年	
	社数	小間数	社数	小間数
通常出展	107 社	278 小間	115 社	294 小間
新興ベンダ（ブース）出展）	36 社	39 小間	54 社	45 小間

厳しい環境の中で、出展者数・小間数ともに大変厳しい数字となったが、新規出展者数は 35 社と昨年の 29 社を上回った。また、今年の大きな特徴として、申込後のキャンセルが続いた。最終的に 7 社 10 小間がキャンセルとなった。

3.1.6 出展者セミナー

1 セッション 45 分間で、30～100 名の適正人数のお客様に向けて集中 PR が行える出展者セミナールームを提供した。2009 年は 10 会場にて 104 セッションを開催した。

聴講者数：2,070 名

3.1.7 キーノートスピーチ

－より速く、より大きく、日本の創造的半導体技術－

首都大学東京　学長

西澤　潤一

(1) 日　時: 1 月 22 日（木）10:30～11:30

(2) 場　所:アネックスホール

(3) 聴講料:無料

(4) 聴講者数：300 名

3.1.8 第 16 回 FPGA/PLD Design Conference

(1) 日　時：1 月 22 日（木）・23 日（金）

(2) 場　所：アネックスホール

(3) 聴講料：事前申込：10,500 円（1 日券）※消費税込

当日申込：12,600 円（1 日券）※消費税込

聴講者数：

セッション 1	53 名	セッション 5	37 名
セッション 2	55 名	セッション 6	35 名
セッション 3	56 名	セッション 7	38 名
セッション 4	46 名	セッション 8	29 名

EDSFair と併設の第 16 回 FPGA/PLD Design Conference が 1 月 22 日（木）と 23 日（金）の両日、アネックスホールにて開催され、全 8 セッションで合計 349 名の聴講者を集めた。

1 月 22 日(木)

セッション 1（10:00～11:20） CPLD/FPGA の基礎と活用方法 大木　真一氏 [(株)武蔵野電波 / 東海大学 専門職大学院]
セッション 2　（11:35～12:55） FPGA 回路設計初心者のための開発トラブルシューティング 豊岡 治宏 氏 [三菱電機マイコン機器ソフトウエア(株) 第 3 事業部 開拓部 開拓第 1 課主査]
セッション 3（14：05～15：25） FPGA に搭載された専用機能を有効活用する～RAM/MAC/PLL/DCM/SerialIO～ 井倉 将実 氏 [来栖川電工(有) 応用技術部 技術取締役]
セッション 4　（15:40～17:00） FPGA に CPU を搭載し、オリジナル SoC を実現する ～　FPGA を中心とした組み込みシステムの開発　～ 実吉 智裕 氏 [株式会社アットマークテクノ 代表取締役]

1 月 23 日(金)

セッション 5（10:00～11:20） FPGA 高速 I/O を設計力で使いこなす ～DDR メモリ I/F, LVDS, PCI-Express の設計ポイント～ 櫻井 剛 氏 [三精システム株式会社 第 1 設計部 部長補佐]
セッション 6（11:35～12:55） 安心できる、高速 IO 系の基板設計初歩の初歩 金子 俊之 氏 [株式会社トッパン NEC サーキットソリューションズ 技術開発本部 設計部 スーパーバイザー]
セッション 7（14:05～15:25） System Verilog による FPGA 設計 岩本 正美 氏 [匠ソリューションズ株式会社 代表取締役社長]
セッション 8（15:40～17:00） プログラマブル・システム LSI 時代の到来 - 続々登場のミックスド・シグナル LSI の技術動向 三上 廉司 氏 [日本サイプレス(株) マーケティング・マネジャー]

3.1.9 特別ゾーン

新企画：電磁界解析・SI/PI テクノロジ・ゾーン

いまや全ての設計者にとって必須の知識となった SI/PI/EMC 対策をテーマに取り上げた、「ここまでできる！最新電磁界シミュレーション技術と SI/PI/EMC 対策」テクノロジ・ゾーンは、基板上の信号劣化要因に対するソリューションを提供する電磁界解析ツール・ベンダおよびその関連企業が一同に集まりこの問題の解決策を展示、紹介した。

出展者：

アジレント・テクノロジー(株)	日本ケイデンス･デザイン･システムズ社
アンソフト･ジャパン(株)	PHYSWARE INC
(株)エーイーティー	Helic, Inc.

また、ゾーンに併設されたミニ・ステージにおいてシステム機器ユーザが SI/PI/EMC の課題に対する取組みをプレゼンテーション形式で紹介した。

◆1 月 22 日（木）　12:00～12:20(19 名)、15:00～15:20(28 名)
プレゼンター：パナソニックエレクトロニックデバイス株式会社　デバイスアプリケーションセンター
顧問　越智　敏　氏

◆1 月 23 日（金）　13:00～13:20(44 名)、15:10～15:30(42 名)
プレゼンター：ソニー（株）モノ造り本部　モノ造り技術部門　技術開発室
担当部長　村山 敏夫　氏

下記ゾーン参加者による、自社ソリューションを紹介するプレゼンテーションも両日行なわれた。
(22 日　5 回　合計 62 名、23 日　7 回　合計 121 名)

- **高速ディジタル信号の解析に最適な MW STUDIO による時間領域シミュレーション**
  （株）エーイーティー
- **Allegro にタイトに統合された、EMS 2D フルウェーブ・フィールドソルバ**
  日本ケイデンス･デザイン･システムズ社
- **シミュレーションで設計期間短縮 アンソフトの SI、PI、EMI ソリューション**
  アンソフト･ジャパン（株）
- **High Speed, high capacity, true-parallel, 3D full-wave solution for chip-package-board SI, PI and EMI**
  パラレル処理と独自のアルゴリズムにより、フル・ウエーブ 3D フィールド・ソルバを実現した Physware のシグナルおよびパワー・インテグリティと電磁界解析のソフト
  PHYSWARE INC
- **An inductance aware RFIC design flow**
  VeloceRF によるプロセス設計キット(PDK)に統合したスパイラル･インダクタ自動生成及び伝送路やインダクタ間で起きる相互インダクタンス解析ソリューション
  Helic, Inc.

新企画：FPGA ビレッジ

出展者を集めた、FPGA ゾーンを設置した。

また、FPGA/PLD Design Conference 実行委員会の協力のもと、特設ステージにおいてエグゼクティブセッションも開催した。

1 月 22 日（木）：15:45～17:15　『FPGA/PLD の未来と課題を探る』

～ユーザの視点から FPGA/PLD の未来予測～

出展者：

(株)アルティマ
特定非営利活動法人 FPGA コンソーシアム
匠ソリューションズ(株)
立野電脳(株)
東京エレクトロンデバイス(株)
(株)トーメンエレクトロニクス
特殊電子回路(株)
(有)ヒューマンデータ
富士エレクトロニクス(株)
プロトタイピングジャパン(株)
三菱電機マイコン機器ソフトウエア(株)
菱洋エレクトロ(株)

新企画：インドパビリオン

近年、日本への進出が目覚ましい、インドの半導体設計技術に関連する企業が集結した、パビリオンを設置した。効率よくインド企業の最新情報、技術の見学が可能となった。

出展者：

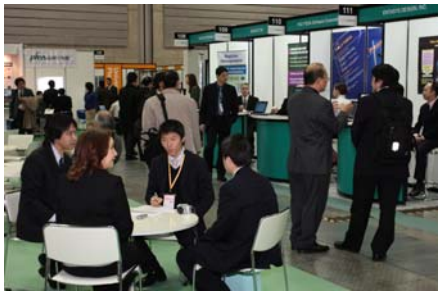
Izumi Network Yugen Kaisha
Wipro Limited
Kitatec KK.
コスミック・サーキット
Zensar technologies Ltd. & Zensar Advanced technologies Co.
ナレッジアイティ(株)
Patni Computer Systems Ltd

新興ベンダエリア

国内外新興企業 25 社の最新ソリューションが集まり、設計開発者向けに最新情報を紹介した。

出展者：

IC サービス(株)
Analog Rails
ALTOS DESIGN AUTOMATION
Incentia Design Systems, Inc.
EDXACT SA
ENTASYS DESIGN, INC.
CONCEPT ENGINEERING GMBH
Sidense Corp.
Certicom
CLK DESIGN AUTOMATION, INC.
Silicon Frontline Technology, Inc.
シンテスト・ジャパン
DUOLOG TECHNOLOGIES
Z Circuit Automation
SOLIDO DESIGN AUTOMATION
SYNFORA
Dorado Design Automation, Inc.
nSys Design Systems Pvt Ltd
Nangate
Pextra Corporation
POLYTEDA Software Corporation
Micrologic Design Automation Inc.
メガシス(株)
Rapid Bridge LLC
Real Intent, Inc.



JEVeC ビレッジ

日本の EDA の発展を目指して設立された「日本 EDA ベンチャー連絡会（JEVeC)」との協力による特別企画。EDA 開発に携わる国内のベンチャー企業が一同に集結し、日本企業ならではの「ものづくり力」を活かした技術や製品をアピールした。

出展者：

(株)アストロン
(株)礎デザインオートメーション
エイシップ・ソリューションズ(株)
ギガヘルツテクノロジー(株)
ケイレックス・テクノロジー(株)
(株)ジーダット
(株)ジェム・デザイン・テクノロジーズ
(株)システム・ジェイディー
(株)数理システム
TOOL(株)
日本 EDA ベンチャー連絡会(JEVeC)

ユニバーシティ・プラザ

産学の交流を促進すると共に、大学研究機関による研究成果を発表する場とする企画である。設計技術に関する研究成果を発表実演した。

22 大学研究室

■ **VLSI における新しい故障モデルに基づく故障検査法の開発**
愛媛大学 大学院理工学研究科 電子情報工学専攻 情報システム工学講座研究室

■ **ユビキタス･センサネットワーク向き低消費電力プロセッサ**
大阪大学 大学院情報科学研究科 情報システム工学専攻 集積システム設計学講座 今井研究室

■ **ディペンダブル再構成可能アーキテクチャとばらつき考慮設計技術**
大阪大学 大学院情報科学研究科 情報システム工学専攻 尾上・橋本研究室

■ **パターンマッチング回路の合成と応用について**
九州工業大学 情報工学研究院 電子情報工学研究系 笹尾研究室

■ **LSI 低消費電力テスト技術**
九州工業大学 情報工学府 梶原・温研究室

■ **アナログ LSI 回路設計自動化環境の構築**
九州工業大学 マイクロ化総合技術センター 中村研究室

■ **高性能・低電力システム LSI アーキテクチャとその電力削減手法**
九州大学 システム LSI 研究センター

■ **耐ソフトエラー設計支援ツールの開発**
九州大学 大学院システム情報科学研究院 情報工学部門 安浦・松永・吉村研究室
九州大学 システム LSI 研究センター
豊橋技術科学大学 工学部 情報工学系 組込みシステム研究室

■ **次世代リコンフィギャラブルロジックとその設計支援技術**
熊本大学 工学部 情報電気電子工学科/大学院自然科学研究科 情報電気電子工学専攻 末吉研究室

■ **インテリジェント・ユビキタスセンサーLSI 技術**
神戸大学 工学研究科 情報知能学専攻 プロセッサ・アーキテクチャ研究室

■ **高シグナルインテグリティ信号波形整形技術**
筑波大学 システム情報工学研究科 コンピュータサイエンス専攻 安永研究室(集積システム研究室)

■ **動的再構成可能 LSI を活用したシステム構築および技術応用**
筑波大学 大学院システム情報工学研究科 コンピュータサイエンス専攻
高度 IT 人材育成のための実践的ソフトウェア開発専修プログラム

■ **LSI 設計のための開発環境**
東海大学 専門職大学院 組込み技術研究科 清水尚彦研究室

■ **キテレツ・アイディア・コンペ**
東京工業高等専門学校

■ **VDEC の活動紹介**
東京大学 大規模集積システム設計教育研究センター

■ **システムレベル設計記述の検証・デバッグ支援技術**
東京大学 工学部電気電子工学科 藤田研究室

■ **組込み RTOS 向けアプリケーションの開発支援に関する研究報告 ～実行時ログの見える化～**
名古屋大学 大学院情報科学研究科 情報システム学専攻 高田・冨山研究室

■ **ディジットシリアル演算を用いた再構成型プロセッサと科学技術用エンジンの紹介**
広島市立大学 大学院情報科学研究科 情報工学専攻 コンピュータアーキテクチャ研究室

■ **FPGA を用いたハイパフォーマンスコンピューティング**
広島市立大学 大学院情報科学研究科 情報工学専攻 論理回路システム研究室

■ **ダイナミック・アーキテクチャ型細粒度並列処理**
広島市立大学 情報科学部 情報工学専攻 高橋隆一研究室

■ **多層プリント配線基板設計支援システム MULTI-PRIDE**
広島大学 大学院工学研究科 情報工学専攻 アルゴリズム論研究室

■ **FPGA を用いたシステム実現 --- エステティック・フィルタ, PLC, 高信頼性回路の実現 ---**
明治大学 理工学部 情報科学科 理工学研究科 基礎理工学専攻 情報科学系 井口研究室

新興ベンダプレゼンテーション

会場内に設置の特設ステージでは、国内外新興企業が、次々とプレゼンテーションを行った。

◆1 月 22 日

12：00～12：20
Ultra-fast Characterization and library validation.
(206)Altos Design Automation,Inc.
Mr.Jim McCanny　CEO

CDC Venfication
(118)REAL INTENT, INC.
Mr.Naoto Kimura,　Director, FAE Japan

15：10～15：40
Functional ECO & MMMC Timing ECO
(120)Dorado Design Automation, Inc.
Mr.J.J. Hsiao,Executive VP

PICO EXTREME-THE MOIST ADVANCED ALGORITHM SYNTHESIS
(112)SYNFORA
Mr.Atsushi Uria,　Director of Technical Support

DUOLOG's Chip Integration Platform
(108)DUOLOG TECHNOLOGIES
Mr.Brian Clinton, Product Manager

◆1 月 23 日

12：00～12：20
システム設計者のためのノイズ対策の勘どころ
(410)ギガヘルツテクノロジー株式会社
河村 隆二 氏, 代表取締役

ケイレックスがご提供する EDA サービスのご紹介
(412)ケイレックス・テクノロジー株式会社
今井 広明 氏, 営業部長

12：20～12：40
STILAccess(TM)によるテストコスト削減事例
(413)株式会社システム・ジェイディー
伊達 博 氏, 代表取締役

SiP Feasibility Study, Before CAD.
(411)Gem Design Technologies, Inc.
Mr.Hiroshi Murata, President

Anti-Counterfeiting: SOC Digital Content Protection and Management
(113)Certicom
Mr.Craig Rawlings,
Senior Director Product Management

15：30～15：50
Translation Service from India
(018)Izumi Network Yugen Kaisha
Mr.Hiroyuki Kanae

India IT Club
(020)Kitatec K.K.
Mr.Harsh Obrai

13：10～13：30
Micrologic Design Automation
(212)Micrologic Design Automation
Mr.Danny Rittman PHD, CTO

Parasitics Analysis and Management
(109)EDXACT SA
Mr.Daniel Borgraeve, VP Sales

13：30～13：50
Automated Standard Cell Library Creation and Optimization
(210)Nangate
Mr.Yutaka Kumagai,
Partner Cubic Micro (Nangate Representative)

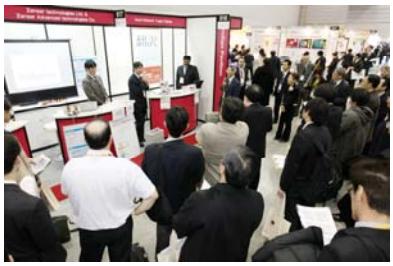
15：50～16：10
情報共有化基盤(インテリジェント Hub)機能付き SaaS サービス
(019)The Knowledge IT 株式会社
ヘアント セティア 氏, 代表取締役

Process Variation Solutions for Transistor-Level Designers
(116)SOLIDO DESIGN AUTOMATION
Mr.Johnson Lau, Senior Director

3.1.10 ガイド・ツアー

日本の設計技術・EDA 技術の第一人者が、ツアー・ガイドとしてブースへ同行訪問し、各社の技術紹介・質疑応答を日本語でサポートする「ガイド・ツアー」を「インド・パビリオン」まで拡大させ実施した。

□新興ベンダ・ツアー参加企業：１９社
　内訳：米国企業：9 社、カナダ企業：3 社、独企業：1 社、アイルランド企業：１社
　インド企業：１社、韓国企業：１社、仏企業：１社、デンマーク：1 社、台湾企業：１社
　実施回数：１月 22 日 2 回、１月 23 日 ２回
　ツアー参加人数：各ツアーとも 5－20 名



□インドパビリオン・ツアー参加企業：6 社
　実施回数:1 月 23 日 1 回
　ツアー参加人数:30 名弱

EDSFair2009 では、ツアー方式のブース訪問を企画。ツアー・ガイドが引率して海外新興ベンダのブースを訪問、コミュニケーションのサポートを日本語で行うことにより、来場者が新興ベンダの技術を理解しやすいように支援を行った。

新興ベンダ・ガイド・ツアーでは、二人の設計技術のエキスパートがツアー・ガイドを努めた。1 月 22 日は、米国 D2S, Inc の日本代表の吉田憲司氏、23 日は、東芝のシステム LSI 設計技術部長の樋渡有氏である。二人とも、LSI 設計技術や EDA に精通し、講演経験も豊富なベテランである。また、インドパビリオン・ツアーには、インド電子機器コンピュータソフトウェア輸出振興会（ＥＳＣ）日本代表のジェネシス株式会社代表取締役、西山 征夫 氏であった。

新興ベンダ・ガイド・ツアーおよびインドパビリオン・ツアーは、まず特設ステージにおいて、ツアー・ガイドによりツアーで訪問するベンダの企業紹介が行われ、各ベンダの代表者もステージ上で紹介した。ブース訪問に先立って、ベンダがどのような会社か、特徴とする製品や技術は何かを事前に理解してもらった。　企業紹介のスライドも日本語で準備されていたので、参加者にとっては、ブース訪問前にベンダについてある程度の知識を得ることができた。　ステージでの企業紹介が一通り終わると、ガイド・ツアーの旗のもと、ツアー・ガイドを先頭に紹介されたベンダのブースを訪問した。
各ブースでは、最初そのベンダの代表がパネルや、PC を使って製品や技術をさらに紹介。中には、日本語デモや、パンフレットなどの資料を準備していた企業もあり、出展者としてもこのツアーによるブース来場者を期待していた様子が見受けられた。　ツアー・ガイドは、必要に応じてベンダの説明を日本語に訳したり、また Q&A では参加者が聞きたいであろう質問を代わって行ったり、日本語でなされた質問を英語で伝えたり、また、回答を日本語で伝えたりと、ツアー参加者の技術の理解のためのコミュニーケションのサポートを行っていた。
各ツアーは、大まかに設計分野ごとでまとめられ各ツアーとも毎回 10－20 名の参加者があり、盛況であった。

3.1.11 特設ステージ

場　所：展示フロア内特設ステージ　参加無料

展示会場内の特設ステージにて、昨年に続いて、ベンダーエグゼクティブを迎えての「エグゼクティブが語る！」3 セッションと「今さら聞けないことがわかる！未来がわかる！」2 セッションを開催し、二日間で計 1049 名の聴講者を集めた。「エグゼクティブが語る！」では LSI 設計と PCB 設計にまたがる技術課題をとりあげたセッション１、FPGA コンファレンスとの連携による FPGA 関連セッション、そして、半導体ベンダー2 社のエグゼクティブとメディアのエグゼクティブを迎えて電子産業の将来と EDA 技術をとりあげたセッションを開催した。「今さら聞けない・・」では「高位合成」と「検証メソドロジ」につき、対話式講義と講演により、それぞれ注目の先端技術につき、基礎から設計適用事例までが紹介された。

今年度は EDSF 実行委員会の他の WG との連携により、特設ステージのセッションテーマと関連した展示エリアの取り組み（セッション１のテーマと関連したテクノロジーテーマゾーンの設置、セッション２に関連して FPGA パビリオン）も行われ、EDSFair 全体の一体感を示すとともに、来場誘致に貢献できた。また、特設ステージ企画 WG メンバとして、EDA 技術専門委員会と EDSF 実行委員会の EDA ベンダ委員の参加を頂き、それぞれセッションオーガナイザとして、活動頂いた。

EDSFair2009 来場者アンケートによると、来場者の約 3 分の 1 は特設ステージを来場目的にあげており、目的達成度も回答数中 44%と否達成 12%を大きく上回った。自由記入でも、「特設ステージは企画がよく、いろいろな意見を聞くことができた」など積極的評価の記載が 5 件と、他のセミナなどとの時間割の整合と、客席の窮屈度の 2 件の改善要望の記載を上回った。

資料スライドの操作、スクリーンを指示するポインタ、各セッションの時間管理などの運営面の反省点は、次年度委員に引き継ぎ、改善をお願いする。

以下、各セッション毎に、企画概要（開催案内）と、実施結果として、聴講数と簡単なコメントを記載する。

1 月 22 日（木）

■『エグゼクティブが語る！』

□13:00～14:30 セッション 1　日英同時通訳付き

「ノイズに悩まされない設計の実現に向けて ～ LSI 設計・PCB 設計・EDA ツールで、今すべきこと ～」

『LSI を PCB に実装したら想定外の電気的ノイズトラブルが発生した』こんな声をよく聞くようになってきたのは、なぜか? その原因は? どうしたら未然に防げるのか? このステージでは、LSI 設計と PCB 設計にまたがる課題を認識し、解決への道筋を探ります。

Dr. Dian Yang ［Apache Design Solutions, Inc., SVP, Product Management］

Mr. A. J Incorvia ［Cadence Design Systems, Inc., VP, Research and Development］

仮屋　和浩　氏 ［(株) 図研　取締役　技術本部長］

中村　篤 氏 ［(株) ルネサステクノロジ］

福場　義憲　氏 [東芝セミコンダクター社 システム LSI 事業部 カスタム SoC 応用設計技術部]

金子　俊之　氏 ［(株) トッパン NEC サーキットソリューションズ　技術開発本部　設計部］

【司　会】　小島 郁太郎 氏 ［㈱日経 BP 社］

【オーガナイザ】　南 文裕氏 ［東芝マイクロエレクトロニクス㈱］

聴講者数　：　225 名

オーガナイザ、司会者の周到な準備により、特設ステージの開幕にふさわしい、活発なセッションであった。会場からも多数の質問がでた。チップ、パッケージ、PCB の各設計間の協力連携が今後、ますます重要となる。会場からの質問にもあった、設計データの、共通化・標準化あるいは相互乗り入れへの取り組みの進展を期待したい。

□15:45～17:15 セッション 2　日英同時通訳付き

「FPGA/PLD の未来と課題を探る～ユーザの視点から FPGA/PLD の未来予測～」

近年、FPGA/PLD は製品づくりに不可欠なデバイスとして日本市場でも認知されつつあります。その FPGA/PLD に今後どのような成長を求め期待するのか、ユーザの視点から技術面、経済面として将来の製品に求められるソリューションを提起します。そして、それに対して、デバイスベンダー、EDA ベンダーとともに、FPGA/PLD のあるべき姿を探りながら課題を議論し、日本市場における FPGA/PLD の未来を予測します。

Mr. Danny Biran ［Altera Corporation, SVP, Product & Corporate Marketing］

小島　洋一　氏 ［ザイリンクス（株）, フィールドアプリケーションエンジニアリング本部　本部長］

Mr. Gary Meyers ［Synopsys, Inc., VP & GM, Synplicity Business Group］

Mr. Simon Bloch ［Mentor Graphics Corp., VP & GM, Design and Synthesis Division］

山本　靖 氏 ［山本靖　&　アソシエーツ］

松本　仁 氏 ［三菱電機（株）コミュニケーション製作所　光技術 G ］

【司　会】　弘中　哲夫 教授 ［広島市立大学　大学院　情報科学研究科 ］

【オーガナイザ】　松本　仁 氏 ［三菱電機（株）コミュニケーション製作所 ］

聴講者数　：　140 名

5 つの特設ステージセッション中では唯一聴講数が 100 人台だったのは残念だったが、ここ数年 EDSF への展示がない 2 大 FPGA ベンダのエグゼクティブと、関連 EDA ベンダーエグゼクティブの参加のもと、

国内 FPGA ユーザ代表とのパネル討論は FPGA/PLD の未来と課題を探るよい機会となった。

1 月 23 日（金）

■『エグゼクティブが語る！』

□10:30～11:30 セッション 3

「電子産業の成長シナリオと EDA 業界の役割」

日本の電子産業の将来像を、「携帯電話・情報家電・パソコンなどの既存製品の将来性」「環境やエネルギー問題へ向けた新製品・ビジネスモデルの創出」「MEMS など次世代技術の将来性」といった視点から語りあって頂きます。日々、難しい技術課題にとりくまれるエンジニア、管理職の方にも参考になる話が山盛りです。

望月　洋介　氏 ［日経 BP 社　電子・企画局　局長補佐］
野口　達夫 氏 ［東芝セミコンダクター社　技師長］
福間　雅夫 氏 ［NEC エレクトロニクス（株）執行役員］
【司　会】 高野　一郎 氏　［㈱テクノアソシエーツ 代表取締役社長］
【オーガナイザ】 荒木　大 氏　［㈱インターデザイン・テクノロジー］
聴講者数　：　218 名

設計や EDA の個々の技術ではなく、半導体ビジネスの視点から技術を語るセッションを、半導体ベンダエグゼクティブ（2 名）の参加を得て開催したのは、今回、初めての取り組みである。世界不況の中、半導体業界も非常に厳しい状況にあるが、このような場での業界および、関連業界関係者への発信はぜひ今後も続けてほしいものである。

■『今さら聞けないことがわかる！未来がわかる！』

□　14:00～15:00 セッション 4

「今さら聞けない高位合成　～一から学ぶ高位合成～」

最近高位合成を用いた設計事例の紹介が聞かれるようになったけど、本当はどうなの？どこまでできるの？

ここでは、高位合成を改めて理解するために、高位合成の仕組み、特長、適用範囲を設計事例も含め対話形式でレクチャー。生徒と先生の学習シリーズ第 3 弾で高位合成をわかりやすくお伝え致します。

山田　晃久 氏 ［シャープ㈱電子デバイス事業本部］
【聞き手】　銭 岩 氏 ［シャープ㈱電子デバイス事業本部］
【司　会】　西本　猛史 氏 ［シャープ㈱電子デバイス事業本部］
【オーガナイザ】　西本　猛史 氏 ［シャープ㈱電子デバイス事業本部］
聴講者数　：　263 名

5 セッション中、聴講数が最大で、高位合成に対する関心の高さが伺えた。高位合成の基本的な技術と、設計からみた動作レベル設計と RTL 設計の違いなどの説明から、実際の LSI 設計への適用事例まで、高位合成のエキスパートである講師の説明と、生徒役の若手エンジニアからの質問でわかりやすく紹介された。

□16:15～17:15 セッション 5

「検証メソドロジ入門から超並列計算機向けインターコネクトへの適用事例まで
～仏作って魂を入れる検証～」

SystemVerilog を用いた検証メソドロジが整備されたことで、検証の枠組みを容易かつ堅牢に構築することが可能になりました。しかしこれで検証の問題は解消されたのでしょうか？違います！これはまだ仏を作った段階。この中に込める「魂」ともいうべき検証項目や検証シナリオと結びついてはじめて、品質と生産性が両立した検証が可能となります。本講演では、仏としてのクラスベースの検証環境構築手法と、魂としての仕様書起点の検証手法を紹介し、検証のための組織の在り方も含めて大規模 LSI への適用事例までをご紹介します。

松岡　正 氏 ［(株）ベリフォア　代表取締役］
高山　浩一郎　氏［富士通（株）次世代テクニカルコンピューティング開発本部　LSI 開発統括部］
【司　会】　三橋　明城男 氏 ［(株）メンター・グラフィックス・ジャパン　テクニカルセールス部］
【オーガナイザ】　三橋　明城男 氏 ［(株）メンター・グラフィックス・ジャパン］
聴講者数　：　233 名

検証メソドロジとは何か？、導入のメリットは？から、超並列計算機向けインターコネクトへの適用事例までを「仏」と「魂」をキーワードに、必要なスキルや組織のありようも含め紹介された。「ぜひ検証メソドロジに飛び込んでほしい」との講師の呼びかけが、導入をためらっていた会社（部署）への、後押しになることを期待したい。

JEITA EDA 技術専門委員会 企画 WG

主　　査	ソニー(株)	齋藤　茂美
委　　員	(株)東芝	南　文裕
委　　員	三菱電機(株)	松本　仁
委　　員	(株)インターデザイン・テクノロジー	荒木 大
委　　員	シャープ(株)	西本　猛史
委　　員	メンター・グラフィックス・ジャパン(株)	三橋　明城男
委　　員	セイコーエプソン(株)	熊谷　敬
委　　員	セイコーエプソン(株)	藤原　安英
委　　員	富士通マイクロエレクトロニクス(株)	河村　薫
委　　員	三洋電機(株)	山田　節

3.1.12 来場者数詳細

2009 年来場者数

1 月 22 日(木)	晴れ	3,953
1 月 23 日(金)	晴れ	5,164
合　　計		9,117

過去の来場者数

開催年	１日目	２日目	合　計
2008 年	4,604	5,827	10,431 名
2007 年	4,956 名	6,180 名	11,136 名
2006 年	5,006 名	5,997 名	11,003 名
2005 年	5,066 名	6,087 名	11,153 名
2004 年	4,764 名	6,023 名	10,787 名

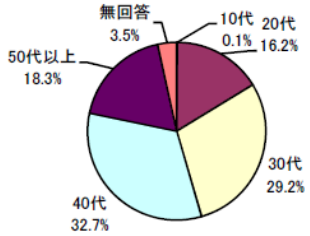
3.1.13 全来場者入場登録票アンケート回答 集計結果

入場登録票アンケートによる来場者プロファイルを以下に示す。

入場登録票アンケート(属性他)

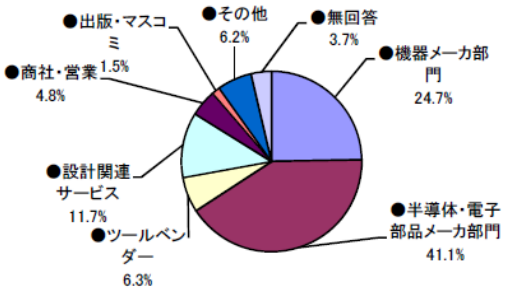
■年代

	2009(今回)	2008(参考)
10代	0.1%	0.1%
20代	16.2%	15.6%
30代	29.2%	34.5%
40代	32.7%	34.3%
50代以上	18.3%	15.4%
無回答	3.5%	0.1%
合計	100.0%	100.0%



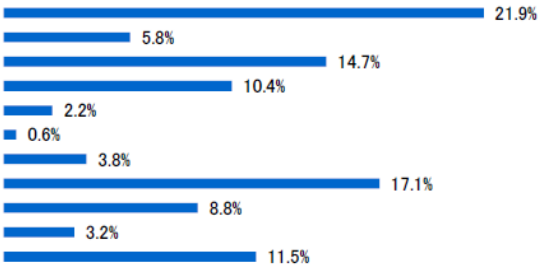
■業種

	2009(今回)	2008(参考)
●機器メーカ部門	24.7%	25.5%
●半導体・電子部品メーカ部門	41.1%	45.4%
●ツールベンダー	6.3%	5.4%
●設計関連サービス	11.7%	10.5%
●商社・営業	4.8%	4.1%
●出版・マスコミ	1.5%	1.1%
●その他	6.2%	7.7%
●無回答	3.7%	0.3%



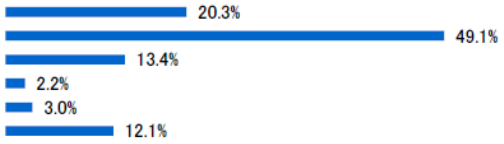
※業種詳細

●機器メーカ部門	2009(今回) 24.7%	2008(参考) 25.5%
コンピュータ関連機器	21.9%	20.9%
ネットワーク関連機器	5.8%	5.7%
一般民生機器	14.7%	16.5%
画像処理機器	10.4%	11.3%
医療機器	2.2%	1.9%
アミューズメント	0.6%	1.0%
自動車・輸送機器	3.8%	6.4%
産業機器(機械・精密機器等)	17.1%	16.1%
通信機器	8.8%	9.2%
放送機器	3.2%	1.7%
その他	11.5%	9.3%

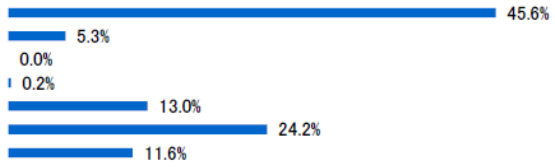


●半導体・電子部品メーカ部門	2009(今回) 41.1%	2008(参考) 45.4%
システムLSI、ASIC、マイコン、メモリ	81.9%	89.3%
FPGA/PLD	5.2%	3.2%
ディスプレイ	1.7%	1.9%
電子コンポーネント	3.1%	2.9%
プリント基板	2.4%	1.2%
その他	5.8%	1.5%

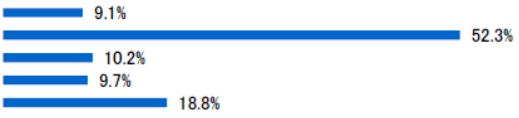
●ツールベンダー	2009(今回) 6.3%	2008(参考) 5.4%
※ツール関連が主要営業品目である商社・代理店も含む		
機能・論理設計ツール	20.3%	16.5%
LSI設計ツール	49.1%	44.2%
プリント基板設計ツール	13.4%	11.2%
マイコンツール	2.2%	3.7%
ハードウェア・ボード機器	3.0%	1.7%
その他	12.1%	22.7%



●設計関連サービス	2009(今回) 11.7%	2008(参考) 10.5%
デザインハウス	45.6%	48.3%
IPプロバイダ	5.3%	6.0%
IP流通サービス	0.0%	0.0%
ネット環境	0.2%	0.4%
教育・コンサルタント	13.0%	8.3%
ソフト開発	24.2%	25.1%
その他	11.6%	11.9%



●商社・営業	2009(今回) 4.8%	2008(参考) 4.1%
※ツール関連が主要営業品目である商社・代理店は除く		
電子機器	9.1%	14.2%
半導体	52.3%	54.7%
電子部品	10.2%	10.9%
ツール	9.7%	9.3%
その他	18.8%	10.9%



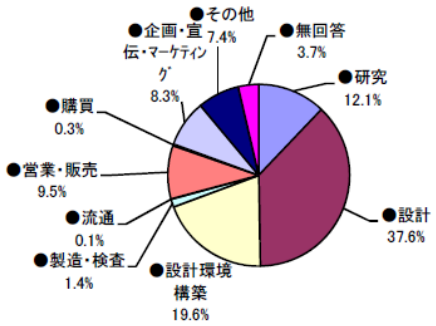
●出版・マスコミ	2009(今回) 1.5%	2008(参考) 1.1%

●その他	2009(今回) 6.2%	2008(参考) 7.7%

●無回答	2009(今回) 3.7%	2008(参考) 0.3%

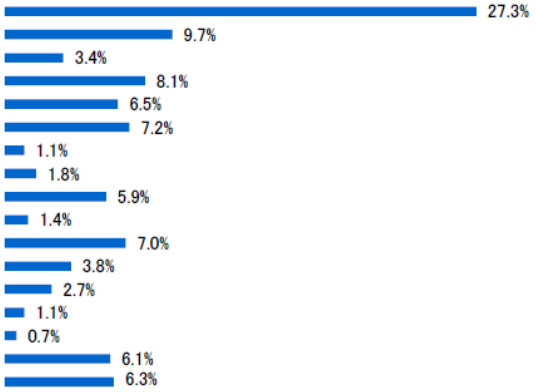
■職務

	2009(今回)	2008(参考)
●研究	12.1%	11.4%
●設計	37.6%	40.6%
●設計環境構築	19.6%	21.5%
●製造・検査	1.4%	1.7%
●流通	0.1%	0.3%
●営業・販売	9.5%	8.6%
●購買	0.3%	0.5%
●企画・宣伝・ﾏｰｹﾃｨﾝｸﾞ	8.3%	7.4%
●その他	7.4%	7.8%
●無回答	3.7%	0.2%

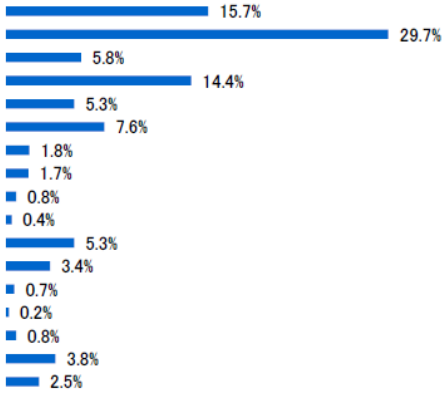


※職務詳細

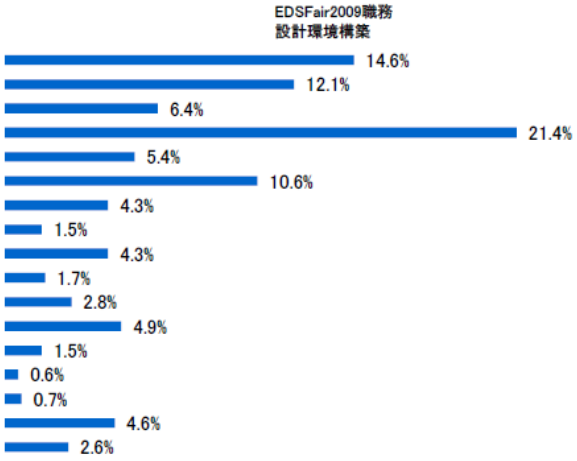
●研究	2009(今回) 12.1%	2008(参考) 11.4%
システムレベル	27.3%	26.5%
機能(RTL)	9.7%	10.4%
論理(ゲートレベル)	3.4%	2.8%
レイアウト	8.1%	6.9%
テスト	6.5%	6.7%
アナログ	7.2%	9.4%
カスタム	1.1%	2.0%
IPマクロ	1.8%	0.6%
リソ/マスク/プロセス/製造	5.9%	6.1%
TCAD	1.4%	1.2%
FPGA/PLD	7.0%	5.5%
PCB	3.8%	3.9%
IC Package	2.7%	2.2%
SiP	1.1%	0.8%
装置実装	0.7%	0.8%
ソフトウェア・ファームウェア	6.1%	6.7%
無回答	6.3%	7.7%



●設計	2009(今回) 37.6%	2008(参考) 40.6%
システムレベル	15.7%	15.3%
機能(RTL)	29.7%	33.9%
論理(ゲートレベル)	5.8%	5.1%
レイアウト	14.4%	12.5%
テスト	5.3%	5.3%
アナログ	7.6%	6.8%
カスタム	1.8%	1.8%
IPマクロ	1.7%	2.6%
リソ/マスク/プロセス/製造	0.8%	1.2%
TCAD	0.4%	0.4%
FPGA/PLD	5.3%	4.0%
PCB	3.4%	2.4%
IC Package	0.7%	0.6%
SiP	0.2%	0.3%
装置実装	0.8%	0.5%
ソフトウェア・ファームウェア	3.8%	4.0%
無回答	2.5%	3.3%



●設計環境構築	2009(今回) 19.6%	2008(参考) 21.5%
システムレベル	14.6%	14.2%
機能(RTL)	12.1%	12.2%
論理(ゲートレベル)	6.4%	6.5%
レイアウト	21.4%	20.7%
テスト	5.4%	7.3%
アナログ	10.6%	11.9%
カスタム	4.3%	5.5%
IPマクロ	1.5%	1.8%
リソ/マスク/プロセス/製造	4.3%	4.0%
TCAD	1.7%	1.8%
FPGA/PLD	2.8%	2.2%
PCB	4.9%	2.9%
IC Package	1.5%	0.7%
SiP	0.6%	0.6%
装置実装	0.7%	0.1%
ソフトウェア・ファームウェア	4.6%	4.5%
無回答	2.6%	3.1%



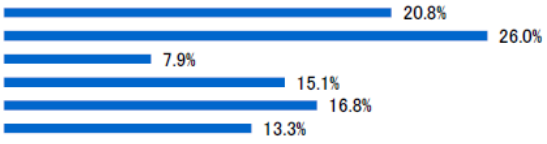
■ご来場の目的

	2009(今回)	2008(参考)
●一般展示	40.0%	43.0%
●特別展示	32.2%	24.8%
●セミナー/カンファレンス	24.2%	30.8%
●その他	0.5%	0.0%
●無回答	3.1%	1.4%

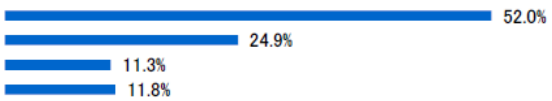
※ご来場の目的詳細

●展示関連	2009(今回) 40.0%	2008(参考) 43.0%
展示ブース	87.1%	85.5%
スイートデモ	12.9%	14.5%

●特別展示	2009(今回) 32.2%	2008(参考) 24.8%
電磁界解析・SI/PI テクノロジ・ゾーン	20.8%	0.0%
FPGAビレッジ	26.0%	0.0%
インドパビリオン	7.9%	0.0%
新興ベンダ・エリア	15.1%	27.5%
JEVeCビレッジ	16.8%	0.0%
ユニバーシティ・プラザ	13.3%	12.0%



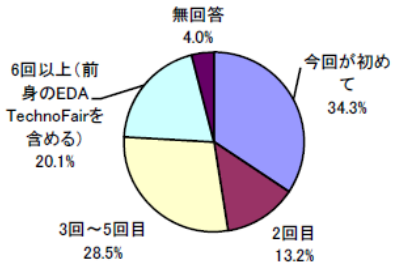
●セミナー/カンファレンス	2009(今回) 24.2%	2008(参考) 30.8%
出展者セミナー	52.0%	54.8%
キーノートスピーチ	24.9%	20.8%
FPGA/PLD Design Conference	11.3%	11.1%
システム・デザイン・フォーラム	11.8%	13.3%



●無回答	2009(今回) 3.1%	2008(参考) 1.4%

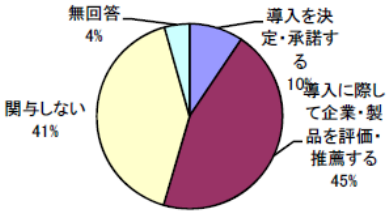
■来場頻度を教えてください

	2009(今回)	2008(参考)
今回が初めて	34.3%	37.5%
2回目	13.2%	13.7%
3回～5回目	28.5%	30.5%
6回以上(前身のEDA TechnoFairを含める)	20.1%	18.0%
無回答	4.0%	0.3%
合計	100.0%	100.0%



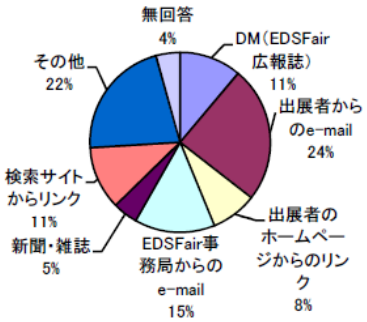
■あなたの製品導入権限について教えてください

	2009(今回)	2008(参考)
導入を決定・承諾する	9.4%	9.1%
導入に際して企業・製品を評価・推薦する	45.2%	51.8%
関与しない	41.1%	38.5%
無回答	4.4%	0.6%
合計	100.0%	100.0%



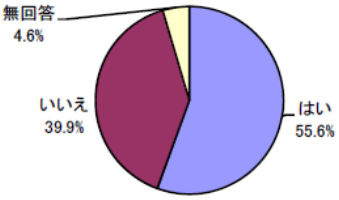
■Electronic Design and Solution Fairをどちらでお知りになりましたか？

	2009(今回)	2008(参考)
DM(EDSFair広報誌)	11.1%	15.6%
出展者からのe-mail	24.4%	26.6%
出展者のホームページからのリンク	8.3%	6.9%
EDSFair事務局からのe-mail	14.4%	11.9%
新聞・雑誌	4.5%	4.7%
検索サイトからリンク	11.3%	11.0%
その他	21.6%	22.0%
無回答	4.3%	1.3%
合計	100.0%	100.0%



■今後、Electronic Design and Solution Fairからの情報配信を希望しますか？

	2009(今回)	2008(参考)
はい	55.6%	57.1%
いいえ	39.9%	41.7%
無回答	4.6%	1.2%
合計	100.0%	100.0%



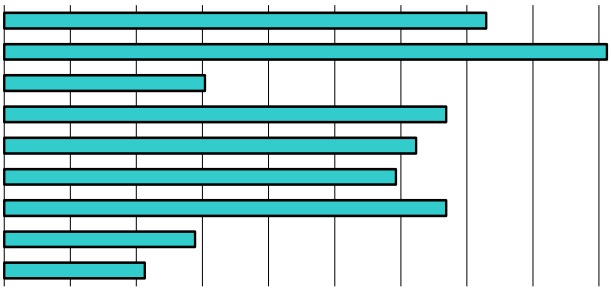
3.1.14 出展社アンケート回答 集計結果

出展社アンケートの回答結果を示す。

出展社 143 社（窓口数 117 社）の内、回答 51 社回答。

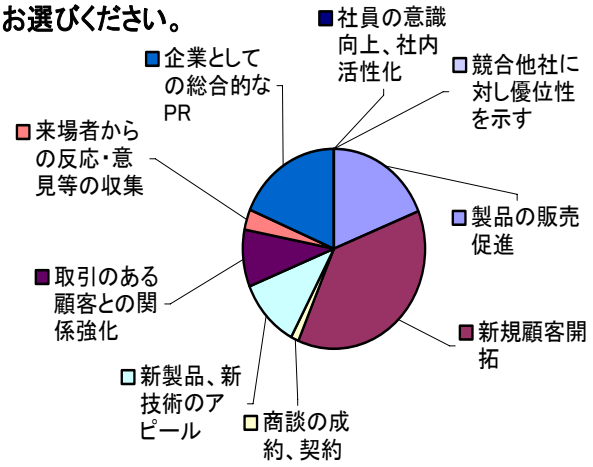
Q1-1. 今回の出展にあたっての目的について該当するものをお選びください。(いくつでも)

	（回答数）	2009年度	2008年度
製品の販売促進	48名	14.6%	16.4%
新規顧客開拓	60名	18.2%	19.6%
商談の成約、契約	20名	6.1%	7.5%
新製品、新技術のアピール	44名	13.4%	15.0%
取引のある顧客との関係強化	41名	12.5%	8.9%
来場者からの反応・意見等の収集	39名	11.9%	11.7%
企業としての総合的なPR	44名	13.4%	14.5%
競合他社に対し優位性を示す	19名	5.8%	4.7%
社員の意識向上、社内活性化	14名	4.3%	1.9%
合計	**329名**	**100.0%**	**100.0%**



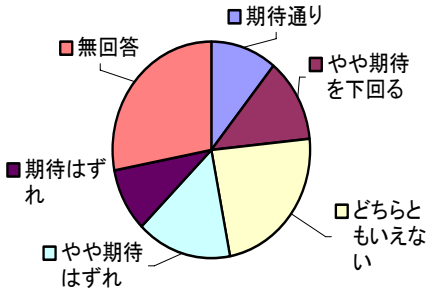
Q1-2. なかでも今回の出展にあたってもっとも重視した目的一箇所をお選びください。

	（回答数）	2009年度	2008年度
製品の販売促進	12名	18.8%	17.6%
新規顧客開拓	24名	37.5%	49.0%
商談の成約、契約	1名	1.6%	2.0%
新製品、新技術のアピール	7名	10.9%	9.8%
取引のある顧客との関係強化	6名	9.4%	2.0%
来場者からの反応・意見等の収集	2名	3.1%	3.9%
企業としての総合的なPR	12名	18.8%	15.7%
競合他社に対し優位性を示す	0名	0.0%	0.0%
社員の意識向上、社内活性化	0名	0.0%	0.0%
合計	**64名**	**100.0%**	**100.0%**



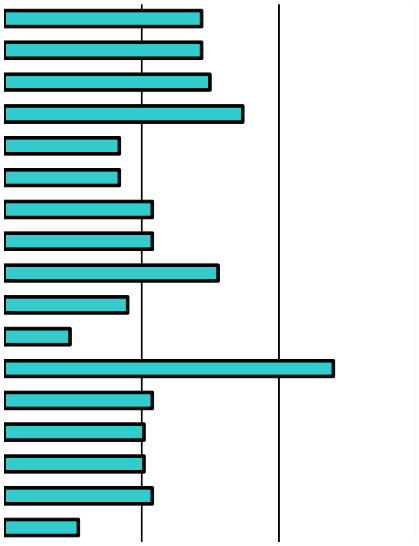
Q1-1で「製品の販売促進」を目的とされていた場合、当項目の評価をお聞かせください。

	（回答数）	2009年度	2008年度
期待通り	7名	10.9%	7.8%
やや期待を下回る	8名	12.5%	7.8%
どちらともいえない	15名	23.4%	29.4%
やや期待はずれ	10名	15.6%	11.8%
期待はずれ	6名	9.4%	2.0%
無回答	18名	28.1%	41.2%
合計	**64名**	**100.0%**	**100.0%**
加重平均値		-1.6%	1.8%



Q2. 出展にあたってターゲットとした来場者の業種について該当するものをお選びください。(いくつでも)

	(今回回答数)		2008年度 下度(参考)
【機器メーカ・部門】コンピュータ関連機器	24名	7.2%	6.9%
【機器メーカ・部門】ネットワーク関連機器	24名	7.2%	6.9%
【機器メーカ・部門】一般民生機器	25名	7.5%	8.2%
【機器メーカ・部門】画像処理機器	29名	8.7%	8.8%
【機器メーカ・部門】医療機器	14名	4.2%	3.5%
【機器メーカ・部門】アミューズメント	14名	4.2%	4.1%
【機器メーカ・部門】自動車・輸送機器	18名	5.4%	6.0%
【機器メーカ・部門】産業機器(機械・精密機器	18名	5.4%	6.0%
【機器メーカ・部門】通信機器	26名	7.8%	7.9%
【機器メーカ・部門】放送機器	15名	4.5%	4.4%
【機器メーカ・部門】その他	8名	2.4%	2.8%
【半導体・電子部品メーカ部門】システムLSI、A	40名	12.0%	12.9%
【半導体・電子部品メーカ部門】FPGA/PLD	18名	5.4%	5.0%
【半導体・電子部品メーカ部門】ディスプレイ	17名	5.1%	4.4%
【半導体・電子部品メーカ部門】電子コンポーネ	17名	5.1%	4.1%
【半導体・電子部品メーカ部門】プリント基板	18名	5.4%	4.4%
【半導体・電子部品メーカ部門】その他	9名	2.7%	3.7%
合計	**334名**	**100.0%**	**100.0%**



なかでも出展にあたって最も重視したターゲット一箇所をお選びください。

	(今回回答数)		2008年度 下度(参考)
【機器メーカ・部門】コンピュータ関連機器	0名	0.0%	2.0%
【機器メーカ・部門】ネットワーク関連機器	1名	2.0%	2.0%
【機器メーカ・部門】一般民生機器	6名	11.8%	12.0%
【機器メーカ・部門】画像処理機器	6名	11.8%	6.0%
【機器メーカ・部門】医療機器	0名	0.0%	0.0%
【機器メーカ・部門】アミューズメント	0名	0.0%	0.0%
【機器メーカ・部門】自動車・輸送機器	0名	0.0%	2.0%
【機器メーカ・部門】産業機器(機械・精密機器	1名	2.0%	0.0%
【機器メーカ・部門】通信機器	1名	2.0%	4.0%
【機器メーカ・部門】放送機器	0名	0.0%	0.0%
【機器メーカ・部門】その他	1名	2.0%	0.0%
【半導体・電子部品メーカ部門】システムLSI、A	22名	43.1%	50.0%
【半導体・電子部品メーカ部門】FPGA/PLD	3名	5.9%	8.0%
【半導体・電子部品メーカ部門】ディスプレイ	0名	0.0%	4.0%
【半導体・電子部品メーカ部門】電子コンポーネ	0名	0.0%	2.0%
【半導体・電子部品メーカ部門】プリント基板	4名	7.8%	2.0%
【半導体・電子部品メーカ部門】その他	1名	2.0%	0.0%
無回答	5名	9.8%	6.0%
合計	**51名**	**100.0%**	**100.0%**

3.1.15 まとめ

EDSFair2009 の過去にない特徴は、年度の開始時点と EDSFair 開催時点で、世の中の情勢が全く異なるものとなっていたことである。実行委員会を立ち上げ、年度の方針や計画を決定した春の時点では、EDSFair2008 を上回る出展者数、来場者数を期待していたが、第三四半期頃から顕在化した不況の嵐は、まさに Fair 開催のころには最悪の状況となっていった。大手企業は軒並み大幅な赤字予想を発表し、電気各社も出張を含めた極めて厳しい予算措置をとっている最中での開催となった。EDSFair2009 も２０％以上の出展者数、来場者数減少を覚悟し、一旦決定していた予算も大幅にカットした。

このような状況の中で、出展者数で１５％、来場者数で１３％の減少で抑えられたのは、まさに関係者の努力によるところが大きいと言える。Tech ON!においても、「EDSF，参加者数 13%減で踏ん張る」という形で記事となった。

出展者から見た今年の目玉は、テクノロジーテーマゾーン、インドパビリオン、FPGA ビレッジの３つの新企画であった。いずれの企画も多数の来場者を集め、アンケートにおいても高い評価を得ており、EDSF でしか得られない情報の発信という当初の目的を十分に達成できた。

来場者の観点では、コンシェルジュサービスや通訳サービスを新規に行った。利用者には満足してもらえたので、次回以降は来場者にさらに周知させることで来場者増につながる企画になると期待したい。

今回は最終的に実現できなかったが、メディア各社がテーマや出展者を独自に集めて特別ゾーンで展示するというメディアゾーン企画は、この不況の嵐さえなければ実現できていたものであった。出展企業の分野拡大の観点からは大変有効であり、来年度以降継続して検討の価値があると思われる。

今回出展者数、来場者数ともに減少ということになったが、これを機として、出展者の間に、現在の延長では来年度の開催が危ないのではないか、新機軸を出さないと自社の出展もできなくなるか、良くても縮小せざるを得ない、結果として出展者、小間数が減った場合には、予算的に開催ができなくなるのではないかという不安と問題意識が一気に広まった。

EDA-TC 幹事会では、元々委員の負荷の軽減という立場から EDSFair 開催方法の見直しを進めていたこともあり、出展者の意見を聞いたうえで、次年度以降の開催方法について早急に方針の決定を行うこととした。

3.2　システム・デザイン・フォーラム 2009

3.2.1　はじめに

最新の EDA 技術の標準化推進、業界内での普及･促進活動の一環として、EDA 技術専門委員会主催による“システム・デザイン・フォーラム 2009”を、EDSFair2009 と同期して 2009 年 1 月 23 日に開催した。

EDA 技術専門委員会は、今回と同様の目的とするフォーラムを、ほぼ毎年継続して開催してきた。まず、1990 年から 1994 年にかけて EDA 標準化活動の発表とその一般への普及を図ることを目的とした“EDA 標準化フォーラム”を 4 回開催し、つづいて、EDA 技術専門委員会の活動に係る内容の発表、討論の場を目的として“EDA フォーラム”を 1999 年から 2002 年にかけて 2 回開催してきた。また、2004 年には、最新の設計技術、課題を設計事例とともに紹介する“システム・デザイン・セミナー”を、2005 年には “システム・デザイン・フォーラム 2005”を、2 日間の日程で開催した。この“システム・デザイン・フォーラム 2005”では、1 日目に SystemVerilog ユーザ・フォーラムと SystemC ユーザ・フォーラムを、2 日目に SoC に関連した設計技術、課題等を含めた設計事例を紹介する 2 セッションと、LSI、パッケージ、基板を含めた統合設計に関するパネル討論のセッションを行っている。2006 年には、“システム・デザイン・フォーラム 2006”として、“SystemVerilog ユーザ・フォーラム”と“SystemC ユーザ・フォーラム”の 2 セッション構成で、両設計言語の標準化動向の紹介、チュートリアル、設計適用事例の紹介を行った。2007 年は“システム・デザイン・フォーラム 2007”として、“SystemVerilog ユーザ・フォーラム”と“SystemC ユーザ・フォーラム”の 2 セッションに、65nm 以下のプロセスノードで深刻化するプロセスばらつきを打破する最新の設計技術動向を紹介するフィジカル･デザイン･フォーラムを新たに加え、計 2 日間 3 セッションを開催した。去年は、“システム・デザイン・フォーラム 2008”として、“SystemC ユーザ・フォーラム”と“Power Format フォーラム”の 2 セッションを開催した。“SystemC ユーザ・フォーラム”では、最新の SystemC 標準化動向、TLM2.0 のチュートリアル、JEITA SystemC ワーキング・グループの取り組みの報告と、設計適用事例の紹介を行った。“Power Format フォーラム”では、最新の低消費電力設計技術の紹介と､個々に Power Format 標準化を目指す二つの団体 Accellera Organization, Inc.、Si2(Silicon Initiative, Inc.双方からの標準化活動の最新状況や設計適用事例の紹介と、JEITA Power Format 検討ワーキング・グループの Power Format の標準化に対する検討状況の報告を行った。

今年は、「SystemC ユーザ・フォーラム 2009」に加えて、新たに、プロセス微細化による製造ばらつきの問題に対して、「最先端統計から見た 32nm ばらつき予測と設計法」をテーマとした、「ナノ世代物理設計フォーラム」を開催した。

SystemC ユーザ・フォーラム 2009 では、OSCI（Open SystemC Initiative）による SystemC の最新動向の紹介、JEITA SystemC ワーキング・グループによる、システム設計から実装、検証を含む SystemC 推奨設計メソドロジの紹介、半導体理工学研究センター（STARC）による TL モデリングガイドの紹介、SystemC を用いた高位合成適用事例、および TLM2.0 を利用した回路設計事例を報告した。

ナノ世代物理設計フォーラムでは、プロセスの微細化により、新たな設計上の課題としてあらわれてきた製造ばらつきによる設計の収束性および製造時の良品率の低下に対処するため、ばらつきの影響を考慮できる統計的な設計手法の現状を報告した。

3.2.2　システム・デザイン・フォーラム 2009 概要

- 開催日時

1 月 23 日(金)	10:00～12:00	**SystemC ユーザ・フォーラム 2009**
	12:45～16:30	**ナノ世代物理設計フォーラム**

- 開催場所

  パシフィコ横浜 アネックスホール

- 聴講料（消費税込）

	事前申込	当日申込
SystemC ユーザ・フォーラム 2009	2,100 円	2,625 円
ナノ世代物理設計フォーラム	3,150 円	4,200 円

- 定員

  200 名/セッション

- 主 催

  社団法人 電子情報技術産業協会 EDA 技術専門委員会

- 協 賛

  OSCI(Open SystemC Initiative)

- 受 付

  インターネットで受け付け

  http://www.edsfair.com/conference/systemdesign.html

- 開催案内プログラム

社団法人電子情報技術産業協会(JEITA)EDA 技術専門委員会では、最新 EDA 技術の普及促進を目的としてシステム・デザイン・フォーラムを 2005 年から開催しております。その中で今年は、「SystemC ユーザ・フォーラム 2009」に加えて、新たに、プロセス微細化による製造ばらつきの問題に対して、「最先端統計から見た 32nm ばらつき予測と設計法」をテーマとした、「ナノ世代物理設計フォーラム」を開催いたします。

SystemC ユーザ・フォーラム 2009 では、OSCI（Open SystemC Initiative）によります SystemC の最新動向の紹介、JEITA SystemC ワーキング・グループによります、システム設計から実装、検証を含む SystemC 推奨設計メソドロジの紹介、半導体理工学研究センター（STARC）によります TL モデリングガイドの紹介、SystemC を用いた高位合成適用事例、および TLM2.0 を利用した回路設計事例のご紹介をいたします。是非、システム LSI 設計の最先端状況の把握と、論議の場としてお役立てください。

LSI物理設計・検証においては、半導体デバイス・配線テクノロジの進化に伴い、新たな設計上の課題があらわれてきています。その一つとして、プロセスの微細化に伴う製造ば

らつきが顕著化し、設計の収束性および製造時の良品率を低下させています。こうしたばらつきに対処するため、ばらつきの影響を考慮できる統計的な設計手法が注目を集めています。そこで、ナノ世代物理設計フォーラムでは、「デバイス・回路ばらつきの要因と統計的側面」および「Statistical回路設計イノベーション」をサブタイトルとして、国内外から著名な先生方をお招きし、ばらつきを考慮した設計方法の現状を紹介していただきます。このフォーラムが、次世代以降のLSI物理設計における新たな課題を知る良い機会となり、また論議する良い場になると確信しております。

なお、ナノ世代物理設計フォーラムは、EDA 技術専門委員会活動開始 10 周年記念事業として、国内外から著名な先生をお招きしており、特別聴講料となっております。奮ってご参加下さい。

● **SystemC ユーザ・フォーラム 2009 (1 月 23 日 10:00～12:00)**

司会：長谷川 隆 氏

（SystemC ワーキング・グループ 主査/富士通マイクロエレクトロニクス）

SystemC は、2005 年 12 月に IEEE 標準仕様として IEEE 1666-2005 が承認された後も、TLM2.0 の LRM 作成、動作合成サブセット策定、アナログ/アナログーデジタル混在拡張仕様の検討が進められており、適用領域がさらに広がっています。実設計においても、C 言語ベースのシステムレベル設計言語の業界標準として、検証、設計分野で幅広く利用されており、今後にも期待が寄せられています。

本セッションでは、1)OSCI による SystemC の最新動向の紹介、2)JEITA SystemC ワーキング・グループによる、システム設計から実装、検証を含む SystemC 推奨設計メソドロジの紹介、3) STARC による TL モデリングガイドの紹介、4) SystemC を用いた高位合成適用事例、5) TLM2.0 を利用した回路設計事例の発表を行います。

① OSCI アップデート

Stan Krolikoski 氏 （OSCI）

② SystemC 推奨設計メソドロジの紹介

SystemC ワーキング・グループ

③ STARC TL モデリングガイドの紹介

吉永和弘 氏 （㈱半導体理工学研究センター）

④ 事例#1：画像処理 IP 開発への高位設計適用事例

浅野哲也 氏 （㈱ルネサステクノロジ）

⑤ 事例#2：TLM2.0 を利用した大規模回路設計

長谷川裕恭 氏 （㈱エッチ・ディー・ラボ）

● ナノ世代物理設計フォーラム（1 月 23 日 12:45～14:30，14:45～16:30）

司会：金本 俊幾 氏

（ナノ世代物理設計ワーキング・グループ主査/ルネサス テクノロジ）

現在量産段階にある 45nm、さらにこの先の 32nm テクノロジノードでは、デバイス特性、回路特性の WID(Within Die)および D2D(Die to Die)ばらつきの増大が、SoC 設計上の深刻な課題となっています。これに対し、さまざまな手法がばらつきに起因する課題を克服する手段として提案されています。

本セッションでは、最先端統計から見た 32nm ばらつき予測と設計法を以下のトピックを通じてお伝えします。

セッション 2A：「デバイス・回路ばらつきの要因と統計的側面」 12:45～14:30

① 最先端統計から見た 32nm デバイス特性ばらつき

平本　俊郎 教授（東京大学 生産技術研究所）

② 最先端技術におけるデバイスばらつきと統計的回路特性

桜井　貴康 教授（東京大学 生産技術研究所）

セッション 2B：「Statistical 回路設計イノベーション」 14:45～16:30

③ Parameter variations and process variation tolerant design of logic and SRAMs

Prof. Kaushik Roy（Purdue University）

④ Variability immune circuits and statistical　estimation of process variation towards 32nm era

Prof. Dennis Sylvester（University of Michigan）

3.2.3 開催結果

各セッションのインターネットでの事前聴講登録数、当日申込数、有料の参加者数、及びアンケート回答数は以下であった。

表 1 システム・デザイン・フォーラム 2009 の参加者数

	SystemC ユーザ・フォーラム	ナノ世代物理設計フォーラム
事前申込者	118	83
当日申込者	9	12
申込者数合計	127	95
参加者数（有料）	123	93
アンケート回答数	113	74

今年は、経済状況が厳しい中にも関わらず、SystemCユーザ・フォーラムには100名を越える参加者があり、システムLSI設計の最先端状況の把握に役立ったと思われる。また、ナノ世代物理設計フォーラムにも100名近い参加者があり、次世代以降のLSI物理設計における新たな課題を知る良い機会となった。

● 各セッションの状況

各セッションとも熱の入った発表、討論となり盛況であった。

■ SystemC ユーザ・フォーラム 2009

毎年恒例の SystemC ユーザ・フォーラムが、2009 年 1 月 23 日の午前にパシフィコ横浜にて「EDS Fair 2009」に併設される形で開催された。OSCI、JEITA SystemC WG、及び STARC による報告、SystemC 適用事例 2 件の盛りだくさんの内容であった。また例年通り、100 名を超える入場者があった。

図 1　SystemC ユーザ・フォーラム 2009

最初にワーキング・グループ主査、長谷川 隆 氏(富士通マイクロエレクトロニクス)の開会挨拶から始まり、まず OSCI（Open SystemC Initiative）の役員である Stan Krolikoski 氏（米 Cadence, Inc.）が、2008 年に OSCI が設立 10 年目を迎え、TLM(Transaction Level Modeling) 2.0 の正式版や AMS(Analog Mixed Signal)のドラフト仕様がリリースされた重要な節目の年であった事を報告した。TLM2.0 に関しては、現在 LRM(Language Reference Manual)を執筆中であり、本年中にリリースされた後に IEEE に移管されて標準化の審議に入るはずだが、その具体的時期については明言

されなかったのが残念である。

AMS 標準に関しては、今回リリースしたドラフトに対するフィードバックを募集している事、合成サブセットワーキング・グループからこの四半期中に次のドラフトがリリースされる予定である事、また、新しいワーキング・グループとして、SystemCで記述されたモデルの内部をＥＤＡツールがアクセスする際の標準ＡＰＩを定めるCCI(Configuration, Control, and Inspection)ワーキング・グループが設立された事が紹介された。

続いて、JEITA SystemCワーキング・グループの清水氏(OKIセミコンダクタ)より、同WGの活動の紹介があった。昨年同WGでは「SystemC推奨設計メソドロジ」の合成編について紹介していたが、本年夏には設計メソドロジ全般を公開予定であり、今回はその概要が紹介された。(http://eda.ics.es.osaka-u.ac.jp/jeita/eda/index-jp.html)
また、先日公開された TLM 2.0 に関する補足説明として、比較的難解なプロトコル拡張について、例を用いてわかりやすく解説された。配布資料には、より詳細な解説書や TLM 2.0Glossary(用語集)の日本語版も添付されており、OSCI 提供の資料を読む際に役立つものである。質疑応答では、「無視できない拡張」を使用した際のエラーメッセージの出方について質問が投げかけられたが、現状では特別なエラーメッセージは出ないため、ログファイルを注意深く見なければいけない点は変らないようだ。

次に、STARC(㈱半導体理工学研究センター)の吉永氏より、TL モデリングの概要とその必要性、そして公開されたばかりの TL モデリングガイドライン第 2 版の紹介があった。この第 2 版では、OSCI の TLM 2.0 に完全準拠しており、これを用いることで設計の容易化や再利用性が高まることが期待される。旧版からの変更点も解説され、スムーズな移行が可能である。TL モデリングガイドは、STARC の WEB から無償ダウンロード可能となっている。また有償で書籍を購入することもできる。
(http://www.starc.jp/tlmg/index-j.html)

後半は、SystemC のユーザー事例が 2 件講演された。
まず、ルネサステクノロジの浅野哲也氏より、SystemC 記述を用いた高位設計環境の紹介があった。同社における高位設計環境の狙いと構成、そして画像処理 IP 開発への適用事例と得られた結果が示された。SystemC 関連のツールとしては、記述チェッカ、シミュレータ、動作合成、等価検証のツールを適用し、いずれも実用レベルであるとのことであった。質疑応答では、動作合成を適用するか否かの判断をどこですべきかの議論になったが、現状では決め手がないようである。

最後を締めくくったのは、エッチ・ディー・ラボの長谷川裕恭氏であり、同社で実際に行われている SystemC を用いた大規模回路設計について紹介された。的確に階層化することにより段階的に詳細化することで動作合成結果と SystemC の混在や既存の RTL 設計との混在シミュレーションが可能であること、また TLM 2.0 によるシステムシミュレーションも実施していることなどが紹介された。また、設計や検証の負担を軽くするため、インタフェース仕様は一本化すべきであるとの意見も示された。

長谷川 隆 ＝ 富士通マイクロエレクトロニクス株式会社
JEITA SystemC ワーキング・グループ

■ ナノ世代物理設計フォーラム

システム・デザイン・フォーラム 2009 のセッションのひとつとして，2009 年 1 月 23 日の午後、パシフィコ横浜においてナノ世代物理設計フォーラムが開催された。LSI におけるプロセス技術や回路設計技術をリードしてきた講師陣を招聘、100 名近い入場者があった。

図 2　ナノ世代物理設計フォーラム

最初にナノ世代物理設計ワーキング・グループ主査の、金本 俊幾 氏（ルネサス テクノロジ）から本フォーラム開催の趣旨説明および本ワーキング・グループの活動紹介がありスタートした。

次にプロセス技術の立場から、東京大学生産技術研究所教授の平本俊郎氏が登壇した。同氏は、半導体 MIRAI プロジェクトでの研究成果である超大規模 DMA-TEG(Device Matrix Array Test Element Group)の測定結果から、MOS トランジスタの閾値 Vth バラつきが正規分布すること、および PMOS よりも NMOS のばらつきが大きいことを示した。また、PMOS トランジスタのばらつきは、製造ラインや製造条件によらず、ほぼチャネルの離散不純物バラつきに依存することと、NMOS トランジスタのばらつきは、離散不純物ばらつきに加えて他の要因が関与していることを示唆した。

将来展望としては、High-k/Metal-Gate の導入により一時的にランダムばらつきは低減するものの、チャネル不純物濃度を低くできる SOI、FinFET、ナノワイヤの追求は続き、さらにメタルソース・ドレインなど、ソース・ドレインの不純物もなくす方

向と予想。ただし、不純物をなくしても、必ず他のばらつき要因が現れ支配的となるため、ナノ領域ではバラつきに対する総合対策が必須と締めくくった。

続いて、回路設計の立場から同じく東京大学生産技術研究所教授の桜井貴康氏が登壇。回路特性に対するデバイスばらつきの影響を論じた。まず、WID（Within Die)バラつきの中のランダムバラつきについて、一般に n 段直列ゲートの遅延バラつきは 1/√n に緩和されると考えられるが、最低動作電圧はそれらのうちワーストなゲートで規定されることを指摘。低消費電力を目指した超低電圧設計において問題になることを示した。

また、規則正しいレイアウトパターンによりバラつきの幅を抑えられるか、という点について Sea-Of-Gates 型と通常の標準セルレイアウトの 90nm における特性比較結果を紹介し、期待したほどランダムバラつき幅は狭まらず、むしろ面積と平均遅延値へのペナルティが大きいことを示した。

なお、講演時間内には十分な時間がなく議論ができなかった D2D(Die to Die)バラつきについてもコメントがあり、今後、設計に対し本質的に重要となるため、設計者としての以下の 2 つの希望を挙げた。

１）なるべく、D2D ばらつきを小さくするプロセス

２）Post-fabrication Vth adjustment (tuning)を許すデバイス（例えば、短チャネルでも基板バイアスの効くデバイス）

3 番目には、「Statistical 回路設計イノベーション」をテーマに米 Purdue University, Roscoe H George Professor の Kaushik Roy 氏が登壇。バラつきに対して敏感な回路である SRAM を中心とした最新の耐バラつき設計を紹介した。

Kaushik Roy 氏は SRAM について、D2D バラつきの影響をリード、ライト、アクセス、ホールドの各動作に分けて論じ、低 Vth 側のコーナーではリード、ホールド、高 Vth 側のコーナーではライト、アクセスの不良が増加することを解説。双方の不良を低減するため、D2D バラつきを確実に捉える SRAM アレイのリーク電流モニタ回路と、その検出値をフィードバックして適切なボディバイアスを SRAM アレイに与える自己修復回路を紹介した。その他、ビット線のリークを補償するキーパー回路のサイズをダイナミックに調整し、遅延ばらつきを補正する手法や、ロジック回路において活性化率の低いクリティカルパスを敢えて 2 サイクル動作とし、低電力と耐バラつき性を向上するアプローチ CRISTA(CRitical path ISolation for Timing Adaptiveness)等、様々な手法が紹介された。結論として、プロセスバラつき、およびプロセス許容性は益々重要な問題となっており、電力、性能、歩留まりを考慮した最適化設計が必要となってきている、と述べた。

本フォーラムの最後は米 University of Michigan, Associate Professor の Dennis Sylvester 氏が登壇。"Coping with Variability in Nanoscale CMOS ～Analyze, Sense, Correct, Exploit" と題して、多角的にバラつき問題を俯瞰した。

Sylvester 氏は、バラつき問題はバラつきそのものではなく、バラつきを見込んで設定する設計マージンの問題である、とした。その上で、まず Sense として信頼性モニタ、具体的には NBTI(Negative Bias Temperature Instability)センサ回路やゲートリークモニタ回路により LSI の寿命を正確に見積もって、設計マージンを適切に制御するアプローチを紹介した。NBTI 特性劣化による PMOS 寿命を平均値だけでなく分布形状まで示すスライドに、初めてこのようなデータを見る！と感嘆する質問があった。実際は実験的に測定したものではなくシミュレーション結果であるという説明であったが、新たに提案された TEG でこのようなデータが測定できる可能性があるということで反響を呼んだ。しかし、このようにして得られたデータを信頼性設計にどのように使うかはまだ明確な方法論が確立されておらず、今後の実用化を期待したい分野である。そのほか、Analyze としてモンテカルロベース SSTA、Correct として劣化に応じた動的電源制御 DVS(Dynamic Voltage Scaling)、Exploit としてストレス効果を制御するための設計ガイドラインの紹介があった。

金本　俊幾 ＝ 株式会社ルネサステクノロジ
JEITA ナノ世代物理設計ワーキング・グループ

3.2.4 まとめ

本年の SystemC ユーザ・フォーラム 2009 の参加人数は、厳しい経済状況の中、例年より少ない人数ではあったが、123 名を集められたことは、依然として SystemC への興味が高いことを表していると考えられる。

講演は 2 時間で 5 講演を実施。OSCI、JEITA SystemC ワーキング・グループ、及び STARC による報告、SystemC 適用事例 2 件と充実した内容であったが、時間的にはやや不足気味であった。質疑応答も活発に行われた(7 件)。

フォーラム終了後のアンケートでは、SystemC ユーザ・フォーラム 2009 の満足度（満足＋まあ満足）が 70%と前年度の 81%を下回り、昨年よりは低い結果となった。しかしながら、「満足」だけを取り上げると、17%から 26%へ増加がみられる。適用が進み、はっきりとした目的意識を持った参加者が多かったのかもしれない。来年度以降開催する場合には、フォーラムの位置づけを見直す必要があると思われる。

ナノ世代物理設計フォーラムの満足度（満足＋まあ満足）が 76%となっており、今後の参加希望についての設問でも、「希望する」（参加する＋内容次第で参加する）が 88%とな

り、フォーラム内容の充実と開催への期待が確認された。

本フォーラムに期待された講演内容については、「デバイス特性ばらつき動向」と「耐ばらつき設計」がともに 35%とあり、副っていると考えられる。またもっとも得られた知見は、という設問には「デバイス特性ばらつき動向」が 52%と高い評価であった。

その他、意見、要望の設問では、「有料でも 2,000～3,000 円であれば、ありがたい」、「DFM、DFY Technology フォーラムを開いて欲しい」等、今後に対する期待をいただいた。

運営に関しては、「始まってからしばらく、カメラのフラッシュがひどかった。有料のセッションではひかえていただきたい」、「受け付けの対応が悪い」、「全体的に講演内容に対して時間が足らない印象。特に 3 番目の講演は、項目を絞った方が良かった」とのコメントをいただいた。今後の反省材料としたい。

また要望として「電子資料の配布」、「予稿集と実際の発表時スライドで差がある。最新の予稿を PDF で入手できるようにして欲しい」、「英語が苦手なので、Q&A の解説がほしかった」とのコメントもあり、検討事項としたい。

3.2.5 システム・デザイン・フォーラム 2009WG 委員（敬称略、順不同）

主　査	江 田　努	ローム㈱
委　員	熊 谷　敬	セイコーエプソン㈱
同	河 村　薫	富士通マイクロエレクトロニクス㈱
同	金 本 俊 幾	㈱ルネサステクノロジ
同	中 森　勉	富士通マイクロエレクトロニクス㈱
同	長谷川 隆	富士通マイクロエレクトロニクス㈱
同	中 西 早 苗	NEC エレクトロニクス㈱
同	西園寺 修	日本シノプシス㈱
同	今 井 正 治	大阪大学
同	若 林 一 敏	日本電気㈱
同	山 田 陽 一	セイコーエプソン㈱
アドバイザ	太 田 光 保	パナソニック㈱
オブザーバ	山 田　節	三洋電機㈱
事 務 局	小田 佳代子	日本エレクトロニクスショー協会

3.3 ASP-DAC 2009

3.3.1 はじめに

ASP-DAC (Asia and South Pacific Design Automation Conference)は、VLSI およびシステム LSI の設計技術や設計自動化技術をテーマにしたアジア太平洋地域での最大規模の国際会議である。ASP-DAC は米国で開催されるこの分野のトップ・コンファレンスである DAC (Design Automation Conference)、ICCAD（International Conference on Computer Aided Design）や欧州で開催される DATE (Design, Automation and Test in Europe)とはシスター・コンファレンスの関係にあり、お互いにリエゾンを交換して協力関係を持っている。

ASP-DAC は、電子情報通信学会や情報処理学会などの学会だけでなく、電機メーカおよび半導体メーカの業界団体である JEITA（会議開始当時は EIAJ）と EDSF（会議開始当時は EDAT）の支援のもとで 1995 年に開始された。業界団体である JEITA が ASP-DAC のような国際会議の支援を行っているのは、次のような理由による。電機メーカや半導体メーカが国際競争力のある電子製品の開発を行うためには、マーケッティングや製品企画だけでなく、大規模・高機能・低消費電力のシステム LSI の最適設計を短期間で行える設計力を持つ必要がある。そのためには、最新の設計自動化技術についての情報収集と研究開発を行う必要がある。一流の国際会議を国内で開催することにより、わが国からより多くの技術者と研究者が参加して最先端の設計技術および設計自動化技術についての情報収集、情報交換などを行うことが可能になる。

3.3.2 会議の開催経緯

ASP-DAC の第 1 回目の会議は 1995 年 8 月 30 日から 9 月 1 日にかけて幕張メッセの日本コンベンションセンターで、情報処理分野の国際学会である IFIP（International Federation on Information Processing）の TC10 WG10.2 および WG10.5 に属する CHDL および VLSI という名称の 2 つの国際会議と並列開催の形で開催された。第 2 回目は 1997 年 1 月に開催され、それ以降毎年 1 月に開催されてきた。この間、1999 年には香港（中国）で、2002 年にはバンガロール（インド）でそれぞれ開催された。2007 年以降は、日本と国外で交互に開催するというローテーションで運営されている。今回の会議（ASP-DAC 2009）は 14 回目で、横浜市パシフィコ横浜で 1 月 19 日（月）から 22 日（木）の日程で開催された。

3.3.3 ASP-DAC 2009 の概要

ASP-DAC 2009 の概要を表 1 に示す。一般講演としては、33 カ国から投稿された 355 編の論文の中から 116 編が採択され、3 日間にわたって並列の 4 つのトラック、24 のセッションで発表された。表 1 からもわかるように、論文の投稿数については前回韓国で開催された ASP-DAC 2008 を越す 355 件で、論文の採択率は前年よりやや低い 33%であり、この分野での他の国際会議（DAC, ICCAD, DATE）とほぼ同じ水準を維持している。これまでどおり、ASP-DAC は名実ともに一流の国際会議であると評価できる。

基調講演のタイトルと講演者を表 2 に、特別セッションのタイトルを表 3 に示す。また、表 4 に、昨年に引き続いて実施されたデザイナーズ・フォーラムのセッション・タイトルを示す。表 5 には、有料チュートリアルのタイトルを示す。

発表された論文の中から、表 6 に示す 2 本の論文が選ばれ、ベストペーパー賞が授与された。また、デザイン･コンテストに応募した作品の中から､表 7 に示すベストデザイン 2 件が選ばれて表彰された。

前回と同じく学生フォーラムも実施された。

表 1: ASP-DAC 2007、2008、2009 の比較

開催年	2007 年	2008 年	2009 年
日時	2007 年 1 月 23 日(火) ～26 日（金）	2008 年 1 月 21 日(月) ～24 日（木）	2009 年 1 月 19 日（月） ～22 日（木）
会場	横浜市（日本) パシフィコ横浜	ソウル (韓国) COEX	横浜市（日本) パシフィコ横浜
併設展示会	EDSF 2007	---	EDSF 2009
論文投稿数	408	350	355
論文投稿国（地域）数	30	27	33
論文採択数 （採択率）	131（32%）	122（35%）	116（33%）
キーノート アドレス	3 件	3 件	3 件 （表 2 参照）
一般講演	27 セッション（131 編）	24 セッション(122 編)	24 セッション（116 編）
特別セッション （招待講演等）	5 セッション	4 セッション	4 セッション （表 3 参照）
デザイン・ コンテスト	1 セッション	1 セッション	1 セッション
学生フォーラム	昼休みに実施 （Student Forum と改称）	昼休みに実施 （Student Forum）	昼休みに実施 （Student Forum）
ポスターボード	―	―	―
有料チュートリアル	6 件 （全日 2 件、半日 4 件）	5 件 （全日 2 件、半日 3 件）	7 件　（表 5 参照） （全日 1 件、半日 6 件）
デザイナーズ・ フォーラム	4 セッション （招待講演 2、 パネル討論 2）	4 セッション （招待講演 2、 パネル討論 2）	4 セッション（表 4 参照） （招待講演 2、 パネル討論 2）

表 2: 基調講演

講演タイトル	講演者
Challenges to EDA system from the view point of processor design and technology drivers	Mitsuo Saito （Toshiba Corp., Japan）
Automated Synthesis and Verification of Embedded Systems: Wishful Thinking or Reality?	Wolfgang Rosenstiel (Univ. of Tuebingen, Germany)
From Restrictive to Prescriptive Design	Leon Stok (IBM Corp., USA)

表 3: 特別セッションのタイトル

種類	セッション・タイトル
招待講演	セッション 2D: EDA Acceleration Using New Architectures
	セッション 3D: Hardware Dependent Software for Multi- and Many-Core Embedded Systems
	セッション 4D: Challenges in 3D Integrated Circuit Design
パネル討論	セッション 9D: Dependable VLSI: Device, Design and Architecture – How should they cooperate? -

表 4: デザイナーズ・フォーラムのタイトル

種類	セッション・タイトル
招待講演	セッション 5D: Consumer SoCs
	セッション 7D: Analog/RF Circuit Designs
パネル討論	セッション 6D: ESL Design Methods
	セッション 8D: Near-Future SoC Architectures – Can Dynamically Reconfigurable Processors be a Key Technology?

表 5： Tutorial のタイトル

トラック	種類	タイトル
1	全日	Software Development and Programming of Multi-core LSI
2	半日	Formal Methods for C-Based Embedded System Design Verification – Technical Trends and Practical Aspects -
3	半日	Statistical Design on the Verge of Maturity: Revisiting the Foundation
4	半日	Circuit Reliability I: Reliability Mechanisms and the Impact on IC Design
5	半日	Circuit Reliability II: Circuit Aging Prediction and Resilient Design
6	半日	Recent Advances in Low-Leakage VLSI Design
7	半日	Memory Architectures and Software Transformations for System Level Design

表 6: ベストペーパー賞が授与された論文

論文タイトル・著者
1C-1: “Fast Yield: Variation-Aware, Layout-Driven Simultaneous Binding and Module Selection for Performance Yield Optimization”, Gregory Lucas, Scott Cromar and Deming Chen (Univ. of Illinois, Urbana-Champaign, USA)
5C-1: “Efficiently Finding the ‘Best’ Solution with Multi-Objectives from Multiple Topologies in Topology Library of Analog Circuit”, Yu Liu, Masato Yoshioka, Katsumi Homma and Toshiyuki Shibuya (Fujitsu Laboratories Ltd., Japan)

表 7: ベストデザイン賞が授与された設計

種類	論文タイトル・著者
Best Design Award	1D-1: "A Wireless Real-Time On-Chip Bus Trace System", Shusuke Kawai, Takayuki Ikari (Keio Univ., Japan), Yutaka Takikawa (Renesas Design Corp., Japan), Hiroki Ishikuro, Tadahiro Kuroda (Keio Univ., Japan)
Special Feature Award	1D-13: "Ultra Low-Power ANSI S1.11 Filter Bank for Digital Hearing Aids", Yu-Ting Kuo, Tay-Jyi Lin, Yueh-Tai Li (National Chiao Tung Univ., Taiwan), Chou-Kun Lin (ITRI, STC, Taiwan), Chih-Wei Liu (National Chiao Tung Univ., Taiwan)

3.3.4 論文の投稿状況

1998 年から 2009 年の、ASP-DAC への論文投稿数の地域別の推移を図 1 に示す。ただし、ASP-DAC 2002 については、インドのバンガロールで同時開催された VLSI Design 2002 への投稿論文を含む。図 1 に示すように、1999 年（香港開催）以降は投稿論文数が増加し、安定的に 300～400 件の投稿がある。名実ともに、ASP-DAC は設計自動化分野の国際会議として定着したと言ってよいであろう。

表 8 に、日本からの論文投稿数の推移と、全世界から投稿された論文に占める割合を示す。日本からの論文投稿数が全体に占める割合は、2000 年をピークにして、10%前後に低下している。今回は、日

本と台湾、北米などの地域からの投稿が増加した。論文投稿数が多かったのは、米国の 105 編（前回は 92 編)、中国の 43 編（前回は 67 編)、日本の 50 編（前回は 31 編)、台湾の 54 編（前回は 34 編)、インドの 21 編（前回は 25 編）であった。

図 1　地域別論文投稿数

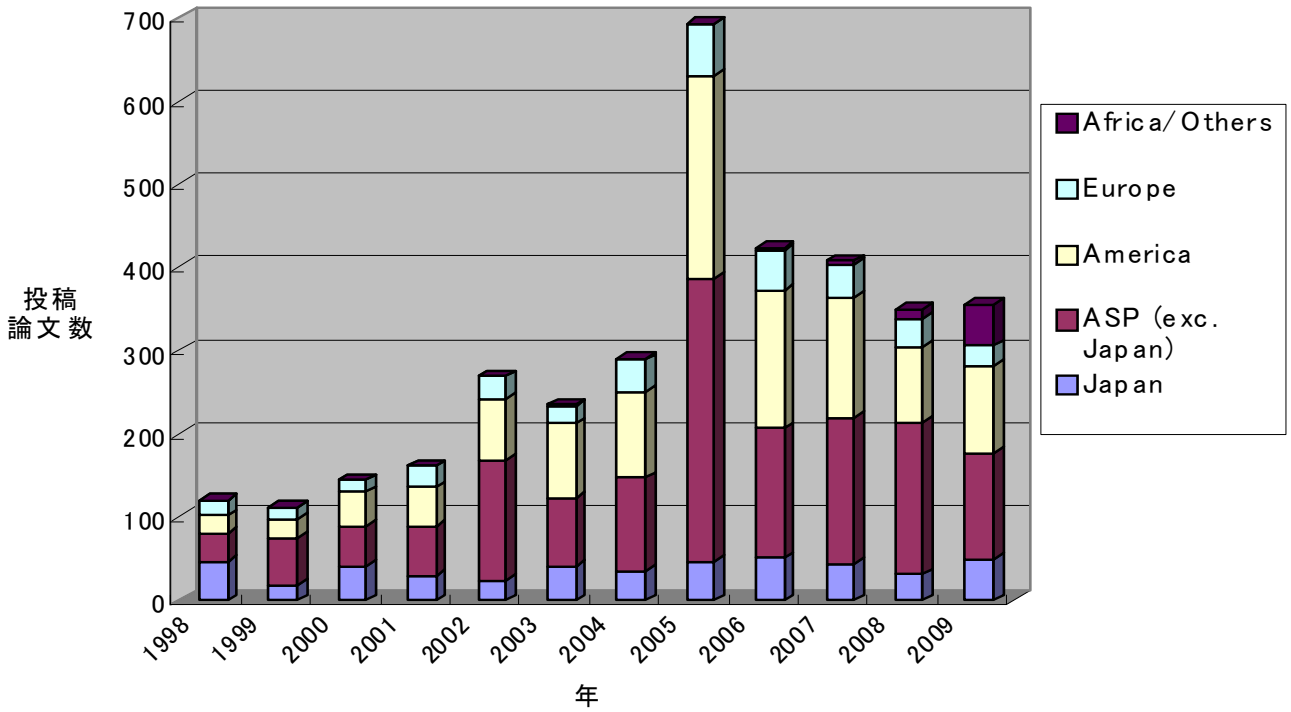


表 8 日本からの論文投稿数と全体に占める割合

年 地域	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
日本 (割合)	46 (38%)	18 (16%)	42 (29%)	29 (18%)	24 (9%)	42 (18%)	36 (12%)	46 (7%)	51 (12%)	44 (11%)	31 (9%)	50 (14%)
全体	121	111	144	161	269	235	291	692	424	408	350	355

次に、研究分野別の論文投稿数および採択論文数を表 9 に示す。ASP-DAC 2009 では、研究分野を 13 種類に分類して論文の査読と採否の決定を行った。今回論文投稿数が多かった分野は、分野 1 のシステムレベル設計手法のセッション、分野 8 のタイミング・消費電力関係のセッション、分野 2 のシステム・アーキテクチャおよび最適化のセッション、分野 3 の組み込みシステムおよびリアルタイム・システムのセッションであった。

表 9: 分野別の論文投稿数と採択論文数

分野	研究分野	投稿数	採択数	採択率
1	System-Level Design Methodology	48	16	33.3%
2	System Architecture and Optimization	40	12	30.0%
3	Embedded and Real-Time Systems	36	12	33.3%
4	High-Level/Behavioral/Logic Synthesis and Optimization	23	8	34.8%
5	Validation and Verification for Behavioral/Logic Design	21	6	28.6%
6	Physical Design (Routing)	17	6	35.3%
7	Physical Design (Placement)	15	5	33.3%
8	Timing, Power, Thermal Analysis and Optimization	41	14	34.1%
9	Signal/power Integrity, Interconnect/Device/Circuit Modeling and Simulation	18	7	38.9%
10	Design for Manufacturability/Yield and Statistical Design	16	6	37.5%
11	Test and Design for Testability	22	9	40.9%
12	Analog, RF and Mixed Signal Design and CAD	26	5	19.2%
13	Emerging Technologies and Applications	32	10	31.3%
	合　　計	355	116	32.7%

3.3.5 参加者の内訳

ASP-DAC への地域別の参加者数の推移を図 2 に示す。また、日本からの参加者の推移を表 10 に示す。今回の全参加者数は 492 名であった。前回（412 名）と比べると、参加者が増加した。日本からの参加者数は全体の 62.0%の 305 名であった。

図 2 地域別参加者の推移

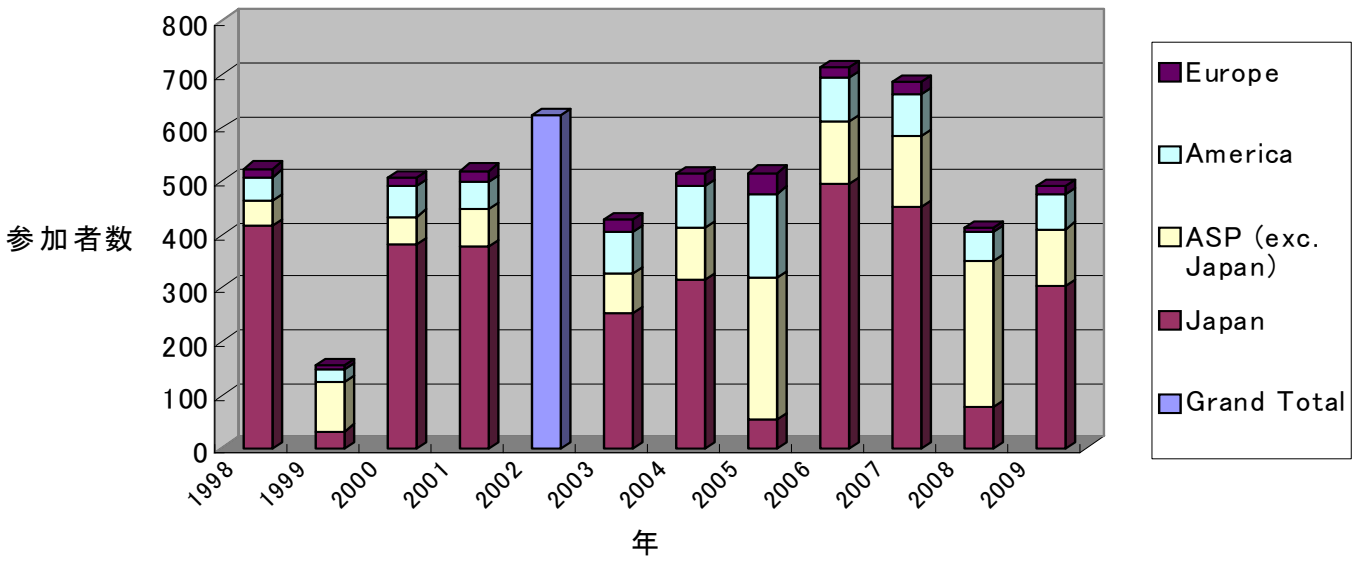


表 10:　日本からの参加者数と全体に占める割合（チュートリアルのみの参加者を除く）

年 地域	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
日本 (割合)	416 (79%)	32 (21%)	383 (76%)	379 (73%)	N/A	253 (59%)	316 (61%)	55 (11%)	494 (70%)	450 (66%)	77 (19%)	305 (62%)
全体	524	156	507	520	623	429	515	516	708	685	412	492

3.3.6 今後の展望

ASP-DAC の今後の開催予定を表 11 に示す。引き続き、隔年で日本開催になる模様である。

表 11:　ASP-DAC の今後の開催予定

年	開催予定地	開催時期	実行委員長
2010 年	Taipei（台湾）	2010 年 1 月 18 日（月）～21 日（木）	Y.L.Lin 氏　（National Tsing Hua Univ.）
2011 年	横浜（日本）	2011 年 1 月 25 日（火）～28 日（金）	浅田邦博 氏 （Univ. of Tokyo.）

3.4　EDA 用語辞典

3.4.1　背景と目的

エレクトロニクス機器の中核部品であるシステム LSI は、半導体微細化技術の目覚ましい進歩の恩恵を受けて大規模・高機能化している。システム LSI のこのような複雑化に伴い、設計上流では超大規模システム LSI の機能・論理の設計や検証問題、設計下流ではバラツキ問題（DFM：design for manufacturability）が顕在化しており、これらの設計課題を解決するために、LSI の自動設計化技術（EDA：electronic design automation)も進歩している。

JEITA（(社)電子情報技術産業協会） EDA 技術専門委員会は、EDA 技術の進歩に伴う最先端技術を分かり易く解説して EDA 技術の普促・推進を行うために、EDA 用語辞典ワーキンググループ(WG)を発足させて、日経 BP 社『Tech-On!』サイトの「EDA 用語辞典 LSI 設計ツール編」(初版 1999 年)を、「改訂版 EDA 用語辞典」として最新版に改訂した。「改訂版 EDA 用語辞典」では、「設計言語の標準化」、「アサーション検証」、「プラットフォーム・ベース設計」、「パワー・フォーマット」、「故障診断」などの新規の用語が用語全体の半分を占めると共に、DFM 技術を中心にして全ての用語が最新技術を反映した内容に更新された。

「改訂版 EDA 用語辞典」の出版に際し、今井正治 教授（大阪大学）を監修者にお招きし、国内を代表する専門家によって EDA 用語辞典 WG を構成し、第一線の先生、研究者、技術者の方々に執筆を頂いた。

3.4.2　概要

本「改訂版EDA用語辞典」は、以下の日経BP社『Tech-On!』サイトに掲載された。
(http://techon.nikkeibp.co.jp/word/column/eda/)
見出し用語数は以下の 44 件である。

分類	見出し用語
LSI設計とEDA技術	[LSI設計とEDA用語辞典]
標準化	[標準化][設計言語の標準化][フォーマット・モデリングの標準化]
システム・レベル設計・検証	[システム設計・検証][システム・レベル記述言語][ハードウェア・ソフトウェア協調設計] [動作合成][プラットフォーム・ベース設計][トランザクション・レベル・モデリング][ASIP]
機能設計・検証	[機能設計・検証][検証カバレッジ][ハードウェア記述言語・検証記述言語] [アサーション・ベース検証][フォーマル検証][非同期検証] [論理エミュレーション、プロトタイピング][論理シミュレーション][論理合成] [コーディング・スタイル・チェック]
LSIのテスト	[LSIのテスト][故障シミュレーション][ATPG][DFT] [スキャン・テスト][BIST][故障診断]
低消費電力設計	[低消費電力設計][パワー・フォーマット]
回路設計・検証	[ミックスド・シグナル][回路シミュレーション]
物理設計・検証	[物理設計・検証][プロトタイピング][シグナル・インテグリティ解析][信頼性解析] [自動レイアウト][レイアウト寄生パラメータ抽出][レイアウト検証][DFM]
タイミング設計・検証	[STA][SSTA][セル・キャラクタライズ][OCV]

3.4.3　参加メンバ

主査	黒川 敦	三洋半導体(株)
監修	今井正治	国立大学法人 大阪大学
専門家(標準化技術)	神戸尚志	近畿大学
専門家(システム)	長谷川隆	富士通マイクロエレクトロニクス(株)

	今井正治(兼)	
専門家(テスト設計)	吉田正昭	ＮＥＣエレクトロニクス(株)
専門家(物理設計)	金本俊幾	(株)ルネサステクノロジ
	黒川敦(兼)	
委員	山田　節	三洋電機(株)
委員	西本猛史	シャープ(株)
委員	熊谷　敬	セイコーエプソン(株)
委員	齋藤茂美	ソニー(株)
委員	南　文裕	(株)) 東芝
委員	太田光保	パナソニック(株)
委員	河村　薫	富士通マイクロエレクトロニクス(株)
委員	秋山俊恭	(株)ルネサステクノロジ
委員	江田　努	ローム(株)
委員	長野義史	(株)エッチ・ディー・ラボ
委員	川原常盛	コーウェア(株)
委員	山城　治	(株)ジーダット
委員	野坂啓介	(株)図研
委員	平尾栄二	凸版印刷(株)
委員	安井孝史	日本ケイデンス・デザイン・システムズ社
委員	飯島一彦	日本シノプシス(株)
委員	船津英世	マグマ・デザイン・オートメーション(株)
委員	瀬谷和宏	丸紅情報システムズ(株)
委員	三橋明城男	メンター・グラフィックス・ジャパン(株)
委員	前野酉治	(株)リコー

４．添付資料

4.1 ﾅﾉ世代物理設計 ﾜｰｷﾝｸﾞｸﾞﾙｰﾌﾟ報告

1. 製造ばらつきに起因する リーク電流変動の低減法

報告内容

- はじめに
- 提案手法
- 提案手法の検証
- まとめ

背景

- LSIの微細化が進むにつれ、リーク電流は、増加傾向にあり、リーク電流の低減は必須である。
- リーク電流はプロセスばらつきに敏感であり、出来上がったチップ毎にリーク電流に大きな差を生じさせ、歩留りの低下につながる。

リーク電流のばらつきの要因を明確化し、ばらつきを低減することは非常に重要である。

課題

- リーク電流のばらつきに対してどのデバイスパラメータがどのように影響するのかを明瞭に報告している例はみられなかった。

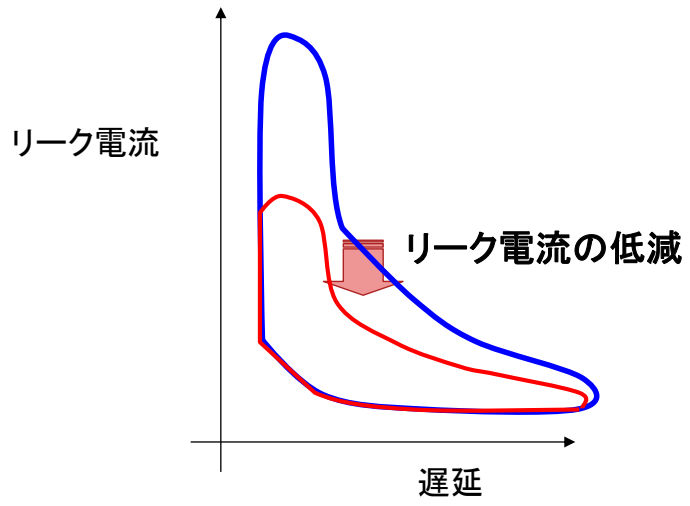
- プロセスばらつきに対して遅延分布を悪化させずに、リーク電流ばらつきを低減する手法は報告されていない。

目的

プロセスばらつきによる遅延時間とリーク電流の関係から、遅延一定となる条件下でリーク電流を変動させるパラメータの関係を吟味し、リーク電流ばらつきを低減する方法を提示し、その有効性を示す。

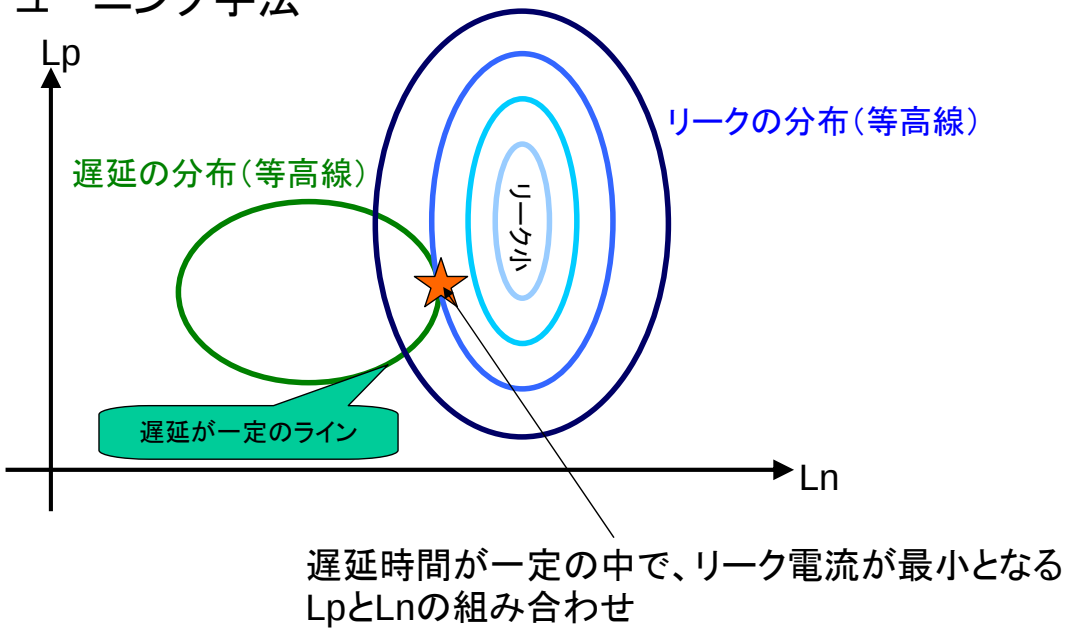
提案手法

- NMOSとPMOSの各種パラメータのバランスを調整し、等しい遅延の中でリーク電流を最小にさせる。



キーアイデア

- スタンダードセルにおけるNMOSとPMOSのパラメータのチューニング手法



遅延時間が一定の中で、リーク電流が最小となるLpとLnの組み合わせ

解析式の導出方法

- 遅延時間とリーク電流を解析式で表現する。

解析式では、閾値電圧V_{th}、チャネル長L、ゲート酸化膜厚Toxを考慮し、それぞれNMOSとPMOSを個別に扱う。解析式の変数は$V_{th,P}$,$V_{th,N}$, L_P, L_N, Tox_P, $Tox_{,N}$ の6個となる。

- インバータ2段分の遅延とリーク電流を取り扱う。
- 遅延とリーク電流の解析式は、応答曲面法により以下の2次の多項式で表現する。

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} \beta_{ij} x_i x_j$$

遅延の解析式

- インバータの遅延はα乗モデル[1]から、以下の式で得られる。

$$T_d = \frac{C_L V_{dd}}{\mu \cdot \frac{\varepsilon_{ox}}{T_{ox}} \cdot \frac{W}{L} (V_{dd} - V_{th})^{\alpha}}$$

$$\rightarrow \quad T_d \propto \frac{1}{(V_{dd} - V_{th})^{\alpha}}, L, T_{ox}$$

遅延の解析式は以下のように表現する。

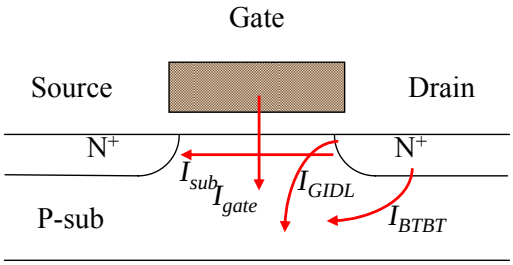
$$T_{d,1} - T_{d,0} = f\left(V_{th,P,1} - V_{th,P,0}, V_{th,N,1} - V_{th,N,0}, \frac{L_{P,1}}{L_{P,0}}, \frac{L_{N,1}}{L_{N,0}}, \frac{T_{ox,P,1}}{T_{ox,P,0}}, \frac{T_{ox,N,1}}{T_{ox,N,0}}\right)$$

*P*と*M*はPMOSとNMOS、*0*はノミナル時の値、*1*は変動した時の値の意味

リーク電流の解析式

- リーク電流はおおよそ以下のトータル電流で表される[2]。

$$I_{total} = I_{BTBT} + I_{sub} + I_{gate} + I_{GIDL}$$



BSIM4[3]で、Isubは以下で表される。

$$I_{sub} = \mu \frac{\varepsilon_{ox}}{T_{ox}} \frac{W}{L} V_T{}^2 e^{\frac{V_{gs} - V_{th}}{nV_T}} \left(1 - e^{\frac{-V_{ds}}{V_T}}\right)$$

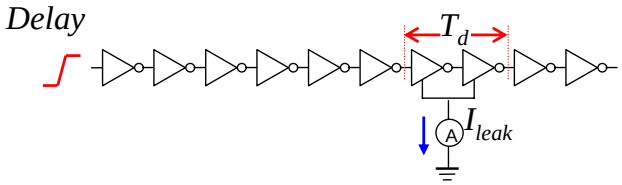
$$\rightarrow \quad \ln(I_{sub}) \propto (V_{dd} - V_{th}), \ln(T_{ox}), \ln(L)$$

リーク電流の解析式は以下のように表現する。

$$\ln\left(\frac{I_{leak,1}}{I_{leak,0}}\right) = g\left(V_{th,P,1} - V_{th,P,1}, V_{th,N,1} - V_{th,N,1}, \ln\left(\frac{L_{P,1}}{L_{P,0}}\right), \ln\left(\frac{L_{N,1}}{L_{N,0}}\right), \ln\left(\frac{T_{ox,P,1}}{T_{ox,P,0}}\right), \ln\left(\frac{T_{ox,N,1}}{T_{ox,N,0}}\right)\right)$$

解析式導出に用いた回路と条件

- 回路は多段インバータとし、波形が安定した2段の遅延(Td)とリーク電流(Ileak)を求める。



項目	値
SPICEモデルパラメータ	PTM 45nm (Bulk)[23]
SPICE電流モデル	BSIM4
電源電圧V_{dd}	1V
基本（×1インバータ）のトランジスタサイズ	P W/L=450nm/45nm
	N W/L=225nm/45nm
閾値電圧（V_{th}）の3σ	42%
チャネル長（L）の3σ	12%
ゲート酸化膜厚（T_{ox}）の3σ	4%

変動パラメータは閾値電圧V_{th}、チャネル長L、ゲート酸化膜厚Toxの3種とし、PMOS、NMOSそれぞれ、各パラメータの値を-3σ,-1σ,0,1σ,3σの5点で振り、計5^6= 15625点のSPICEシミュレーションを行う。求めた結果を遅延とリーク電流の解析式に当てはめ、係数を抽出する。

インバータにおける解析式の結果

- Vth, L, ToxのPMOS、NMOSトランジスタ全6パラメータを用いた遅延解析式とリーク電流解析式の係数

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} \beta_{ij} x_i x_j$$

遅延解析式係数 と解析式

β0	β1	β2	β3	β4	β5	β6
2.38E-11	1.11E-10	-8.70E-11	-2.86E-11	8.51E-12	-4.72E-11	-4.02E-11

βij	β1	β2	β3	β4	β5	β6
β1	3.07E-11	-3.08E-11	-7.54E-11	-2.51E-11	-1.81E-11	-1.09E-11
β2		2.30E-11	3.77E-11	4.00E-11	1.05E-11	1.49E-11
β3			-8.61E-12	3.00E-11	3.29E-11	1.36E-11
β4				-2.67E-11	8.86E-12	2.32E-11
β5					3.93E-12	3.35E-12
β6						3.18E-12

$$y = T_{d,1} - T_{d,0},\ x_1 = V_{th,P,1} - V_{th,P,0},\ x_2 = V_{th,N,1} - V_{th,N,0},$$
$$x_3 = \frac{L_{P,1}}{L_{P,0}},\ x_4 = \frac{L_{N,1}}{L_{N,0}},$$
$$x_5 = \frac{T_{ox,P,1}}{T_{ox,P,0}},\ x_6 = \frac{T_{ox,N,1}}{T_{ox,N,0}}$$

リーク電流解析式係数 と解析式

β0	β1	β2	β3	β4	β5	β6
6.17E-01	1.02E+01	-8.80E+00	-1.57E+01	-1.10E+01	-1.03E+00	-2.53E+00

βij	β1	β2	β3	β4	β5	β6
β1	1.79E+01	3.06E+01	-3.77E+01	3.57E+01	-8.35E+00	1.31E+01
β2		1.58E+01	-4.35E+01	3.37E+01	-3.37E+00	-5.51E-01
β3			7.98E+01	-5.30E+01	-7.18E+00	-1.58E+01
β4				5.55E+01	-3.79E+00	-1.37E+01
β5					4.44E-01	-1.64E+00
β6						3.85E+00

$$y = \ln\left(\frac{I_{leak,1}}{I_{leak,0}}\right),\ x_1 = V_{th,P,1} - V_{th,P,1},\ x_2 = V_{th,N,1} - V_{th,N,1},$$
$$x_3 = \frac{L_{P,1}}{L_{P,0}},\ x_4 = \frac{L_{N,1}}{L_{N,0}},$$
$$x_5 = \frac{T_{ox,P,1}}{T_{ox,P,0}},\ x_6 = \frac{T_{ox,N,1}}{T_{ox,N,0}}$$

インバータにおける解析式の結果

- V_{th}、L、Toxそれぞれ個別に扱った遅延解析式とリーク電流解析式の係数

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} \beta_{ij} x_i x_j$$

遅延解析式係数 と解析式

Vth	β0	β1	β2	β11	β22	β12
	−6.42E−15	−1.86E−11	1.63E−11	3.01E−11	2.35E−11	−2.97E−11
L	β0	β3	β4	β33	β44	β34
	−5.27E−11	2.20E−11	4.04E−11	−9.73E−12	−2.53E−11	2.54E−11
Tox	β0	β5	β6	β55	β66	β56
	−2.98E−12	−3.29E−12	−1.68E−12	2.87E−12	2.43E−12	2.65E−12

リーク電流解析式係数 と解析式

Vth	β0	β1	β2	β11	β22	β12
	2.26E−01	1.14E+01	−9.43E+00	3.21E+01	2.66E+01	4.37E+01
L	β0	β3	β4	β33	β44	β34
	1.42E−01	−1.88E+01	−1.34E+01	1.14E+02	8.34E+01	−9.11E+01
Tox	β0	β5	β6	β55	β66	β56
	−6.36E−06	−1.60E+00	−2.29E+00	9.17E−01	6.33E+00	−3.64E+00

V_{th}: $y = T_{d,1} - T_{d,0},\ x_1 = V_{th,P,1} - V_{th,P,0},\ x_2 = V_{th,N,1} - V_{th,N,0}$

L: $y = T_{d,1} - T_{d,0},\ x_3 = \frac{L_{P,1}}{L_{P,0}},\ x_4 = \frac{L_{N,1}}{L_{N,0}}$

Tox: $y = T_{d,1} - T_{d,0},\ x_5 = \frac{T_{ox,P,1}}{T_{ox,P,0}},\ x_6 = \frac{T_{ox,N,1}}{T_{ox,N,0}}$

V_{th}: $y = \ln\left(\frac{I_{leak,1}}{I_{leak,0}}\right),\ x_1 = V_{th,P,1} - V_{th,P,1},\ x_2 = V_{th,N,1} - V_{th,N,1}$

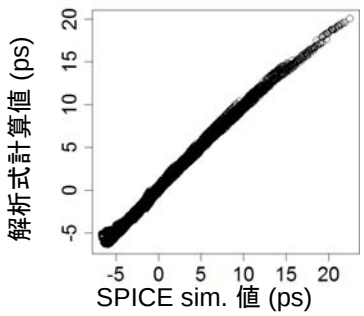
L: $y = \frac{I_{leak,1}}{I_{leak,0}},\ x_3 = \frac{L_{P,1}}{L_{P,0}},\ x_4 = \frac{L_{N,1}}{L_{N,0}}$

Tox: $y = \frac{I_{leak,1}}{I_{leak,0}},\ x_5 = \frac{T_{ox,P,1}}{T_{ox,P,0}},\ x_6 = \frac{T_{ox,N,1}}{T_{ox,N,0}}$

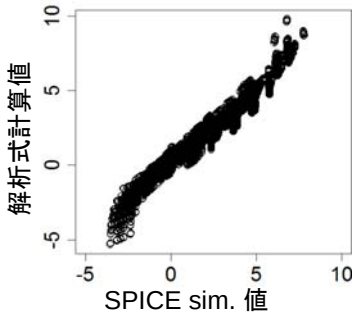
解析式の精度

- 本解析では、解析式を用いて評価を実施するので、SPICEシミュレーション結果と解析式から計算した結果を比較し、解析（回帰）式の精度を確認する。

遅延変動値



Log(リーク変動率)

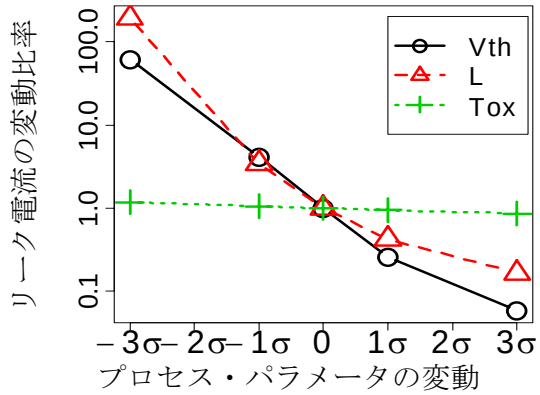


遅延とリーク電流の回帰式の決定係数R^2 は、それぞれ0.995と0.968であった。V_{th}, L, Tox個別の係数、回路をNAND、NORに変更した場合においても、決定係数は0.95以上となり、解析式で十分高い精度が得られることを確認した。

プロセス・パラメータがリーク電流へ与える影響

- プロセス・パラメータが変動した時のリーク電流への影響を示す。

ノミナル値1に対するリーク電流の変動率

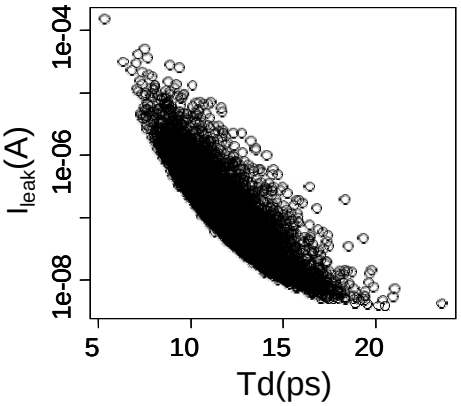


Vthの3σ	42%
Lの3σ	12%
Toxの3σ	4%

Toxはリーク電流のばらつきに対して寄与が少ないことがわかる。

プロセスばらつきによる遅延とリーク電流の相関

- V_{th}, L, Toxのパラメータがばらついた場合の、遅延とリーク電流の分布を示す。横軸が遅延、縦軸がlogをとったリーク電流値 である。

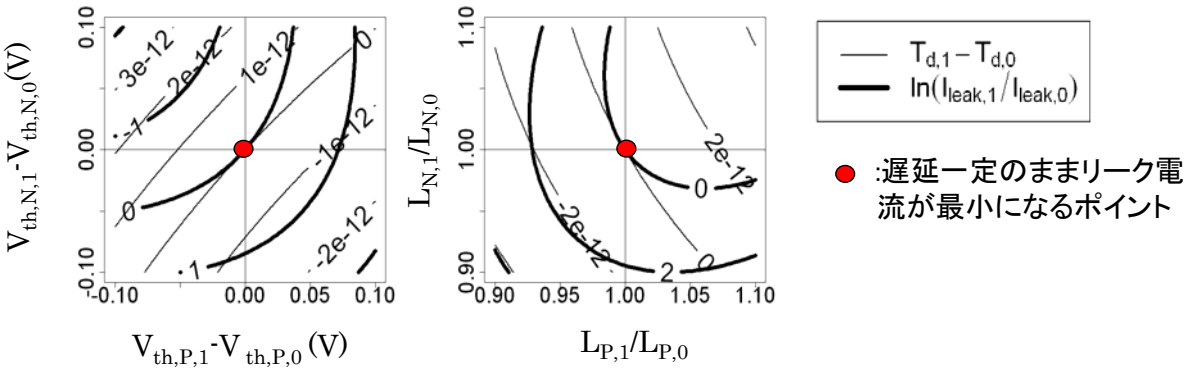


試行回数1万回のモンテカルロ解析の結果

遅延とリーク電流は1対1に対応しているわけではなく、遅延が一定でも様々なリーク電流値になる条件が存在する。このような幅を持つことは、実測値[4]でもみられている。

解析式による遅延とリーク電流の評価

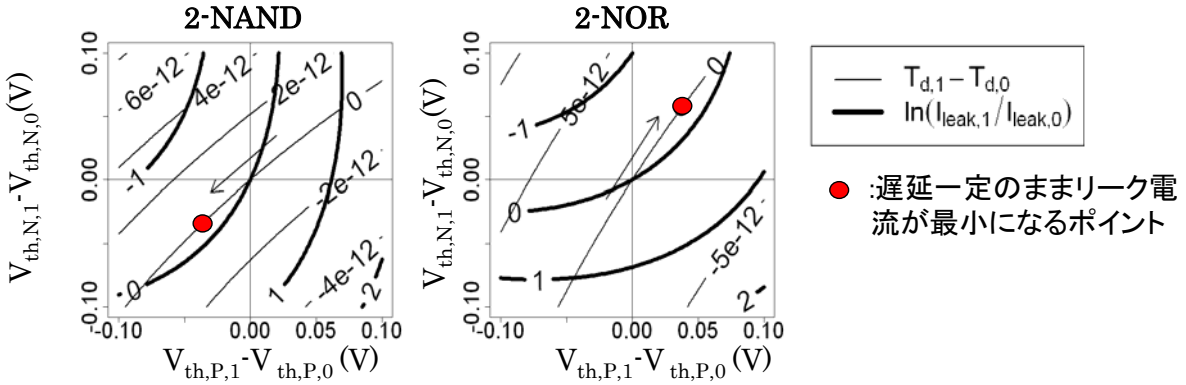
- INVでのリーク電流と遅延の分布を求めた。遅延とリーク電流の解析式は2次式のフィッティングであるため、$V_{th,P}$,$V_{th,N}$の2次元のパラメータ空間の中ではリーク電流と遅延値は楕円の分布を形成する。



パラメータのノミナル値を遅延の等高線が0の線に沿って動かすことにより、遅延が一定のまま、リーク電流が変動する。インバータの例では、初期値がリーク電流最小に近い条件であることがわかった。

解析式による遅延とリーク電流の評価

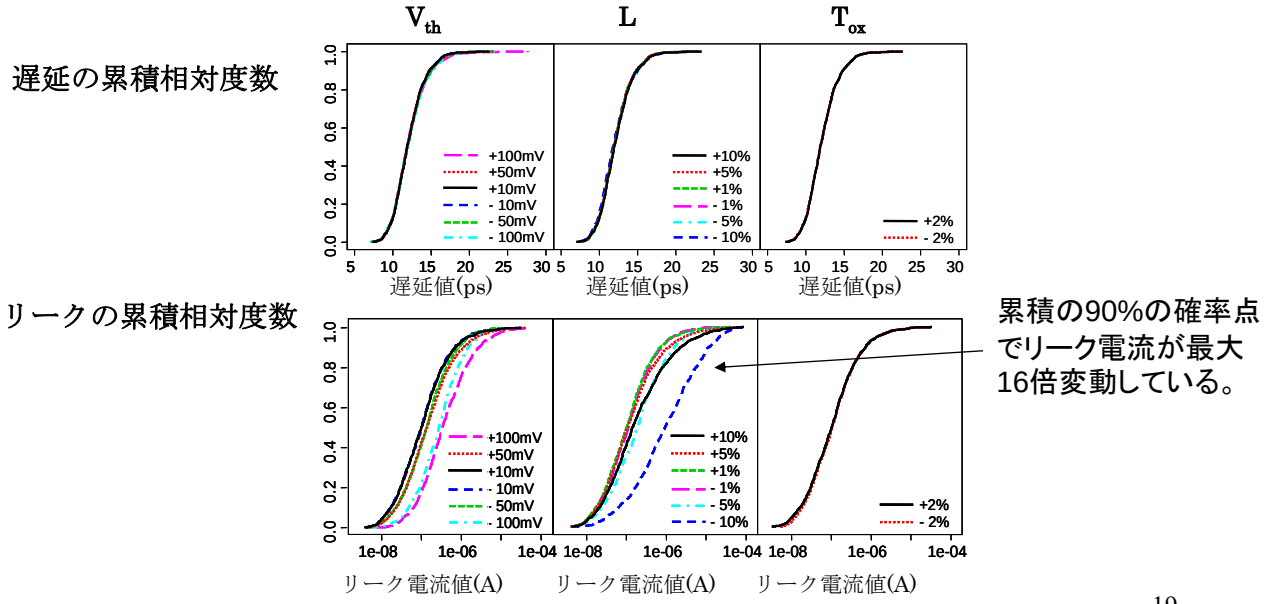
- 2入力NOR、2入力NANDでのリーク電流と遅延の分布を求めた。



パラメータのノミナル値(0,0)を遅延一定のままリーク電流最小の位置に移動させることにより、移動させる前よりもリーク電流が最小に近い値で分布し、リーク電流のばらつきは低減されることが予想される。

2パラメータ間の遅延とリーク電流

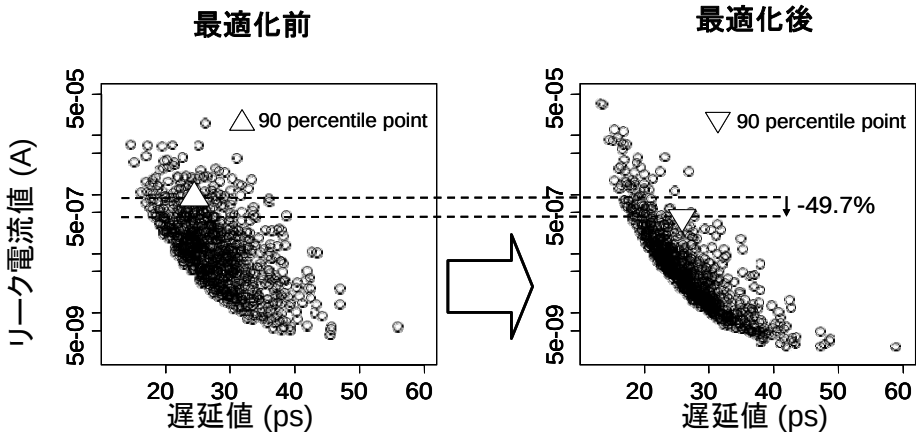
- PMOS、NMOSトランジスタのV_{th}, L, Tox各々について、遅延とリーク電流値の累積分布として表した。



累積の90%の確率点でリーク電流が最大16倍変動している。

全パラメータを用いた改善

- 2入力NORについて、Vth, L, ToxのP/N　6パラメータ全てを用いて遅延一定のままリーク電流小のポイントを抽出して、モンテカルロ解析を行った結果である。



リーク電流値のばらつきが狭まっていることがわかる。90%の確率点で、リーク電流に対して49.7%の低減効果があった。

まとめ

・遅延ばらつきとリーク電流ばらつきを解析・評価するための解析式を提案し、45nmテクノロジモデルを例に係数を導出した。

・解析式を元にして遅延を一定にしながらリーク電流ばらつきを削減する方法を提案し、提案手法による実験を行った。

・本提案手法の有効性が確認され、リーク電流ばらつきコーナーが削減されることを示した。

参考文献

[1]T. Sakurai and A. R. Newton, “Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas,” IEEE J. Solid-State Circuits, vol. 25, pp. 584–594, Apr. 1990.
[2]W. Nabel and J. Mermet, “Low power design in deep submicron electronics,” Kluewer Academic Publishers, 1997.
[3]BSIM4, UC Berkeley, [online]. Available: http://www-device.eecs.berkeley.edu/~bsim3/bsim4.html
[4]H. F. Dadgour, S.-C. Lin, and K. Banerjee, “A statistical framework for estimation of full-chip leakage-power distribution under parameter variations,” IEEE Trans. Electron Devices, vol. 54, no. 11, pp. 2930–2945, Nov. 2007.

2. 32nmプロセスにおける配線自己発熱の信号伝播遅延に対するインパクト調査

背景

- 「ばらつき」におけるローカルシステマティック(Deterministic)成分として、配線の自己発熱(Self-heating)考慮要否、あるいは考慮要となるシチュエーションをスクリーニングしたい。
  - 配線-基板間で数十℃も温度差が生じるシチュエーションがあるかどうか[1)]
  - 現実的な構造、物性値を使って配線伝播遅延へのインパクトを評価

配線のSelf-heating

- 配線のSelf-heating とは：
  電荷の充放電等にともなう配線を流れる電流による発熱
  → 配線と基板との間に温度差を生じる
- MOSトランジスタの発熱による基板面内温度分布は議論が活発
  → 配線Self-heating による垂直方向の温度差はOpen issue

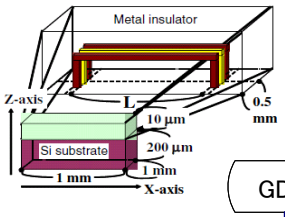
評価指針

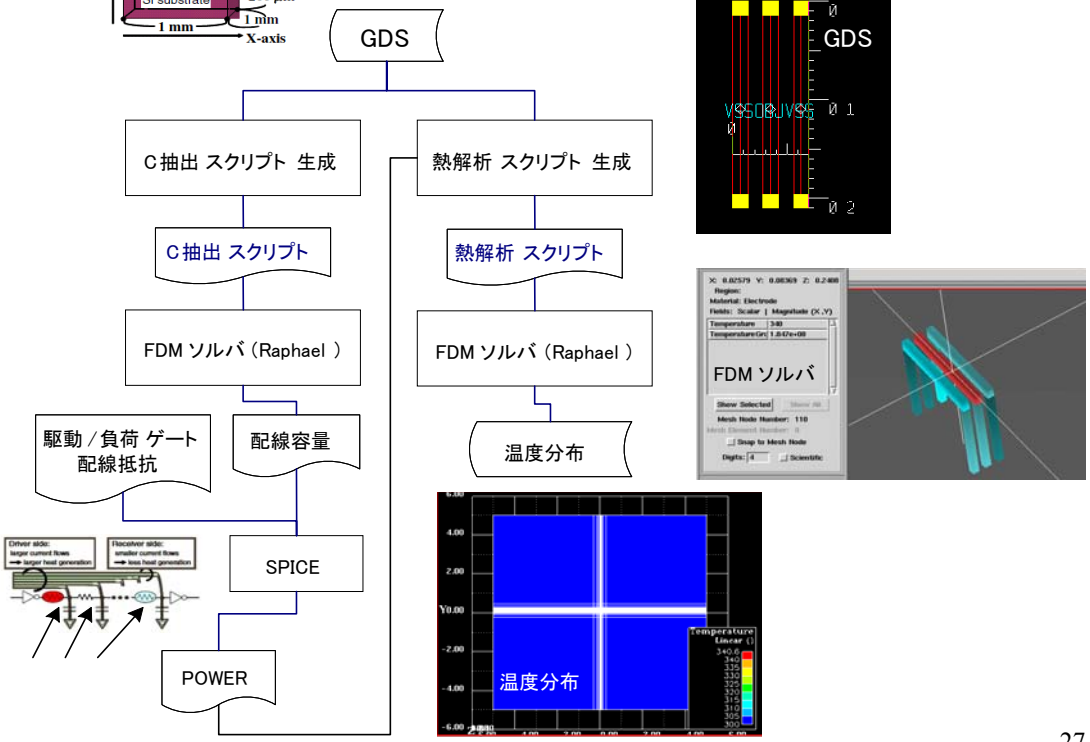
- ITRS2007[2)]の物理パラメータ、PTM[3)] の回路パラメータ、Nangate[4)]の設計データから32nmプロセス相当の評価データを策定
- 配線自己発熱がSignificantになると思われるケースを抽出

→ 短距離配線で大きな容量負荷を駆動[1)]

- 解析手段：有限差分法容量/熱解析、SPICE回路シミュレーション

評価フロー



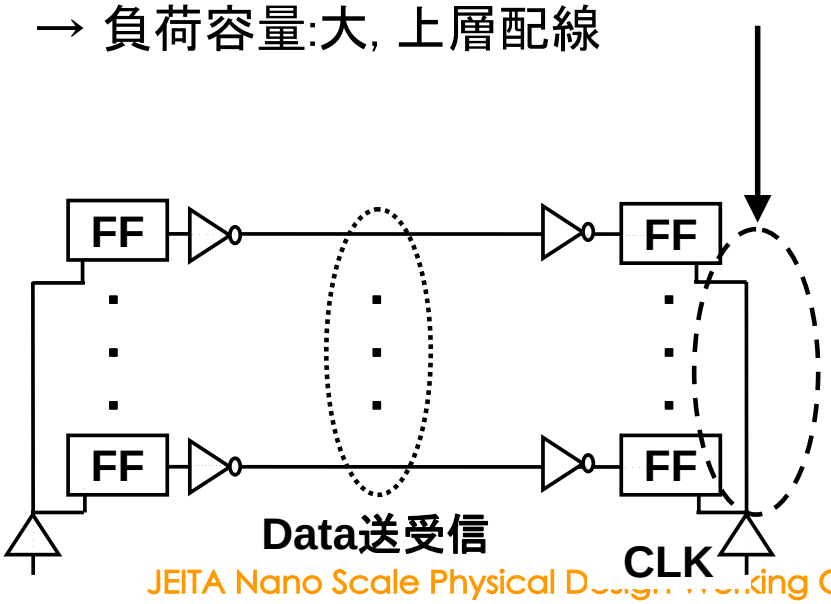


評価ケース策定

- 配線自己発熱による温度差が大きくなる要因
  - 配線の電流量:大 → 負荷容量:大
  - 体積あたりの消費電力:大
    → 短距離配線、細線
  - 基板からの距離:大 → 上層配線
  - 絶縁膜の熱抵抗:大 → 空孔を含む材質
    （Porous-silicaなど）

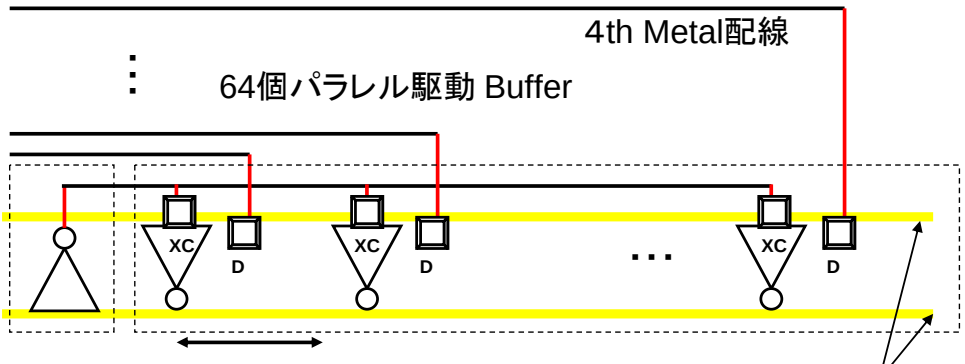
ケース1: 64bit-FF駆動モデル

- 64bitMPUのFullAdder[5]、Busモデル[6]
- 64bitのデータ(送)受信部のFF CLK配線
  → 負荷容量:大, 上層配線



Driver/Receiver/配線モデル

- 典型的な64bit-FF駆動モデルを想定



配線51ピッチ分
(**Scan Flip-flop with Set and Reset**
～Nangate 45nm Open Cell Library
からピッチを決定→32nm相当に変換)

電源
1Metal
+
VDD:4Metal ＋
GND:4Metal

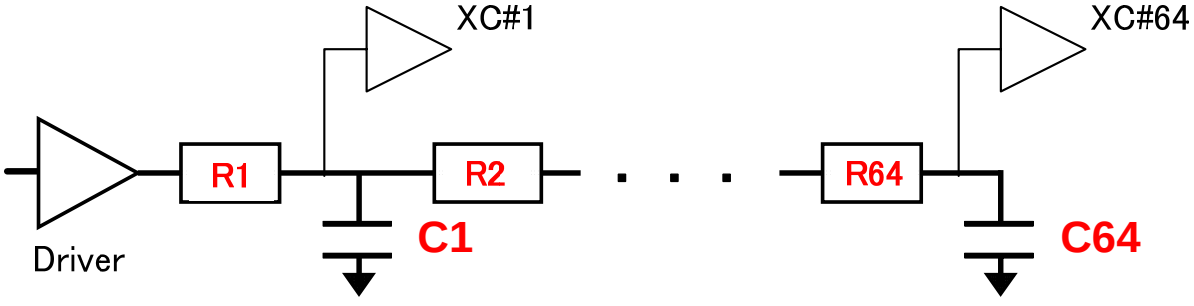
配線断面寸法・物理定数

- ITRS2007

Category	Year of Production		2013
	DRAM ½ Pitch (nm) (contacted)		32
	MPU/ASIC Metal 1 (M1) ½ Pitch (nm)		32
	Interlevel metal insulator – effective dielectric constant (κ)	εr	2.8
	Cu Temperature Coefficient of Resistance		0.0044
	On-chip local clock frequency (MHz)	f	7344
Interconnects	Metal 1 wiring pitch (nm)	p	64
	Metal 1 A/R (for Cu)		1.9
	Metal 1 wiring width (nm)	w	32
(Metal 1)	Metal 1 wiring thickness (nm)	t	60.8
	Height (dielectric thickness) between Metal 1 and substrate (nm)	h	60.8
	Barrier/cladding thickness (for Cu Metal 1 wiring) (nm)		2.4
	Intermediate wiring pitch (nm)	p	64
	Intermediate wiring dual damascene A/R (Cu wire)		1.9
	Intermediate wiring dual damascene A/R (Cu via)		1.7
	Intermediate wiring width (nm)	w	32
(Intermediate)	Intermediate wiring thickness (nm)	t	60.8
	Height (dielectric thickness) between Intermediate wiring levels (nm)	h	54.4
	Barrier/cladding thickness (for Cu intermediate wiring) (nm)		2.4
	Cu thinning at minimum intermediate pitch due to erosion (nm), 10% × height, 50% area density, 500 µm square array		6
	Conductor effective resistivity (µΩ-cm) Cu intermediate wiring including effect of width-dependent scattering and a conformal barrier of thickness specified below		4.83

等価回路

- ITRS2007 32nmプロセスの配線寸法を代入、配線長から抵抗、電磁界解析で容量を算出
- 上記を3Dモデルとして、熱解析を実施



物理構造パラメータの決定(1)

- ドライバ・レシーバサイズ

レシーバ入力Inverter:

Scan FF CLK入力

NMOS:W/L=0.4μm/32nm,

PMOS:W/L=2.0μm/32nm

ドライバ出力Inverter:

64FF, および配線負荷を7.3GHzで駆動可能な寸法。

Rise/Fallの電流のピーク値が等しくなるよう

P/N間でゲート幅を調整

NMOS:W/L=25.6μm/32nm,

PMOS:W/L=128μm/32nm

物理構造パラメータの決定(2)

- 配線:4M(Intermediate層)

配線幅:最小寸法×5 = 0.16μm

→最遠方のFFを7.3GHzで駆動可能な最小寸法

隣接電源配線(幅/間隔= 0.16μm /32nm)

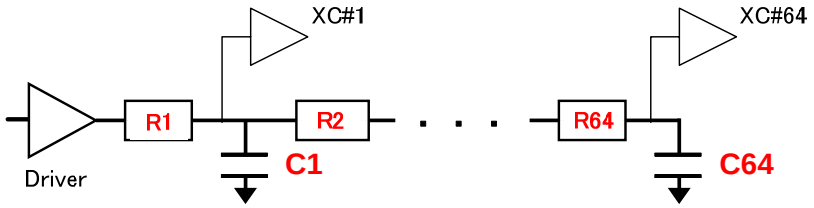
配線容量 Ctotal=C1+・・・+C64

容量シミュレーション結果より

Ctotal= 4.34e-14 F → **C1=C2=・・・=C64= 6.77E-16 F**

配線抵抗 Rtotal=R1+R2+・・・+R64= 1037 Ω

→ **R1=R2=・・・=R64= 16.2 Ω**



配線断面寸法・物理定数

- 配線絶縁膜の熱伝導率導出

  Step.1: SiO2にAirを混合した材質と仮定

  Step.2: 比誘電率が等価になるように混合比を決定

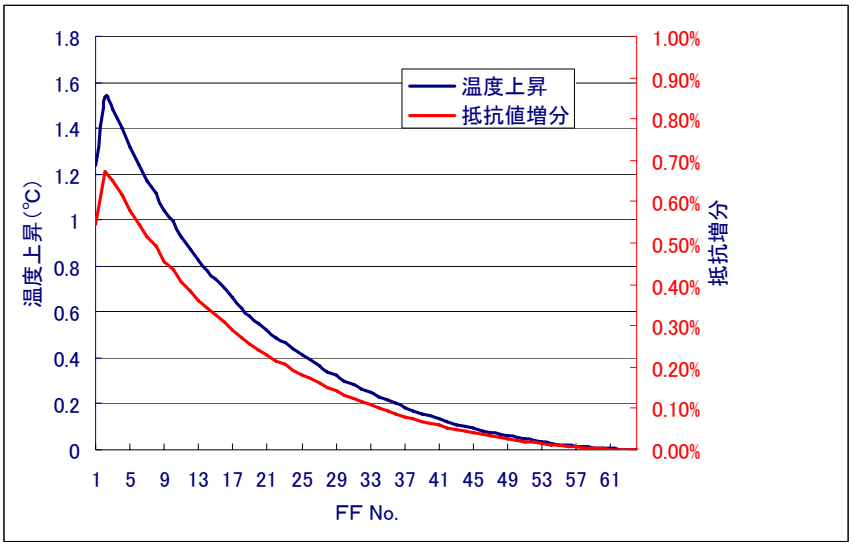
  Step.3: 求めた混合比に対する熱伝導率

	SiO2	Air	ITRS
εr	4.2	1	2.8
熱伝導率 W/meter・k	1.38	0.0241	0.1409

ケース1　解析結果

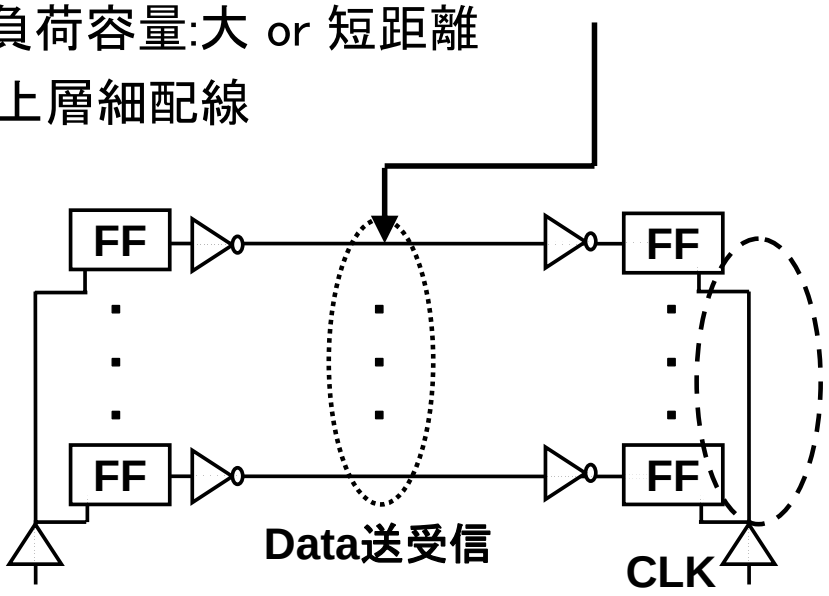
- 熱解析結果

  最大温度差: 1.53℃　→ 配線抵抗増加: 0.67%



ケース2: リピータ間配線

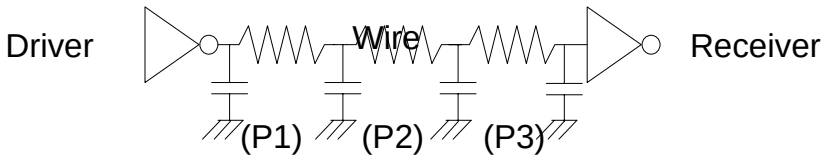
- データ送受信のリピータ間配線
  → 負荷容量:大 or 短距離
  上層細配線



物理構造パラメータの決定

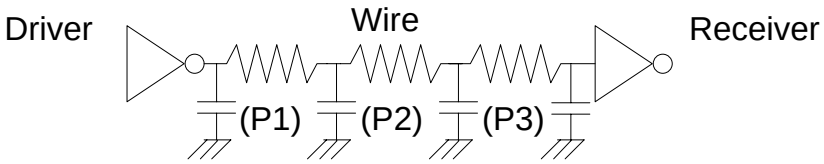
- 配線:4M(Intermediate層)
  配線幅:最小寸法 = 32nm
  隣接電源配線(幅/間隔= 32nm /32nm)

  配線容量 Cwire = 1.644E-16 F/μm
  配線抵抗 Rwire = 24.825 Ω/μm



物理構造パラメータの決定

- 配線長、ドライバ・レシーバサイズ決定条件
  1. ドライバ・レシーバは同一サイズ
  2. ドライバ遅延+配線遅延=クロック周期の1/7 [1)]
  3. ドライバ遅延と配線遅延の比:
     1/8, 1/4, 1/2, 1, 2, 4, 8
     (2. の条件を満たす範囲)
- 配線モデル: 3段に分割し、電力見積り→熱解析→遅延時間評価



物理構造パラメータの決定

- 単位Inverter諸元

Pmos(Unitサイズ)		
ε0	8.85E-12	F/m
epsrox	3.9	
cgso	8.50E-11	F/m
cgdo	8.50E-11	F/m
cgbo	2.56E-11	F/m
toxm	1.75E-09	m
L	3.20E-08	m
W	9.00E-07	m
Cgate	**7.22E-16**	**F**

Nmos(Unitサイズ)		
ε0	8.85E-12	F/m
epsrox	3.9	
cgso	8.50E-11	F/m
cgdo	8.50E-11	F/m
cgbo	2.56E-11	F/m
toxm	1.65E-09	m
L	3.20E-08	m
W	4.00E-07	m
Cgate	**3.37E-16**	**F**

- Elmore遅延で求めたドライバ・レシーバサイズ、配線長

Td_drv/Td_wire		1/8	1/4	1/2	1	2	4	8
Driverサイズ (Unit比)		22.4	7.2	3.6	2.0	1.2	0.8	0.5
配線長	μm	26.8	52.5	59.7	57.1	48.9	38.8	29.3

ケース2　解析結果

- 熱解析結果(1)

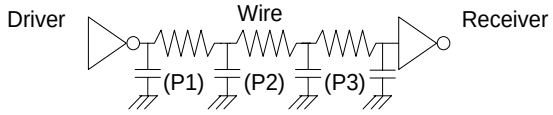
ドライバ遅延が配線遅延の1/8となるケースで
配線自己発熱による温度差が最大(P1: 0.162℃)
→ このケースに対し、当該配線を最小間隔で敷き詰めた状態で、熱解析をさらに実行
(Data送受信配線64bit分並走を想定)

Tdriver/Twire比		1/8	1/4	1/2	1	2	4	8
Power(W)	P1	8.99E-06	2.76E-06	1.96E-06	1.45E-06	9.70E-07	6.00E-07	3.05E-07
	P2	8.34E-06	1.93E-06	1.16E-06	7.92E-07	5.13E-07	3.14E-07	1.57E-07
	P3	7.77E-06	1.37E-06	6.39E-07	3.65E-07	2.11E-07	1.23E-07	5.87E-08
	Total	2.51E-05	6.06E-06	3.76E-06	2.61E-06	1.69E-06	1.04E-06	5.21E-07
⊿T(℃)	P1	1.62E-01	2.60E-02	1.60E-02	1.30E-02	1.00E-02	8.00E-03	6.00E-03
	P2	1.49E-03	1.77E-04	9.37E-05	6.74E-05	5.04E-05	3.85E-05	2.53E-05
	P3	1.37E-03	1.25E-04	5.19E-05	3.11E-05	2.08E-05	1.50E-05	9.80E-06

ケース2　解析結果

- 熱解析結果(2)

最大温度差: 5.49℃　→ 配線抵抗増加: 4.82%
Drv.入力－Rcv.入力
Delay増加: 2.72%



Tdriver/Twire		1/8
Power(W)	P1	8.99E-06
	P2	8.34E-06
	P3	7.77E-06
	Total	2.51E-05
⊿T(℃)	P1	5.49E+00
	P2	1.10E+01
	P3	5.49E+00
⊿R(Ω)	P1	2.41%
	P2	4.82%
	P3	2.41%

Summary

- 配線の自己発熱（Self-heating）要考慮シチュエーションをスクリーニング
- 32nmプロセスを想定した現実的な構造、物性値を使って配線伝播遅延へのインパクトを評価
- 64bitのデータ送受信モデル

1）データ受信部のFF CLK配線
→ 配線-基板間最大温度差 1.53℃、配線抵抗増加 0.53%

2）データ送受信のリピータ間配線
→ 最大温度差 5.49℃、配線抵抗増加 4.82%
→ Drv.入力－Rcv.入力Delayの増加： 2.72%

参考文献

1) K. Shinkai, M. Hashimoto, T. Onoye, “Future Prediction of Self-heating in Short Intra-block Wires, ” Proceedings of International Symposium on Quality Electronic Design (ISQED), pp. 660-665, March 2007.
2) Semiconductor Industry Association,”International Technology Roadmap for Semiconductors 2007 Edition,” 2007. http://www.itrs.net/Links/2007ITRS/Home2007.htm
3) W. Zhao, Y. Cao, “New generation of Predictive Technology Model for sub-45nm design exploration,” Proceedings of International Symposium on Quality Electronic Design (ISQED), pp.585-590, March 2006.
4) Nangate Inc.,”45nm Open Cell Library,” http://www.nangate.com/
5) Radu Zlatanovici,” Power - Performance Optimization for Digital Circuits,” EECS Department, U.C.Berkeley Technical Report No. UCB/EECS-2006-164, December, 2006.
6) P.P. Sotiriadis, A.P. Chandrakasan, “A Bus Energy Model for Deep Submicron Technology,” IEEE Transactions on VLSI Systems, Vol.10, No.3, pp341-350, June 2002

3. 配線ばらつきの表現手段の調査、検討

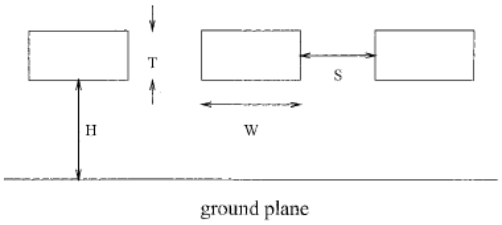
もくじ

- 配線ばらつきのインパクト
  - 配線ばらつきの要因
  - ロジック回路性能への影響
    - 遅延
    - 電源ノイズ
- 配線ばらつきを考慮した設計フロー
  - タイミング解析
  - クロストーク、Power解析
- Sensitivity SPEF標準化状況
  - Sensitivity SPEFフォーマット
  - IEEE標準化委員会の状況
  - 現フォーマットの課題
- まとめ

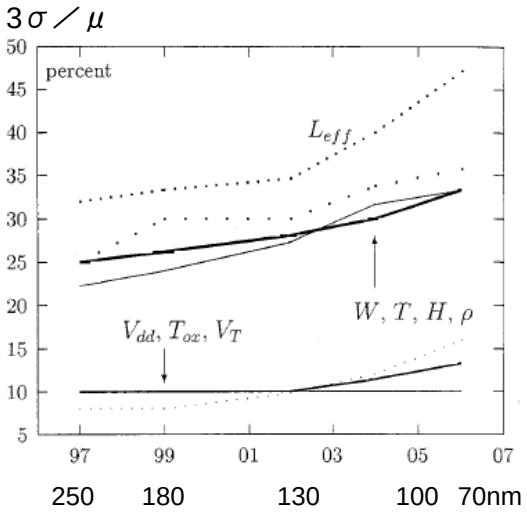
配線ばらつきの要因

- リソグラフィー
- エッチング
- Copper Electroplating
- CMP (Chemical-Mechanical Polishing)

配線ばらつきの大きさ



Year	L_{eff} nm	T_{ox} nm	V_{dd} V	V_T mV	W μ	H μ	ρ $\frac{m\Omega}{\square}$
1997	80	0.4	0.25	50	0.2	0.3	10
1999	60	0.36	0.18	45	0.17	0.3	12
2002	45	0.39	0.15	40	0.14	0.27	15
2005	40	0.42	0.12	40	0.12	0.27	19
2006	33	0.48	0.09	40	0.1	0.25	25



S. Nassif, “Delay Variability: Sources, Impacts and Trends” ISSCC 2000.

配線ばらつきのロジック回路への影響

- 遅延
- クロストークノイズ
- 電源ノイズ
- 消費電力

配線ばらつきの信号伝播遅延への影響(1)

Nassifの予測

	1997	1999	2002	2005	2006
Constant W/L, wire length $= L_{max}$					
Device (%)	47	47	44	44	45
Wire (%)	53	53	56	56	54
Case (A): no scaling of device width					
Device (%)	47	43	37	35	34
Wire (%)	53	57	63	65	66
Case (B): scale wire length with L_{eff}					
Device (%)	47	51	52	55	60
Wire (%)	53	49	48	45	40

S. Nassif, “Delay Variability: Sources, Impacts and Trends” ISSCC 2000.

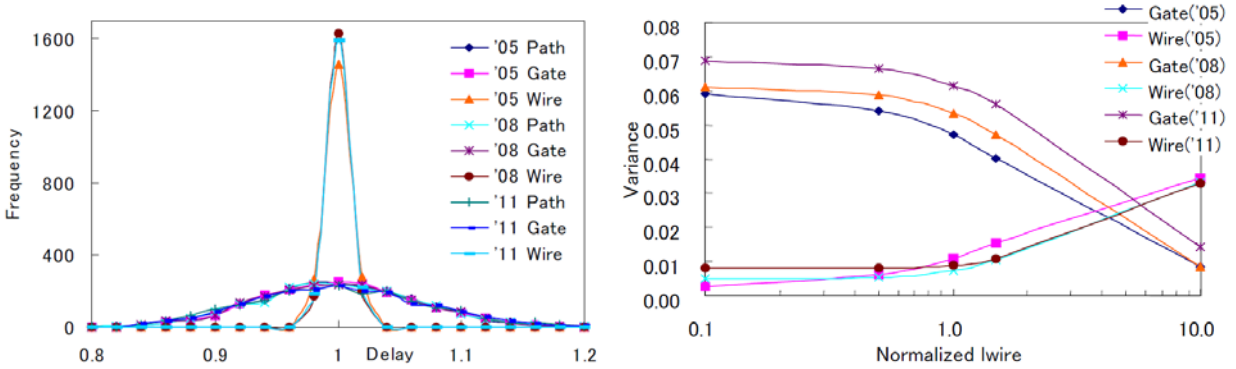
配線ばらつきの信号伝播遅延への影響(2)

DMD研究会の予測(2002)

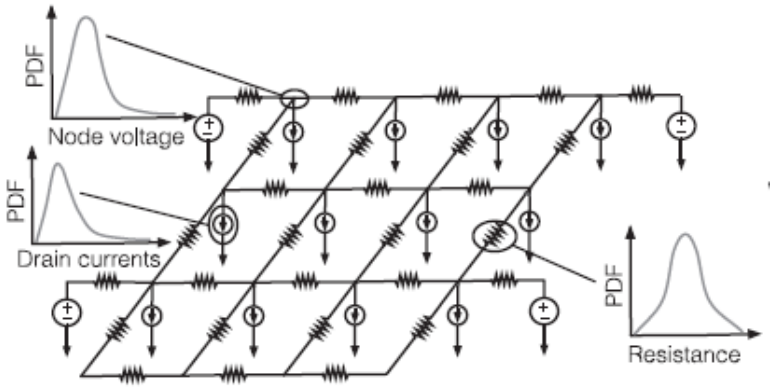
	Lg(p/n)	Tox(p/n)	Vth(p/n)	Vdd	配線幅 Ww	配線膜厚 Tw	ビア高 Hw
ばらつき設定量 (3σ)	10%	4%	12.5%	10%	10%	10%	10%

世代によらず一定

'05, '08, '11=100, 70, 50nm



配線ばらつきの電源ノイズへの影響



- ばらつきパラメータ: 配線W, T, デバイスWd, Vth ($3\sigma/\mu$=30%)
  → ノード電圧のσ=15～28%
- ばらつきパラメータ: デバイスVth ($3\sigma/\mu$=30%)
  → ノード電圧のσ=10～20%

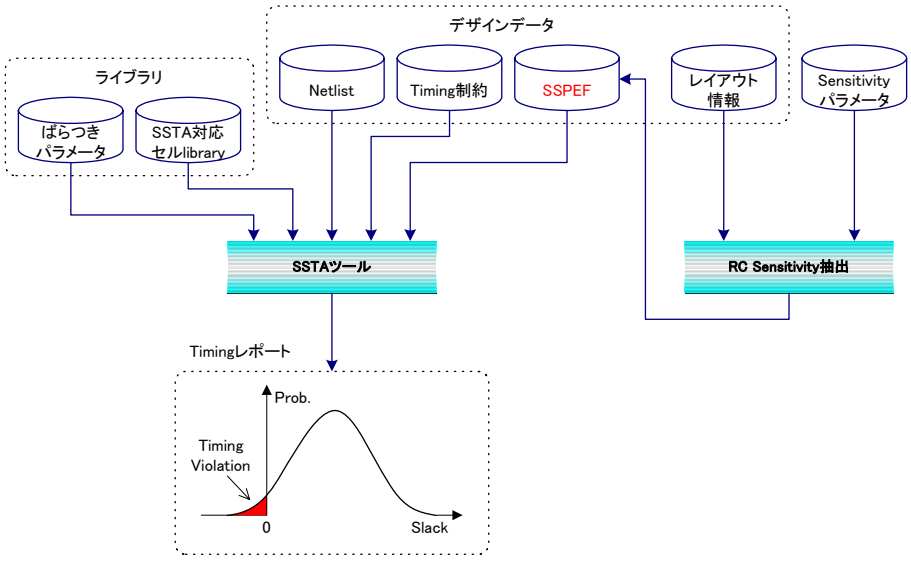
P. Ghanta et. al. “Analysis of Power Supply Noise in the Presence of Process Variation”, IEEE Design & Test of Computers, 2007.

配線ばらつきを考慮したLSI設計

- 配線ばらつき情報(SSPEF)を用いた設計フロー
  - タイミング解析
    - Statistical STA
    - 従来フローの延長（STA）
  - クロストーク、Power解析

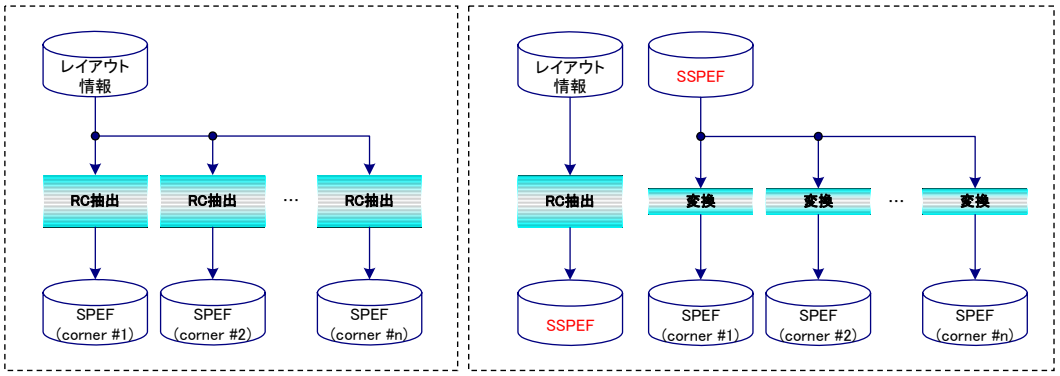
タイミング解析

- Statistical STAに使用



タイミング解析

- 従来フローの延長
  - 従来：コーナー条件毎にSPEFを作成
  - SSPEFからコーナーSPEFを生成
    - RC抽出の実行回数が低減できる



クロストーク、Power解析

- RC値に関して、コーナー条件での値ではなく統計的に処理
  - ツールの対応が必要
  - クロストーク解析
    - Critical Path解析に活用
      - 配線ばらつきを考慮したインクリメンタルSDF
      - Glitchエラー緩和への利用
  - Power解析
    - より実chipに近いPower見積り値
    - EM、IR-Drop解析時に適用し、marginの取り過ぎを防止

Sensitivity–SPEF フォーマット

$$cn_j = \frac{\partial C(p)/C_o}{\partial p_j/NF(p_j)} = \left(\frac{\partial C(p)}{\partial p_j}\right)\left(\frac{NF(p_j)}{C_o}\right)$$

$$V.C.(p_i) = \frac{\sigma(p_i)}{N.F.}$$

$$\Delta v_i = VC(p_i) * VM(p_i)$$

where,
VC(pi) is the variation coefficient defined in the header section.
VM(pi) is the variation multiplier, with a conventional value of 3

W/T/H Variation (VC) **W/T/H Mean (NF)**

```
*VARIATION_PARAMETERS
0 M2_W N D X 0.0330000 0.600000
1 M2_T N D X 0.0670000 0.700000
   :
```

param_id param_name param_type_for_cap param_type_for_res param_type_for_induc var_coeff normalization_factor

```
*D_NET RCXX_NET1 122.995
   :
*CAP
1 RCXX_NET1:24 RCXX_NET2:48 0.0235753 *SC 0:0.497 1:0.445
2 RCXX_NET1:24 RCXX_NET2:49 2.17010 *SC 0:0.436 1:0.387
   :
*RES
1 RCXX_NET1:24 RCXX_NET1:23 16.5000 *SC 0:0.375 1:1.000
```

Cap Mean (C0)

Sensitivities (cnj)

$$C(p) = C_o \frac{(1+\sum_j cn_j \Delta v_j)}{(1+\sum_i cd_i \Delta v_i)}$$

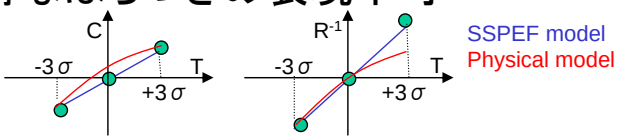
Param_type: N (numerator), D (denominator) or X (not applicable).

Sensitivity SPEF標準化

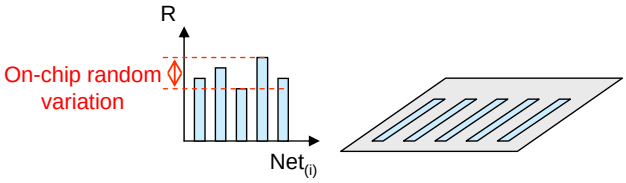
- IEEE標準化委員会状況
  - IEEE Std. 1481-1999 Standard Parasitic Exchange Format に配線ばらつきに関する記述を追加(SSPEF)
  - 現在、最終ドラフト作成完了、投票前レビュー中
  - 最終ドラフト版をNPD-WGとして正式に入手、ML参加
- NPD今後のアクション
  - IEEE最終ドラフト版に至った背景調査
  - 現状フォーマットの課題調査
  - ドラフト版に対する修正提案
  - SSPEFの適用方法検討・提案

SSPEFの課題

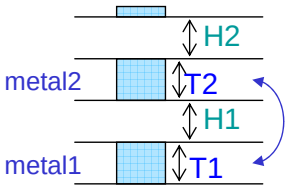
- ばらつきパラメータとRLCの関係式は、比例もしくは反比例のみ表現可能
- 正側負側非対称なばらつきの表現不可



- D2D（W2W,L2Lを含む）成分のみ表現可能
- WID成分の表現不可



- 距離依存、配線層間等、パラメータ間相関係数の定義不可



まとめ

- 微細化に伴いLSI性能に与える配線ばらつきは増大している。配線ばらつきを精度よくモデル化し解析に用いなければ、過剰/過小設計を強いられることになる。
- 配線ばらつきを反映した設計フローとしては、SSTAに代表されるタイミング解析、クロストーク解析、Power解析等が想定される。
- 主要な配線ばらつきの表現フォーマットとしては、Sensitivity SPEFをIEEEで標準化作業中。Sensitivity SPEFでは、配線のRLCに対するばらつきパラメータ感度が表現可能。
- ただし、Sensitivity SPEFはまだ実用実績はなし。ばらつき表現上の課題もあり。今後、適用方法、および次期改訂に向けたばらつき表現上の改善提案を行う予定。

JEITA Nano Scale Physical Design Working Group

4.2 SystemCワーキンググループ 2008年度活動報告

JEITA EDA技術専門委員会

標準化小委員会

SystemCワーキンググループ

JEITA

© Copyright 2009 JEITA, All rights reserved

SystemCワーキンググループメンバー

主査	長谷川　隆	（富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ）
副主査	今井　浩史	（東芝）
委員	中西　早苗	（NECエレクトロニクス）
	小島　智	（NECシステムテクノロジ）
	清水　靖介	（ﾛｰﾑｸﾞﾙｰﾌﾟ・OKIｾﾐｺﾝﾀﾞｸﾀ）
	逢坂　孝司	（ケイデンス）
	長尾　文昭	（三洋半導体）
	西園寺 修	（シノプシス）
	柿本　勝	（ソニー ）
	竹村　和祥	（パナソニック）
	牧野　潔	（メンター）
	大島　良紀	（ルネサステクノロジ）
	長谷川 裕恭	（HD LAB）
	東島　清宏	（CoWare）
客員	今井　正治	（大阪大学）
		（計15名）

© Copyright 2009 JEITA, All rights reserved

目次

© Copyright 2009 JEITA, All rights reserved

4.2.1 SystemCワーキンググループ概要

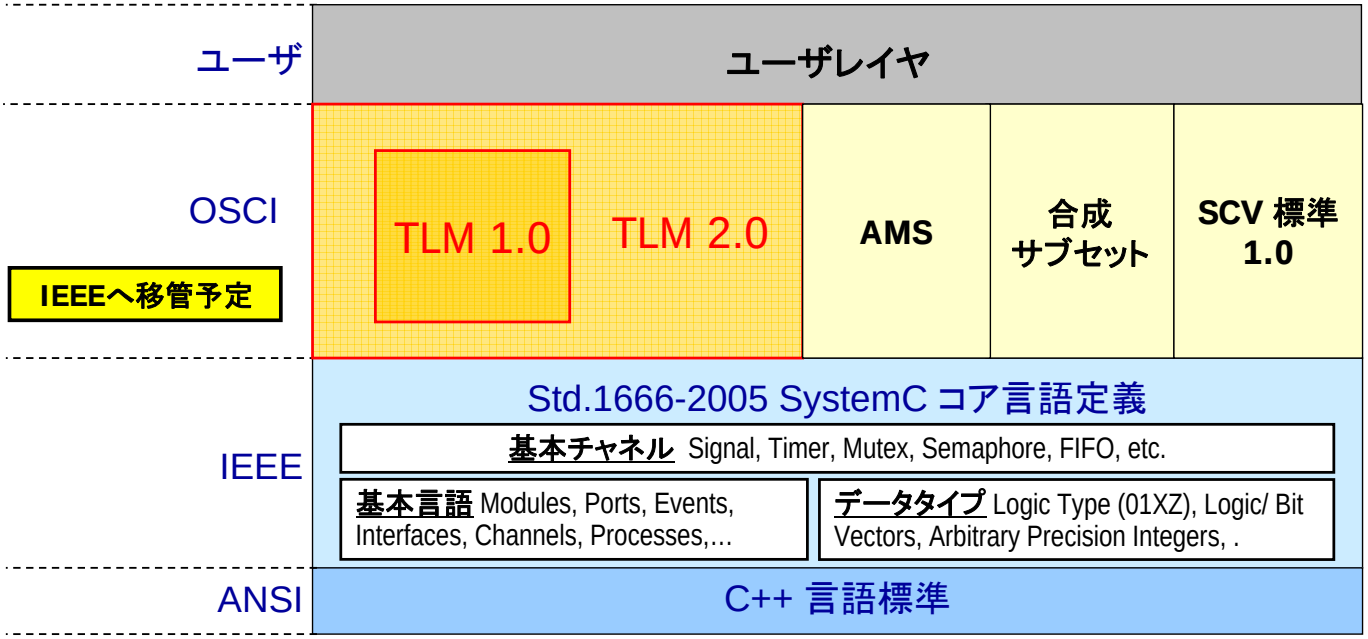
© Copyright 2009 JEITA, All rights reserved

SystemC とは

- C++言語をベースとした、システムレベル設計言語の代表的な言語である
  - Open SystemC Initiative (OSCI)という標準化組織により、言語仕様(LRM)とリファレンスシミュレータが策定され、無償提供されている
    http://www.systemc.org/
  - 2005年12月に、SystemC 2.1のLRMがIEEE Std. 1666として標準化された
  - 現在は合成サブセットやTLM(Transaction Level Modeling)の標準化案が検討されている
- C++の文法を保持したまま、クラスライブラリの形で以下のような言語拡張がなされている
  - 並列動作を可能とするシミュレーションエンジン(クロック、イベント、等)
  - 抽象化された通信手段(Channels, Interfaces)
  - ハード実装に必要なデータタイプ (固定小数点、固定長ビット、等)
  - 0, 1, Z, X 等の信号値等

© Copyright 2009 JEITA, All rights reserved

SystemCの標準化の階層



OSCI資料(2007)に加筆

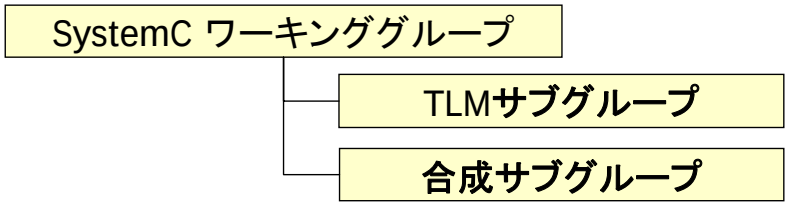
© Copyright 2009 JEITA, All rights reserved

SystemCワーキンググループについて

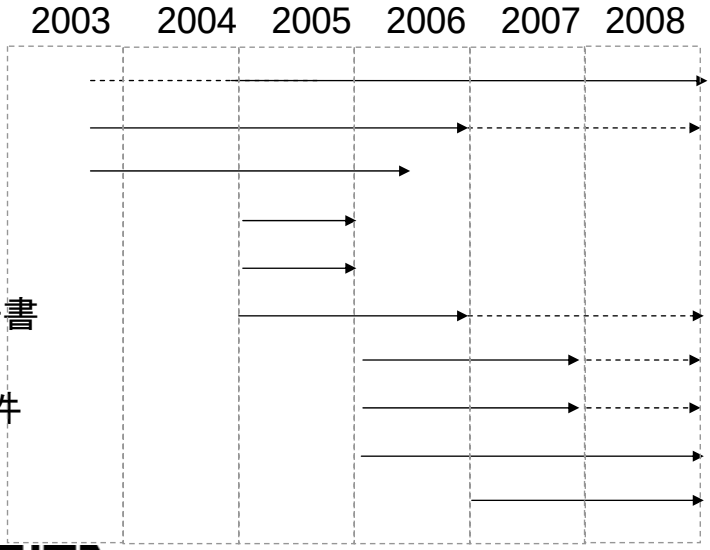
- 設立と名称変更
  - 2003年10月に、JEITA EDA技術専門委員会 標準化小委員会内にSystemCタスクグループとして設置(SystemVerilogタスクグループと同時)
  - 2007年4月度よりSystemCワーキンググループに名称変更して活動継続
- 目的
  - 日本国内における唯一のSystemCの標準化関連組織として、OSCIやIEEE P1666ワーキンググループと連携しつつ、日本国内の事情・要求事項を取り込むべくSystemCの国際標準化を進めていく。
  - SystemCに関連した調査結果を積極的に情報発信を行うことで、国内普及を図る。これらにより日本の産業界の国際競争力を高めることを目指す。

© Copyright 2009 JEITA, All rights reserved

SystemCワーキンググループの活動



- これまでの歩み
  - SystemCユーザ・フォーラム
  - SystemC動向調査
  - OSCI LRMレビュー
  - IEEE P1666 WG 標準化活動
  - SystemC 2.1調査
  - SystemC-SystemVerilog共通辞書
  - 合成サブセットレビュー
  - 動作合成スタイルガイド構成要件
  - TLM標準化(レビュー)
  - SystemC推奨設計メソドロジ



© Copyright 2009 JEITA, All rights reserved

これまでの活動内容と主な成果

- SystemC標準化活動
  - IEEE P1666 WGに投票権のあるメンバーとして参加し、SystemC言語仕様書のレビューにて50件以上の改善提案によりIEEE Std. 1666-2005の策定に貢献(2005年度)
  - OSCI動作合成サブセットdraft、TLM2.0 draft1に対するレビューを実施し、OSCIへフィードバックを提出。
  - TLM 2.0 draft2のレビューを実施し、OSCIへ10件のフィードバックを提出した(説明不足7件、例題追加要望3件) (2007年度)
  - 2008年6月にリリースされたTLM 2.0正式版ユーザマニュアルのレビューを実施し、OSCIへ7件のフィードバックを提出。レビュー作業の過程を日本語抄訳としてまとめ、本アニュアルレポートに掲載(2008年度)
- SystemC技術調査
  - 2000年～2005年度における国内外でのSystemCの利用状況について調査を実施
  - SystemC 2.1について調査を行い、その特長を日本語で紹介(2005年度)
  - 国内外におけるTLM(Transaction Level Modeling)の利用状況調査を実施(2006-2007年度)
  - OSCIより公開されている合成サブセットのドキュメントのレビュー及び抄訳の作成を実施(2006年度)
  - SystemC動作合成ガイドライン構成要件の検討と、ガイド準拠のドキュメントのレビュー(2007年度)
  - SystemC推奨設計メソドロジの検討を行い、合成編について審議完了(2007年度)
  - SystemC推奨設計メソドロジの総合編の検討・レビューを行い審議完了し、本アニュアルレポートに掲載(2008年度)
- SystemC普及活動
  - SystemCユーザフォーラムを開催し、積極的に情報発信を行いSystemCを利用した設計の普及をはかる
  - SystemC Japan 2007にてこれまでの活動成果を講演(2007年度)
  - JEITA EDA-TCのWEBページを活用し、これまでの活動成果を一挙掲載(2007年度)
    http://eda.ics.es.osaka-u.ac.jp/jeita/eda/index-jp.html

© Copyright 2009 JEITA, All rights reserved

2008年度報告資料について

4.2.2 TLM 2.0 ユーザマニュアルの日本語要約
- OSCI TLM-2.0 User Manual (JA22)の日本語抄訳

4.2.3 TLM 2.0 補足解説
- TLM 2.0 Draft2からの変更点について解説

4.2.4 TLM 2.0 のOSCIへのフィードバック
- OSCI へ提出した TLM2.0 における改善要望

4.2.5 SystemC推奨設計メソドロジ 2008年度版
- 推奨設計メソドロジとしてSystemCの効果的な使い方を提案

4.2.6 SystemCユーザフォーラム2009開催報告
- SystemCユーザフォーラムの開催報告とアンケート結果

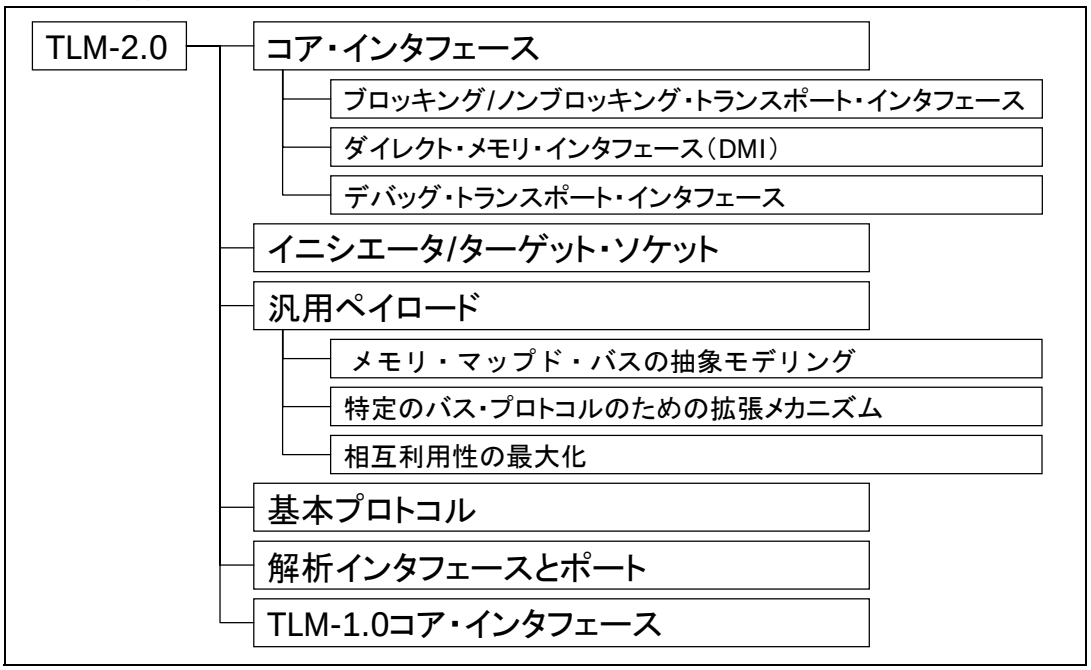
© Copyright 2009 JEITA, All rights reserved

4.2.2 TLM 2.0 ユーザ・マニュアルの日本語抄訳

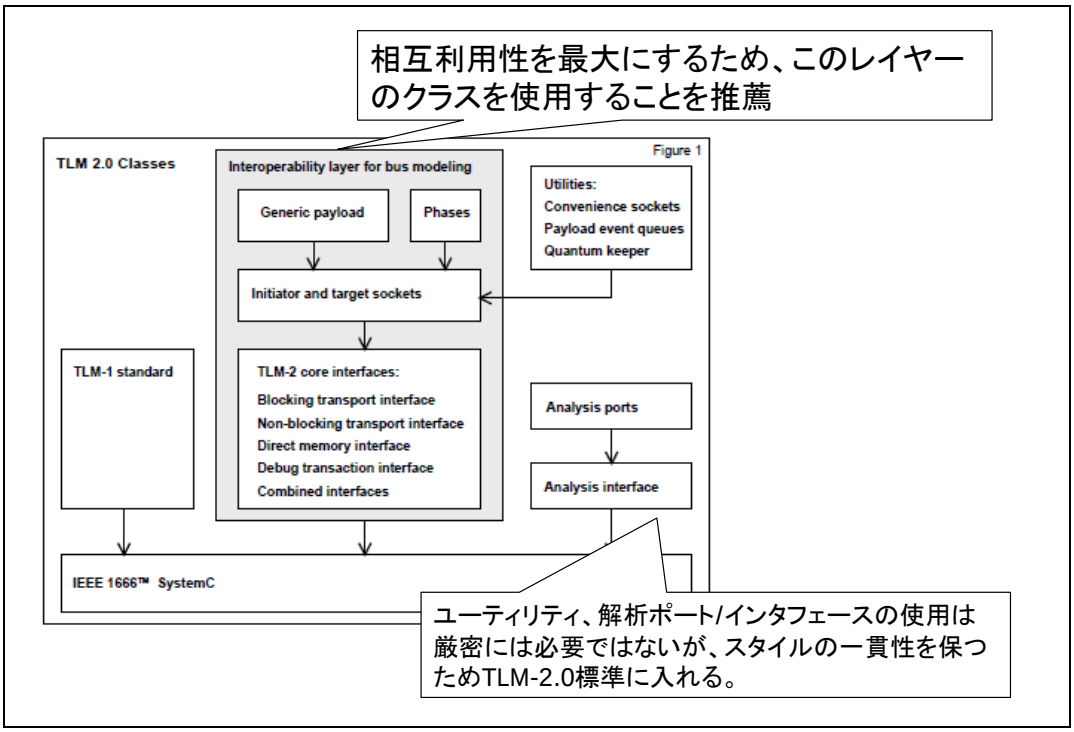
本節は、OSCI の TLM2 USER MANUAL Software version: TLM-2.0、Document version: JA22 を日本語抄訳したものである。

1. あらまし

- TLM-2.0 の内容



- TLM-2 クラス



- 汎用ペイロード
  - メモリ・マップド・バスのモデリングを主なターゲット（非バス・プロトコルにも使用可能）とする。
  - アトリビュート、フェーズは拡張可能（特定なプロトコルをモデリングするため）。
    - 相互利用性を阻害するので使用には注意が必要。
- コーディング・スタイル

- モデルとスタイルの関係
  - ・ルーズリー・タイムド・モデル ⇒ ブロッキング・トランスポート・インタフェース、DMI、テンポラル・デカップリング
  - ・アプロキシメイトリー・タイムド・モデル ⇒ ノンブロッキング・トランスポート・インタフェース、ペイロード・イベント・キュー

  上記は推奨される使い方であるが、これらに限定される訳ではない。

1.1 スコープ

- このドキュメントの焦点
  - TLM-2 コア・インタフェースとクラスのキー・コンセプトとセマンティクスである。
- TLM-1 コア・インタフェースをリストアップするがそのセマンティクスを定義はしない。
- 言語リファレンス・マニュアルではない。
- TLM-2.0 を如何に使うかといった、より実践的なガイドラインを追加していくつもりである。

1.2 ソースコードとドキュメント

- TLM-2.0 リリースのディレクトリ構造

include/tlm/	:ソースコード、readme ファイル、リリース・ノート
include/tlm/tlm_h/tlm_req_rsp	:TLM-1 標準
include/tlm/tlm_h/tlm_req_rsp/tlm_1_interfaces	:TLM-1 コア・インタフェース
include/tlm/tlm_h/tlm_req_rsp/tlm_channels	:TLM-1 fifo、req-rsp チャネル
include/tlm/tlm_h/tlm_req_rsp/tlm_ports	:TLM-1 ノンブロッキング・ポート、イベント・ファインダー
include/tlm/tlm_h/tlm_req_rsp/tlm_adapters	:TLM-1 スレーブ-トランスポート、トランスポート-マスタ・アダプタ
include/tlm/tlm_h/tlm_trans	:TLM-2 相互利用性クラス
include/tlm/tlm_h/tlm_trans/tlm_2_interfaces	:TLM-2 コア・インタフェース
include/tlm/tlm_h/tlm_trans/tlm_generic_payload	:TLM-2 汎用ペイロード
include/tlm/tlm_h/tlm_trans/tlm_sockets	:TLM-2 ソケット
include/tlm/tlm_h/tlm_quantum	:TLM-2 グローバル・クォンタム
include/tlm/tlm_h/tlm_analysis	:TLM-2 解析インタフェース、ポート
include/tlm/tlm_utils	:TLM-2 ユーティリティ・クラス
docs	:ドキュメント（ユーザ・マニュアル、ホワイト・ペーパー、Doxygen）
examples	:コード例
unit_test	:リグレッション・テスト

2. リファレンス

- 本標準は以下の文書とともに使用すること。
  - ISO/IEC 14882:2003, Programming Languages-C++
  - IEEE Std 1666-2005, SystemC Language Reference Manual
  - Requirements Specification for TLM 2.0, Version 1.1, September 16, 2007

2.1 参考文献

- Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Frank Ghenassia, published by Springer 2005, ISBN 10 0 387-26232-6(HB), ISBN 13 978-0-387-26232-1(HB)
- Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms, by Tim Kogel, Rainer Leupers, and Heinrich Meyr, published by Springer 2006, ISBN 10 1-4020-4825-4(HB), ISBN 13978-1-4020-4825-4(HB)
- ESL Design and Verification, by Brian Bailey, Grant Martin and Andrew Piziali, published by Morgan Kaufmann/Elsevier 2007, ISBN 10 0 12 373551-3, ISBN 13 978 0 12 373551-5

3. イントロダクション

3.1 背景

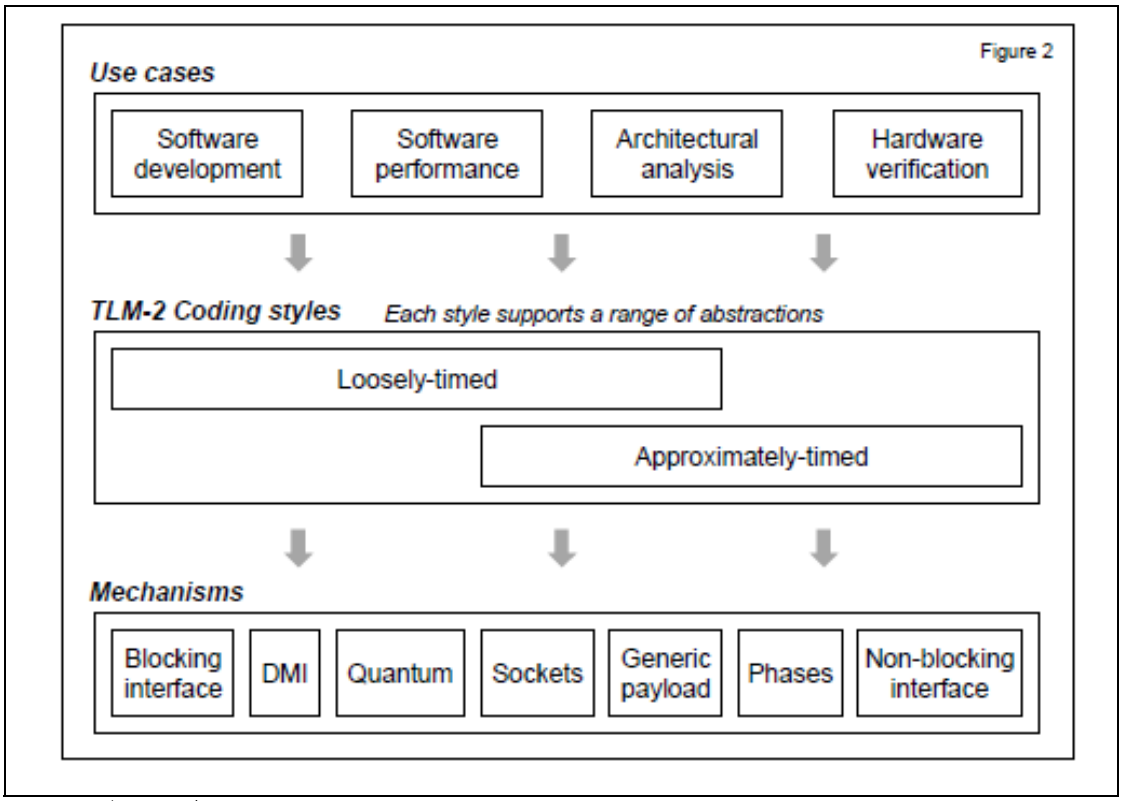
- TLM-1 標準は、メモリ・マップド・バス、オンチップ・コミュニケーション・ネットワークをモデリングする上で 3 つの欠点がある。

欠点	TLM2 での解決
トランザクション・クラスの標準が無い。	汎用ペイロード

タイミング・アノテーションをサポートしない。 （モデル間でタイミング情報を交換する標準的方法がない。wait 文で遅延を実装する。）	ブロッキング/ノンブロッキング・トランスポート・インタフェースでのタイミング・アノテーション
トランザクションを値もしくは参照渡しする。	トランザクション・オブジェクトが幾つかのトランスポート・コールに渡る仕組み（新しいトランスポート・インタフェースがサポート）

3.2 トランザクション・レベル・モデリング、ユースケース、抽象度

- 分類法を変更
  - ユースケースに応じた抽象度を定義する代わりに、インタフェース（API）とコーディング・スタイルを区別するといアプローチをとる。
  - 様々なユースケースに適するコーディング・スタイルを説明する。
- TLM-2 インタフェースは、相互利用性を保証する。
- 各コーディング・スタイルは様々な抽象度をサポートできる。
- アンタイムド機能モデル（アルゴリズム・モデル）
  - 唯一つの SystemC プロセスからなる。
  - トランザクション・レベルではない（プロセス間の通信がないから）。
- トランザクション・レベル
  - 複数のプロセスから成る。
  - 他のプロセスへ制御を譲渡する仕組み（wait 文）。
  - 強い同期と弱い同期
    - 強い同期：一連のコミュニケーションがあらかじめ正確に決まっている。
      FIFO、セマフォで実現可能。これは完全なアンタイムド・モデリング・スタイルなので、TLM-2.0 の範囲外。
    - 弱い同期：一連のコミュニケーションが個別のプロセスの詳細タイミングで部分的に決まる。
- ルーズリー・タイムド・コーディング・スタイル
  - 複数の組込みソフトウェア・スレッドの並列実行を許すバーチャル・プラットフォーム・モデルは、強い、もしくは、弱い同期を使う。このモデルに適したスタイル。
- アプロキシメイトリー・タイムド・コーディング・スタイル
  - プロトコル固有の複数のタイミング・ポイント（プロトコルの各フェーズの開始・終了のタイミング等）をトランザクションに関連付けるためには、より詳細なトランザクション・レベル・モデルが必要なケースがある。適切なタイミング・ポイントを選ぶことにより、コンポーネント・モデルをクロック精度で動作させることなくタイミング精度の高いコミュニケーションをモデリング可能。このようなコーディング・スタイル。



3.3　コーディング・スタイル

- 本ドキュメントでは、2 つのコーディング・スタイルについてのみ詳しく述べる。
  - ルーズリー・タイムド
  - アプロキシメイトリー・タイムド
- コーディング・スタイルは厳密に定義されるわけではない。
- TLM-2 コア・インタフェースを律するルールはコーディング・スタイルとは別に定義される。

3.3.1　アンタイムド・コーディング・スタイル

- アンタイムド・コーディング・スタイルは規定しない。
  - 現代のバス・ベース・システムは組込みプロセッサ上で動作するソフトウェアをモデリングするため、何らかのタイミングの概念が要求される。
- アンタイムド・モデリングは TLM-1 コア・インタフェースでサポートされる。

3.3.2　ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング

- ブロッキング・トランスポート・インタフェースを使用。
  - 2 つのタイミング・ポイント（トランスポート関数コールと戻り）
    - リクエスト開始とレスポンス開始にゆるく対応。
    - 同じシミュレーション時刻でも、異なる時刻でも良い。
- ユースケース：SW 開発（マルチ・プロセッサのバーチャル・プラットフォーム・モデル）
  - タイマーと割込みのモデリングをサポート（OS ブートとコード実行に十分）。
- 特徴：テンポラル・デカップリング
- テンポラル・デカップリングをサポート
  - モデルは他モデルとの同期ポイントに到達するまで自分のローカル時間を先行して実行可能。
    - 各プロセッサは次のプロセッサが実行される前に、ある時間（クォンタム）だけ実行
  - ⇒高速なシミュレーション
    - スケジューリングのオーバーヘッド低減。
- SystemC スケジューラからの考察
  - イベントが発生する時刻まで時間を進め、その時刻で動作すべきプロセスを実行。
  - プロセスの実行が終了すると、シミュレーション・カーネルに制御が返る。
  - モデルが詳細だと、イベント・スケジューリングとプロセスのコンテクスト・スイッチのオーバーヘッドがシミュレーション速度の支配的要因となる。
  - ⇒シミュレーション高速化の一つの方法： 現在時刻に先行してプロセスを実行すること。
    - テンポラル・デカップリング

- テンポラル・デカップリングの実装
  - プロセスは、以下の 2 つの外部依存のケースに出会うまで時刻を先行して実行可能。このとき、現状の値を受け入れて実行継続か、シミュレーション・カーネルに制御を返す。
    - 他プロセスにより値を書きかえられる変数に依存
    - 他プロセスと情報交換
  - 2 つの対処法
    - 同期を強制（シミュレーション時間に追いつくまで他のプロセスに制御を譲渡 = 標準の SystemC シミュレーション・セマンティクス)。
    - 現在の値を受入れ、実行継続（値の早期サンプリングがダメージを与えない、以降の値変化は以降のプロセス実行中で取込まれると仮定。ソフトウェア・スタックがハードウェアの詳細タイミングに依存しないバーチャル・プラットフォーム・モデルではこの仮定が成り立つ)。
- クォンタム
  - 制限なしにシミュレーション時刻を先行してプロセスが実行 ⇒ SystemC スケジュール動作しない ⇒ 他プロセスは決して動作しない。
  - ⇒グローバル・クォンタムで制限。
    - 先行実行の上限を設定。
    - アプリケーションが設定。
    - シミュレーション速度と精度のトレードオフ
      小さすぎる値 ⇒ 低速。制御の譲渡と同期が頻繁。
      大きすぎる値 ⇒ システム機能停止。タイミングの一貫性が保てない。
- 例：プロセッサ、メモリ、タイマー、低速なペリフェラルから構成されるシステム。
  - ソフトウェアはその実行時間の大半をメモリからの命令のフェッチと実行に費やし、システムの残りとはタイマーからの割込み（例えば、1 ms 毎）が発生したときのみ情報交換。
  - ISS モデルは 1 ms のクォンタムまで先行実行可能（メモリ・モデルには直接アクセスし、タイマー割込みでペリフェラルとのみ同期)。この間は他のハードウェア・モデルのクロックにロックされる必要なし。
  - 1000 倍のスピード・アップが可能。
- テンポラル・デカップリングしたモデル、しないモデルが混在する場合、テンポラル・デカップリングしないモデルがシミュレーションスピードのボトルネックになる可能性が高い。
- TLM-2 でのサポート
  - テンポラル・デカップリング
    - ↑ tlm_global_quantum クラス、ブロッキング/ノンブロッキング・トランスポートのタイミング・アノテーション
  - グローバル・クォンタム
    - ↑ tlm_quantumkeeper ユーティリティ・クラスによるアクセス

3.3.3 ルーズリー・タイムド・モデルの同期

- アンタイムド・モデル
  - 明示的同期ポイント（wait 文、ブロッキング・メソッドのコール）
- ルーズリー・タイムド・モデル
  - 明示的同期 ⇒ 予測可能な実行を保証
  - イニシエータはクォンタムが終了するまで先行実行可能（明示的同期ポイントなし）
  - 要求ベース同期（クォンタム終了前にスケジューラへ制御を譲渡） ⇒ タイミング精度向上
- タイム・クォンタム・メカニズム
  - 正確なシステム同期を保証するためのものではない。
  - スケジューラの仕組みに基づいたシミュレーション環境で、複数イニシエータが動作する仕組みを提供。
  - システム・レベルでの明示的同期スキームを設計するための仕組みではない。

3.3.4 アプロキシメイトリー・タイムド・コーディング・スタイル

- インタフェース：ノンブロッキング・トランスポート・インタフェース
  - タイミング・アノテーション
  - 複数フェーズ
  - トランザクションのライフタイム中のタイミング・ポイント

- ユースケース：アーキテクチャ探索、性能解析
- トランザクションのフェーズ
  - 4つのタイミング・ポイント
    - リクエストの開始と終了、レスポンスの開始と終了
  - 特定プロトコルでは、タイミング・ポイントの追加が必要なケースあり ⇒ 汎用ペイロードとの相互利用性が失われる。
  - 2つのタイミング・ポイント（トランザクションの開始と終了）も可能だが、ルーズリー・タイムド・モデルでは、ブロッキング・トランスポート・インタフェースの使用が望ましい。
- アプロキシメイトリー・タイムド・モデルの特徴
  - テンポラル・デカップリングを使わない（要求されるタイミング精度のため）。
  - プロセスの情報交換に2種類のディレイをアノテート。
    - ターゲットのレイテンシ。
    - ターゲットの開始間隔（アクセプト・ディレイ）
    - wait(delay)もしくはnotify(delay)で実装。

3.3.5 ルーズリー・タイムドとアプロキシメイトリー・タイムド・コーディング・スタイルの特徴

- コーディング・スタイルはタイミング・ポイントとテンポラル・デカップリングで特徴付けられる。

	タイミング・ポイント	テンポラル・デカップリング（タイミング制御）
ルーズリー・タイムド	2つ （トランザクション開始と終了）	使用。 プロセスは先行実行した時間情報を持つ。 明示的同期ポイント、クォンタムを使いきったら制御を譲渡。
アプロキシメイトリー・タイムド	4つ （基本プロトコル）	SystemCスケジューリング・セマンティクスに従う。 プロセス間の情報交換時にディレイとアノテート。 wait(delay)、notify(delay)で実装。
アンタイムド	なし	明示的に予め決定されている同期ポイントで制御を譲渡。

3.3.6 ルーズリー・タイムドとアプロキシメイトリー・タイムド・コーディング・スタイル間の切換え

- シミュレーション中にモデルのルーズリー・タイムド・コーディング・スタイルとアプロキシメイトリー・タイムド・コーディング・スタイル間のスイッチが可能。
- 利用例
  - ルーズリー・タイムド・レベルでリセットとブートを高速に実行し、興味のあるステージに到達したら、より詳細な解析をするためアプロキシメイトリー・タイムド・モデリングへスイッチする。

3.3.7 サイクル精度モデリング

- 現時点ではTLM-2の範囲外
  - TLM-1でモデリング可能。
  - 標準化は未解決の課題（将来のOSCI標準ではカバーされるかもしれない）
- 原理的には、アプロキシメイトリー・タイムド・コーディング・スタイルに含まれるように拡張可能と思われる。
  - 適切なフェーズとルールの定義により。
  - TLM-2.0リリースはこれに十分な機構が含まれるが、詳細はまだ詰められていない。

3.3.8 ブロッキングとノンブロッキング・トランスポート・インタフェースの対比

- 異なるタイミング・レベルをサポートする別個のインタフェース

ブロッキング	・トランザクションの開始と終了をモデリング可能。 ・1 回の関数コールでトランザクションを完了。
ノンブロッキング	・トランザクションを複数のタイミング・ポイントへ分割。 ・1 回のトランザクションを完了するのに複数の関数コールが必要。

- 相互利用性：1 つのインタフェースにまとまっている。
  - コーディング・スタイルに従って、ブロッキング、ノンブロッキングのいずれか、もしくは、両方を使用。
  - TLM-2 トランスポート・インタフェースを提供するモデルは、相互利用性を最大にするため、ブロッキングとノンブロッキングの両方を提供すること（実装はかならずしも必要ない）。
- 便利ソケット
  - ブロッキング → ノンブロッキング、ノンブロッキング → ブロッキングのトランスポート・コール変換を提供。
  - 変換にコストがかかる（特に、ノンブロッキング → ブロッキング変換）
    - アプロキシメイトリー・タイムド・モデルがシミュレーション時間を支配しているので、このコスト・オーバーヘッドはおそらく緩和される。
- トランスポート・コールの混在と順序のルール
  - イニシエータは動的にブロッキング、ノンブロッキングのどちらをコールするか選べる。
  - 本標準には、同じターゲットへのブロッキング/ノンブロッキング・トランスポート・コールの混在と順序のルールが含まれる。
- 各トランスポート・インタフェースの強み

ブロッキング	・コーディング・スタイルが単純（1 回の関数コールでトランザクションが完了）。 ・テンポラル・デカップリングを有効に利用可能。
ノンブロッキング	・複数タイミング・ポイントのサポート。 （テンポラル・デカップリングも可能だが、・複数タイミング・ポイントのサポートがテンポラル・デカップリングの利点を無効化。）

3.3.9 ユースケースとコーディング・スタイル

ユースケース	コーディング・スタイル
ソフトウェア・アプリケーション開発	ルーズリー・タイムド
ソフトウェア・パフォーマンス解析	ルーズリー・タイムド
ハードウェア・アーキテクチャ解析	ルーズリー・タイムド アプロキシメイトリー・タイムド
ハードウェア・パフォーマンス検証	アプロキシメイトリー・タイムド サイクル精度
ハードウェア機能検証	アンタイムド（検証環境） ルーズリー・タイムド アプロキシメイトリー・タイムド

3.4 イニシエータ、ターゲット、ソケット、ブリッジ

- コンポーネントの種類

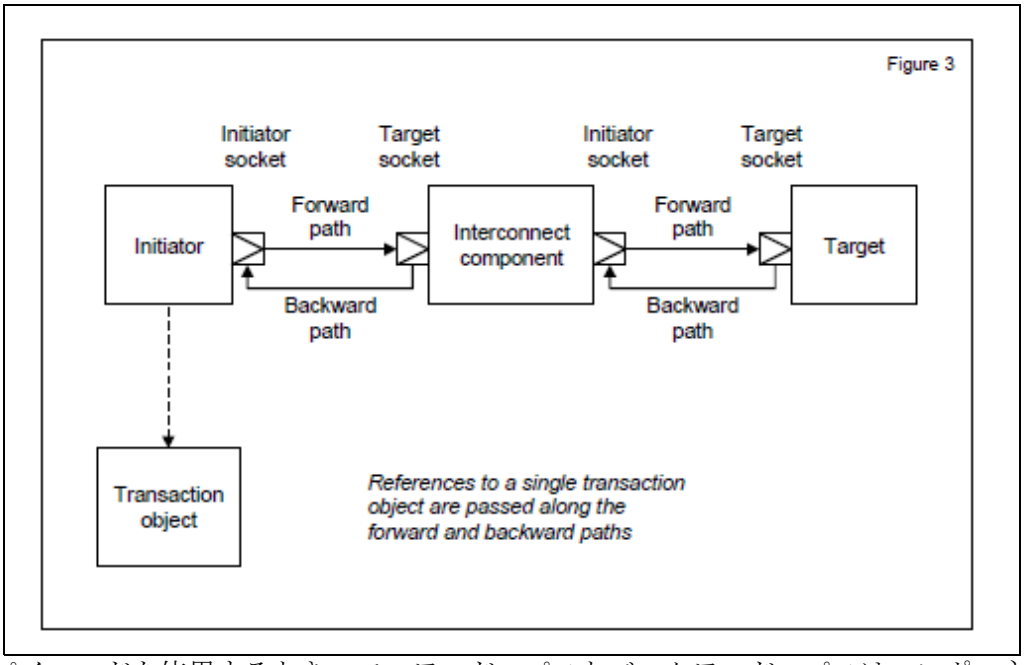
イニシエータ	トランザクションを開始 ・トランザクション・オブジェクトを生成。 ・コア・インタフェースの 1 つのメソッドをコールしてトランザクション・オブジェクトを渡す。
ターゲット	トランザクションの最終目的地 ・ライト・トランザクションでは、イニシエータ（プロセッサ）がターゲット（メモリ）にデータを書き込む。 ・リード・トランザクションでは、イニシエータがターゲットからデータを読む。
インターコネクト・コンポーネント	イニシエータもしくはターゲットとして振舞う ・例：アービタ、ラウター

- トランザクションのライフタイム

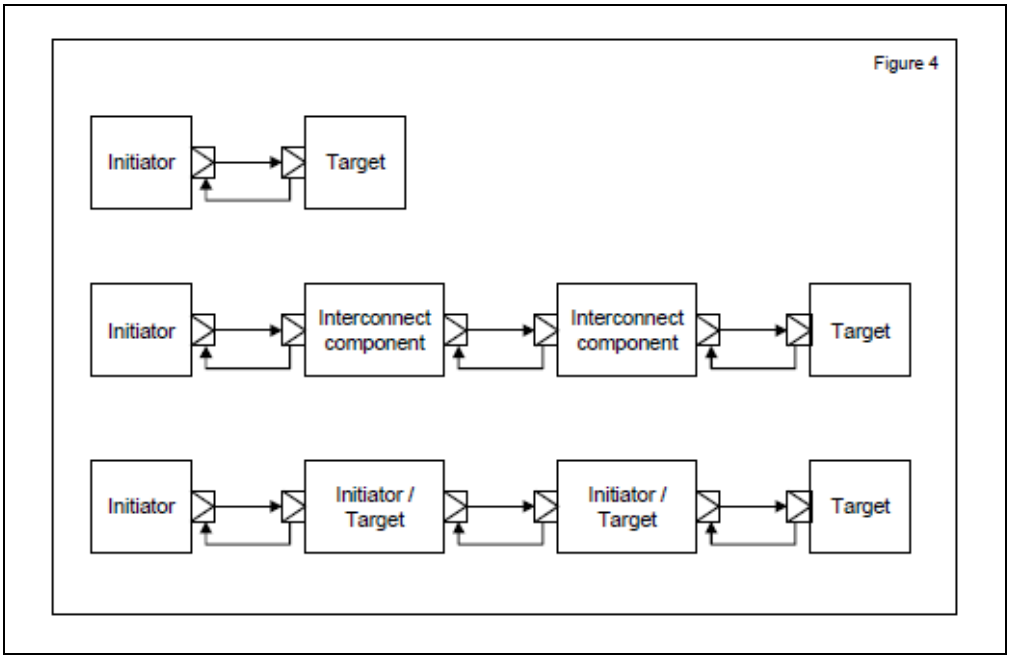
①イニシエータがトランザクション・オブジェクトを生成。

②フォワード・パス
・アービタのようなインターコネクト・コンポーネントが実装するトランスポート・インタフェース・メソッドに渡される。
・インターコネクト・コンポーネントで、さらにトランスポート・メソッド（ラウターのような2番目のインターコネクト・コンポーネントで実装）に渡される。
・2番目のインターコネクト・コンポーネントで、さらにトランスポート・メソッド（メモリにようなターゲットで実装）に渡される。
③ターゲットでトランザクションが処理される。
④イニシエータへトランザクション・オブジェクトを返す（ノンブロッキング・トランスポート・メソッドの返り値により決まる）。
(1)メソッド・コールの返り値 ＝ リターン・パス
(2)逆方向のメソッド・コール ＝ バックワード・パス

- フォワード・パスとバックワード・パス



- 汎用ペイロードを使用するとき、フォワード・パスとバックワード・パスはコンポーネントとソケットの同じ組（もしくは逆向き）を通る。



- ソケット
– フォワード・パスとバックワード・パスをサポートするため、ポートとエクスポートが要求

される。

ソケット	インタフェース・メソッド・コール	
	フォワード・パス	バックワード・パス
イニシエータ	ポート	エクスポート
ターゲット	エクスポート	ポート

- イニシエータは、少なくとも 1 つのイニシエータ・ソケットを持つ。
- ターゲットは、少なくとも 1 つのターゲット・ソケットを持つ。
- 異なるトランザクション・タイプを転送するコンポーネントはいくつかのソケットを持つ。
  - イニシエータとターゲットの役割。
  - TLM-2 トランザクション間のブリッジ。

- バス・ブリッジのモデリング方法

(1)インターコネクト・コンポーネント
- 1 つのトランザクション・オブジェクトのポインタを渡す。 ⇒ シミュレーション高速化に適する。

(2)TLM-2 トランザクションのブリッジ
- トランザクションをコピー。
- 別のアトリビュートを付加できる（より柔軟)。

- TLM-2 ソケットの使用を推薦
  - 相互利用性を最大化
  - 便利
  - 一貫性を持つコーディング・スタイル

3.5 DMI とデバッグ・トランスポート・インタフェース

- ターゲットの持つメモリへの直接アクセス、デバッグ・アクセスを提供するインタフェース。
- インターコネクト・コンポーネントを通る通常のパスをバイパス。

インタフェース	目的	提供するパス
DMI	ルーズリー・タイムド・シミュレーションでの通常のメモリ・トランザクションの高速化。	・フォワード・パス （イニシエータ→ターゲット） ・バックワード・パス （ターゲット→イニシエータ）
デバッグ・トランスポート・インタフェース	通常のトランザクションに付随するディレイ、副作用を引き起こさないデバッグ・アクセスを提供。	・フォワード・パス

3.6 統合インタフェースとソケット

ソケット	提供するインタフェース
イニシエータ ターゲット	・ブロッキング・トランスポート・インタフェース ・ノンブロッキング・トランスポート・インタフェース ・DMI ・デバッグ・トランスポート・インタフェース

- 4 つのインタフェースと汎用ペイロード
- ⇒ TLM-2 標準を使ったモデルの相互利用性を保証
- ターゲット・ソケット
  - 4 つのインタフェースを提供 ⇒ 事実上、すべてを実装
    - ブロッキング⇄ノンブロッキング変換の用意があれば、両方実装する必要なし。
    - モデルのスピードと精度の要求に応じてどちらかを実装してもよい。
  - イニシエータは、スピードと精度の要求に応じて、コア・インタフェースのどのメソッドをコールしてもよい。
- コーディング・スタイルは適切なインタフェースの選択のガイドとなる
- 典型例

コーディング・スタイル （イニシエータ）	インタフェース

ルーズリー・タイムド	・ブロッキング・トランスポート ・DMI ・デバッグ
アプロキシメイトリー・タイムド	・ノンブロッキング・トランスポート ・デバッグ

3.7 ネームスペース

namespace	クラス
tlm	メモリ・マップド・バスのモデリング用の相互利用インタフェースのクラス
tlm_utils	ユーティリティ・クラス （相互利用性には必ずしも必要ではないが、TLM-2 標準の一部）

3.8 ヘッダー・ファイルとバージョン

- アプリケーションは、以下のヘッダー・ファイルをインクルードすること。
  - include/tlm 下の tlm.h
  - include/tlm/tlm_utils 下のヘッダー・ファイル（ユーティリティ・クラスを使用する場合）
- 最新版の OSCI シミュレータを使用する場合は、を SystemC ヘッダー・ファイルをインクルードする前に、SC_INCLUDE_DYNAMIC_PROCESSES マクロを定義すること。
- include/tlm/tlm_h/tlm_version.h はマクロと OSCI TLM-2 ソースコードのバージョン・ナンバーの定数を含んでいる。アプリケーションはこれらを使ってもよい。
  - tlm_release メソッドは､sc_core::sc_release メソッドが返すのと同じフォーマットの文字列を返す。

4. TLM-2 コア・インタフェース

TLM-1 からのコア・インタフェースに加えて、TLM-2 ではブロッキングとノンブロッキングのトランスポート・インタフェース、ダイレクト・メモリ・インタフェース(DMI)、およびデバッグ用トランスポート・インタフェースが追加される。

4.1 トランスポート・インタフェース

トランスポート・インタフェースはイニシエータとターゲット、そしてインターコネクト・コンポーネントの間のトランザクションを転送するのに使用される主要なインタフェースである。ブロッキング及びノンブロッキングのトランスポート・インタフェースの両方ともタイミング・アノテーションとテンポラル・デカップリングをサポートするが、ノンブロッキング・トランスポート･インタフェースだけがトランザクションのライフタイム中の複数フェーズをサポートする。ブロッキング・トランスポートは明示的なフェーズ引数を持たず、ブロッキング・トランスポートとノンブロッキング・トランスポート・インタフェースのフェーズの間のどんな関連でも純粋に抽象的である。ノンブロッキング・トランスポートのメソッドだけが、リターン・パスが使用されたかどうかを示す値を返す。

トランスポート・インタフェースと汎用ペイロードは、メモリ・マップド・バスの抽象モデリングで高速で使われる為に設計された。トランスポート・インタフェースは、それらが汎用ペイロードから単独に使用される事を許可するトランザクション･タイプ上にテンプレート化されるが、しかし相互利用性の利益の多くが失われるだろう。

トランザクション・オブジェクトのメモリ管理、トランザクションの順序、及び許可された関数呼び出しのシーケンスの規則は、トランスポート・インタフェースのテンプレート引数として渡された特定のトランザクション・タイプに依存しており、同様にソケットのテンプレート引数として渡されたプロトコル・タイプ・クラスに依存している。(もしソケットが使用されているならば)

4.1.1 ブロッキング・トランスポート・インタフェース

4.1.1.1 イントロダクション

新しい TLM-2 のブロッキング・トランスポート・インタフェースは、ルーズリー・タイムド・コーディング・スタイルをサポートする事を意図している。ブロッキング・トランスポート・インタフェースは、イニシエータが 1 回の関数呼び出しの経過の間、ターゲットと一緒にトランザクションを完了することをのぞむ所、トランザクションの開始と終了を記録する唯一の重要なタイミング・ポイントにおいて適切である。

ブロッキング・トランスポート・インタフェースは、イニシエータからターゲットまではフォワード・パスを使用するだけである。

TLM-2 のブロッキング・トランスポート・インタフェースは、TLM-2 の規格の一部としてまだ TLM-1 からのトランスポート・インタフェースとの計画的な類似性を持っているが、TLM-1 のトランスポート・インタフェースと TLM-2 のブロッキング・トランスポート・インタフェースは同一ではない。特に、新しい b_transport メソッドは、non-const 参照渡しによる単一のトランザクション引数とタイミングをアノテートする為の第 2 引数を持っているのに対し、TLM-1 の transport メソッドは、タイミング・アノテーションを持っていない単一の const 参照の要求引数を取り、応答の値を返す。TLM-1 は、値が渡される（もしくは const の参照による）分離された要求と応答オブジェクトを仮定するのに対し、TLM-2 ではブロッキング・インタフェースを使用するかノンブロッキング・インタフェースを使用するかに関わらず、参照渡しによる単一のトランザクション・オブジェクトを仮定する。
b_transport メソッドはタイミング・アノテーション引数を持っている。この単一の引数は、現在のシミュレーション時間に相対的なトランザクションの開始及び終了の各々の時間を示すために、b_transport の呼び出しと b_transport からの応答の両方で使用される。

4.1.1.2 クラス定義

```
namespace tlm {
  template <typename TRANS = tlm_generic_payload>
  class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual void b_transport(TRANS& trans, sc_core::sc_time& t) = 0;
  };
} // namespace tlm
```

4.1.1.3 TRANS テンプレート引数

このコア・インタフェースがあらゆるタイプのトランザクションを転送するのに使用されてもよい事が意図されている。特別なトランザクション・タイプである tlm_generic_payload は、トランザクション属性の正確な詳細があまり重要でない場合のモデル間の相互利用性の簡便さの為に提供される。
最大限の相互利用性の為には、アプリケーションは基本プロトコルと一緒に標準のトランザクション・タイプである tlm_generic_payload を使うべきである。特定のプロトコルをモデル化するために、アプリケーションはそれら自身のトランザクション・タイプを代えることができる。異なるトランザクション・タイプで特殊化されたインタフェースを使用するソケットは、バインド出来ない。そして、コンパイル時チェックが提供されるが、相互利用性を制限する。

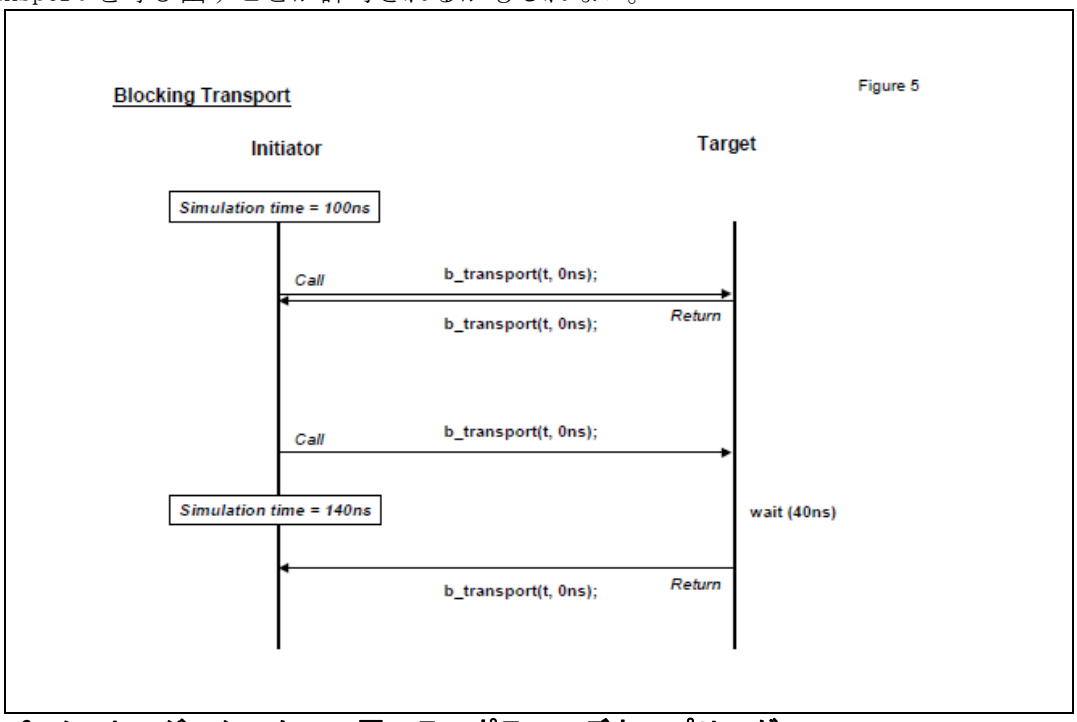
4.1.1.4 規則

a) b_transport メソッドは wait を直接もしくは間接的に呼び出すことができる。
b) b_transport メソッドはメソッド・プロセスからは呼び出されないものとする。
c) イニシエータは、１つの呼び出しから次の呼び出し、トランスポート・インタフェース、DMI、及びデバッグ・トランスポート・インタフェースに渡る呼び出しでトランザクション・オブジェクトを再利用することができる。
d) b_transport の呼び出しは、トランザクションの最初のタイミング・ポイントである事を示すものとする。b_transport からの応答は、トランザクションの最終的なタイミング・ポイントである事を示すものとする。
e) タイミング・アノテーション引数は、関数呼び出し及び応答が実行されるシミュレーション時間（sc_time_stamp()で返された値）からのオフセットのタイミング・ポイントである事を示す。
f) 呼び出し先関数は、トランザクション・クラス TRANS によって課せられたどんな制約にも従うようにトランザクション・オブジェクトを変更もしくはアップデートしてもかまわない。
g) トランザクション・オブジェクトにタイミング情報を含まないことを推奨する。タイミングは b_transport の sc_time 引数を使ってアノテートするべきである。
h) b_transport が nb_transport_fw を呼び出す事を許可されるかどうかは、プロトコルと連携されたルールに依存する。基本プロトコルにおいて、自動的にこの変換をする事が出来る便利ソケットの simple_target_socket が提供される。5.3.2 章「Simple sockets」を参照の事。

4.1.1.5 メッセージ・シーケンス図－ブロッキング・トランスポート

ブロッキング・トランスポート・メソッドは、すぐに応答するか（すなわち、現在の SystemC 評価フェーズで)、スケジューラに制御を譲り、シミュレーション時間より遅れたポイントでイニシエータに戻るかもしれない。イニシエータのスレッドはブロックされるかもしれないが、イニシエータの他のスレッドは、プロトコルに依存して、最初の関数呼び出しが応答する前に

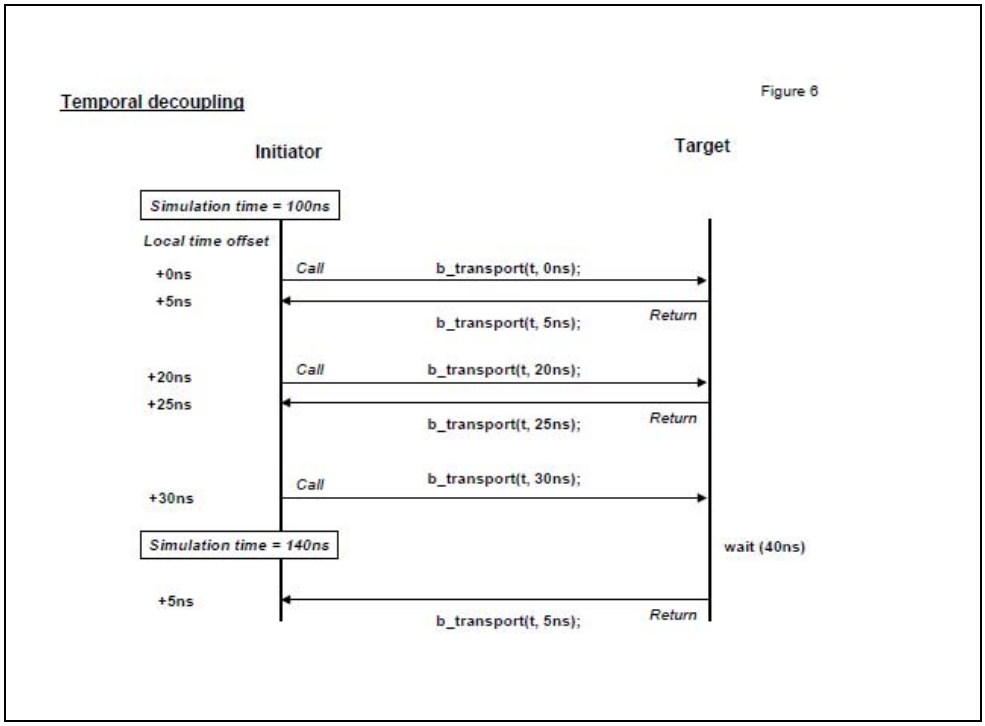
b_transport を呼び出すことが許可されるかもしれない。



4.1.1.6 メッセージ・シーケンス図ーテンポラル・デカップリング

テンポラル・デカップリングされたイニシエータは、現在のシミュレーション時間に先立って抽象的なローカル時間で走るかもしれないが、その場合以下に示すように、時間引数の為の非ゼロの値を b_transport に渡すべきである。イニシエータとターゲットは、時間引数の値を増加させる事で、各々のローカル時間のオフセットを進めるであろう。呼び出し先から返された時間引数を加えた現在のシミュレーション時間は、各々のトランザクションが完了するまでの抽象的な時間を与えるが、シミュレーション時間自身はイニシエータのスレッドが切り替わるまで進むことが出来ない。

b_transport の本体は、それ自身で wait を呼び出すかもしれないが、その場合はローカル時間のオフセットはゼロにリセットすべきである。以下のダイアグラムでは、イニシエータからの最後の応答は 140ns のシミュレーション時間後に起こるが、5ns の遅延がアノテートされたため、有効なローカル時間としては 145ns となる。

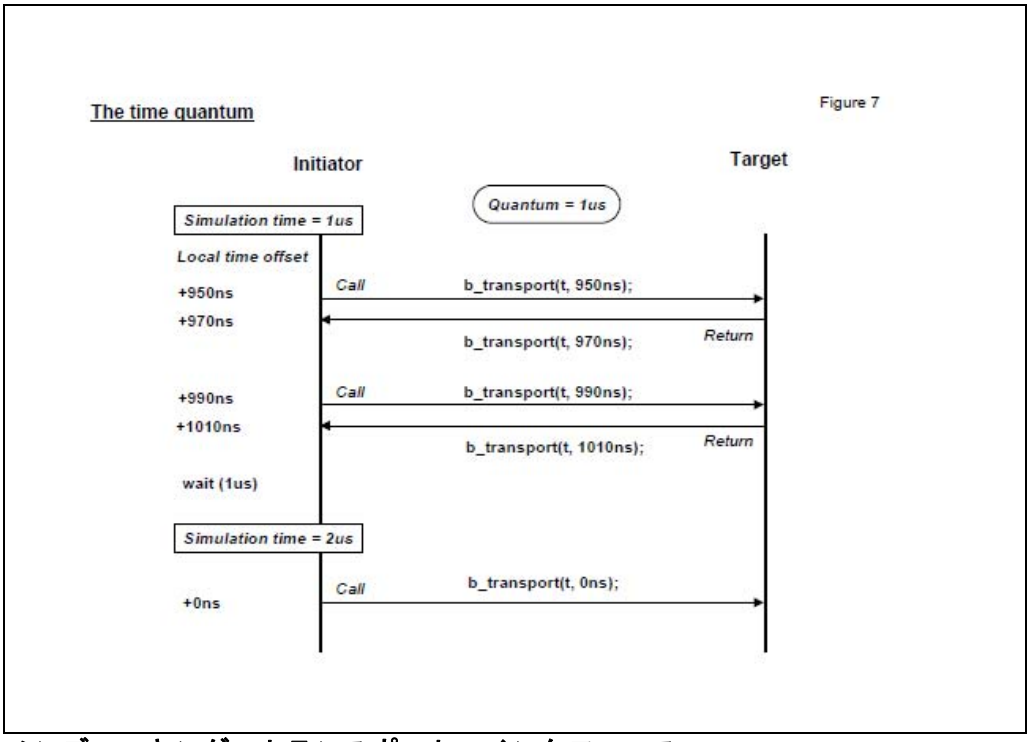


4.1.1.7 メッセージ・シーケンス図ータイム・クォンタム

テンポラル・デカップリングされたイニシエータは、タイム・クォンタムが超えるまでローカル時間で進み続けるであろう。その時、イニシエータは次のクォンタムの境界まで実行を停止し同期せざるをえないだろう。これはモデルの他のイニシエータが動作して、追いつくのを許す。そして、それは効果的に、イニシエータが順番に実行することを意味する、各々がいつそれ自身のローカル時間の経過を追うことによって制御を返すべきかを決定する役割を果たす。オリジナルのイニシエータは、シミュレーション時間が次のクォンタムに達した後に再び走るだけであるべきである。

ルーズリー・タイムド・コーディング・スタイル中の遅延の第一の目的は、各々のイニシエータがいつ制御を返すべきかを決定する事を考慮する為のものである。もし正確に機能するためのタイミングの詳細を当てにしないのであれば、それが最良である。

各クォンタム中、所定のイニシエータによって生成されたトランザクションは厳密な連続した順序で起こるが、シミュレーション時間は進まない。ローカル時間は SystemC のスケジューラによって追跡されない。



4.1.2 ノンブロッキング・トランスポート・インタフェース

4.1.2.1 イントロダクション

ノンブロッキング・インタフェースは、アプロキシメイトリー・タイムドによるコーディング・スタイルのサポートを意図しており、各トランザクションが通過するイニシエータとターゲット間の相関関係の詳細な流れをモデリングするのに適している。言い換えると、各フェーズ・トランザクションが明示的にタイミング・ポイントをいずれかで検出し、複数フェーズにトランザクションを分解する事である。

タイミング・ポイント数を２つに制限する事によってルーズリー・タイムドにも使用する事は出来るが、一般的には推奨されない。ルーズリー・タイムドにおいてはブロッキング・インタフェースの容易性が好まれる。ノンブロッキング・インタフェースはブロッキング・トランスポート・インタフェースの使用が難しいパイプライン化されたトランザクション・モデリングに適している。

ノンブロッキング・インタフェースはイニシエータからターゲット（フォワード・パス）とターゲットからイニシエータ（バックワード・パス）の両方で使用する。それら二方向で使用される tlm_fw_nonblocking_transport_if と tlm_bw_nonblocking_transport_if の２つの異なるインタフェースが用意されている。

ノンブロッキング・インタフェースは、ノンブロッキング・インタフェース・メソッドがトランザクション・オブジェクトとタイミング・アノテーションの非コンスト参照を渡す点だけが新しいブロッキング・インタフェースの引数渡しのメカニズムと良く似ている。ノンブロッキング・

インタフェース・メソッドはまた、トランザクションの状態を示す為のフェーズを渡し、関数からの戻りがフェーズ・トランザクションを表すかを示す為に列挙型の値を返す。
ブロッキングとノンブロッキングの両方のトランザクションがタイミング・アノテーションをサポートするが、ノンブロッキングだけがトランザクションのそのライフタイムの間、複数フェーズをサポートしている。ブロッキング、ノンブロッキング・トランザクション・インタフェースと汎用ペイロードは、高速で抽象モデリングされるメモリ・マップド・バスと共に使用する事を前提に設計されている。しかし、トランスポート・インタフェースは特定のプロトコルをモデルする為の汎用ペイロードから分けて使用出来る。トランザクションの型とフェーズの型はいずれもノンブロッキング・トランスポート・インタフェースのテンプレート引数である。

4.1.2.2 クラス定義

```
namespace tlm {
  enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };
  template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
  class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual tlm_sync_enum nb_transport_fw(TRANS& trans, PHASE& phase,
      sc_core::sc_time& t) = 0;
  };
  template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
  class tlm_bw_nonblocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual tlm_sync_enum nb_transport_bw(TRANS& trans, PHASE& phase,
      sc_core::sc_time& t) = 0;
  };
} // namespace tlm
```

4.1.2.3 TRANS と PHASE テンプレート引数

ノンブロッキング・トランスポート・インタフェースは、任意のフェーズ数やタイミング・ポイントと共に、あらゆる型のトランスポート・トランザクションで使用しても良い事が意図されている。詳細なトランザクションのアトリビュートが重要にならないようなモデル間の相互利用性向上向けに、特別なトランザクション型である tlm_generic_payload 型が提供され、また基本プロトコルで使用される特別な型である tlm_phase 型も提供されている。相互利用性を最大限に活用するには基本プロトコルとしてデフォルト・トランザクション型の tlm_generic_payload とデフォルト・フェーズ型の tlm_phase を使用すべきである。

4.1.2.4 nb_transport_fw と nb_transport_bw 呼び出し

a) フォワード・パス上で使用する nb_transport_fw と、バックワード・パス上で使用する nb_transport_bw の、二つのノンブロッキング・トランスポート・メソッドが用意されている。名前と呼び出しの方向は異なるが、これら二つのメソッドのセマンティクスはよく似ている。本書ではイタリック体で書かれている nb_transport は、二つのメソッドを区別する必要が無い状況において、両方のメソッドを説明する際に使用される。
b) 基本プロトコルの場合、フォワードとバックワードのパスは逆並びで同じコンポーネントとソケットのシーケンスを通じて渡すべきである。
c) nb_transport_fw はフォワード・パスのみで呼ばれるものとし、nb_transport_bw はバックワード・パスのみで呼ばれるものとする。
d) フォワード・パス上の nb_transport_fw 呼び出しは、バックワード・パス上の nb_transport_bw を直接的または間接的に呼び出さないものとする。また逆も同様である。
e) nb_transport メソッドは直接、または間接的に wait を呼ばないものとする。
f) nb_transport メソッドはスレッド・プロセスまたはメソッド・プロセスから呼ばれる。
g) nb_transport では b_transport 呼び出す事が許可されていない。一つの解決策は、オリジナルの nb_transport_fw メソッドによって生成または通知された別のスレッド・プロセスから b_transport を呼び出す事である。基本プロトコルでは便利な simple_target_socket が提供されており、この変換を自動的に行わせる事が出来る。詳細は 5.3.2 の『シンプル・ソケット』を参照。
h) ノンブロッキング・トランスポート・インタフェースは、明確にパイプライン化されたトランザクションのサポートを意図している。言い換えれば、最初のトランザクションの完了を待つ必要はなく、同じプロセスから nb_transport_fw が連続して呼ばれ、それぞれ個別のトランザクシ

ョンを開始する事が出来る。
i) トランザクションの最終タイミング・ポイントは、フォワード・パスまたはバックワード・パスのいずれかにおける nb_transport を呼び出しか、そこから戻る事によって検出される。

4.1.2.5 trans 引数

a) 与えられたトランザクション・オブジェクトのライフタイムは、nb_transport への一連の呼び出しがイニシエータ、インターコネクト・コンポーネント、ターゲット間で一つのトランザクション・オブジェクトをフォワードとバックワードへ渡すことができるよう、nb_transport からの戻りを超えて拡張しても良い。
b) イニシエータは、一つの呼び出しから次の呼び出し、トランスポート・インタフェース、DMI、およびデバッグ・トランスポート・インタフェースに渡る呼び出しでトランザクション・オブジェクトを再利用してもよい。
c) トランザクション・オブジェクトのライフタイムは nb_transport の呼び出し毎に拡張できる為、呼び出し元関数と呼び出し先関数のいずれかは、トランザクション・クラスの TRANS によって課せられたなんらかの制約条件の下に、そのトランザクション・オブジェクトの変更または更新を行っても良い。例えば、汎用ペイロードでは、そのターゲットはリードコマンドの場合にそのトランザクション・オブジェクトのデータ配列をアップデートしても良いが、コマンド・フィールドを更新すべきではない。

4.1.2.6 phase 引数

a) 各 nb_transport 呼び出しはフェーズ・オブジェクトの参照を渡す。あるフェーズから他のフェーズへの変化はタイミング・ポイントを検出する。基本プロトコルの場合、同一のフェーズにおける nb_transport の連続的な呼び出しは許可されない。sc_time 引数を使用したタイミング・アノテーションは、もしそれが一つならフェーズ変化を遅らせるものとする。
b) トランザクションのアトリビュートは各フェーズにおいて基本的にステーブルであり、フェーズ変化が検出されたタイミング・ポイントの時のみ変化する。フェーズ途中で起こっているトランザクション・オブジェクトのあらゆる変化は、次のタイミング・ポイント時に他のコンポーネントに見えるようになるだけなはずである。
c) フェーズ引数は参照渡しされる。呼び出し元関数または呼び出し先関数のいずれかがそのフェーズを変更してもよい。
d) トランザクションのあらゆる状態変更は、一つの呼び出しから次の呼び出しのフェーズ引数の値を比較する事によって呼び出し元関数か、呼び出し先関数のいずれかでその変更が検出出来るようにフェーズ引数の変更を伴わせるべきである。
e) フェーズ引数の値は、呼び出し元関数と呼び出し先関数間における通信用プロトコル・ステートマシンの現在の状態を表す。単一トランザクション・オブジェクトは二つ以上のコンポーネント（イニシエータ、インターコネクト、ターゲット）間のいずれかに渡され、各呼び出し元関数／呼び出し先関数の接続は、別々のプロトコル・ステートマシンを（少なくとも概念的には）要求する。
f) トランザクション・オブジェクトは一つの nb_transport 呼び出しを超えるライフタイムとスコープを持つのに対して、フェーズ・オブジェクトは一般的に呼び出し元関数においてローカルである。トランザクションを与える為の各 nb_transport 呼び出しは異なるのフェーズ・オブジェクトを持っても良い。異なる呼び出し元関数／呼び出し先関数上に対応するフェーズ・トランザクションは、シミュレーション時間内の異なるポイントで発生しても良い。
g) デフォルト・フェーズ型の tlm_phase は基本プロトコル向けである。その他プロトコルについては、拡張型の tlm_phase、またはそれ自身のフェーズタイプ（相互利用性低下への対応と共に）を代入出来る。7.1 章　フェーズを参照。

4.1.2.7 tlm_sync_enum の戻り値

a) 同期の概念は方々で示されている。同期する事とは、いくつかプロセスが動作している中で SystemC スケジューラに制御を譲渡する事であるが、テンポラル・デカップリング用に追加的な意味も含まれる。 これは別の場所でより多くの議論がされている。8.1.3 章　テンポラル・デカップリングを使用するプロセス向けの一般的なルールを参照。
b) 原則として同期は制御の譲渡によって実行可能であるが(スレッド・プロセスの場合 wait を呼ぶか、またはメソッド・プロセスでカーネルに戻る)、テンポラル・デカップリングされたイニシエータは、tlm_quantum_keeper クラスの sync メソッドを呼ぶことによって同期する。一般的に他の SystemC プロセスが実行される事を許す一方で、時々同期するイニシエータにはそれが必要

である。
c) 下記のルールはフォワードとバックワード・パスの両方に適用する。
d) nb_transport の戻り値の意味は固定されており、トランザクション型またはフェーズ型によって変わらない。したがって、以下の規則は tlm_phase と tlm_generic_payload の制限を受けないが、しかしノンブロッキング・トランスポート・インタフェースをパラメタライズする際に使用される、あらゆるトランザクションとフェーズ型に適用する。
e) TLM_ACCEPTED： 呼び出し先関数は呼び出しを許可される。呼び出し先関数は呼び出し中のトランザクション・オブジェクト、フェーズ、または時間引数の状態を変更しないものとする。言い換えると、TLM_ACCEPTED はリターン・パスが使用されていない事を示す。フェーズ・トランザクションを無視している呼び出し先関数は TLM_ACCEPTED を返すべきである。呼び出し元関数は、その呼び出しに続く nb_transport 引数の値は変化しない為無視すべきである。
f) TLM_UPDATE： 呼び出し先関数はトランザクション・オブジェクトを更新する。呼び出し先関数は、呼び出し中にフェーズ引数の状態変更、トランザクション・オブジェクトの状態変更、時間引数の値の増加を行える。言い換えると、TLM_UPDATED はリターン・パスが使用されている事を示し、そして呼び出し先関数は、トランザクションと共に連想されるプロトコル・ステートマシンの状態を進める。呼び出し先関数が各々引数を変更する事を実際に行う義務があるかどうかはそのプロトコルに依存する。nb_transport 呼び出しに続き、その呼び出し先関数は、フェーズ、トランザクション、または時間引数を検査し、適切な動作をすべきである。
g) TLM_COMPLETED： 呼び出し先関数はトランザクション・オブジェクトを更新して、そしてそのトランザクションは完了する。呼び出し先関数は、呼び出し中にフェーズ引数の状態変更、トランザクション・オブジェクトの状態変更、時間引数の値の増加を行える。呼び出し先関数は、最終フェーズのトランザクションを TLM_COMPLETED の値を暗黙的に示している為フェーズ引数を更新する義務を持たない。呼び出し元関数は、トランザクションの最終タイムポイントが sc_time_stamp()+t（t は時間引数）時間に起こるよう振舞うべきである。言い換えると、TLM_COMPLETED はリターン・パスが使用されており、そのトランザクションは完了である事を示す。nb_transport 呼び出しに続くその呼び出し先関数は、フェーズ、トランザクション、または時間引数を確認し、適切な動作をすべきである。これ以上フォワードまたはバックワードのいずれかに沿って、このトランザクションに関連して呼び出されるトランスポートはない。この意味上の完了は、必ずしも無事に完了したという意味ではなく、その為トランザクション型に依存して呼び出し元関数はトランザクション・オブジェクト内に埋め込まれた応答状態の検査を必要とする可能性がある。
h) ３つの戻り値のいずれであってもプロトコルによって、nb_transport の呼び出しに続く呼び出し先関数はトランザクション・オブジェクトの応答を生成するか、またはリリースする為に呼び出し先関数を含むコンポーネントを許可する上で制御の譲渡を必要とする可能性がある。

4.1.2.8 tlm_sync_enum のまとめ

tlm_sync_enum	Transaction object	Phase on return	Timing annotation on return
TLM_ACCEPTED	Unmodified	Unchanged	Unchanged
TLM_UPDATED	Updated	Changed	May be increased
TLM_COMPLETED	Updated	Ignored	May be increased

4.1.2.9 メッセージ・シーケンス・チャート ― バックワード・パスを使用しているケース

記のメッセージ・シーケンス・チャートは、nb_transport のシーケンスを呼び出すバリエーションを図示している。nb_transport へ、または nb_transport から渡される引数と戻り値は、return、phase、delay の表記で表される。return は関数呼び出しからの戻り値、phase はフェーズ引数の値、delay は sc_time 引数の値を意味し、 “－”の表記は値が使用されない事を示す。
下記のメッセージ・シーケンス・チャートを例として、BEGIN_REQ や END_REQ 等の基本プロトコルのフェーズを使用する。アプロキシメイトリー・タイムド・コーディング・スタイルと基本プロトコルで、トランザクションはイニシエータとターゲット間を前後２回渡される。他のプロトコルではフェーズ数とその名前は異なる可能性がある。
nb_transport 呼び出しの受け取り側が、その次のトランザクションの状態、または次のフェーズ・トランザクションの遅延を即座に計算出来ない場合、それは TLM_ACCEPTED の値を戻すべきである。呼び出し元関数は、スケジューラへの制御の譲渡し、呼び出し先関数が応答する準備が完了した時に、反対の経路から nb_transport の呼び出しを受ける事を想定すべきである。
トランザクションはパイプラインされても良い。イニシエータは前のトランザクションの最終タ

イミング・ポイントを確認する以前に、ターゲットに対して他のトランザクションを送る為の nb_transport を呼び出す事が出来る。
プロセスはシミュレーション時間を進める事を許可する為の一般的な制御の譲渡をしているのであり、アプロキシメイトリー・タイムド・コーディング・スタイルは、ルーズリー・タイムド・コーディング・スタイルより遅いシミュレーションを実行する事が想定されている。

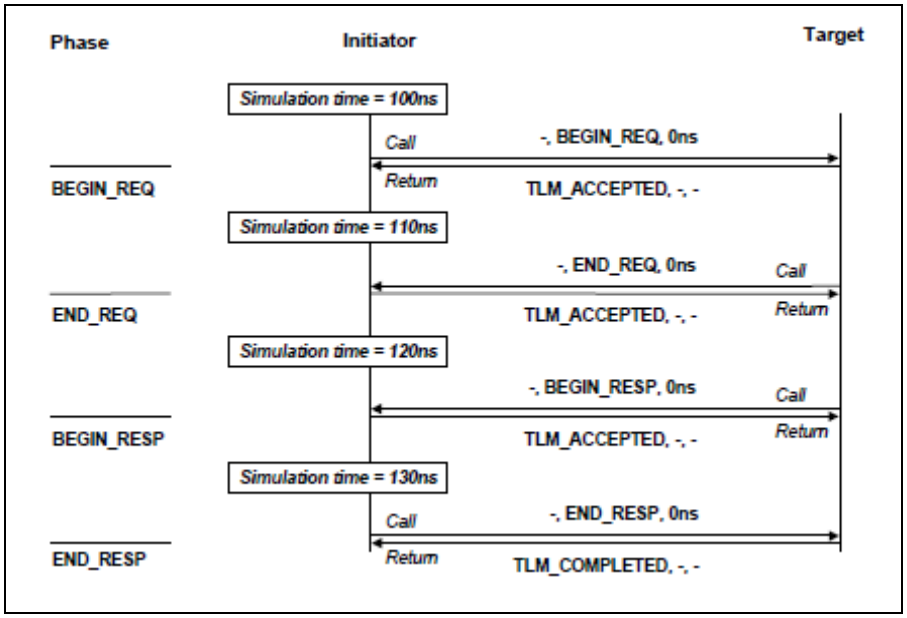


図 8 バックワード・パス使用

4.1.2.10 メッセージ・シーケンス・チャート ― リターン・パスを使用しているケース

nb_transport 呼び出しの受け取り側が、そのトランザクションの次の状態、または次のフェーズ・トランザクションの遅延を即座に計算出来る場合、それは反対の経路を使用するのではなく nb_transport からのリターンにより新しいステートを戻す可能性がある。このタイミング・ポイントがトランザクションの最後を示さない場合に、その戻り値の TLM_UPDATED が提供され、トランザクションの最後を示す場合は、TLM_COMPLETED が提供される。トランザクションの完了時に呼び出し先関数は、それが他のフェーズを先取りして最終フェーズまでジャンプした事を呼び出し元関数に示す為に、どのステージでも TLM_COMPLETED を返すことが出来る。これはイニシエータとターゲットのいずれも適用される。
TLM_UPDATED であるなら、呼び出し先関数は、トランザクションとフェーズの更新とフェーズ・トランザクションに遅延をアノテートすべきである。
TLM_COMPLETED であるなら、トランザクションの最終フェーズが暗黙的に示されるので、フェーズ引数の値は呼び出し元関数によって無視されるべきである。

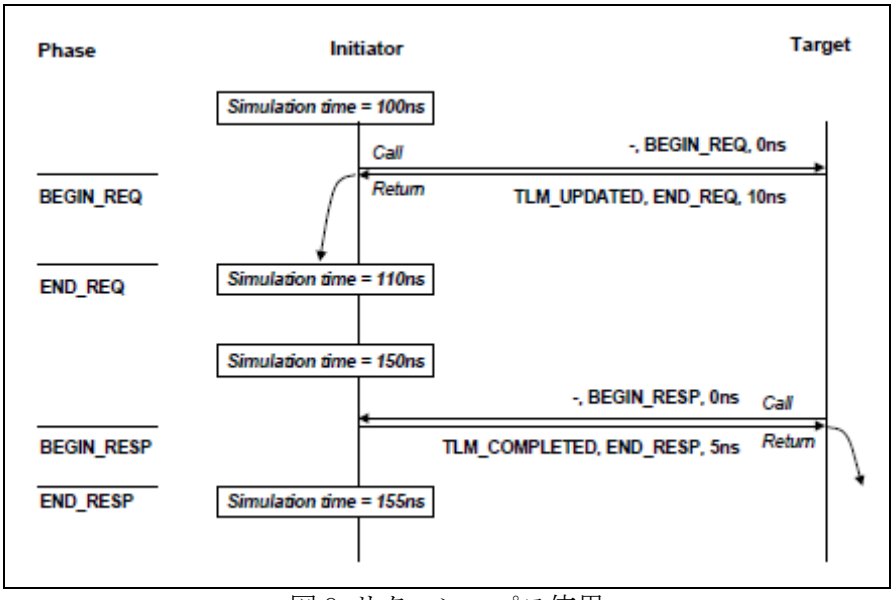


図 9 リターン・パス使用

4.1.2.11 メッセージ・シーケンス・チャート ― 早期完了

プロトコルに依存して、イニシエータやターゲットは早期にトランザクションを完了する為に nb_transport から TLM_COMPLETED を任意のタイミング・ポイントで返しても良い。イニシエータとターゲットのいずれも、このトランザクション・インスタンスの nb_transport 呼び出しを行わなくても良い。イニシエータやターゲットは、このようなトランザクションのショートカットを受け入れられるか否かは、その特定のプロトコルのルールに依存する。

以下のダイアグラムのリターン・パスにおけるタイミング・アノテーションは、与えられた遅延後に最終タイミング・ポイントが起こる事をイニシエータに指示している。

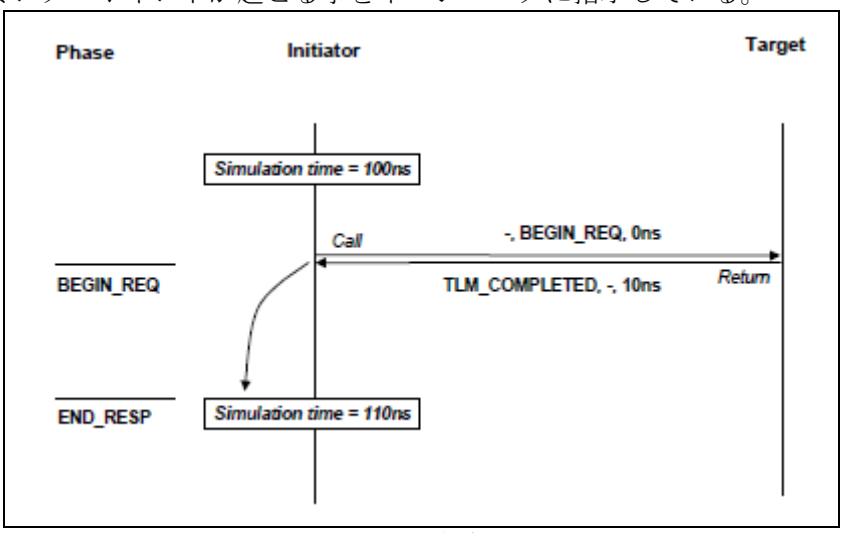


図 10 早期完了

4.1.2.12 メッセージ・シーケンス・チャート ― タイミング・アノテーション

呼び出し元関数は、nb_transport 呼び出しに遅延をアノテートしても良い。これは、呼び出し元関数に対応するフェーズがあたかも与えられた遅延の後に、それを受け取ったかのように処理されるべきである事を示す。アプロキシメイトリー・タイムドな呼び出し先関数は、このシミュレーション時間がアノテートされた時間に到達際、処理する為のペイロード・イベント・キューにトランザクションを入れる事によって、通常この状況をハンドルする。

遅延はフォワードまたはバックワード・パスのいずれかの呼び出しにアノテートさせる事が出来る。

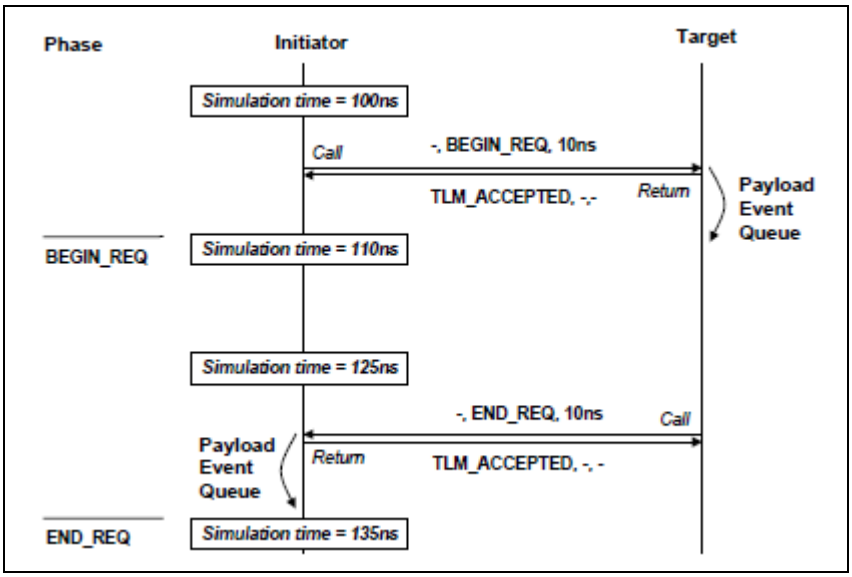


図 11 タイミング・アノテーション

4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーション

タイミング・アノテーションはブロッキングまたはノンブロッキング・トランスポート・インタフェースの共有の機能であり、b_transport、nb_transport_fw、nb_transport_bw メソッドに sc_time 引数を使用して表される。下記の章では、イタリック体の transport は、b_transport、nb_transport_fw、nb_transport_bw の３つのメソッドの意味で用いられる。

4.1.3.1 sc_time 引数

a) トランザクション・オブジェクトは、タイミング情報を含まない事が推奨される。あらゆるタイミング･アノテーションは transport の sc_time 引数を使用してアノテートされるべきである。
b) 時間引数はマイナスにはならず、常にカレント・シミュレーション時間の sc_time_stamp()に連動して表される。
c) 時間引数は transport の呼び出しと transport からの戻りの両方で適用されるものとする。（tlm_sync_enum のルールに従って nb_transport の値を戻す）
d) nb_transport メソッドはそれ自身に時間引数の値を増加する可能性があるが値は減少しない。b_transport メソッドは wait が呼ばれる場合にのみ時間引数の値を減少させる事でシミュレーション時間に同期する。このルールは SystemC シミュレーションにおいて時間が戻らない事と同様である。
e) 以下の説明では、transport 呼び出しにおけるトランザクションの受け取り側は呼び出し先関数であり、そして transport からの戻りにおけるトランザクションの受け取り側は呼び出し元関数である。
f) 受け取り側は、sc_time_stamp()+t（t は時間引数の値）の時間でトランザクションを受け取るように振舞うものとする。言い換えると、受け取り側は対応しているフェーズ・トランザクションをマークするタイミング・ポイントが sc_time_stamp()+t の時間に起こるかのように振舞うものとする。
g) トランザクションが処理される事により時間順番を増やす可能性のある sc_time_stamp()+t 時間の transport を呼び出すシーケンスが与えられる。一般的には呼び出し元、または呼び出し先関数は、タイミング・アノテーションをいかなる特定順番で生成する事を強いられない。アウト・オブ・オーダーのタイミング・アノテーション付きトランザクションをハンドルする為の責任は受け取り側が持つ。
h) タイミング・アノテーションと共にトランザクションを受け取る上で、受け取り側はスピードと精度間におけるトレードオフをモデリングに反映させるいくつかの選択肢を持つ。言い換えると、モデルは速くして精度を落とす（ルーズリー・タイムド）事も出来るし、遅くして精度を上げる事も出来る。ブロッキング・トランスポートはルーズリー・タイムド・モデルが推奨され、ノンブロッキング・トランスポートはアプロキシメイトリー・タイムドが推奨される。その選択はトランスポート・インタフェースによって実施されるのでは無く、プロトコル型クラスや、コーディング・スタイルの部分に記載される。
i) もし受け取り側がタイミング並びに実行順番の正確なモデルを実装する事があるなら、それはトランザクションが相互作用の可能性があるその他の SystemC プロセスに基づいた正しい時間に処理される事を確実にするべきである。SystemC における将来の時間に起きるイベントのスケジュールを適切に行うメカニズムは時間イベント通知である。容易性を考慮し、TLM-2 では SystemC の通常のセマンティクス（8.2 章　ペイロード・イベント・キューを参照）に従った適切なシミュレーション時間に処理する為、トランザクションをキューイング出来るペイロードのイベント・キューを知るユーティリティ・クラスのファミリを提供する。言い換えると、アプロキシメイトリー・タイムド（原文のまま）の受け取り側は、ペイロードのイベント・キューに通常トランザクションを置くべきである。
j) もしタイミングの精度、またはそれらタイミング・アノテーションによって与えられた順番に来ているトランザクションのシーケンスを処理する事が受け取り側に関係がない場合、それは遅延を除いて各トランザクションを即座に処理出来る。またその場合受け取り側は、トランザクションを処理するのに必要な時間をモデルする為の、タイミング・アノテーションの値を増やす事を選択出来る。言い換えると、ルーズリー・タイムドの受け取り側は通常テンポラル・デカップリングを使用出来る。このシナリオは、システム設計が正しいイベントの連続性を実施する為の明確的なメカニズム（上記の TLM-2 インタフェースの上で）が存在する為、アウト･オブ･オーダー実行を許容出来るとみなせる。
k) トランザクションを処理するいかなる場合にも、受け取り側はトランザクションの変更と、同一のフェーズとタイミング・アノテーションを使用している事を除き、更なる transport からの呼び出し、またはリターンと共にトランザクションを渡しても良い。
l) またタイミング・アノテーションについては、テンポラル・デカップリングの観点でも説明される。ゼロでないタイミング・アノテーションは、“タイム・ワープ”が受け取り側に要請されたと見なす事が出来る。受け取り側は、タイム・ワープに入るか、後処理とイールド制御の為にキ

ューにトランザクションを置くか選択する事が出来る。ルーズリー・タイムド・モデルにおいて、タイム・ワープは一般的に有効である。もう一方では、もしターゲットが他の非同期イベントに依存関係を持つ場合、そのターゲットは確実に先のトランザクションの状態を予測する事が出来るまでは、シミュレーション時間の更新を待たなければならない可能性がある。
m) テンポラル・デカップリングの説明については、3.3.2 章　ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリングを参照。
n) クォンタムの説明については、8.1 章　グローバル・クォンタムとクォンタム・キーパーを参照。

4.1.4 TLM-1 からのマイグレーション・パス

旧 TLM-1 と新 TLM-2 インタフェースはどちらも TLM-2 標準の一部である。TLM-1 ブロッキングとノンブロッキング・インタフェースは現在も使用出来る。例えばベンダの何社かは、HDL 設計向けに機能検証環境の構築にこれらインタフェースを使用している。
新旧ブロッキング・トランスポート・インタフェース間の類似性は、TLM-1 インタフェースを使用した古いモデルと、新しい TLM-2 インタフェース間におけるアダプタを構築する作業を容易にする事を意図している。

4.2 ダイレクト・メモリ・インタフェース

4.2.1 イントロダクション

ダイレクト・メモリ･インタフェース（DMI）は、イニシエータがターゲット内のメモリ領域に直接アクセスできる手段を提供する。つまりトランスポート・インタフェースを使わずに直接ポインタを使ってメモリアクセスが可能である。DMI によって、イニシエータからインターコネクト・コンポーネントに接続されたターゲットへの複数の b_transport 関数や nb_transport 関数の呼び出しをバイパスできるため、イニシエータとターゲット間のメモリアクセスを高速化することが可能となる。
ダイレクト・メモリ・インタフェースには２つのインタフェースがあり、ひとつはイニシエータからターゲットへのフォワード・パス呼び出しともうひとつはターゲットからイニシエータへのバックワード・パス呼び出しである。フォワード・パスは、ある指定されたアドレスに対する DMI アクセス（読み込みまたは書き込み）の特定モードを要求するために利用され、DMI 領域の境界を含む tlm_dmi 型の DMI 記述子へのリファレンスを返す。バックワード・パスは、フォワード・パスによって確立されたＤＭＩポインタが無効な場合にターゲットから呼び出される。フォワード・パスとバックワード・パスは、全く通らないか、ひとつまたは複数のインターコネクト・コンポーネントを通ってもよいが、同じソケットを通る対応するトランスポート呼び出しにおいては、フォワードとバックワードは同一の経路を通るべきである。
DMI ポインタは、フォワード・パスにそってトランザクションを通過することによって要求される。デフォルトの DMI トランザクション型は、tlm_generic_payload であり、トランザクション・オブジェクトのコマンドとアドレス・アトリビュートのみ使用する。DMI は、トランスポート・インタフェースの拡張と同じ仕組みである。つまり、DMI 要求には、無視可能拡張があってもよいが、必須拡張は、新しいプロトコル型のクラス定義を必要とする。(tlm_generic_payload の型定義が記載されている 6.2.2 章の新しいプロトコル型のクラスの定義を参照)
DMI 記述子は、イニシエータで消費されるレイテンシ値を返し、その結果ルーズリー・タイムド・モデリングに十分なタイミング精度を確保することができる。
DMI ポインタはデバッグ用途にも使えるが、通常デバッグのトラフィックよりもメモリアクセスのトラフィックの方が支配的であるため、デバッグ用途では、DMI ではなくデバッグ・トランザクション・インタフェースで十分である。DMI ポインタをデバッグ用に使う場合、レイテンシ値は無視すべきである。

4.2.2 クラス定義

```
class tlm_dmi
{
public:
  tlm_dmi() { init(); }
  void init();

  enum dmi_access_e {
    DMI_ACCESS_NONE = 0x00,
    DMI_ACCESS_READ = 0x01,
    DMI_ACCESS_WRITE = 0x02,
```

```
    DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ | DMI_ACCESS_WRITE
  };
  unsigned char* get_dmi_ptr() const;
  sc_dt::uint64 get_start_address() const;
  sc_dt::uint64 get_end_address() const;
  sc_core::sc_time get_read_latency() const;
  sc_core::sc_time get_write_latency() const;
  dmi_access_e get_granted_access() const;
  bool is_none_allowed() const;
  bool is_read_allowed() const;
  bool is_write_allowed() const;
  bool is_read_write_allowed() const;

  void set_dmi_ptr(unsigned char* p);
  void set_start_address(sc_dt::uint64 addr);
  void set_end_address(sc_dt::uint64 addr);
  void set_read_latency(sc_core::sc_time t);
  void set_write_latency(sc_core::sc_time t);
  void set_granted_access(dmi_access_e t);
  void allow_none();
  void allow_read();
  void allow_write();
  void allow_read_write();
};

template <typename TRANS = tlm_generic_payload>
class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
  virtual bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data) = 0;
};
class tlm_bw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) = 0;
};

} // namespace tlm
```

4.2.3 get_direct_mem_ptr メソッド

a) get_direct_mem_ptr メソッドは、ターゲットではなく、イニシエータまたはインターコネクト・コンポーネントからのみ呼び出されるべきである。
b) trans 引数には、イニシエータで生成された DMI トランザクション・オブジェクトへの参照値。
c) dmi_data 引数には、イニシエータで生成された DMI 記述子への参照値。
d) すべてのインターコネクト・コンポーネントでは、get_direct_mem_ptr 呼び出しをイニシエータからターゲットへのフォワード・パスに沿って通過させる必要がある。言い換えれば、インターコネクト・コンポーネントのターゲット・ソケットで、get_direct_mem_ptr の実装は、イニシエータ・ソケットで get_direct_mem_ptr メソッドを呼び出すことかもしれない。
e) それぞれの get_direct_mem_ptr 呼び出しは、イニシエータからターゲットへのトランスポート呼び出しの対応するセットと全く同一のパスに従う。言い換えれば、それぞれの DMI 要求は、一つのイニシエータと一つのターゲット間の相互作用を含む。これら二つのコンポーネント（イニシエータとターゲット）もまた、トランスポート・インタフェースを通る一つのトランザクション・オブジェクトのイニシエータとターゲットとしての役割をする。DMI は二つ目のトランザクション・オブジェクト（例えばバス幅変換のようなもの）を発生するようなコンポーネントを通るパス上では使うことができない。（もし DMI がシミュレーション速度が非常に要求される場合には、シミュレーション・モデルを再構成する必要があるかもしれない）
f) すべてのインターコネクト・コンポーネントは、フォワード方向での trans と dmi_data 引数を伝える必要がある。唯一許されている修正は、以下のように記述される DMI トランザクション・オブジェクトのアドレス引数値である。トランザクションと DMI 記述子のアドレス・アトリビュ

ートは両方とも get_direct_mem_ptr メソッド終了後に修正されるかもしれない。つまり、ターゲットからイニシエータへの関数呼び出しを巻き戻しているとき。
g) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていれば、以下に記述するように、DMI 記述子のメンバーをセットし、関数の返り値として true をセットする必要がある。
h) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていなければ、以下に記述するように、アドレス範囲と DMI 記述子の型メンバーのみセットし、関数の返り値として false をセットする必要がある。
i) get_direct_mem_ptr を複数回呼び出した場合、ターゲットは同じメモリ領域、同じ時刻に DMI アクセスを複数のイニシエータに許可するかもしれない。アプリケーションは同期とデータの一貫性を保証する必要がある。
j) それぞれの get_direct_mem_ptr の呼び出しで隣り合うメモリ領域への単一 DMI ポインタのみを返すので、実際にはそれぞれの DMI 要求は一つのターゲットでのみ実行可能である。別の言い方をすれば、もしメモリ領域が複数のターゲットに分散している場合、例えアドレス範囲が隣りあっていても、それぞれのターゲットは分割された DMI リクエストを要求するだろう。

4.2.4 テンプレート引数と tlm_generic_payload クラス

a) tlm_fw_direct_mem_if テンプレートは DMI トランザクション・クラスの型でパラメタライズされる。
b) トランザクション・オブジェクトはダイレクト・メモリ・アクセスが要求するアドレスを指し示すアトリビュート及び、要求されるアクセスの型、つまり指定されたアドレスに対する読み込みアクセスか、書き込みアクセスかを含む。
c) TRANS テンプレートの引数のデフォルト値は tlm_generic_payload クラスである。
d) 相互利用性を最大化するために、DMI トランザクション・クラスは tlm_generic_payload にすべきである。必須拡張または他のトランザクション型を使うことによって相互利用性は制限される。
e) イニシエータは DMI トランザクション・オブジェクトを生成及び管理する必要があり、また get_direct_mem_ptr への引数として渡される前に適切な属性を設定する必要がある。
f) トランザクション・オブジェクトのコマンド属性は、要求されている DMI アクセスの種類、つまり読み込みアクセスの TLM_READ_COMMAND または書き込みアクセスの TLM_WRITE_COMMAND かを示すために、イニシエータによって設定される。
g) トランザクション・オブジェクトのアドレス属性は、ダイレクト・メモリ・アクセスが要求しているアドレスを指し示すために、イニシエータによって設定される。
h) フォワード・パスに沿って DMI トランザクション・オブジェクトが通過するインターコネクト・コンポーネントによって同じソケットの対応するトランスポート・インタフェースにトランザクションのアドレス属性をデコード及び、必要に応じて修正すべきである。例えば、インターコネクト・コンポーネントによって、ターゲットのアドレス幅やシステム・メモリ・マップにおける場所に応じて、アドレスをマスク(上位ビットの数を削減) する必要がある。
i) インターコネクト・コンポーネントは,アドレスエラーを検知した場合、get_direct_mem_ptr 呼び出しを通過させる必要はない。
j) 基本プロトコルの場合、イニシエータは、コマンドとアドレスを除いて汎用ペイロードの属性を設定する必要はない。また、ターゲットとどんなインターコネクト・コンポーネントでも、他のすべての属性を無視してもよい。特に、応答ステータスの属性や許可されている DMI 属性は無視してもよい。許可されている DMI 属性は、トランスポート・インタフェースとともに利用することだけを想定している。
k) イニシエータは、一つの DMI 呼び出しから次の呼び出し、さらには DMI やトランスポート・インタフェース、そしてデバッグ・トランスポート・インタフェースの呼び出しに渡ってトランザクション・オブジェクトは再利用されるかもしれない。
l) もしアプリケーションが、要求されている DMI アクセスの種類が決定したときターゲットによって利用される DMI トランザクション・オブジェクトにさらに属性を追加する必要があるならば、無関係なトランザクション・クラスに置き換えるよりも、汎用ペイロードを拡張することを推奨する。例えば、DMI トランザクションは、CPU の種類に応じて異なった DMI 要求をターゲットが処理するために、CPU Id が必要かもしれない。そのような拡張が無視不可能な場合は、新しいプロトコル型クラスを定義する必要がある。

4.2.5 tlm_dmi クラス

a) DMI 記述子は tlm_dmi クラスのオブジェクトである。DMI 記述子は、イニシエータで生成されるが、そのメンバーはインターコネクト・コンポーネントまたはターゲットで設定される。イニシエータは DMI 記述子を再利用してもよい。その場合イニシエータは get_direct_mem_ptr 関数を呼び出す間に init メソッドを呼び出し再度初期化する。

b) DMI 記述子は、次の属性を持つ：DMI ポインタ属性、許可されたアクセス型の属性、開始アドレス属性、終了アドレス属性、読み込みレイテンシ属性、そして書き込みレイテンシ属性

c) インターコネクト・コンポーネントは DMI 記述子をフォワード・パスで修正してはいけないので、DMI 記述子はターゲットで受け取ったときに初期状態にあるべきである。

d) init メソッドはメンバーを以下に記述したように、初期化する。

e) set_dmi_ptr メソッドは DMI ポインタ・アトリビュートを引数で渡される値に設定される。get_dmi_ptr メソッドは DMI ポインタ・アトリビュートの現在の値を返す。

f) DMI ポインタ・アトリビュートは開始アドレス・アトリビュートの値に対応するストレージ場所を示すためにターゲットによって設定される。これは get_direct_mem_ptr 呼び出しで要求されるアドレス以下である。初期値は 0 である。

g) DMI 領域のストレージは unsigned char*の型で表現される。ストレージは汎用ペイロードのデータ配列と構成を持つ。もしターゲットがその構成のメモリ領域へのポインタを返すことが出来ない場合は、ターゲットは DMI をサポートできず、get_direct_mem_ptr は偽の値を返すべきである。TLM2.0 におけるメモリ構成とエンディアンの扱い方の詳細は、6.17 節を参照。

h) インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しからのリターン・パスにおいて DMI アクセスが許可された領域を制限するために DMI ポインタ・アトリビュートを修正することが許されている。

i) set_granted_access メソッドは、許可されたアクセス型アトリビュートを引数で渡された値で設定される。get_granted_access メソッドは、許可されたアクセス型アトリビュートの現在の値を返す。

j) allow_none、allow_read、allow_write そして allow_read_write メソッドは、許可されたアクセス型アトリビュートをそれぞれ DMI_ACCESS_NONE、DMI_ACCESS_READ、DMI_ACCESS_WRITE、または DMI_ACCESS_READ_WRITE の値に設定する。

k) is_none_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_NONE の値の場合及びその場合のみ真の値を返す。is_read_allowed メソッドは、 許可されたアクセス型アトリビュートが DMI_ACCESS_READ または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_write_allowed メソッドは、 許可されたアクセス型アトリビュートが DMI_ACCESS_WRITE または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_read_write_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。

l) ターゲットは、許可されたアクセス型アトリビュートに許可されたアクセスの型を設定する。ターゲットは、読み込みまたは読み込み／書き込みアクセスが許可されることよって読み込みアクセスの要求に答えることができ、同様に書き込みみまたは読み込み／書き込みアクセスが許可されることよって書き込みアクセスの要求に答えることができる。インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しのリターン・パスにおいて、DMI_ACCESS_READ_WRITE の値を DMI_ACCESS_READ または DMI_ACCESS_WRITE で上書きすることによって許可されたアクセスを制限ことができる。

m) ターゲットは、イニシエータへの読み込みや書き込みまたは読み込み／書き込みアクセスが許可されていないが、DMI トランザクション・オブジェクトへの拡張によって要求されるあるほかのアクセスの種類を許可していることを示すために、DMI_ACCESS_NONE を許可されたアクセス型に設定すべきである。この値は、DMI トランザクション・オブジェクトの拡張によって事前に定義されたアクセス型(読み込み、書き込みと読み込み／書き込み) が不要または意味がなくなる場合にのみ利用されるべきである。この値は基本プロトコルの場合に利用されるべきではない。

n) イニシエータは、許可されたアクセス型アトリビュートを使ってターゲットによって許可されている（または基本プロトコルとは別に、汎用ペイロードの拡張やまたほかの DMI トランザクション型を使って許可されている）DMI アクセスのこれらのモード（インターコネクトによって修正される可能性はあるが）のみを利用する責任がある。

o) set_start_address と set_end_address メソッドは、開始と終了アドレス・アトリビュートを

それぞれ引数で渡される値を設定する。get_start_address と get_end_address メソッドは、開始と終了アドレス・アトリビュートの現在の値をそれぞれ返す。
p) 開始と終了アドレス・アトリビュートは DMI 領域における最初と最後のバイトのアドレスを示すためにターゲットによって設定（またはインターコネクトによって修正）される。DMI 領域が有効か無効かが get_direct_mem_ptr メソッド返り値（true または false）によって決まる。
q) ターゲットは、それぞれの get_direct_mem_ptr 呼び出しに対して一つの隣接するメモリ領域を許可または拒絶しかできない。ターゲットは、開始と終了アドレス・アトリビュートを同じにすることによってメモリ領域を一つのアドレスに設定することができ、DMI 領域を任意の大きさに設定することもできる。
r) 指定された領域への指定された型の DMI アクセスが許可されているとき、イニシエータはその領域が無効になるまでその領域のどこでも指定された型のアクセスを実行することができる。別の言い方でいうと、DMI 要求によって指定されたアドレスはアクセス制限されない。
s) get_direct_mem_ptr 呼び出しが通過するいかなるインターコネクト・コンポーネントでも、アドレス引数を変換するように開始と終了アドレス・アトリビュートを変換する必要がある。DMI 記述子におけいるアドレスのいかなる変換も、記述子が get_direct_mem_ptr 関数呼び出しのリターン・パスを通過するときに実行される。例えば、ターゲットは開始アドレス・アトリビュートをそのターゲットが知っているメモリ・マップ内での相対アドレスに設定する。その場合インターコネクト・コンポーネントは相対アドレスをシステム上のメモリ・マップにおける絶対アドレスに変換する必要がある。初期値はそれぞれ 0 と sc_dt::uint64 の取り得る最大値である。
t) インターコネクト・コンポーネントは、DMI アクセスが許可されている領域を制限するため、または DMI アクセスが許可されていない領域を拡張するために、開始と終了アドレス・アトリビュートを修正することができる。
u) もし get_direct_mem_ptr が true を返す場合は、開始と終了アドレス・アトリビュートで示される DMI 領域が DMI アクセスできる領域である。もし get_direct_mem_ptr が false を返す場合は、DMI アクセスできない領域である。
v) ターゲットまたはインターコネクト・コンポーネントが二つまたはそれ以上の get_direct_mem_ptr 呼び出しを受け取る場合は、二つまたはそれ以上の重なり合う DMI アクセス許可領域、または二つまたはそれ以上の重なり合う DMI アクセス禁止領域を返すかもしれない。
w) ターゲットまたはインターコネクト・コンポーネントは、同じアクセス型（例えば、読み込み及び読み込み／書き込みの両方や書き込み及び読み込み／書き込みの両方）に対して、最初の領域を無効にするために invalidate_direct_mem_ptr を途中で呼び出さない場合には、許可領域と禁止領域がオーバーラップする DMI 領域を返してはならない。
x) 別の言葉で言えば、DMI 領域の定義は invalidate_direct_mem_ptr を途中呼び出しによって最初の領域が無効化されない限り、生成された順番に依存しない。特に禁止 DMI 領域の生成は、同じアクセス型ですでに許可された DMI 領域に穴をあけることができない。逆もまた真。
y) ターゲットは DMI アクセスをすべてのアドレス領域（開始アドレスを 0、終了アドレスを最大値）で禁止してもよい。ターゲットが DMI アクセスを全くサポートしないようなケースである。その場合、インターコネクト・コンポーネントはこのターゲットが占有しているメモリ・マップの一部だけを禁止領域として制限するべきである。もしインターコネクト・コンポーネントがアドレス領域を制限することに失敗したら、イニシエータはすべてのシステム上のアドレス空間のすべてが DMI 禁止されていると勘違いをするだろう。
z) set_read_latency と set_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・アトリビュートに引数で渡された値を設定する。get_read_latency と get_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・アトリビュートの現在の値を返す。
aa) 読み込み及び書き込みレイテンシ・アトリビュートは、それぞれ読み込み及び書き込みのメモリ・トランザクションにかかるレイテンシを設定する。初期値は SC_ZERO_TIME。インターコネクト・コンポーネントとターゲットの両方が読み込み、書き込みのどちらかのレイテンシの値を増やし、その結果 DMI 記述子が get_direct_mem_ptr メソッドからのリターン・パスにおいてイニシエータからターゲットへ戻るときにレイテンシが積算される。許可されたアクセス型アトリビュートの値によって、一つまたは両方のレイテンシが有効になる。
bb) イニシエータはダイレクト・メモリ・ポインタを使ってメモリにアクセスするときはいつでも、レイテンシを考慮する必要がある。もしイニシエータがレイテンシを無視すれば、タイミングが不正確になる。

4.2.6 invalidate_direct_mem_ptr メソッド

a) invalidate_direct_mem_ptr メソッドはターゲットまたはインターコネクト・コンポーネントだけによって呼び出される。
b) ターゲットは存在する DMI 領域の有効性やアクセス型を修正する変更の前に invalidate_direct_mem_ptr メソッドを呼び出す必要がある。例えば、存在する DMI 領域のアドレス領域を制限する前や、アクセス型を読み込み／書き込みから読み込みに変更する前や、アドレス空間をリマップする前などである。
c) start_range と end_range の引数は、DMI 領域が無効化されるアドレス範囲の最初と最後のアドレスを示す。
d) イニシエータが invalidate_direct_mem_ptr を受け取ったらすぐに指定されたアドレス範囲と重なり合う DMI 領域(get_direct_mem_ptr で事前に獲得していたもの)を無効にして廃棄する。
e) 部分的に重なり合う場合、つまり存在する DMI 領域の一部分だけ無効な場合には、イニシエータは存在する領域の境界を調整するか、または全体の領域を無効化する。
f) 各 DMI 領域は、ターゲットが invalidate_direct_mem_ptr 呼び出しで明示的に無効化するまでは有効のままである。各イニシエータは、有効な DMI 領域の管理表を管理し、無効化されるまで各領域を利用し続ける。
g) すべてのインターコネクト･コンポーネントは、デコードや必要に応じて対応するトランスポート・インタフェースのためにアドレス引数を修正するときに、invalidate_direct_mem_ptr 呼び出しをイニシエータからターゲットへのバックワード・パスに沿って通過させる義務がある。トランスポート・インタフェースがフォワード・パスと DMI のバックワード・パスでアドレスを変換するので、トランスポートと DMI 変換は相互に逆であるべきである。
h) もし一つの invalidate_direct_mem_ptr をターゲットから呼び出すならば、インターコネクト・コンポーネントは複数の invalidate_direct_mem_ptr をイニシエータへ呼び出してもよい。複数のイニシエータが存在しそれぞれが同じターゲットに対してダイレクト・メモリ・ポインタを得るかもしれないので、インターコネクト・コンポーネントがすべてのイニシエータに対して invalidate_direct_mem_ptr を呼び出すのが安全な実装方法である。
i) インターコネクト・コンポーネントは、start_range を 0、end_range を sc_dt::uint64 の最大値に設定することによって、イニシエータにあるすべてのダイレクト・メモリ・ポインタを無効化することができる。

4.2.7 DMI ヒントを使った最適化

a) DMI ヒントは DMI アクセスの繰り返しポーリングをなくすことによってシミュレーションスピードを最適化するメカニズムである。DMI ポインタが使える状態か確認するために get_direct_mem_ptr を呼び出すかわりに、イニシエータがトランスポート・インタフェースを通過する通常のトランザクションの DMI ヒントを確認することができる。
b) 汎用ペイロードによって DMI ヒントは提供される。ユーザ定義のトランザクションによって同じメカニズムを実装できる。その場合ターゲットによって DMI ヒントの値を適切に設定すべきである。
c) DMI ヒントを使うかどうかは自由に選択できる。イニシエータは汎用ペイロードの DMI ヒントを自由に無視できる。
d) イニシエータが DMI ヒントを有効にするための推奨手順は下記に示す。
   i. イニシエータが有効な DMI 領域のキャッシュに対するアドレスを確認する。
   ii. もし DMI ポインタが存在しないなら、イニシエータはトランスポート・インタフェースを通して通常のトランザクションを実行する。
   iii. その後、イニシエータはトランザクションの DMI ヒントを確認する。
   iv. もし DMI ヒントが DMI を許可していることを示していれば、イニシエータは get_direct_mem_ptr を呼び出す。

4.3 デバッグ・トランスポート・インタフェース

4.3.1 イントロダクション

デバッグ・トランスポート・インタフェースによって、遅延、待ち、イベント通知または通常のトランザクションに対する副作用を一切発生させないでターゲットのストレージに対して読み込み及び書き込みが可能である。デバッグ・トランスポート・インタフェースは、トランスポート・インタフェースと同じパスを通るため、デバッグ・トランスポート・インタフェースの実装は通常のトランザクションと同じアドレス変換を実行する。

例えばデバッグ・トランスポート・インタフェースによって、ISS に接続されたソフトウェアデバッガがシミュレーションされるシステム内のメモリデータを読み出したり、書き込んだりできる。またデバッグ・トランスポート・インタフェースによって、イニシエータがシステムメモリの内容をシミュレーション中に解析目的でスナップショットを取ることや、システム内のメモリのある領域を初期化することも可能である。
デフォルトのデバッグ・トランザクション型は tlm_generic_payload であり、トランザクション・オブジェクトのコマンド、アドレス、データ長及びデータ・ポインタ・アトリビュートのみを利用する。デバッグ・トランザクションは、トランスポート・インタフェースにおける拡張と同じアプローチを取る。つまり、デバッグ・トランザクションは無視可能拡張を含むかもしれないが、必須拡張の場合は新しいプロトコル型クラスを定義する必要がある（tlm_generic_payload の型定義の内容を含む 6.2.2 節 新しいプロトコル型クラスの定義を参照)。

4.3.2 クラス定義

```
namespace tlm {

template <typename TRANS = tlm_generic_payload>
class tlm_transport_dbg_if : public virtual sc_core::sc_interface
{
public:
  virtual unsigned int transport_dbg(TRANS& trans) = 0;
};

} // namespace tlm
```

4.3.3 TRAN テンプレート引数と tlm_generic_payload クラス

a) tlm_transport_dbg_if テンプレートはデバッグ・トランザクション・クラスの型でパラメタライズされている。
b) デバッグ・トランザクション・クラスには、デバッグ・アクセスに必要なコマンド、アドレス、データ長及びデータ・ポインタをターゲットに教えるためのアトリビュートが含まれる。基本プロトコルの場合、これらは汎用ペイロードの対応するアトリビュートである。
c) TRANS テンプレート引数のデフォルト値は tlm_generic_payload クラスである。
d) 相互利用性を最大化するために、デバッグ・トランザクション・クラスは tlm_generic_payload であるべき。必須拡張や他のトランザクション型を使えば、相互利用性が制限される。
e) もしアプリケーションがデバッグ・トランザクションにさらにアトリビュートを追加する必要がある場合には、関連性のないトランザクション・クラスに置き換えるよりも汎用ペイロードを拡張する方法を推奨する。その拡張が無視不可能の場合には、新しいプロトコル型クラスを定義する必要がある。

4.3.4 ルール

a) transport_dbg への呼び出しは、通常のトランザクションで利用されるトランスポート・インタフェースと同じフォワード･パスに従う。
b) trans 引数はデバッグ・トランザクション・オブジェクトへの参照渡しである。
c) イニシエータはデバッグ・トランザクション・オブジェクトを生成し、管理する必要がある。また transport_dbg への引数で渡す前にオブジェクトの適切なアトリビュートを設定する必要がある。
d) トランザクション・オブジェクトのコマンド・アトリビュートはイニシエータによって要求されているデバッグ・アクセスの種類（TLM_READ_COMMAND はターゲットへの読み込みアクセス、TLM_WRITE_COMMAND はターゲットへの書き込みアクセス）を示すために設定される。
e) アドレス･アトリビュートはイニシエータによって読み込みまたは書き込みされる領域の最初のアドレスに設定される。
f) フォワード・パスに沿ってデバッグ・トランザクション・オブジェクトを通るインターコネクト・コンポーネントで、同じソケットの対応するトランスポート・インタフェースに、トランザクション・オブジェクトのアドレス属性をデコード及び必要に応じて修正されるべきである。例えば、インターコネクト・コンポーネントで、ターゲットアドレス幅とシステム・メモリマッ上の場所に従ってアドレスをマスク（上位ビットを削減）する必要がある。
g) インターコネクト・コンポーネントでは、アドレッシング・エラーを検知した場合 transport_dbg 呼び出しを通過させる必要はない。

h) アドレス・アトリビュートは、もしデバッグ・ペイロードが複数のインターコネクト・コンポーネントを通って転送されれば、複数回修正されているかもしれない。デバッグ・ペイロードがイニシエータに戻ってきた時には、アドレス・アトリビュートのもともとの値は上書きされているかもしれない。
i) データ長アトリビュートはイニシエータによって読み込むまたは書き込むバイト数に設定される。
j) データ・ポインタ・アトリビュートはイニシエータによってターゲットにコピーされる値（書き込み時）またはターゲットからコピーされる値（読み込み時）の配列のアドレスに設定される。この配列はイニシエータによって割り当てられ、transport_dbg から戻ってくる前に削除されない。配列サイズは少なくともデータ長アトリビュートの値と同じである。
k) もし可能であれば、ターゲットにおける transport_dbg の実装は、指定されたアドレスを使って（インターコネクトを通してアドレス変換後に）指定されたバイト数を読み込みまたは書き込むようにする。
l) トランスポート・インタフェースと一緒に利用されたときに、データ配列は汎用ペイロードのデータ配列と同じ構成を持つ。transport_dbg の実装は、ターゲット内のローカル・データ・ストレージの構成と汎用ペイロードの構成間で変換する必要がある。
m) 基本プロトコルの場合、イニシエータはコマンド、アドレス、データ長及びデータ・ポインタ以外の汎用ペイロードのアトリビュートを設定する必要はない。ターゲットとインターコネクト・コンポーネントは他のすべてのアトリビュートを無視してもよい。特に、応答ステータス・アトリビュートは無視される。
n) イニシエータは一つの呼び出しから次の呼び出し、デバッグ・トランスポート・インタフェース、トランスポート・インタフェース及び DMI に渡る呼び出しでトランザクション・オブジェクトを再利用してもよい。
o) transport_dbg は、実際に読み込みまたは書き込みしたバイト数の合計を返す。それは num_bytes よりも少ないことがある。もしターゲットが実行できない時には、0 の値を返す。
p) transport_dbg は、待ち(wait)を呼ばない、イベント通知を生成しない、ターゲットやインターコネクト・コンポーネントへの副作用がない。

5. 結合インタフェースとソケット

5.1 統合インタフェース

5.1.1 イントロダクション

- フォワード・トランスポート I/F 、バックワード・トランスポート I/F は、TLM-2 コア・インタフェースをグループ化している。これは、イニシエータ/ターゲット・ソケットによりグループ化される。
- コア・インタフェースには、トランスポート、DMI、デバッグ・トランスポート・インタフェースを含むが、TLM-1 コア・インタフェースは含まない。
- フォワード I/F は、イニシエータ・ソケットからターゲット・ソケットへのフォワード・パスにおけるメソッド呼び出しを提供する。バックワード I/F は、ターゲット・ソケットからイニシエータ・ソケットへのバックワード・パスにおけるメソッド呼び出しを提供する。ブロッキング・トランスポート I/F とデバッグ・トランスポート I/F はバックワード・パスを必要としない。
- 標準のイニシエータ・ソケットやターゲット・ソケットに関連のない新しいソケットクラスを定義することは、技術的には可能であるが、相互利用性の観点からは推奨しない。一方、標準のソケットクラスから新しいソケットクラスを派生させる方法は、推奨する。
- 結合された I/F のテンプレートは、プロトコル・タイプ・クラスでパラメタライズされる。プロトコル・タイプ・クラスは、フォワード I/F、バックワード I/F で使われるタイプ、すなわちペイロード・タイプとフェーズ・タイプを定義する。プロトコル・タイプ・クラスは、特定のプロトコルに関連付けられる。デフォルトのプロトコル・タイプは tlm_base_protocol_types である。

5.1.2 クラス定義

```
namespace tlm {
  // The default protocol types class:
  struct tlm_base_protocol_types
  {
```

```
    typedef tlm_generic_payload tlm_payload_type;
    typedef tlm_phase tlm_phase_type;
    };
    // The combined forward interface:
    template< typename TYPES = tlm_base_protocol_types >
    class tlm_fw_transport_if
    : public virtual tlm_fw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
      typename TYPES::tlm_phase_type>
    , public virtual tlm_blocking_transport_if< typename TYPES::tlm_payload_type>
    , public virtual tlm_fw_direct_mem_if< typename TYPES::tlm_payload_type>
    , public virtual tlm_transport_dbg_if< typename TYPES::tlm_payload_type>
    {};
    // The combined backward interface:
    template < typename TYPES = tlm_base_protocol_types >
    class tlm_bw_transport_if
    : public virtual tlm_bw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
      typename TYPES::tlm_phase_type >
    , public virtual tlm_bw_direct_mem_if
    {};
  } // namespace tlm
```

5.2 イニシエータ・ソケットとターゲット・ソケット

5.2.1 イントロダクション

- ソケット：port と export を組み合わせたもの
  - ソケットの種類
    - イニシエータ・ソケット：フォワード・パス用 port とバックワード・パス用 export
    - ターゲット・ソケット：フォワード・パス用 export とバックワード・パス用 port
  - ソケットは、port と export を接続先の port と export とをバインドする機能を持つ。
  - 階層をまたいでバインドを行う場合は、順序に注意が必要。
  - アプリケーションが通常使用するソケット
    - tlm_initiator_socket
    - tlm_target_socket
  - これらのソケットは、プロトコル・タイプ・クラスでパラメタライズされる。
  - ソケットは、プロトコル・タイプが同じである場合にのみバインド可能となる。デフォルトのプロトコル・タイプは tlm_base_protocol_types である。
  - 新規にプロトコル・タイプを定義する場合は、汎用ペイロードを使用するかどうかに関わらず、新しいプロトコル・タイプ・クラスを持つ結合された I/F テンプレートをインスタンス化するべきである。
- ソケットの利点

  a. フォワード・パス、バックワード・パスに対するトランスポート、ダイレクトメモリ I/F、デバッグ・トランスポート I/F を 1 つのオブジェクトにグループ化する。

  b. フォワード・パスとバックワード・パスそれぞれの port と export を、 1 回の呼び出しでバインドするメソッドを提供。

  c. 互換性のないプロトコル・タイプでパラメタライズされたソケットに対する、強力なタイプ・チェック機能を提供。

  d. トランザクションを中断させるために利用可能なバス幅パラメタを持つ。
- tlm_initiator_socket と tlm_target_socket クラスは、インタオペラビリティ・レイヤに属する。更に、ユーティリティ・ネームスペース内に便利ソケットとして知られる派生ソケットが定義されている。

5.2.2 クラス定義

```
namespace tlm {
  // Abstract base class for initiator sockets
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
  >
  class tlm_base_initiator_socket_b
```

```
{
public:
  virtual ~tlm_base_initiator_socket_b() {}
  virtual sc_core::sc_port_b<FW_IF> & get_base_port() = 0;
  virtual BW_IF & get_base_interface() = 0;
  virtual sc_core::sc_export<BW_IF> & get_base_export() = 0;
};
// Abstract base class for target sockets
template <
  unsigned int BUSWIDTH = 32,
  typename FW_IF = tlm_fw_transport_if<>,
  typename BW_IF = tlm_bw_transport_if<>
>
class tlm_base_target_socket_b
{
public:
  virtual ~tlm_base_target_socket_b();
  virtual sc_core::sc_port_b<BW_IF> & get_base_port() = 0;
  virtual sc_core::sc_export<FW_IF> & get_base_export() = 0;
  virtual FW_IF & get_base_interface() = 0;
};
// Base class for initiator sockets, providing binding methods
template <
  unsigned int BUSWIDTH = 32,
  typename FW_IF = tlm_fw_transport_if<>,
  typename BW_IF = tlm_bw_transport_if<>,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_base_initiator_socket : public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
public sc_core::sc_port<FW_IF, N, POL>
{
public:
  typedef FW_IF fw_interface_type;
  typedef BW_IF bw_interface_type;
  typedef sc_core::sc_port<fw_interface_type, N, POL> port_type;
  typedef sc_core::sc_export<bw_interface_type> export_type;
  typedef tlm_base_target_socket<BUSWIDTH, fw_interface_type, bw_interface_type, N ,POL >
    target_socket_type;
  typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
    base_target_socket_type;
  typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
    base_type;
  tlm_base_initiator_socket();
  explicit tlm_base_initiator_socket(const char* name);
  unsigned int get_bus_width() const;
  void bind(base_target_socket_type& s);
  void operator() (base_target_socket_type& s);
  void bind(base_type& s);
  void operator() (base_type& s);
  void bind(bw_interface_type& ifs);
  void operator() (bw_interface_type& s);
  // Implementation of pure virtual functions of base class
  virtual sc_core::sc_port_b<FW_IF> & get_base_port() { return *this; }
  virtual BW_IF & get_base_interface() { return m_export; }
  virtual sc_core::sc_export<BW_IF> & get_base_export() { return m_export; }
protected:
  export_type m_export;
};

// Base class for target sockets, providing binding methods
template <
```

```
  unsigned int BUSWIDTH = 32,
  typename FW_IF = tlm_fw_transport_if<>,
  typename BW_IF = tlm_bw_transport_if<>,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_base_target_socket : public tlm_base_target_socket_b<BUSWIDTH, FW_IF, BW_IF>,
public sc_core::sc_export<FW_IF>
{
public:
  typedef FW_IF fw_interface_type;
  typedef BW_IF bw_interface_type;
  typedef sc_core::sc_port<bw_interface_type, N, POL> port_type;
  typedef sc_core::sc_export<fw_interface_type> export_type;
  typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
  base_initiator_socket_type;
  typedef tlm_base_initiator_socket<BUSWIDTH, fw_interface_type, bw_interface_type, N, POL>
    initiator_socket_type;
  typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
    base_type;
  tlm_base_target_socket();
  explicit tlm_base_target_socket(const char* name);
  unsigned int get_bus_width() const;
  void bind(base_initiator_socket_type& s);
  void operator() (base_initiator_socket_type& s);
  void bind(base_type& s);
  void operator() (base_type& s);
  void bind(fw_interface_type& ifs);
  void operator() (fw_interface_type& s);
  int size() const;
  bw_interface_type* operator-> ();
  bw_interface_type* operator[] (int i);
  // Implementation of pure virtual functions of base class
  virtual sc_core::sc_port_b<BW_IF> & get_base_port() { return m_port; }
  virtual FW_IF & get_base_interface() { return *this; }
  virtual sc_core::sc_export<FW_IF> & get_base_export() { return *this; }
protected:
  port_type m_port;
};

// Principle initiator socket, parameterized with protocol types class
template <
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm_base_protocol_types,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_initiator_socket : public tlm_base_initiator_socket <
  BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
{
public:
  tlm_initiator_socket();
  explicit tlm_initiator_socket(const char* name);
};

// Principle target socket, parameterized with protocol types class
template <
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm_base_protocol_types,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
```

```
  class tlm_target_socket : public tlm_base_target_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
  {
  public:
    tlm_target_socket();
    explicit tlm_target_socket(const char* name);
  };
} // namespace tlm
```

5.2.3 tlm_base_initiator_socket_b と tlm_base_target_socket_b

a. 抽象ベースクラス tlm_base_initiator_socket_b と tlm_base_target_socket_b は、複数の純粋仮想関数を備えている。これらはソケットと関連付けられた port、export、インタフェースオブジェクトを返すためのもので、派生クラスで上書きされる必要がある。
b. これらソケットは、通常アプリケーションから直接使われることはない。

5.2.4 tlm_base_initiator_socket と tlm_base_target_socket

a. tlm_initiator_socket クラスでは、コンストラクタで指定されたキャラクタ文字列を、ベースクラスである sc_port のコンストラクタに対して引き渡し、インスタンス名として設定する。また、同じ文字列に”_export”を付加した名前を、backward パスの sc_export に対してインスタンス名として設定する。これらは sc_gen_unique_name の呼び出しにより設定される。

e.g.) tlm_initiator_socket(“foo”)と指定
→ sc_port　：foo
→ sc_export：foo_export

e.g.) デフォルト
→ sc_port ： sc_gen_unique_name(“tlm_base_initiator_socket”)
→ sc_export: sc_gen_unique_name("tlm_initiator_socket_export")

b. tlm_target_socket クラスでは、コンストラクタで指定されたキャラクタ文字列を、ベースクラスである sc_export のコンストラクタに対して引き渡し、インスタンス名として設定する。また､同じ文字列に”_port”を付加した名前を､sc_port に対してインスタンス名として設定する。これらは sc_gen_unique_name により設定される。

e.g.) tlm_target_socket(“foo”)と指定
→ sc_export：foo
→ sc_port　：foo_port

e.g.) デフォルト
→ sc_port ： sc_gen_unique_name("tlm_base_target_socket")
→ sc_export: sc_gen_unique_name("tlm_target_socket_port")

c. get_bus_width メソッドは、テンプレート引数 BUSWIDTH の値を返す。
d. テンプレート引数 BUSWIDTH は、ソケットを介して転送される、個々のデータ・ワードのワード長を決定するもので、ワード中のビット数で表される。バースト転送では、BUSWIDTH は各ビート中のビット数を決定する。本アトリビュートの厳密な解釈はトランザクション・タイプに依存する。汎用ペイロードにおける BUSWIDTH の意味は、「6.11 データ長アトリビュート」を参照。
e. ソケットとソケットをバインドする場合、二つのソケットは同一の BUSWIDTH 値を持たなければならない。
f. bind、operator () メソッドがソケットを引数とする場合、メソッドが属するソケット・インスタンスが、引数で渡されたソケット・インスタンスの当該メソッドにバインドされる。
g. bind、operator () メソッドがインタフェースを引数とする場合、メソッドが属するソケット・インスタンスの export が、引数で渡されたチャネル･インスタンスにバインドされる（チャネルはインタフェースを実装するクラスを表す SystemC 用語)。
h. イニシエータ・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、イニシエータ・ソケットの port をターゲット・ソケットの export にバインドする。また、ターゲット・ソケットの port とイニシエータ・ソケットの export をバインドする。これは、双方のソケットが同じ階層に置かれている場合に適用される。
i. イニシエータ・ソケットはターゲット・ソケットに対して、どちらのソケットの bind メソッド、または operator()メソッドでも、バインドすることができる。
j. イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合、または、ターゲット・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、それぞれ

のソケットの port と port をバインドする。また、それぞれのソケットの export と export をバインドする。これは、階層をまたぐバインド、すなわち、子ソケットから親ソケット、または、親ソケットから子ソケットへ接続される場合、トランザクションがモジュール階層上下に受け渡される場合に適用される。
k. 階層バインディングにおいては、正しい順序でバインドする必要がある。イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合は、子ソケットが親ソケットに対してバインドされなければならない。ターゲット・ソケットをターゲット・ソケットにバインドする場合は、親ソケットが子ソケットに対してバインドされなければならない。このルールは、tlm_base_initiator_socket が sc_port から派生しており、tlm_base_target_socket が sc_export から派生していることと一致している。階層をさかのぼる場合は、port から port へ、トップ階層においては port から export へ、階層を下がる場合は export から export へバインドされなければならない。
l. tlm_base_initiator_socket と tlm_base_target_socket の二つのソケットを相互にバインドする場合、フォワード・インタフェース・タイプ、バックワード・インタフェース・タイプ、バス幅は同じでなければならない。
m. ターゲット・ソケットの size メソッドは、バックワード・パスのターゲット・ソケットのポートの size メソッドを呼び出し、そのポートの size で返される値を返す。
n. ターゲット・ソケットの operator->メソッドは、バックワード・パスのターゲット・ソケットのポートの operator->メソッドを呼び出し、そのポートの operator->で返される値を返す。
o. ターゲット・ソケットの operator[]メソッドは、バックワード・パスのターゲット・ソケットのポートの operator[]メソッドを同じ引数で呼び出し、そのポートの operator[]で返される値を返す。
p. tlm_base_initiator_socket と class tlm_base_target_socket は、マルチ・ソケットとして振舞う。すなわち、一つのイニシエータ・ソケットは、複数のターゲット・ソケットとバインドすることができ、一つのターゲット・ソケットは、複数のイニシエータ・ソケットとバインドすることができる。インデックスは、bind や operator()メソッドが呼ばれた順序に依存する。
q. tlm_base_initiator_socket または tlm_base_target_socket が複数回バインドされた場合、operator[]メソッドは接続先の対応するオブジェクトを指定する目的に使用できる。インデックス値は bind または operator()メソッドがよびだされた順序に依存する。しかしながら、受け取り側のソケットは、呼び出し側を特定するメカニズムを持たないため、受け取り側のインタフェース・メソッド呼び出しは、anonymous となる。このメカニズムについては、便利ソケットの側で提供される。「5.3.4 マルチソケット」を参照。
r. 例えば、一つのソケットが、二つのターゲットにバインドされる場合を考える。二つの呼び出し、socket[0]->nb_transport_fw(...)、socket[1]->nb_transport_fw()は、ターゲットを区別することはできるが、これら二つのターゲットからの nb_transport_bw()は、呼び出し側は区別することができない。
s. 仮想メソッド get_base_port と get_base_export は、ソケットの port と export のオブジェクトを返す。仮想メソッド get_base_interface は、イニシエータ・ポートの場合は export オブジェクトを、ターゲット・ソケットの場合はソケット・オブジェクトそのものを返すように実装するべきである。

5.2.5 tlm_initiator_socket と tlm_target_socket

a. tlm_initiator_socket と tlm_target_socket は、プロトコル・タイプ・クラスをテンプレート・パラメタとしてとる。これらのソケット(またはこれらから派生する便利ソケット)は、通常アプリケーションが利用する。
b. tlm_initiator_socket と tlm_target_socket を相互にバインドする場合は、同じプロトコル・タイプ・クラス（デフォルトは tlm_base_protocol_types)、同じバス幅を持つ必要がある。新規プロトコル・タイプに対しても、新しいプロトコル・タイプ・クラスを定義することにより、ソケット間の強力なタイプ・チェックが可能である。これは汎用ペイロードをベースにしたプロトコルでなくても可能である。
例

```
#include <systemc>
#include "tlm.h"
using namespace sc_core;
using namespace std;
```

```
struct Initiator: sc_module, tlm::tlm_bw_transport_if<> // Initiator implements the bw interface
{
  tlm::tlm_initiator_socket<32> init_socket; // Protocol types default to base protocol
  SC_CTOR(Initiator) : init_socket("init_socket") {
    SC_THREAD(thread);
    init_socket.bind( *this ); // Initiator socket bound to the initiator itself
  }
  void thread() { // Process generates one dummy transaction
    tlm::tlm_generic_payload trans;
    sc_time delay = SC_ZERO_TIME;
    init_socket->b_transport(trans, delay);
  }
  virtual tlm::tlm_sync_enum nb_transport_bw(
    tlm::tlm_generic_payload& trans,
    tlm::tlm_phase& phase,
    sc_core::sc_time& t) {
      return tlm::TLM_COMPLETED; // Dummy implementation
  }
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
  { } // Dummy implementation
};

struct Target: sc_module, tlm::tlm_fw_transport_if<> // Target implements the fw interface
{
  tlm::tlm_target_socket<32> targ_socket; // Protocol types default to base protocol
  SC_CTOR(Target) : targ_socket("targ_socket") {
    targ_socket.bind( *this ); // Target socket bound to the target itself
  }
  virtual tlm::tlm_sync_enum nb_transport_fw(
  tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t) {
    return tlm::TLM_COMPLETED; // Dummy implementation
  }
  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
  { } // Dummy implementation
  virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data)
  { return false; } // Dummy implementation
  virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
  { return 0; } // Dummy implementation
};

SC_MODULE(Top1) // Showing a simple non-hierarchical binding of initiator to target
{
  Initiator *init;
  Target *targ;
  SC_CTOR(Top1) {
    init = new Initiator("init");
    targ = new Target("targ");
    init->init_socket.bind(targ->targ_socket); // Bind initiator socket to target socket
  }
};

struct Parent_of_initiator: sc_module // Showing hierarchical socket binding
{
  tlm::tlm_initiator_socket<32> init_socket;
  Initiator* initiator;
  SC_CTOR(Parent_of_initiator) : init_socket("init_socket") {
    initiator = new Initiator("initiator");
    initiator->init_socket.bind( init_socket ); // Bind initiator socket to parent initiator socket
  }
};

struct Parent_of_target: sc_module
```

```
{
  tlm::tlm_target_socket<32> targ_socket;
  Target* target;
  SC_CTOR(Parent_of_target) : targ_socket("targ_socket") {
    target = new Target("target");
    targ_socket.bind( target->targ_socket ); // Bind parent target socket to target socket
  }
};

SC_MODULE(Top2)
{
  Parent_of_initiator *init;
  Parent_of_target *targ;
  SC_CTOR(Top2) {
    init = new Parent_of_initiator("init");
    targ = new Parent_of_target("targ");
    init->init_socket.bind(targ->targ_socket); // Bind initiator socket to target socket at top level
  }
};
```

5.3 便利ソケット

5.3.1 イントロダクション

要素モデルをより簡単に書け、追加機能性を実装した便利ソケットファミリーがある。 便利ソケットは、tlm_initiator_socket クラスと tlm_target_socket クラスから派生される。便利ソケットは、TLM-2 の相互利用性レイヤ(インタオペラビリティ・レイヤ) の一部ではないが、ネームスペース tlm_utils で参照できる。

- コールバック登録とは、ソケットは、インタフェース・メソッド読み出しに対応するコールバック・メソッドを提供している。簡単に言うと、コールバックは、応答するオブジェクトをソケットに登録する。
- マルチポートとは、単一イニシエータ・ソケットを複数のターゲット・ソケットにバインドすることができる。また、逆に単一ターゲット・ソケットを複数のイニシエータ・ソケットにバインドすることができる。ソケット・クラス・テンプレートは、バインド数とテンプレート・アーギュメントのバインディング方針を提供する。
- b-nb 変換とは、ターゲット・ソケットは、呼び出された b_transport 関数を nb_transport_fw 関数呼出しに変換したり、その逆をすることができる。
- タグ付けとは、インタフェース・メソッド呼び出しは、id でタグ付けをされている。id は、どのソケットに到着したかを示す。
- 階層バインドとは、ソケットは、階層化されている状態で、イニシエータ・ソケットからイニシエータ・ソケット、またイニシエータ・ソケットからターゲットにバインドできる。

クラス名	コールバック登録	マルチポート	b/nb 変換	タグ付け	階層バインド
tlm_initiator_socket	no	yes	-	no	yes
tlm_target_socket	no	yes	no	no	yes
simple_initiator_socket	yes	no	-	no	no
simple_initiator_socket_tagged	yes	no	-	yes	no
simple_target_socket	yes	no	yes	no	no
simple_target_socket_tagged	yes	no	yes	yes	no
passthrough_target_socket	yes	no	no	no	no
passthrough_target_socket_tagged	yes	no	no	yes	no
multi_passthrough_initiator_socket	yes	yes	-	yes	yes
multi_passthrough_target_socket	yes	yes	no	yes	yes

5.3.2 シンプル・ソケット

5.3.2.1 イントロダクション

シンプル・ソケットは、使用するのがとても簡単なので”シンプル・ソケット”と呼ぶ。 シンプル・ソケットは、互利用性レイヤ・ソケット tlm_initiator_socket と tlm_target_socket から派生し、そのタイプを持つソケットに直接バインドできる。

シンプル・ソケットは、双方向にバインドする代わりに、コールバック関数を登録する方法を提供する。コールバック関数は、インタフェース・メソッド呼び出しが到着する毎に、呼び出される。コールバック・メソットは、ソケットによってサポートされた、関数登録のメソッドと言える。
ユーザは、あらゆるインタフェース・メソッド対応する為、シンプル・ソケットにすべてのコールバック関数を登録する必要があると考えるかもしれない。しかし、強制されていない。例えば、シンプル・ターゲット・ソケットにおいては、ユーザは b_transport 関数と nb_transport_fw 関数の両方、あるいは片方を登録するだけで、自動的に、登録されていないインタフェース・メソッドを、登録されたインタフェース・メソッドに変換できる。この変換過程は、重要である。これは、イニシエータとターゲットが尊重する基本プロトコルの規則に依存している。
passthrough_target_socket は、simple_target_socket の一種で、ブロッキングとノンブロッキング呼び出しの間の変換をサポートしていない。
現在、シンプル・ソケットインプリメンテーションとして、ダイナミックプロセスを利用している。したがって、OSCI(proof-of-concept)SystemC シミュレータの現在のリリースでは、シンプル・ソケットを使用しているアプリケーションをコンパイルする際、 SystemC のヘッダー・ファイルをインクルードする前に、SC_INCLUDE_DYNAMIC_PROCESSES マクロを定義する必要がある。

5.3.2.2 クラス定義

```
namespace tlm_utils {
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_initiator_socket : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    explicit simple_initiator_socket(const char* n = "simple_initiator_socket");
    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
    void register_invalidate_direct_mem_ptr(
      MODULE* mod,
      void (MODULE::*cb)(sc_dt::uint64, sc_dt::uint64));
  };
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    explicit simple_target_socket(const char* n = "simple_target_socket");
    tlm::tlm_bw_transport_if<TYPES> * operator ->();
    void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
    void register_b_transport(
      MODULE* mod,
      void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));
    void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(transaction_type&));
```

```
    void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
  };
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class passthrough_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    explicit passthrough_target_socket(const char* n = "passthrough_target_socket");
    void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
    void register_b_transport(
      MODULE* mod,
      void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));
    void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(transaction_type&));
    void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
  };
} // namespace tlm_utils
```

5.3.2.3 規則

a) シンプル・イニシエータ・ソケット、シンプル・ターゲット・ソケットまたはパススルー・ターゲット・ソケットは、階層バインドできない。コールバック関数登録によって、入ってきたインタフェース・メソッドだけを実行する。

b) シンプル・イニシエータ・ソケットは、bind()か operator()どちらかの呼び出しによって、正確に同じ動作をするよう、シンプル・ターゲット・ソケットにバインドできる。

c) nb_transport_fw が登録されているシンプル・ターゲット・ソケットでは、b_transport コールバック関数の登録は強制されない。入力された b_transport 呼び出しは、自動的に登録されている nb_transport_fw 呼び出しに変換される。(5.3.2.4 シンプル・ターゲット・ソケット b/nb 変換を参照)

d) b_transport が登録されているシンプル・ターゲット・ソケットでは、nb_transport_fw コールバック関数の登録は強制されない。入力された nb_transport_fw 呼び出しは、自動的に登録されている b_transport 呼び出しに変換される。

e) ターゲットのシンプル・ターゲット・ソケットに b_transport も nb_transport_fw も登録されていない場合、インタフェース・メソッドの応答は、実行時エラーになる。(コンパイルエラーにはならない)

f) ターゲットは、b_transport と nb_transport_fw コールバック関数をパススルー・ターゲット・ソケットに登録するべきである。もし、そうしないとインタフェース・メソッドの応答時に、実行時エラーとなる。

g) 入力された transport_dbg 呼び出しに対し 0 の値をリターンする場合、ターゲットはシンプル・ターゲット・ソケットかパススルー・ターゲット・ソケットに transport_dbg コールバック関数の登録は強制されない。

h) 入力された get_direct_mem_ptr が false 値をリターンする場合、ターゲットはシンプル・ターゲット・ソケットかパススルー・ターゲット・ソケットに get_direct_mem_ptr コールバック関数の登録は強制されない。

i) イニシエータは、nb_transport_bw コールバック関数をシンプル・イニシエータ・ソケットに登録するべきである。もし、そうしないと nb_transport_bw メソッドの応答時に、実行時エラー

となる。
j) イニシエータは、invalidate_direct_mem_ptr を使わないのであれば、シンプル・イニシエータ・ソケットで invalidate_direct_mem_ptr の登録を強制されない。

5.3.2.4 シンプル・ターゲット・ソケット b/nb 変換

a) b_transport か nb_transport_fw メソッドが simple_target_socket クラスのソケットを通して呼ばれて、どんな対応するコールバック関数も登録されていない場合、シンプル・ターゲット・ソケットは、２つのインタフェースを結ぶアダプタのように振舞う。
b) シンプル・ターゲット・ソケットがアダプタとして機能する時、イニシエータの観点から、また、ターゲットにある b_transport か nb_transport_fw メソッドをインプリメントした観点から、基本プロトコルの規則を尊重する。
c) ソケットは、インタフェース・メソッドに何ら変更を加えなければ、トランザクション・オブジェクトを通り抜けるだけで、新しいトランザクション・オブジェクトを生成しない。
d) ターゲットに登録されているのが、nb_transport_fw コールバック関数だけの場合、イニシエータからコールされた b_transport メソッドがまだ進行中に、イニシエータが、nb_transport_fw と呼ぶことが許可されない。これはシンプル・ターゲット・ソケットの現在のインプリメンテーションの制限である。

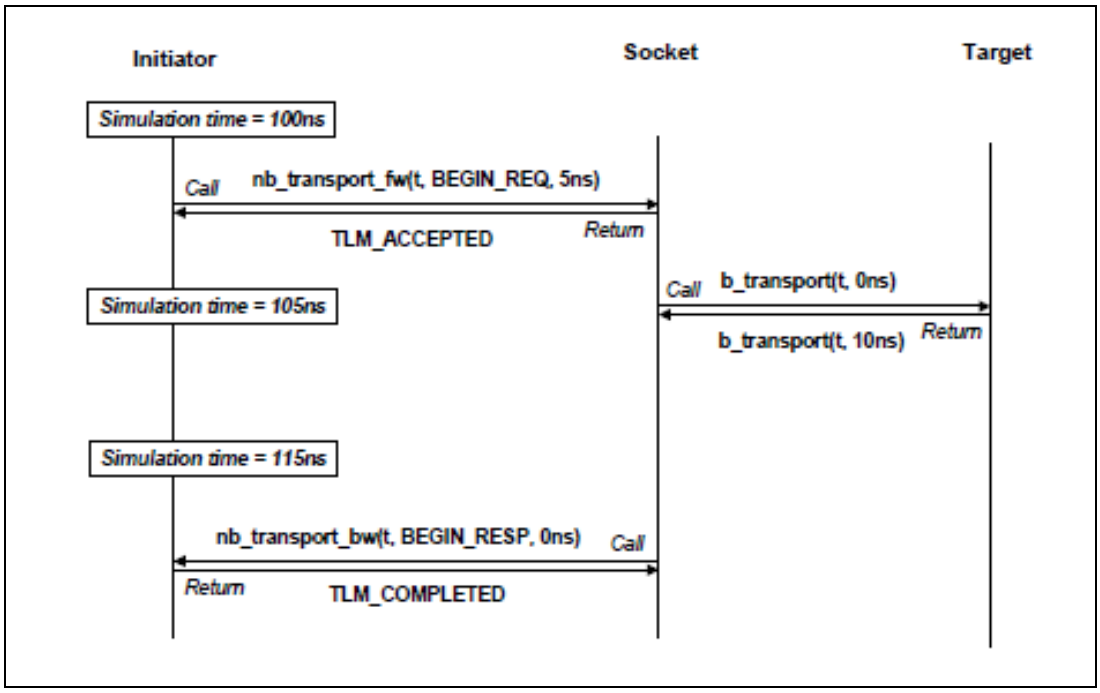


図 12 Simple target socket nb/b adapter

e) 図 12 は、イニシエータがどう nb_transport_fw メソッドをコールする様子を示している。イニシエータは nb_transport_fw メソッドをコールするが、ターゲットは、シンプル・ターゲットケットに b_transport コールバック関数が登録されているだけである。イニシエータは BEGIN_REQ を送る。そして、ソケットは TLM_ACCEPTED をイニシエータにリターンする。ソケットは、次に、b_transport をコールして、リターンされたら、BEGIN_RESP をイニシエータにリターンする。最後は、イニシエータは TLM_COMPLETED をソケットにリターンする。SystemC では、ノンブロッキング・メソッドから直接ブロッキング・メソッドを呼ぶのが許されない。したがって、ソケットは、nb_transport_fw メソッドから b_transport 関数をコールするのではなく、別のプロセスから b_transport をコールする。
f) 図 12 は 1 つの可能なシナリオを示している。他の例は、図の一番最後の遷移で、イニシエータは TLM_COMPLETED ではなく、TLM_ACCEPTED をリターンした場合である。その場合、ソケットはイニシエータからこの後に END_RESP を受け取ると予想される。また、ターゲットは、b_transport が呼ばれた場合、wait を発行する場合もある。
g) 図 13 は、イニシエータは、b_transport メソッドをコールする様子を示している。イニシエータは、b_transport メソッドをコールするが、ターゲットでは、シンプル・ターゲット・ソケットに nb_transport_fw コールバック関数が登録されている。イニシエータは、b_transport メソッドをコールする。そして、ソケットとターゲットは、基本プロトコルの規定に従って、

nb_transport_fw を使いハンドシェークする。ターゲットは、END_RESP を送るかもしれない。あるいは、いきなり BEGIN_RESP フェーズまでジャンプするかもしれない。ソケットは、コールされた nb_transport_bw が BEGIN_RESP であるのを見て TLM_COMPLETED をリターンする。

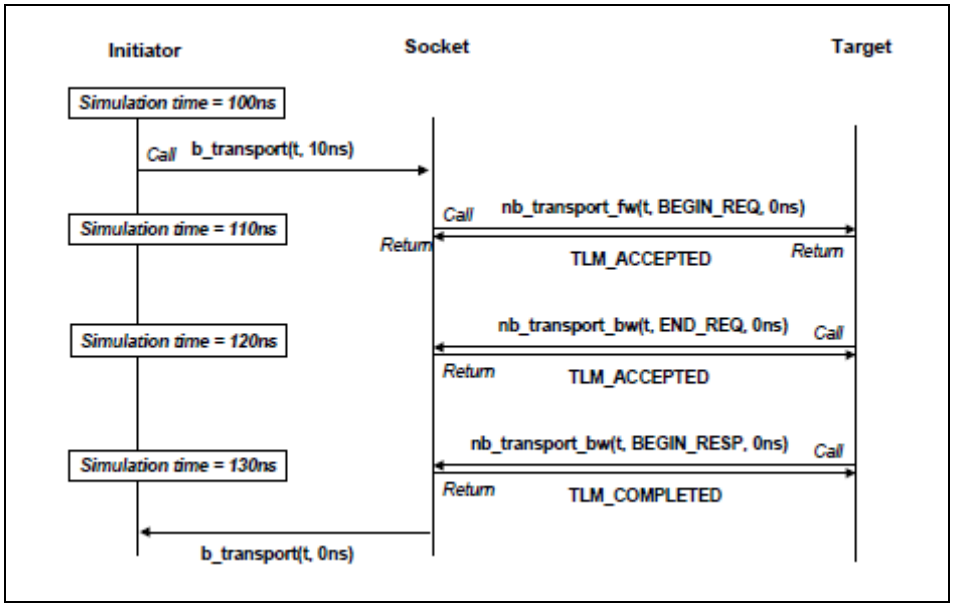


図 13 Simple target socket b/nb adapter

例：

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h" // Header files from utilities
#include "tlm_utils/simple_target_socket.h"
struct Initiator: sc_module
{
  tlm_utils::simple_initiator_socket<Initiator,32,tlm::tlm_base_protocol_types> socket;
  SC_CTOR(Initiator)
  : socket("socket") // Construct and name simple socket
  { // Register callbacks with simple socket
    socket.register_nb_transport_bw( this, &Initiator::nb_transport_bw );
    socket.register_invalidate_direct_mem_ptr( this, &Initiator::invalidate_direct_mem_ptr );
  }
  virtual tlm::tlm_sync_enum nb_transport_bw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
      return tlm::TLM_COMPLETED; // Dummy implementation
  }
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,sc_dt::uint64 end_range)
  { } // Dummy implementation
};
struct Target: sc_module // Target component
{
  tlm_utils::simple_target_socket<Target, 32, tlm::tlm_base_protocol_types> socket;
  SC_CTOR(Target)
  : socket("socket") // Construct and name simple socket
  { // Register callbacks with simple socket
    socket.register_nb_transport_fw( this, &Target::nb_transport_fw );
    socket.register_b_transport( this, &Target::b_transport );
    socket.register_get_direct_mem_ptr( this, &Target::get_direct_mem_ptr );
    socket.register_transport_dbg( this, &Target::transport_dbg );
  }
  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
  { } // Dummy implementation
  virtual tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
      return tlm::TLM_ACCEPTED; // Dummy implementation
  }
```

```
    virtual bool get_direct_mem_ptr( tlm::tlm_generic_payload& trans,tlm::tlm_dmi& dmi_data)
    { return false; } // Dummy implementation
    virtual unsigned int transport_dbg(tlm::tlm_generic_payload& r)
    { return 0; } // Dummy implementation
  };
  SC_MODULE(Top)
  {
    Initiator *initiator;
    Target *target;
    SC_CTOR(Top) {
      initiator = new Initiator("initiator");
      target = new Target("target");
      initiator->socket.bind( target->socket ); // Bind initiator socket to target socket
    }
  };
```

5.3.3 タグ付きシンプル・ソケット

5.3.3.1 イントロダクション

タグ付きシンプル・ソケットは、シンプル・ソケットの一種である。タグ付きシンプル・ソケットは、コールバックが届いた時、どのソケットに到着したかを特定できる整数 id をタグとして持っている。これは、同じコールバック・メソッドが複数のイニシエータ・ソケットかマルチ・ターゲット・ソケットに登録される場合で役に立つ。

5.3.3.2 クラス定義

```
namespace tlm_utils {
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_initiator_socket_tagged : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    explicit simple_initiator_socket_tagged(const char* n = "simple_initiator_socket_tagged");
    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&),
      int id);
    void register_invalidate_direct_mem_ptr(
      MODULE* mod,
      void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64),
      int id);
  };
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef tlm::tlm_fw_transport_if<TYPES> fw_interface_type;
    typedef tlm::tlm_bw_transport_if<TYPES> bw_interface_type;
    typedef tlm::tlm_target_socket<BUSWIDTH, TYPES> base_type;
    explicit simple_target_socket_tagged(const char* n = "simple_target_socket_tagged");
    tlm::tlm_bw_transport_if<TYPES> * operator ->();
    void register_nb_transport_fw(
```

```
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
    int id);
  void register_b_transport(
    MODULE* mod,
    void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
    int id);
  void register_transport_dbg(
    MODULE* mod,
    unsigned int (MODULE::*cb)(int id, transaction_type&),
    int id);
  void register_get_direct_mem_ptr(
    MODULE* mod,
    bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
    int id);
};
template <
  typename MODULE,
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm::tlm_base_protocol_types
 >
class passthrough_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
  typedef typename TYPES::tlm_payload_type transaction_type;
  typedef typename TYPES::tlm_phase_type phase_type;
  typedef tlm::tlm_sync_enum sync_enum_type;
  explicit passthrough_target_socket_tagged(const char* n = "passthrough_target_socket_tagged");
  void register_nb_transport_fw(
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
    int id);
  void register_b_transport(
    MODULE* mod,
    void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
      int id);
  void register_transport_dbg(
    MODULE* mod,
    unsigned int (MODULE::*cb)(int id, transaction_type&),
    int id);
  void register_get_direct_mem_ptr(
    MODULE* mod,
    bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
    int id);
};
} // namespace tlm_utils
```

5.3.3.3 規則

a) タグ付きシンプル・ソケットは、int id タグが別な引数として追加されるだけで、タグ付けをされていないシンプル・ソケットと全く同じ振る舞いをする。

b) 与えられたコールバック・メソッドは、異なったタグを使用する複数のソケットとして登録できる。

c) タグ(int id)は、コールバック関数登録時のインタフェース・メソッドの最後のアーギュメントになる。しかし、ソケットは、コールバック関数実装の本体には最初のアーギュメントにこのタグを付ける。

d) タグ付きシンプル・ソケットは、マルチソケットではない。

e) タグ付きシンプル・ソケットは、マルチソケットや他のコンポーネントにバインドすることができない。(5.3.4 マルチソケットを参照)

5.3.4 マルチソケット

5.3.4.1 イントロダクション

マルチソケットは、タグ付きシンプル・ソケットの一種である。マルチソケットは、単一のソケ

ットが他のコンポーネントの複数のソケットに割り当てられることを許可する。タグ付きシンプル・ソケットはどのソケットを通して呼び出されたのか特定可能であるが、マルチソケットのコールバックもマルチポートのインデックス番号をタグとして、インタフェース・メソッド呼び出しが別のコンポーネントのどのソケットから到着するかを特定できる。また、マルチソケットは、他の便利ソケットと異なり、イニシエータ、ターゲット双方で、子から親へのソケット階層バインドをサポートする。TLM-2.0 キットのマルチソケットのインプリメンテーションはブーストライブラリを使用している。ユーザは、www.boost.org からブーストライブラリをダウンロードして、コンパイラのインクルードパスに適切なディレクトリを加える必要がある。

5.3.4.2 クラス定義

```
namespace tlm_utils {
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
  class multi_passthrough_initiator_socket : public multi_init_base< BUSWIDTH, TYPES, N, POL>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef multi_init_base<BUSWIDTH, TYPES, N, POL> base_type;
    typedef typename base_type::base_target_socket_type base_target_socket_type;
    multi_passthrough_initiator_socket(const char* name);
    ~multi_passthrough_initiator_socket();
    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&));
    void register_invalidate_direct_mem_ptr(
      MODULE* mod,
      void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64));
    // Override virtual functions of the tlm_initiator_socket:
    virtual tlm::tlm_bw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_bw_transport_if<TYPES> >& get_base_export();
    void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);
    // SystemC standard callback
    // multi_passthrough_initiator_socket::before_end_of_elaboration must be called from
    // any derived class
    void before_end_of_elaboration();
    // Bind multi initiator socket to multi initiator socket (hierarchical bind)
    void bind(base_type& s);
    void operator() (base_type& s);
    tlm::tlm_fw_transport_if<TYPES>* operator[](int i);
    unsigned int size();
  };
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
  class multi_passthrough_target_socket : public multi_target_base< BUSWIDTH, TYPES, N, POL>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
```

```
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef sync_enum_type
    (MODULE::*nb_cb)(int, transaction_type&, phase_type&, sc_core::sc_time&);
    typedef void (MODULE::*b_cb)(int, transaction_type&, sc_core::sc_time&);
    typedef unsigned int (MODULE::*dbg_cb)(int, transaction_type& txn);
    typedef bool (MODULE::*dmi_cb)(int, transaction_type& txn, tlm::tlm_dmi& dmi);
    typedef multi_target_base<BUSWIDTH, TYPES, N, POL> base_type;
    typedef typename base_type::base_initiator_socket_type base_initiator_socket_type;
    typedef typename base_type::initiator_socket_type initiator_socket_type;
    multi_passthrough_target_socket(const char* name);
    ~multi_passthrough_target_socket();
    void register_nb_transport_fw (MODULE* mod, nb_cb cb);
    void register_b_transport (MODULE* mod, b_cb cb);
    void register_transport_dbg (MODULE* mod, dbg_cb cb);
    void register_get_direct_mem_ptr(MODULE* mod, dmi_cb cb);
    // Override virtual functions of the tlm_target_socket:
    virtual tlm::tlm_fw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_fw_transport_if<TYPES> >& get_base_export();
    // SystemC standard callback
    // multi_passthrough_target_socket::end_of_elaboration must be called from any derived class
    void end_of_elaboration();
    void bind(base_type& s);
    void operator() (base_type& s);
    tlm::tlm_bw_transport_if<TYPES>* operator[] (int i);
    unsigned int size();
  };
} // namespace tlm_utils
```

5.3.4.3 規則

a) マルチバインド能力と int id(タグ)の解釈は別として、multi_passthrough_initiator_socket は simple_initiator_socket_tagged と同様に振舞う。そして、multi_passthrough_target_socket もまた passthrough_target_socket_tagged と同様に振舞う。

b) クラス multi_passthrough_initiator_socket は、複数のターゲット・ソケットにバインドする１つのイニシエータ・ソケットとして、multi_passthrough_target_socket は、複数のイニシエータ・ソケットにバインドする１つのターゲット・ソケットとして振舞う。この２つクラステンプレートは、バインド数とバインドポリシーを定義するテンプレート・パラメータを持つ。このパラメータは、sc_port テンプレートインスタンスをパラメタライズするクラスインプリメンテーションとして使われる。

c) multi_passthrough_initiator_socket は、異なる multi_passthrough_initiator_socket と階層バインドできる。multi_passthrough_target_socket は、異なる multi_passthrough_target_socket と階層バインドできる。

d) バインディング・オペレータは、直接 initiator-socket-to-target-socket で使える。言い換えれば、multi_passthrough_target_socket には、クラスの tlm_target_socket と simple_target_socket と異なって、ターゲット・ソケットをイニシエータ・ソケットに割り当てる演算子がない。

e) もし、multi_passthrough_initiator_socket か multi_passthrough_target_socket が複数回バインドされていた場合、メソッド operator [] を、 ソケットにバインドしている応答オブジェクトにあてはめることができる。 インデックス値は、bind メソッドか operator()がソケットをバインドする為に呼ばれた順番で決まる。これと同じインデックス値は、コールバックで id タグを決めるのに使わる。

f) 例えば、multi_passthrough_initiator_socket が２つの別なターゲットにバインドされている場合を考えてみる。

```
socket[0]->nb_transport_fw(...)
socket[1]->nb_transport_fw()
```

は２つのターゲットに割り当てられる。これら２つのターゲットからの nb_transport_bw()メソッド呼び出しはそれぞれ 0 と 1 のタグを持つ。

g) メソッドサイズは、現在接続されているマルチソケットのインスタンス数をリターンする。SystemC のマルチポートのように、サイズがエラボレーション中すなわち、end_of_elaboration

コールバックの前の値なら、その値は、インプリメンテーション定義の値をリターンするはずだ。なぜなら、ポートがバインドされている時間は、インプリメンテーション定義だからである。

6. 汎用ペイロード

6.1 イントロダクション

汎用ペイロードは MMB モデルの相互利用性を高めることを目的に、モデル間の通信に必要となるアトリビュート(コマンド、アドレス、データなど) を定義したクラスである。汎用ペイロードは全ての MMB プロトコルで必要となるアトリビュートを含んでいるわけではない。しかし、そのための拡張メカニズムが用意されている。拡張されたとしても汎用ペイロードは共通のベース型となり、異なるプロトコルを繋ぐブリッジ開発の工数低減に寄与する。
汎用ペイロードはイニシエータ・ソケット/ターゲット・ソケットの両方で使用されることが奨められる。汎用ペイロードをベースとしているか否かで異なるプトロコル型のチェックを行うことが出来る。ブロッキングとノンブロッキング・トランスポート・インタフェースの両方と共に汎用ペイロードを使用できる。また、ダイレクト・メモリ、デバック・トランスポート・インタフェース共に汎用ペイロードを使用できる。その場合、制限されたアトリビュートセットだけが使用される。

6.2 拡張性と相互利用性

3 つの推奨する拡張手法がある。これらは blocking、non-blocking transport I/F のトランザクションテンプレート引数 TRANS と複数の I/F が組合されて定義された TYPES を置き換えることで実現するものである。

実現方法	TRANS トランザクション型	TYPES プロトコル型
A	tlm_generic_payload	tlm_generic_payload_types
B	tlm_generic_payload	ユーザ定義型
C	ユーザ定義型	ユーザ定義型

A→B→C の順に相互利用性が損なわれる。これら 3 つの方法で実現したモデルは、一つのシステムで混在されることも考えられている。

6.2.1 無視可能な拡張を含む、Generic Payload（汎用ペイロード）を直接利用

- トランザクション型は tlm_generic_payload を、プロトコル型は tlm_generic_payload_types を指定する
- 拡張部分は"無視可能"とする。"無視可能"とはインターコネクト/ターゲット共この拡張に関してエラーや転送失敗などを起こさないこと
- コンパイル時に型のチェックを行うことが出来る

6.2.2 tlm_generic_payload の型を含む新しいプロトコル型の定義

- トランザクション型は tlm_generic_payload であるが、プロトコル型は tlm_base_protocol_types を用いず、独自で定義した型を利用する
- コンパイル時の型チェックは行える
- 拡張されたアトリビュートを"無視可能"と実現/義務化することは出来るが意味定義には十分な注意が必要
- 推奨される 2 つの利用パターンがある
- イニシエータ/インターコネクト/ターゲット全てが共通に同じ新しいプロトコル型を使用
- 末端だけが新しいプトロコルで、その間のモデルは generic_base_protocol_types を使用
- この拡張は、各モデルがそれを知る/知らないに関わらず正しく転送が行えるよう考慮が必要

6.2.3 新しいプロトコル型とトランザクション型の定義

- トランザクション型、プロトコル型共新しい型を定義/利用する
- この手法は実現するプロトコルと汎用ペイロードが著しくことなる場合に利用する
- 相互利用性を考えるとこの手法よりは、前 2 つの手法を推奨する

6.3 Generic Payload（汎用ペイロード）アトリビュートとメソッド

汎用ペイロードクラスはプライベートなアトリビュートとそれにアクセスするパブリックはメソッドを持つ。各アトリビュートの変更は原則イニシエータが行うが、アドレスやステータスなどはインターコネクトやターゲットが変更する場合がある。リードコマンドにおけるデータ配列はターゲットが更新する。

6.4 クラス定義

```
namespace tlm {
```

```
class tlm_generic_payload;
class tlm_mm_interface {
public:
  virtual void free(tlm_generic_payload*) = 0;
  virtual ~tlm_mm_interface() {}
};
class tlm_extension_base
{
public:
  virtual tlm_extension_base* clone() const = 0;
  virtual void free() { delete this; }
  virtual void copy_from(tlm_extension_base const &) = 0;
protected:
  virtual ~tlm_extension_base() {}
};
template <typename T>
tlm_extension : public tlm_extension_base
{
public:
  virtual tlm_extension_base* clone() const = 0;
  virtual void copy_from(tlm_extension_base const &) = 0;
  virtual ~tlm_extension() {}
  const static unsigned int ID;
};
enum tlm_command {
  TLM_READ_COMMAND,
  TLM_WRITE_COMMAND,
  TLM_IGNORE_COMMAND
};
enum tlm_response_status {
  TLM_OK_RESPONSE = 1,
  TLM_INCOMPLETE_RESPONSE = 0,
  TLM_GENERIC_ERROR_RESPONSE = -1,
  TLM_ADDRESS_ERROR_RESPONSE = -2,
  TLM_COMMAND_ERROR_RESPONSE = -3,
  TLM_BURST_ERROR_RESPONSE = -4,
  TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
};
#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff
class tlm_generic_payload {
public:
  // Constructors and destructor
  tlm_generic_payload();
  explicit tlm_generic_payload( tlm_mm_interface* );
  virtual ~tlm_generic_payload();
private:
  // Disable copy constructor and assignment operator
  tlm_generic_payload( const tlm_generic_payload& );
  tlm_generic_payload& operator= ( const tlm_generic_payload& );
public:
  // Memory management
  void set_mm( tlm_mm_interface* );
  bool has_mm();
  void acquire();
  void release();
  int get_ref_count();
  void reset();
  void deep_copy_from( const tlm_generic_payload& ) const;
  void update_extensions_from(const tlm_generic_payload & );
  void free_all_extensions();
  // Access methods
```

```
    tlm_command get_command() const;
    void set_command( const tlm_command );
    bool is_read();
    void set_read();
    bool is_write();
    void set_write();
    sc_dt::uint64 get_address() const;
    void set_address( const sc_dt::uint64 );
    unsigned char* get_data_ptr() const;
    void set_data_ptr( unsigned char* );
    unsigned int get_data_length() const;
    void set_data_length( const unsigned int );
    unsigned int get_streaming_width() const;
    void set_streaming_width( const unsigned int );
    unsigned char* get_byte_enable_ptr() const;
    void set_byte_enable_ptr( unsigned char* );
    unsigned int get_byte_enable_length() const;
    void set_byte_enable_length( const unsigned int );
    // DMI hint
    void set_dmi_allowed( bool );
    bool is_dmi_allowed() const;
    tlm_response_status get_response_status() const;
    void set_response_status( const tlm_response_status );
    std::string get_response_string();
    bool is_response_ok();
    bool is_response_error();
    // Extension mechanism
    template <typename T> T* set_extension( T* );
    tlm_extension_base* set_extension( unsigned int , tlm_extension_base* );
    template <typename T> T* set_auto_extension( T* );
    tlm_extension_base* set_auto_extension( unsigned int index , tlm_extension_base* );
    template <typename T> void get_extension( T*& ) const;
    template <typename T> T* get_extension() const;
    tlm_extension_base* get_extension( unsigned int ) const;
    template <typename T> void clear_extension( const T* );
    template <typename T> void clear_extension();
    template <typename T> void release_extension(T* ext);
    template <typename T> void release_extension();
    void resize_extensions();
  };
} // namespace tlm
```

6.5 汎用ペイロード・メモリ管理

a) イニシエータは既存ストレージにデータ・ポインタ、バイトイネーブルポインタ・アトリビュートを設定する。ストレージはスタティック、オートマティック（stack)、ダイナミック（new)で確保されている。トランザクションのライフタイムが完了するまで、イニシエータはこのストレージを削除しない。汎用ペイロードのデストラクタはこれら 2 つのアレイを削除しない。

b) この節に関連して 6.20 の汎用ペイロード拡張を読むこと。

c) 汎用ペイロードはメモリ管理への 2 つの異なったアプローチをサポートする。1 つは明白なメモリ・マネージャ、もう一つはイニシエータによる臨時のメモリ管理。混在は可能である。いずれのアプローチもトランザクション・オブジェクトとその拡張の両方を管理すること。

d) メモリ・マネージャは抽象的な基底クラス tlm_mm_interface の free メソッドを実行するユーザによって定義されたクラスである。そのクラスは汎用ペイロード・トランザクション・オブジェクトを割り当てるためにメソッドを提供する

e) メソッド set_mm、acquire、release、get_ref_count、reset はメモリ管理に存在する。汎用ペイロードのオブジェクトはメモリ管理セットを持っていない

f) イニシエータによる臨時のメモリ管理の場合、TLM-2 コア・インタフェース呼び出しの前にイニシエータはトランザクション・オブジェクトのためのメモリを割り当てる。また、呼出し後にイニシエータはトランザクション・オブジェクトとその拡張オブジェクトを削除あるいはプールする。

g) 汎用ペイロードがブロッキング・トランスポート・インタフェース、ダイレクト・メモリ・インタフェースまたはデバッグ・トランスポート・インタフェースと共に使用されるとき、いずれのアプローチも使用される。イニシエータによる臨時のメモリ管理は十分である。メモリ・マネージャが不在のとき b_transport、get_direct_mem_ptr、または transport_dbg メソッドが、トランザクション・オブジェクトとその拡張はリターンの時に無効にされるか、または削除されると仮定するべきである。
h) 汎用ペイロードがノンブロッキング・トランスポート・インタフェースと共に使用されるとき、メモリ・マネージャは使用されるものとする。これは呼び出し元関数がイニシエータ、インターコネクト・コンポーネント、またはターゲットであることにかかわらず適用される。
i) なにも既に存在していないなら、ブロッキングからノンブロッキングへのトランスポートアダプターはメモリ・マネージャを設定しなければならない
j) メモリ・マネージャを使用するとき、ヒープ(new か malloc で)からトランザクション・オブジェクトとその拡大オブジェクトも割り当てるものとする。
k) 臨時のメモリ管理を使用するとき、トランザクション・オブジェクトとその拡大オブジェクトはヒープまたはスタックに割り当てられるものとする。スタックに割り当てられた時、拡大オブジェクトのメモリ管理によって取られる必要があることに注意を払うこと。
l) メソッド set_mm は汎用ペイロード・オブジェクトのメモリ・マネージャを設定するものとする。引数として、アドレスが通過される。引数が null である場合、既存のメモリ・マネージャはトランザクション・オブジェクトから切り離される。また、それ自体は、削除されない。メモリ・マネージャと参照が 0 以上を数えるトランザクション・オブジェクトのために set_mm を呼ばないものとする。
m) メモリ・マネージャが用意できていた場合にだけ、メソッド has_mm が true を返すものとする。nb_transport メソッドのボディーから呼ばれると、has_mm は true を返すはずである。
n) b_transport、get_direct_mem_ptr、または transport_dbg メソッドのボディーから呼ばれると、has_mm は true か false を返す。インターコネクト・コンポーネントは、has_mm と呼んで、トランザクションにはメモリ・マネージャがいるかどうかによる適切な行動を取る。さもなければ、インターコネクト・コンポーネントはメモリ・マネージャ(例えば、ヒープ割り付け)がいるトランザクションのすべての義務を引き受けるものとする。そして、メモリ・マネージャ(例えば、acquire)の存在を要求するメソッドのいずれも呼ばないだろう。
o) それぞれの汎用ペイロード・オブジェクトは参照カウントを持つ。そのデフォルト値はゼロ。
p) メソッド acquire は参照カウントの値を増加するものとする。メモリ・マネージャが不在のとき、acquire が呼ばれると、ランタイムエラーが発生する。
q) メソッド release は参照カウントの値を減少させるものとする。参照カウントの値が 0 となると、メモリ・マネージャ・オブジェクトの free メソッド呼んで、引数としてトランザクション・オブジェクトのアドレスを渡す。メモリ・マネージャが不在のとき、acquire が呼ばれると、ランタイムエラーが発生する。
r) メソッド get_ref_count は参照カウントの値を返すものとする。メモリ・マネージャが不在のとき、返された値は 0 であるだろう。
s) メモリ・マネージャで、そのオブジェクトを最初にインタフェース・メソッド呼び出しの引数のとして、渡す前に、各イニシエータは、それぞれのトランザクション・オブジェクトの acquire メソッドを呼ぶ。そして、オブジェクトはもう必要でないときに、そのトランザクション・オブジェクトの release メソッドを呼ぶべきである。
t) メモリ・マネージャで、現在のインタフェース・メソッド呼び出しを超えてトランザクション・オブジェクトのライフタイムを延長するときは、各インターコネクト・コンポーネントとターゲットは acquire メソッドを呼ぶべきである。そして、オブジェクトはもう必要でないときに release メソッドに呼ぶ。
u) 解析目的のためにトランザクション・オブジェクトのライフタイムを無期限に広げるのならば、各インターコネクト・コンポーネントとターゲットは参照カウント・メカニズムを使用するよりもトランザクション・オブジェクトのクローンを作るべきである。言い換えれば、プロトコルの正常なフェーズを超えてトランザクション・オブジェクトのライフタイムを広げるのに参照カウントを使用するべきでない。
v) メモリ・マネージャで、参照カウントが、イニシエータ自体以外のどんなコンポーネントにもトランザクション・オブジェクトの参照がまだないのを示すまで、トランザクション・オブジェ

クトを新しいトランザクションを表すのに再使用されないものとするか、異なったインタフェースと共に再使用しないものとする。参照カウントが 1 と等しくなるまで、イニシエータは、トランザクション・オブジェクトのために acquire を呼ぶことを仮定する。このルールは、同じインタフェースのトランザクションの再利用、トランスポートの転送、ダイレクト・メモリ、デバックトランスポート・インターフェイスの時、適用される。
w) メソッド reset は自動削除のためにマークされた拡張を削除して、対応する拡張へのポインタを null に設定する。各拡張は拡張オブジェクトのメソッド free と呼ぶことによって、削除されるものとする。ユーザが拡張オブジェクトのための明白なメモリ管理を提供したいなら、多分 free メソッドをオーバーロードできる。
x) 拡張オブジェクトは set_extension コールで追加され、release_extension コールで削除される。clear_extension コールは拡張のポインタをクリアし、拡張オブジェクトを削除しない。トランザクション・オブジェクトがメモリ・マネージャなしでスタックに割り当てられて、且つ拡張オブジェクトがプールになる場合、この後者の振舞いが必要である。
y) メモリ・マネージャが不在の場合、b_transport、get_direct_mem_ptr、または transport_dbg からコントロールを返す前に、コンポーネントが割り当てるあるいは与えられた拡張にセットするのいずれの場合でも、その同じ拡張を削除またはクリアする。たとえば、b_transport で実装され、set_mm にてトランザクション・オブジェクトをメモリ・マネージャに追加した場合のインターコネクト・コンポーネントは、トランザクション・オブジェクトとその拡張が削除されるまで、b_transport から返らない。(下流コンポーネントも同様に拡張が既に削除されていることを仮定する)
z) メモリ・マネージャが存在する場合、拡張は set_auto_extension コールで追加され、メモリ・マネージャで自動的に削除あるいはプールされる。set_extension コールで追加された拡張はスティッキー拡張と呼ぶ。スティッキー拡張とは、トランザクション参照カウントが 0 の場合でも、自動的に削除されないものを指す。
aa) メモリ・マネージャの存在の有無が分からない場合、拡張は set_extension コールで追加、release_extension コールで削除すべきである。この呼出し手順はメモリ・マネージャの存在の如何によらず安全である。この状況は has_mm コールしないインターコネクト・コンポーネントかターゲットの中で起こることができる。(イニシエータの中では、常に、メモリ・マネージャの存在の有無は分かっている、且つ has_mm コールはメモリ・マネージャの有無に関わらず常に自明である)
bb) メソッド free_all_extensions は自動削除のためにマークした他を含むすべての拡張を削除して、対応する拡張ポインタを null に設定する。各拡張は拡張オブジェクトの free メソッド・コールで削除される。ユーザが拡張オブジェクトのための明白なメモリ管理を提供したいなら、多分 free メソッドをオーバーロードできる。
cc) Free_all_extensions はメモリ・マネージャを使用しないでプールされたトランザクション・オブジェクトから拡張を削除する時に便利である。メモリ・マネージャを使う時、自動削除のためにマークされた拡張は削除される。スティッキー拡張は削除されない。
dd) deep_copy_from メソッドは、別のトランザクション・オブジェクトをコピーすることによって、現在のトランザクション・オブジェクトのアトリビュートと拡張を変更するものとする。もし２つの（コピー元とコピー先）トランザクションのポインタが non-null ならば、データとバイト・イネーブル配列は deep コピーされる。アプリケーションは現在のトランザクションで配列が十分に大きいことを確認することを保証する。他のトランザクションの拡張が現在のトランザクションに存在しているならば、拡張クラスの copy_from メソッドのコールでコピーされる。さもなければ、新しい拡張オブジェクトは拡張クラスの clone メソッド・コールで作成され、現在のトランザクションにセットされる。クローニングの場合で、もしメモリ・マネージャが現在のトランザクションのために存在するならば、新しい拡張は自動削除のマークがされる。
ee) 言い換えれば、メモリ・マネージャが存在する場合で、deep_copy_from は現在のオブジェクトに存在しない新しい拡張の全てに自動削除をマークする。メモリ・マネージャが存在しない場合は、全ての拡張は、スティッキー拡張である。
ff) メソッド update_extensions_from は、別のトランザクション・オブジェクトからそれらの現在のオブジェクトに存在している拡張をコピーすることによって、現在のトランザクション・オブジェクトの拡張を変更するものとする。拡張は拡張クラスの copy_from メソッド・コールによってコピーされる。

gg) Deep_copy_from と update_extensions_from の一般的なユースケースはターゲット・ソケットを通して到着するトランザクション・オブジェクトを deep コピーして、イニシエータ・ソケットを通してコピーを出すことである。そして（backward パスあるいは return パスをつかって）トランザクションが返る時、最初のトランザクション・オブジェクトに拡張をコピーバックするために update_extensions_from をコールする。下流で追加された拡張は無視される。
hh) これらの義務は汎用ペイロードに適用される。原則として、同様の義務は汎用ペイロードに関係ないトランザクション・タイプにも適用される。

6.6 コンストラクタ、代入、デストラクタ

a) デフォルトコンストラクタは、汎用ペイロード・アトリビュートにデフォルト値を設定する。
b) コンストラクタ tlm_generic_payload(tlm_mm_interface*)はそれらのデフォルト値に汎用ペイロード・アトリビュートを設定して、アドレスが引数として通過されるオブジェクトに汎用ペイロード・オブジェクトのメモリ・マネージャを設定する。
c) コピーコンストラクタと代入演算は不可である。
d) 仮想のデストラクタ~tlm_generic_payload はすべての拡張を削除するものとする（自動削除のためにマークした他を含む)。各拡張は拡張オブジェクトの free メソッド・コールで削除される。デストラクタはデータ配列、バイト・イネーブル配列を削除しない。

6.7 デフォルト値と標準的なアトリビュート

アトリビュート	デフォルト値	インターコネクトでの変更	ターゲットでの変更
コマンド	TLM_IGNORE_COMMAND	No	No
アドレス	0	Yes	No
データ・ポインタ	0	No	No
データ配列	-	No	Yes(read cmd)
データ長	0	No	No
バイト・イネーブル・ポインタ	0	No	No
バイト・イネーブル配列	-	No	No
バイト・イネーブル長	0	No	No
ストリーミング幅	0	No	No
DMI 許可	False	Yes	Yes
レスポンス・ステータス	TLM_INCOMPLETE_RESPONSE	No	Yes
拡張ポインタ	0	Yes	Yes

6.8 コマンド・アトリビュート

- set_command メソッドは値渡しの引数でコマンド・アトリビュートを設定し、get_command メソッドは現在のコマンド・アトリビュートを値で返す。
- set_read メソッドと set_write メソッドはそれぞれ TLM_READ_COMMAND、TLM_WRITE_COMMAND をコマンド・アトリビュートに設定する。is_read メソッドと is_write メソッドはそれぞれ現在のコマンド・アトリビュートの値が TLM_READ_COMMAND、TLM_WRITE_COMMAND かどうかを判定しそれを返す。
- リードコマンドはコマンド・アトリビュートが TLM_READ_COMMAND と等しい汎用ペイロード・トランザクションであり、ライトコマンドはコマンド・アトリビュートが TLM_WRITE_COMMAND と等しい汎用ペイロード・トランザクションである。
- リードコマンドを受取った時、ターゲットはローカル配列の内容をデータ・ポインタ・アトリビュートとなるよう配列ポインタにコピーする。
- ライトコマンドを受取った時、ターゲットはデータ・ポインタ・アトリビュートにより指定されている配列をターゲット内のローカル配列にコピーする。
- もしターゲットがリードまたはライトコマンドを実行することが出来ない場合は、標準的なエラーレスポンスを生成する。推奨するレスポンス・ステータスは TLM_COMMAND_ERROR_RESPONSE。
- 汎用ペイロード・トランザクションのコマンド・アトリビュートが TLM_IGNORE_COMMAND であった場合、ターゲットはライトまたはリードコマンドを実行してはならない。
- コマンド・アトリビュートはイニシエータにより設定され、インターコネクトやターゲットは書き換えない。
- コマンド・アトリビュートのデフォルト値は TLM_IGNORE_COMMAND である。

6.9 アドレス･アトリビュート

a) set_address は、アドレス・アトリビュートに引数で与えられた値を設定し，get_address は

現在値を返す。
b) read コマンドや write コマンドに対して、ターゲットはアドレス：アトリビュートの現在値が read/write される連続するデータ・ブロックの最初のバイトを指しているものと解釈する。ただし、データ・ポインタで示される配列の先頭バイトを指しているかどうかは、ホスト計算機のエンディアンに依存する。
c) データ配列中の特定のバイトのアドレスは、アドレス・アトリビュート、配列インデックス、ストリーミング幅・アトリビュート、ホスト計算機のエンディアン、ソケット幅で決まる。6.17 エンディアン を参照のこと。
d) アドレス・アトリビュートの値は、ワード・アラインメントである必要はない。 （もしアドレス・アトリビュートがバイトで表現されるローカルソケット幅の倍数であれば、アドレス計算が大幅に単純化できる）
e) ターゲットが指定されたアドレス･アトリビュートでトランザクションを実行できない場合は、標準エラーを生成する。推奨のエラーコードは、TLM_ADDRESS_ERROR_RESPONSE。
f) アドレス・アトリビュートはイニシエータで設定されるが、一つ以上の接続コンポーネントにより上書きされることがある。上書きは、接続コンポーネントがメモリの絶対アドレスからターゲットが認識できる相対アドレスへ変換する場合などのために必要である。アドレス・アトリビュートが上書きされた場合、明示的に別の場所に退避しない限り、上書き前の値は失われる。
g) アドレス・アトリビュートのデフォルト値は‘0’

6.10 データ・ポインタ･アトリビュート

a) set_data_ptr はデータ・ポインタ・アトリビュートに、引数で与えられた値を設定し、get_data_ptr は現在値を返す。データ・ポインタ・アトリビュートは、データ配列へのポインタで、ポインタ値の設定や取得をするもので、データ配列の内容の設定や取得をするものではない。
b) ターゲットは read/write コマンドで、それぞれデータのデータ配列からの読み出し/書き込みを行う。
c) データ配列用の領域はイニシエータが確保する。その領域は、イニシエータ内のレジスタ・ファイルやキャッシュ・メモリのようなデータ領域や、トランザクションレベル・インタフェースのデータ転送に使うテンポラリ･バッファである。
d) 一般には、汎用ペイロードのデータ配列の構成は、イニシエータやターゲットのローカル・レジスタの構成とは独立である。しかしほとんどの場合、汎用ペイロードはターゲットとのデータ・コピーは memcopy 一回で行うよう設計されるので、ターゲットのレジスタは汎用ペイロードと同じ構成であることを想定している。この前提はあくまでもシミュレーション速度のためであり、汎用ペイロードの表現能力を制約するものではない。ターゲットはデータアレイとのコピーについて任意のデータ変換をすることができる。
e) トランザクション･オブジェクトが Null(0)ポインタでトランスポート・インタフェースを呼び出すとエラーとなる
f) データ配列長はデータ長アトリビュートの値以上であること（単位はバイト）
g) データ・ポインタ・アトリビュートはイニシエータが設定するもので、他のコンポーネントやターゲットによって上書きされない
h) write コマンドや TLM_IGNORE_COMMAND では、データ配列はイニシエータが設定するもので、他のコンポーネントやターゲットによって上書きされない
i) read コマンドでは、データ配列はバイト・イネーブルの設定に従って、ターゲットで上書きされるが、他のコンポーネントから上書きはできない
j) データ・ポインタ・アトリビュートのデフォルト値は 0（null）

6.11 データ長アトリビュート

a) set_data_length はデータ長・アトリビュートに引数で与えられた値を設定し，get_data_length は現在値を返す。
b) read コマンドや write コマンドに対して、ターゲットはデータ長アトリビュートの現在値を、バイト・イネーブル・アトリビュートでディスエーブルされているバイトも含めた、コピーされるデータ･アレイのバイト数であると解釈する。
c) 値はイニシエータが設定して、他のコンポーネントやターゲットによって上書きされない
d) ‘０’に設定してはいけない。ゼロバイト転送の場合は、コマンド・アトリビュートを’TLM_IGNORE_COMMAND’とする
e) 相互利用性レイヤの標準ソケットクラスもしくはその派生クラスを使って、バースト転送をす

る場合、転送のワード長はソケットの BUSWIDTH テンプレート・パラメータで指定される。BUSWIDTH はデータ長アトリビュートとは独立で、ビット数で表される。もし、データ長が BUSWIDTH／８と同じか小さいとシングルワード転送を、それより大きいとバースト転送をトランザクションを効率的にモデリングしていることになる。一つのトランザクションを異なるバス幅のソケットで渡すことが出来る。BUSWIDTH は転送のレイテンシ計算に使用される。
f) ターゲットは、ターゲットのワード長より大きなデータ長のトランザクションをサポートするかもしれないし、サポートしないかもしれない。そのときのワード長は、BUSWIDTH テンプレート・パラメータで与えられるか、もしくは他の値で与えられる。
g) ターゲットは与えられたデータ長のトランザクションが実行できなかった場合は、標準エラーレスポンスを返して、データ配列の内容を変更してはいけない。推奨レスポンス・ステータスは、’TLM_BURST_ERROR_RESPONSE’
h) データ長アトリビュートのデフォルト値は’0’で、無効な値である。故に、データ長・アトリビュートはインタフェース・メソッド・コールでトランザクション・オブジェクトが渡される前に明示的に設定されるべきである。

6.12 バイト･イネーブル･ポインタ･アトリビュート

a) set_byte_enable_ptr で、バイト･イネーブル配列へのポインタに引数で与えられた値を設定し、get_byte_enable_ptr でバイト･イネーブル･ポインタ・アトリビュートの値を返す
b) バイト･イネーブル配列の各要素は次のように解釈される。’0’は、対応するバイトがディスエーブルされていることを示し、‘0xff’は対応するバイトがイネーブルされていることを示す。その他の値は未定義である。‘0xff’は、バイト・イネーブル配列がそのままマスクとして使えるように値が選ばれた。TLM_BYTE_DISABLED と TLM_BYTE_ENABLED の２つのマクロが用意されている
c) バイト・イネーブルは、それぞれのビートのアドレスの増加分がビートの最上位バイトより大きいときや、バスの選ばれたバイト･レーンに複数ワードを設定するバースト転送に使われる。より抽象度の高いレベルでは、データ配列に歯抜けがある状態でバースト転送を行う時に使用する
d) 小さなパターンを繰り返し使用する場合や、データ配列全体をカバーする大きなパターンがある場合に、byte enable mask が定義される。6.13 バイト･イネーブル長アトリビュートを参照のこと
e) バイト･イネーブル配列のエレメント数は、バイト･イネーブル長アトリビュートで与えられる
f) バイト･イネーブル･ポインタが’0’(null)に設定されている場合は、その転送ではバイト･イネーブルは使用されず、バイト･イネーブル長は無視される
g) バイト･イネーブルが使われる場合は、バイト･イネーブル･ポインタ･アトリビュートはイニシエータが設定する。バイト･イネーブル配列領域はイニシエータが確保して、バイト･イネーブル配列の内容もイニシエータが設定し、イネーブル配列の内容は他のコンポーネントやターゲットで上書きされない。
h) バイト・イネーブル・ポインタが null でないときは、ターゲットは以下に定義された動作を実装するか、もしくは標準エラー応答を生成する。推奨エラーは’TLM_BYTE_ENABLE_ERROR_RESPONSE’
i) write コマンドでは、データ配列のディスエーブルされたバイトのデータは接続コンポーネントやターゲットでは無視されなければならない。ディスエーブルされたバイトは、接続コンポーネントやターゲットの動作に影響を与えないことが推奨される。これらの無視されるバイトに対しては、イニシエータはどのような値を書き込んでも良い。
j) write コマンドで、ターゲットがトランザクション･データ配列からローカル配列へバイト単位のコピーをしている場合には、ターゲットは、汎用ペイロードのディスエーブル･バイトに対応するローカル配列のバイト値を書き換えてはならない。
k) read コマンドでは、データアレイ中のディスエーブルされたバイトの値を、接続コンポーネントやターゲットが変更してはならない。イニシエータは、これらのディスエーブルされたバイトは接続コンポーネントやターゲットが変更しないものとみなすことができる。
l) read コマンドでは、ターゲットがローカル配列からトランザクション･データ配列へバイト単位でコピーする場合、ターゲットは、汎用ペイロードのディスエーブル･バイトに対応するローカル配列の値は無視すべきである。
m) もし、アプリケーションでこれらの規定を破らなければならない場合や、本書の汎用ペイロードの規定を破らなければならない場合は、新しくプロトコル･タイプ･クラスを作ることを推奨す

る。6.2.2　tlm_generic_payload の型を含む新しいプロトコル型の定義を参照のこと
n) イネーブル･ポインタ･アトリビュートのデフォルト値は’0’。Null ポインタ

6.13 バイト･イネーブル長アトリビュート

a) set_byte_enable_length は、引数で与えられた値をバイト･イネーブル長アトリビュートに設定し、get_byte_enable_length は現在の値を返す
b) read/write コマンドに対して、ターゲットは、バイト･イネーブル長アトリビュートをバイト・イネーブル配列の要素数として解釈する
c) バイト･イネーブル長アトリビュートの値はイニシエータが設定して、接続コンポーネントやターゲットは上書きしない
d) データ配列のあるエレメントに適用するバイト・イネーブルは byte_enable_array_index=data_array_index%byte_enable_length で与えられる。つまり、バイト・イネーブル配列はデータ配列に繰り返し適用される
e) バイト･イネーブル長がデータ配列の長さより大きい場合、余分なバイト･イネーブルは read/write コマンドに影響しない。ただし、拡張に使用することはできる
f) バイト･イネーブル･ポインタが 0、つまり null pointer の場合、バイト・イネーブル長の値は接続コンポーネントやターゲットから無視される。もし、バイト･イネーブル･ポインタが’0’でなければ、バイト･イネーブル長は’0’ではない
g) ターゲットが、指定されたバイト・イネーブル長を使ってトランザクションが実行できないときは、エラーを出す。推奨は、TLM_BYTE_ENABLE_ERROR_RESPONSE
h) バイト･イネーブル長アトリビュートのデフォルト値は、’0’

6.14 ストリーミング幅アトリビュート

a) set_streaming_attribute は、引数の値をストリーミング幅アトリビュートに設定し、get_streaming_attribute は現在値を返す
b) read/write コマンドに対して、ターゲットはストリーミング幅の現在値に従って動作する
c) ストリーミングは、コンポーネントがデータ配列をどのように解釈するかに影響する。ストリームは、継続するビートで発生するデータ転送のシーケンスを構成し、各々のビートは汎用ペイロードのアドレス･アトリビュートで与えられる同じスタートアドレスを持つ。ストリーミング・幅アトリビュートはストリームの幅を決定し、それは、各々のビートで転送されるバイト数である。すなわち、データアレイの構成にストリーミングは影響しない。
d) データ配列中のバイトは、汎用ペイロード・トランザクションにアクセスしているコンポーネント内のローカルアドレスの順序に対応している。最下位アドレスはアドレス・アトリビュートで与えられる。最上位アドレスは、address_attribute+streaming_width-1 で与えられる。ターゲットにコピーされる各々のバイトのアドレスは、各々のビートのスタートでアドレス・アトリビュートの値として設定される。
e) データ配列の実装としては、ストリーミングの幅を持つ単一のトランザクションは、一連のトランザクションと機能的には等価である。そのトランザクションは、オリジナルトランザクションと同じアドレスで、オリジナルストリーミング幅と同じデータ長・アトリビュートをもち、それぞれのビートでオリジナルデータ配列の異なるサブセットをデータ配列としたものである。このサブセットは、オリジナルデータ配列のデータ順を維持したものである。
f) 0 のストリーミング幅は無効である。もし、ストリーミング転送が必要でなかったなら、ストリーミング幅アトリビュート は、データ長・アトリビュートと同じがそれ以上の値にすべきである。
g) ストリーミング幅アトリビュートはデータ配列の長さやデータ配列に格納されているバイト数には影響しない。
h) ストリーミング幅がソケットの幅と異なる場合には、幅変換の問題が生じる。　6.17 エンディアンを参照のこと
i) ターゲットが、指定されたストリーミング幅を使ってトランザクションが実行できないときは、標準エラーを出す。推奨は、TLM_BURST_ERROR_RESPONSE
j) ストリーミングは、バイト･イネーブルと組み合わせて使われるときがあり、その場合通常は、ストリーミング幅はバイト・イネーブル長 と同じ値になる。バイト･イネーブル長がストリーミング幅の倍数ならば、各々のビートで異なるバイト数がイネーブルされることを意味する
k) ストリーミング幅アトリビュートのデフォルト値は、0

6.15 DMI 許可アトリビュート

a) set_dmi_allowed は、DMI 許可アトリビュートに引数の値を設定し、get_dmi_allowed は、現在の値を返す

b) DMI 許可アトリビュートは、ダイレクト・メモリ・ポインタを取得できる可能性をイニシエータに示すもので、DMI 経由で現在のトランザクションが実行できた場合は、ターゲットが DMI 許可アトリビュートを’true’にセットする。4.2.7 4.2.7DMI ヒントを使った最適化を参照のこと

c) DMI 許可アトリビュートのデフォルト値は、false

6.16 応答ステータス・アトリビュート

a) set_response_status は、引数の値を response status attribute へ設定し, get_response_status は、現在値を返す

b) 応答ステータスの現在値が TLM_OK_RESPONSE の場合にのみ、is_response_ok は’true’を返す。 is_response_error は、応答ステータスの現在値が TLM_OK_RESPONSE と異なる場合にのみ’ture’を返す

c) get_response_string は現在の状態を文字列で返す

d) 原則としてターゲットは汎用ペイロードの全ての機能を実装することが推奨されているが、そのようになっていない場合はエラーを返す.6.16.1　標準エラー応答を参照のこと

e) 応答ステータス･アトリビュートには、イニシエータが TLM_INCOMPLETE_RESPONSE を設定して、ターゲットが上書きする。TLM_INCOMPLETE_RESPONSE は、トランザクションがターゲットに届いていないことを示すものなので、他の接続しているコンポーネントが上書きをしてはならない

f) ターゲットは処理が成功した場合には、応答ステータス･アトリビュートに TLM_OK_RESPONSE を設定して、そうでなければ、以下のテーブルにある 5 つのエラーレスポンスを設定してエラーであることを示す。ターゲットは、エラーの原因に応じて適切なエラーを選ぶべきである。

Error response	Interpretation
TLM_ADDRESS_ERROR_RESPONSE	アドレス:アトリビュートで与えられた値では動作できない、もしくはアドレス範囲を超えている
TLM_COMMAND_ERROR_RESPONSE	与えられたコマンドを実行できない
TLM_BURST_ERROR_RESPONSE	与えられたデータ長、もしくはストリーミング幅では動作できない
TLM_BYTE_ENABLE_ERROR_RESPONSE	与えられたバイト･イネーブルでは動作できない
TLM_GENERIC_ERROR_RESPONSE	その他のエラー

g) ターゲットがエラーを検出しても、特定のエラーを選ぶことが出来ないときは、、TLM_GENERIC_ERROR_RESPONSE を設定してよい

h) 応答ステータス･アトリビュート のデフォルト値は、TLM_INCOMPLETE_RESPONSE

i) ターゲットは、トランザクションのライフタイムの適当なところで、イニシエータは応答ステータス･アトリビュートを設定しなければならない。ブロッキングトランスポートインタフェースの場合、b_transport から制御が戻る前に設定する必要がある。ノンブロッキング・インタフェースと基本プロトコルでは、BIGIN_RESP フェーズを設定する前か、TLM_COMPLETE を返す前になる。

j) イニシエータは、BEGIN_RESP まで、もしくはトランザクションが完了するまで、受け取るトランザクションの応答ステータス･アトリビュートを常にチェックすることを推奨される。前もって、TLM_OK_RESPONSE であることが分かっている場合には、イニシエータは、応答ステータス･アトリビュートを無視することを選ぶかもしれない。たとえば、イニシエータが、常に TLM_OK_RESPONSE を返すターゲットにのみ接続していることを前もって知っている場合。ただし、常にそうであるとは限らないので、イニシエータは自らの責任で、応答ステータス･アトリビュートを無視することになる。

6.16.1 標準エラー応答

ターゲットが汎用ペイロードのトランザクションを受け取ったときには、次のいずれか一つのみの処理をすべきである

a) 汎用ペイロードのアトリビュートやモデル化されたコンポーネントの公にドキュメント化されているセマンティックスに従って、トランザクションで示されるコマンドを実行して、レスポンス・ステータスを TLM_OK_RESPONSE に設定する

b) 先に示した５つのエラーメッセージのうちの一つを汎用ペイロードのレスポンス・ステータスとして設定する

c) SystemC の標準レポートハンドラを使って、コマンド実行失敗、コマンド無視を含む SystemC

の４つの重要度レベルのひとつでレポートを生成して、レスポンス・ステータスを
TLM_OK_RESPONSE に設定する

ターゲットは上記の３つのうちのいずれか一つを実行することが推奨されるが、実装は、この推奨を押しつけられるものではない。

汎用ペイロード以外のトランザクション・タイプに対しても、上記と同様の応答を推奨する。すなわち、コマンドを期待されるとおりに実行するか、トランザクション・アトリビュートを使ってエラー応答を生成するか、もしくは SystemC レポートを生成する。しかしながら、処理の詳細とエラー応答メカニズムはこの標準の範囲外である。

上記の a)を満たすための条件は、コンポーネントのユーザが見ることのできるターゲットコンポーネントの期待されている動作によって決まる。汎用ペイロードの・アトリビュートはメモリ・マップド・バスにおける慣例適用法に対応したセマンティックで規定されているが、それは、ターゲットが RAM のような動作をすることを必ずしも想定するものではない。現実には、多くの異なった状況がある。

i. ターゲットが write コマンドと read コマンドの両方をサポートしているメモリマップドレジスタを持っていて、write コマンドがターゲットのステータスを変更しても、write コマンドの後の read コマンドが直前に書き込まれた値ではなく、ターゲットのステータスで決まる値を返すことがある。もしこれが、コンポーネントに期待される正常な動作であれば、a)でカバーされる

ii. ターゲットの write コマンドがデータ・アトリビュートを無視してビットを設定するように実装されていることがある。もしこれが、期待されている正常な動作であれば、a)でカバーされる

iii. ROM は、応答ステータス･アトリビュートを使ったエラー応答をイニシエータに出すことなく無視することがある。write コマンドはターゲットのステートを変更せず、無視されているが、ターゲットは、少なくとも SC_INFO か SC_WARNING の重要度レベルの SystemC レポートを生成すべきである

iv. ターゲットは read コマンドを実行する write コマンドや write コマンドを実行する read コマンドを実装してはならない。これは汎用ペイロードの基本的な使い方に違反するものである。

v. ターゲットは、汎用ペイロードのルールに対応はしているが、副作用を持つ read コマンドを実装するかもしれないが、これは a)でカバーされる

vi. アドレス指定可能なレジスターファイルで構成したメモリマップドレジスタを持つターゲットが、アドレス範囲外の write コマンドを受け取った場合には、トランザクションの応答ステータス･アトリビュートに TLM_ADDRESS_ERROR_RESPONSE を設定するか、SystemC レポートを生成すべきである

vii. 受動的シミュレーション・バスモニター・ターゲットが、バスの物理範囲を越えるアドレスのトランザクションを受け取った時には、a)の対応として、エラーの可能性があるトランザクションを後処理のためにログ記録して、b)や c)によるエラー生成はしないのがよい。代替として、c)に基づくレポートを生成しても良い。

言い換えれば、a),b),c)の選択は、最終的に具体的な判断はケースバイケースであるが、汎用ペイロードに対しては、いずれかの対応をすることが明確なルールとなっている。

例

```
// Showing generic payload with command, address, data, and response status

// The initiator
void thread() {
  tlm::tlm_generic_payload trans; // Construct default generic payload
  sc_time delay;
  trans.set_command(tlm::TLM_WRITE_COMMAND); // A write command
  trans.set_data_length(4); // Write 4 bytes
  trans.set_byte_enable_ptr(0); // Byte enables unused
  trans.set_streaming_width(4); // Streaming unused
  for (int i = 0; i < RUN_LENGTH; i += 4) { // Generate a series of transactions
    int word = i;
    trans.set_address(i); // Set the address
    trans.set_data_ptr( (unsigned char*)(&word) ); // Write data from local variable ‘word’
```

```
    trans.set_dmi_allowed(false); // Clear the DMI hint
    trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE ); // Clear the response status
    init_socket->b_transport(trans, delay);
    if (trans.get_response_status() <= 0) // Check return value of b_transport
      SC_REPORT_ERROR("TLM-2", trans.get_response_string().c_str());
    ...
}
...

// The target
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64 adr = trans.get_address();
  unsigned char* ptr = trans.get_data_ptr();
  unsigned int len = trans.get_data_length();
  unsigned char* byt = trans.get_byte_enable_ptr();
  unsigned int wid = trans.get_streaming_width();
  if (adr+len > m_length) { // Check for storage address overflow
    trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
    return;
  }
  if (byt) { // Target unable to support byte enable attribute
    trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
    return;
  }
  if (wid < len) { // Target unable to support streaming width attribute
    trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
    return;
  }
  if (cmd == tlm::TLM_WRITE_COMMAND) // Execute command
    memcpy(&m_storage[adr], ptr, len);
  else if (cmd == tlm::TLM_READ_COMMAND)
  memcpy(ptr, &m_storage[adr], len);
  trans.set_response_status( tlm::TLM_OK_RESPONSE ); // Successful completion
}

// Showing generic payload with byte enables

// The initiator
void thread() {
  tlm::tlm_generic_payload trans;
  sc_time delay;
  static word_t byte_enable_mask = 0x0000ffffful; // MSB..LSB regardless of host-endianness
  trans.set_command(tlm::TLM_WRITE_COMMAND);
  trans.set_data_length(4);
  trans.set_byte_enable_ptr( reinterpret_cast<unsigned char*>( &byte_enable_mask ) );
  trans.set_byte_enable_length(4);
  trans.set_streaming_width(4);
  ...
...
// The target
virtual void b_transport(
tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64 adr = trans.get_address();
  unsigned char* ptr = trans.get_data_ptr();
  unsigned int len = trans.get_data_length();
  unsigned char* byt = trans.get_byte_enable_ptr();
  unsigned int bel = trans.get_byte_enable_length();
  unsigned int wid = trans.get_streaming_width();
```

```
    if (cmd == tlm::TLM_WRITE_COMMAND) {
      if (byt) {
        for (unsigned int i = 0; i < len; i++) // Byte enable applied repeatedly up data array
          if ( byt[i % bel] == TLM_BYTE_ENABLED )
            m_storage[adr+i] = ptr[i]; // Byte enable [i] corresponds to data ptr [i]
          }
          else
            memcpy(&m_storage[adr], ptr, len); // No byte enables
      } else if (cmd == tlm::TLM_READ_COMMAND) {
        if (byt) { // Target does not support read with byte enables
          trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
          return;
        }
        else
          memcpy(ptr, &m_storage[adr], len);
      }
    trans.set_response_status( tlm::TLM_OK_RESPONSE );
  }
```

6.17 エンディアン

6.17.1 イントロダクション

イニシエータとターゲット間でデータ転送するために汎用ペイロードを使う時、ホストマシンのエンディアン（ホストエンディアン）とイニシエータのエンディアンとモデル化されるターゲット（モデル化エンディアン）の両方が関連する。この章では汎用ペイロードを使用しているイニシエータとターゲット間の相互利用性を確保するための規則を定義する。すなわち、汎用ペイロードのデータ配列とバイト・イネーブル配列の構造に特に関係する。しかしながら、ここで示す規則は汎用ペイロードの直接スコープを超えてエンディアンをモデル化するいくつかの選択肢に影響することがある。

TLM 2.0 アプローチの基本原則は汎用ペイロード・データ配列の構造がイニシエータ、インターコネクト・コンポーネントまたはターゲット内部で局所的に知りえる情報だけに依存する。特に、これはソケット、ホストマシンのエンディアンとモデル化されたコンポーネントの幅に依存する。汎用ペイロードの構造とエンディアンのアプローチは、ある共通システムのシナリオにおいてシミュレーション速度を最大限に引き出すために選択された。これから述べる規則は、汎用ペイロードの構造を示している。そして、これはモデル化されたシステムの構造から独立である。例えば、汎用ペイロード内部の Word は必ずしもモデル化されたアーキテクチャ内部のどんな Word の内部表現とは対応しない。

マクロ視点では、主原則は汎用ペイロードが MSB 対 MSB と LSB 対 LSB で結ばれたエンディアン混在システムにおけるコンポーネントを仮定している。換言すると、１Word が異なったエンディアンのコンポーネント間で転送されるなら、MSB ... LSB 関係が保存される。しかし各コンポーネント内部で見られる各 Byte のローカルアドレスはアドレス・スウィズリング(address swizzling)と一般的に呼ばれる変換を用いて必ず変換される。これは、モデル化されたシステムと TLM2.0 モデルの両方に当てはまる。一方、もしエンディアン混在システムが、各コンポーネント内部でローカルアドレスが変化しないように接続されていれば、(すなわち、各 Byte がどんなコンポーネントから参照されても同じアドレスを持つ時)、明示的 Byte スワップが TLM2.0 モデルに挿入される必要がある。

Helper 関数群がデータ配列の構造を支援するために準備されている。6.19 章を参照。

6.17.2 規則

（省略）

6.18 ホストエンディアンを決定する Helper 関数

6.18.1 イントロダクション

Helper 関数群がホストマシンのエンディアンを決めるために提供されている。これらは、汎用ペイロード・データ配列を作ったり解釈する時に使うことを目的としている。

6.18.2 定義

namespace tlm {
enum **tlm_endianness** {

TLM_UNKNOWN_ENDIAN, TLM_LITTLE_ENDIAN, TLM_BIG_ENDIAN };
inline tlm_endianness **get_host_endianness**(void);
inline bool **host_has_little_endianness**(void);
inline bool **has_host_endianness**(tlm_endianness endianness);
} // namespace tlm

6.18.3 規則
（省略）

6.19 エンディアン変換用 Helper 関数
6.19.1 イントロダクション
汎用ペイロード・データ配列の構造を統括する規則は十分定義されているし、多くの簡単なケースでは、データ配列を作り解釈するためのホストとは独立なＣ＋＋コードを書くことは明確な作業となる。しかしながら、規則はモデル化されたコンポーネントのエンディアンとホストエンディアン間の関係に依存する。そこでホストとは独立なコードを書くことは、ソケット幅と異なるアラインされないアドレスとデータ Word 幅を含む場合では極めて複雑になる。Helper 関数群はこの作業を支援するために準備されている。
エンディアンに関して、相互利用性はエンディアン規則だけに依存する。Helper 関数は必ずしも相互利用性に重要ではない。
エンディアン変換関数の背後にあるモチベーションは、ホストエンディアンとは関係なく一度書かれたイニシエータのための汎用ペイロード・トランザクションを作るＣ＋＋コードを可能にすること、さらに一度の関数呼び出で、ホストエンディアンにマッチするように変換されたとトランザクションを持つことである。各変換関数は、存在する汎用ペイロード・トランザクションを行いそして修正する。変換関数は対になっており、to_hostendian 関数と from_hostendian 関数はいつも一緒に使われる。to_hostendian 関数はトランスポート・インタフェースを介してとトランザクションを送る前にイニシエータにより呼ばれる。from_hostendian 関数は受取った後に呼ばれる。
４つの関数ペアが提供されている。もっとも一般的で強力な_generic、制限された場合だけに使える_word、_aligned、_single である。コスト面で_generic 関数の使用が推奨される。
変換関数には Arithmetic モードと Byte order モードがある。性能面で、Arithmetic モードが推奨される。変換関数は data word コンセプトが使われており、これは TLM2 ソケット幅と汎用ペイロード・データ配列とは独立である。

6.19.2 定義
namespace tlm {
template<class DATAWORD>
inline void **tlm_to_hostendian_generic**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_generic**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_to_hostendian_word**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_word**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_to_hostendian_aligned**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_aligned**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_to_hostendian_single**(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void **tlm_from_hostendian_single**(tlm_generic_payload *, unsigned int);
inline void **tlm_from_hostendian**(tlm_generic_payload *);
} // namespace tlm

6.19.3 規則
（省略）

6.20 汎用ペイロードの拡張

6.20.1 イントロダクション

拡張メカニズムは汎用ペイロードに組み込まれていて、汎用ペイロード無しに使うことはできない。その目的は、汎用ペイロードにアトリビュートの追加を許すことである。

拡張は無視可能(ignorable)か必須(mandatory)のどちらかにする。無視可能拡張は汎用ペイロード・トランザクションを受け取るどのインターコネクト・コンポーネントやターゲットからも無視されるかもしれない。無視可能拡張の主な意図はダウンストリームコンポーネントの機能性に直接の影響を及ぼさない補助的な情報やシミュレーション生成物やサイドバンド情報やメタデータをモデル化することである。無視可能拡張は基本プロトコルによって許される。必須拡張はトランザクションを受け取るどんな内部コンポーネントもターゲットも検査し動作することを義務付けられている。必須拡張の主な意図は、特定のプロトコルの詳細をモデル化するために汎用ペイロードを適応するときに使うことである。必須拡張には、新しいプロトコル型のクラス定義が必要である。

（言い換えれば、）イニシエータの観点から見て、標準の汎用ペイロード・トランザクションのアトリビュートの機能的な意味を変えるか、基本プロトコルの規則を変えるのであれば、その拡張を必須にすることを考慮しなければならない。ターゲットの観点から見て、その拡張がないときに使うことのできるその拡張に代わる適切なデフォルト値がなければ、その拡張を必須にすることを考慮しなければならない。

6.20.2 原理

拡張メカニズムの原理は、汎用ペイロードのコアアトリビュート一式の変化を運ぶ TLM ポートやソケットに同じトランザクション型で特殊化させることを許すことである。そうして、適応や橋渡しの必要なしに TLM ポートやソケットが互いに接続することを許す。拡張メカニズムなしに汎用ペイロードに新しいアトリビュートを追加すると、そのコアインタフェースクラスの新しいテンプレートの特殊化を導く新しいプロトコルクラスの定義が必要である。そのクラスは汎用ペイロードやこのように特殊化されたほかのものと型非互換である。拡張メカニズムは、TLM ポートの型互換を犠牲にすることなく汎用ペイロードに主要でない変更を加えることを許す。そうして、少し異なる情報を伝達するポートを接続するために必要なコード作成の仕事を減らす。

6.20.3 拡張ポインタとオブジェクトとブリッジ

拡張は tlm_extension クラスから派生した型のオブジェクトである。汎用ペイロードは拡張オブジェクトへのポインタ配列を含む。すべての汎用ペイロードのオブジェクトは拡張のすべての型の単体のインスタンスを運ぶことができる。

拡張を指し示すポインタ配列はすべての登録された拡張のために１つのスロットを持つ。

set_extension メソッドは単純にポインタを上書きし、原則としてイニシエータか、インターコネクト・コンポーネントかターゲットから呼び出されることができる。これはとても自由度の高い低レベルのメカニズムを提供するが、使用間違いも起こしやすい。拡張オブジェクトの所有権と削除はユーザによって良く理解され注意深く考慮されなければならない。

２つの別の汎用ペイロード・トランザクションの間にブリッジを作るとき、もし要求されたら入ってくるトランザクション・オブジェクトから出て行くトランザクション・オブジェクトへすべての拡張をコピーし出て行くトランザクションとその拡張を所有し管理するのはブリッジの責任である。(同じことがデータ配列とバイト・イネーブル配列にも言える。) deep_copy_from メソッドは、データ配列とバイト・イネーブル配列や拡張オブジェクトを含むトランザクション・オブジェクトの深いコピーをブリッジが行えるようにするために提供されている。もしそのブリッジが出て行くトランザクションにさらなる拡張を加えるなら、それらの拡張はそのブリッジによって所有される。

拡張の管理は 6.5 章「汎用ペイロードのメモリ管理」に十分に記述されている。

6.20.4 規則

a) 拡張はイニシエータかインターコネクトかターゲットのコンポーネントにより加えることができる。特に、拡張の作成はイニシエータには制限がない。

b) どんな数の拡張でも汎用ペイロードの個々のインスタンスに加えることができる。

c) 無視可能な拡張の場合、どのインターコネクトやターゲットのコンポーネントも自由に与えられた拡張を無視できるようにすることを推奨する。しかし、これは実装によって強制されるべきではない。インターコネクトやターゲットのコンポーネントが拡張を持たないという理由で標準エラー応答を行うようにすることは可能であるが、推奨される動作ではない。

d) 無視可能な拡張の場合、与えられた拡張の有無がどのコンポーネントにおいても主要な機能に影響を与えないことを推奨する。しかし、たとえば診断レポートやデバッグや最適化などであれば影響してもよい。
e) 与えられた拡張の存在を強制する組み込みのメカニズムはない。
f) 個々の拡張の意味はアプリケーションが定義する。あらかじめ定義された拡張はない。
g) 拡張は tlm_extension クラスからユーザ定義クラスを派生させて作り、ユーザ定義クラスそれ自身の名前を tlm_extension のテンプレート引数として渡してそのクラスのオブジェクトを作らなければならない。ユーザ定義クラスは汎用ペイロードの拡張されたアトリビュートを表すメンバーを含んで良い。
h) tlm_extension_base クラスの free 仮想メソッドは、拡張オブジェクトを削除しなければならない。このメソッドは、拡張のユーザ定義のメモリ管理の実装を上書きするかもしれない。しかし、これは必須ではない。
i) tlm_extension クラスの clone 純粋仮想関数は、すべての拡張されたアトリビュートを含む拡張オブジェクトのクローンを作るためにユーザ定義の拡張クラスの中に定義されなければならない。この clone メソッドは、汎用ペイロードのメモリ管理と共に使うことを意図されている。それは、すべての拡張オブジェクトのコピーを作って、オリジナルのオブジェクトのデストラクションがあっても副作用なくそのコピーが残るようにしなければならない。
j) tlm_extension クラスの copy_from 純粋仮想関数は、別の拡張オブジェクトのアトリビュートをコピーすることによって自身のオブジェクトを修正するためにユーザ定義の拡張クラスの中で定義されなければならない。
k) tlm_extension クラステンプレートをインスタンスすると ID 公開データ・メンバが初期化され、これは汎用ペイロード・オブジェクトを持つ与えられた拡張を再登録しその拡張にユニークな ID を割り当てる効果を持たなければならない。その ID はプログラム実行中のすべてに対してユニークでなければならない。
l) 汎用ペイロードは、可変サイズの配列に拡張のポインタを蓄え、拡張の ID はその配列の中への拡張ポインタのインデックスを与えるようにふるまわなければならない。拡張の汎用ペイロードへの登録は、その拡張のための配列インデックスを保持しなければならない。個々の汎用ペイロードのオブジェクトは現在実行中のプログラムの中で登録されるすべての拡張へのポインタを蓄える能力のある配列を含まなければならない。
m) 拡張配列の中のポインタはトランザクションが生成されたときには空でなければならない。
n) 個々の汎用ペイロードのオブジェクトはすべての与えられた拡張型のせいぜい 1 つのオブジェクトへのポインタしか蓄えることができない。(しかし、異なる拡張型のたくさんのオブジェクトを蓄える。)
o) メソッド set_extension と set_auto_extension、get_extension、clear_extension、release_extension は複数の形態で提供され、関数テンプレートの使用や拡張ポインタ引数の使用、ID 引数の使用のような異なる方法でアクセスされてその拡張を認識する。ID 引数を受け取る関数は、汎用ペイロードのオブジェクトのクローンを作るときのような特別なプログラムタスクのために意図されており、アプリケーションで一般的に使うためのものではない。
p) set_extension(T*)メソッドは、ポインタの配列の中の T 型の拡張オブジェクトを示すポインタを引数の値で置き換えなければならない。その引数は登録された拡張へのポインタでなければならない。関数の戻り値は、この呼び出しで置き換えられた汎用ペイロードの中のポインタの前の値でなければならない。それは空(null)のポインタかもしれない。set_auto_extension(T*)メソッドは、その拡張が自動削除にマークされるのを除き、同様にふるまわなければならない。
q) set_extension(unsigned int, tlm_extension_base*)メソッドは、第 1 引数で与えられる配列インデックスの示す、配列のポインタの中の拡張オブジェクトへのポインタを第 2 引数の値で置き換えなければならない。与えられたインデックスは、拡張 ID として登録されていなければならない。そうでない場合の関数のふるまいは未定義である。関数の戻り値は、与えられた配列インデックスへのポインタの前の値でなければならない。それは空(null)のポインタかもしれない。set_auto_extension(unsigned int, tlm_extension_base*)メソッドは、その拡張が自動削除にマークされるのを除き、同様にふるまわなければならない。
r) メモリ管理が存在するときの与えられた拡張のための set_auto_extension の呼び出しは、set_extension の呼び出しに続いてすぐに同じ拡張の release_extension を呼び出すのに等しい。メモリ管理のないときに set_auto_extension を呼び出すとランタイムエラーを起こす。

s) もし拡張が自動削除とマークされたら、与えられた拡張オブジェクトはユーザ定義のメモリ管理の free メソッドの実装によって削除されるかプールされなければならない。free メソッドは、トランザクション・オブジェクトの参照数が 0 に到達したときに呼び出される。拡張オブジェクトは tlm_generic_payload クラスの reset メソッドか拡張オブジェクトの free メソッドを呼び出すことによって削除されても良いし、プールされても良い。
t) もし汎用ペイロードのオブジェクトがすでにその型の拡張への空 (null) でないポインタを含んでいたら、古いポインタは上書きされる。
u) メソッド関数 get_extension(T*&)と T* get_extension()は、もしそれが存在するなら、与えられた型の拡張オブジェクトへのポインタを、存在しなければ空(null)のポインタを返さなければならない。T 型は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。この関数テンプレートを使って存在しない拡張の検索を試すことはエラーではない。
v) get_extension(unsigned int)メソッドは、引数によって与えられた ID の拡張オブジェクトへのポインタを返さなければならない。与えられたインデックスは拡張の ID として登録されていなければならない。そうでない場合のこの関数のふるまいは未定義である。もし与えられたインデックスへのポインタが拡張オブジェクトへのポインタではなかった場合、この関数は空(null)のポインタを返さなければならない。
w) メソッド clear_extension(const T*)と clear_extension()は、汎用ペイロード・オブジェクトから与えられた拡張を取り除かなければならない。すなわち、拡張配列の中の対応するポインタを空(null)に設定しなければならない。拡張は、引数として拡張オブジェクトのポインタを渡すか、clear_extension<ext_type>()のように関数テンプレートのパラメータの型を使うことで特定される。もし存在するならば、その引数は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。clear_extension メソッドはその拡張オブジェクトを削除してはならない。
x) メソッド release_extension(T*)と release_extension()は、もしトランザクション・オブジェクトがメモリ管理を持つならば自動削除としてその拡張をマークしなければならない。そうでない場合、これらのメソッドは、拡張オブジェクトの free メソッドの呼び出しと拡張配列の中の対応するポインタに空(null)を設定することで与えられた拡張を削除しなければならない。拡張は、引数として拡張オブジェクトのポインタを渡すか、release_extension<ext_type>()のように関数テンプレートのパラメータの型を使うことで特定される。もし存在するならば、その引数は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。
y) release_extension メソッドのふるまいは、トランザクション・オブジェクトがメモリ管理を持つかどうかに依存することに注意せよ。メモリ管理がある場合は、拡張はまれに自動削除にマークされ、アクセス可能な状態で続く。メモリ管理がない場合は、拡張ポインタがクリアされるだけでなく拡張オブジェクトそれ自身も削除される。存在しない拡張オブジェクトを開放しないように気をつけなければならない。開放した場合にはランタイムエラーになる。
z) メソッド clear_extension と release_extension は、たとえば set_auto_extension でセットされた拡張のように自動削除としてマークされた拡張や release_extension ですでに開放された拡張のために呼び出してはならない。もしそうすると、ランタイムエラーになるかもしれない。
aa) 個々の汎用ペイロードのトランザクションは、すべての登録された拡張へのポインタを蓄えられる十分な領域を割り当てられなければならない。これは次の 2 つの方法のどちらかで達成できる。 1 つは C++の静的な初期化の後にトランザクション・オブジェクトを構築するとであり、もう 1 つは静的な初期化の後で最初にトランザクション・オブジェクトを使う前に resize_extensions メソッドを呼び出すことである。前に述べたの方法において、拡張配列の大きさを設定するのは汎用ペイロードのコンストラクタの責任である。後に述べた方法において、最初に拡張にアクセスする前に resize_extensions メソッドを呼び出すのはアプリケーションの責任である。
bb) resize_extensions メソッドは、すべての登録された拡張に応じられるように汎用ペイロードの中の拡張配列のサイズを増加させなければならない。
例

```
// Showing an ignorable extension
// User-defined extension class
struct ID_extension: tlm::tlm_extension<ID_extension>
{
ID_extension() : transaction_id(0) {}
```

```
virtual tlm_extension_base* clone() const { // Must override pure virtual clone method
ID_extension* t = new ID_extension;
t->transaction_id = this->transaction_id;
return t;
  }
// Must override pure virtual copy_from method
virtual void copy_from(tlm_extension_base const &ext) {
transaction_id = static_cast<ID_extension const &>(ext).transaction_id;
  }
unsigned int transaction_id;
};

// The initiator
struct Initiator: sc_module
{ ...
void thread() {
tlm::tlm_generic_payload trans;
...
ID_extension* id_extension = new ID_extension;
trans.set_extension( id_extension ); // Add the extension to the transaction

for (int i = 0; i < RUN_LENGTH; i += 4) {
...
++ id_extension->transaction_id; // Increment the id for each new transaction
...
socket->b_transport(trans, delay);
...

// The target
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{ ...
ID_extension* id_extension;
trans.get_extension( id_extension ); // Retrieve the extension
if (id_extension) { // Extension is not mandatory
char txt[80];
sprintf(txt, "Received transaction id %d", id_extension->transaction_id);
SC_REPORT_INFO("TLM-2", txt);
  }
...

// Showing a new protocol types class with a mandatory extension
struct cmd_extension: tlm::tlm_extension<cmd_extension>
{ // User-defined mandatory extension class
cmd_extension(): increment(false) {}
virtual tlm_extension_base* clone() const {
cmd_extension* t = new cmd_extension;
t->increment = this->increment;
return t;
  }
virtual void copy_from(tlm_extension_base const &ext) {
increment = static_cast<cmd_extension const &>(ext).increment;
  }
bool increment;
};

struct my_protocol_types // User-defined protocol types class
{
typedef tlm::tlm_generic_payload tlm_payload_type;
typedef tlm::tlm_phase tlm_phase_type;
};
```

```
struct Initiator: sc_module
{
tlm_utils::simple_initiator_socket<Initiator, 32, my_protocol_types> socket;
...
void thread() {
tlm::tlm_generic_payload trans;
cmd_extension* extension = new cmd_extension;
trans.set_extension( extension ); // Add the extension to the transaction
...
trans.set_command(tlm::TLM_WRITE_COMMAND); // Execute a write command
socket->b_transport(trans, delay);
...
trans.set_command(tlm::TLM_IGNORE_COMMAND);
extension->increment = true; // Execute an increment command
socket->b_transport(trans, delay);
...
...

// The target
tlm_utils::simple_target_socket<Memory, 32, my_protocol_types> socket;

virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
tlm::tlm_command cmd = trans.get_command();
...
cmd_extension* extension;
trans.get_extension( extension ); // Retrieve the command extension
...
if (!extension) { // Check the extension exists
trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
return;
  }
if (extension->increment) {
if (cmd != tlm::TLM_IGNORE_COMMAND) { // Detect clash with read or write
trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
return;
    }
++ m_storage[adr]; // Execute an increment command
memcpy(ptr, &m_storage[adr], len);
  }
...
```

6.21 インスタンス固有の拡張

6.21.1 イントロダクション

汎用ペイロードは、個々のトランザクション・オブジェクトが個々の拡張型の１つまでのインスタンスを含むことができるように拡張オブジェクトへのポインタの配列を持たなければならない。このメカニズム単体で、与えられたトランザクション・オブジェクトに同じ拡張の複数のインスタンスを直接加えることはできない。この節では、インスタンス固有の拡張を提供するユーティリティ一式、すなわち、単体のトランザクション・オブジェクトに加えられた同じ型の複数の拡張について述べる。

インスタンス固有の拡張の型は、tlm_extension クラスと同様な使い方で instance_specific_extension クラステンプレートを使うことで作られる。tlm_extension と違って、アプリケーションは仮想メソッド clone と copy_from の実装を要求されないし許されない。そのアクセス方法は set_extension と get_extension、clear_extension、resize_extension に制限される。インスタンス固有拡張の自動削除はサポートされない。だから、set_extension を呼ぶコンポーネントは clear_extension も呼び出さなければならない。tlm_extension クラスと同様に、resize_extensions メソッドはトランザクション・オブジェクトが静的初期化の間に作られたときだけ呼び出されなければならない。

インスタンス固有の拡張は、instance_specific_extension_accessor クラスのオブジェクトを使ってアクセスされる。このクラスはただ１つの operator()メソッドを提供し、そのメソッドはア

クセスメソッドを呼び出すことができる間の代理オブジェクトを返す。
instance_specific_extension_accessor クラスの個々のオブジェクトは、同じトランザクション・オブジェクトに使われているときでさえ、拡張オブジェクトの別個のセットへのアクセスを与える。
以下のクラス定義においてイタリックの単語はアプリケーションによって直接使われてはならない。それは実装において定義される名前である。

6.21.2 クラス定義

```
namespace tlm_utils {

template <typename T>
class instance_specific_extension : public implementation-defined {
public:
virtual ~instance_specific_extension();
};

template<typename U>
class proxy {
public:
template <typename T> T* set_extension(T* ext);
template <typename T> void get_extension(T*& ext) const;
template <typename T> void clear_extension(const T* ext);
void resize_extensions();
};

class instance_specific_extension_accessor {
public:
instance_specific_extension_accessor();

template<typename T> proxy< implementation-defined >& operator() ( T& );
};

} // namespace tlm_utils
```

例

```
struct my_extn : tlm_utils::instance_specific_extension<my_extn> {
int num; // User-defined extension attribute
};

struct Interconnect: sc_module
{
tlm_utils::simple_target_socket<Interconnect> targ_socket;
tlm_utils::simple_initiator_socket<Interconnect> init_socket;
...
tlm_utils::instance_specific_extension_accessor accessor;
static int count;

virtual tlm::tlm_sync_enum nb_transport_fw(
tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay )
  {
my_extn* extn;
accessor(trans).get_extension(extn); // Get existing extension
if (extn) {
accessor(trans).clear_extension(extn); // Delete existing extension
} else {
extn = new my_extn;
extn->num = count++;
accessor(trans).set_extension(extn); // Add new extension
    }
return init_socket->nb_transport_fw( trans, phase, delay );
} ...
};
```

```
... SC_CTOR(Top) {
// Transaction object passes through two instances of Interconnect
interconnect1 = new Interconnect("interconnect1");
interconnect2 = new Interconnect("interconnect2");
interconnect1->init_socket.bind( interconnect2->targ_socket );
...
```

7. フェーズと基本プロトコル

7.1 フェーズ

7.1.1 イントロダクション

- tlm_phase クラスは、non-blocking transport インタフェース・クラスのテンプレートと基本プロトコルによって使用されるデフォルト・フェーズ・タイプである。
- tlm_phase オブジェクトは unsigned int 値でフェーズを表す。
- unsigned int タイプ tlm_phase クラスは、4 つのベースプロトコル・フェーズ BEGIN_REQ、END_REQ、BEGIN_RESP、および END_RESP の列挙体に割り当てる。
- DECLARE_EXTENDED_PHASE マクロを使用して tlm_phase_enum として提供された 4 つのフェーズのセットは拡張することができる。
- このマクロは対応するオブジェクトを返す get_phase メソッドを持つ tlm_phase から拡張された別物のクラスを作成する。そのオブジェクトは新しいフェーズとして使用可能である。
- 各 application では相互利用性を最大限に生かすには tlm_phase_enum の 4 つのフェーズのみを使用すべきである。もし、 更なるフェーズが特定のプロトコルの詳細をモデル化するのに必要であるなら、DECLARE_EXTENDED_PHASE を使用すべきである。
- DECLARE_EXTENDED_PHASE は tlm_phase タイプとの互換性を有す。
- 無視可能 vs 必須な拡張の本質は汎用ペイロード拡張と同様にフェーズにも適用される。つまり、無視可能フェーズは基本プロトコルに許容される。
- 無視可能フェーズは、ターゲットとイニシエータ両方から無視可能にする必要がある。ターゲットの場合ターゲットがフェーズを見ていないかのように単に動作できるという意味での無視可能であって、イニシエータではターゲットからどんな応答がないときでもイニシエータが続けることができるという意味で無視可能でなければならない。
- この意味でフェーズを無視できないなら、新しいプロトコル・タイプのクラスを定義するべきである。

7.1.2 クラス定義

```
namespace tlm {
  enum tlm_phase_enum {
    UNINITIALIZED_PHASE=0, BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };
  class tlm_phase{
  public:
    tlm_phase();
    tlm_phase( unsigned int );
    tlm_phase( const tlm_phase_enum& );
    tlm_phase& operator= ( const tlm_phase_enum& );
    operator unsigned int() const;
  };
  inline std::ostream& operator<< ( std::ostream& , const tlm_phase& );
  #define DECLARE_EXTENDED_PHASE(name_arg) ¥
  class tlm_phase_##name_arg : public tlm::tlm_phase{ ¥
  public:¥
    static const tlm_phase_##name_arg& get_phase();¥
    implementation-defined ¥
  }; ¥
  static const tlm_phase_##name_arg& name_arg=tlm_phase_##name_arg::get_phase()
} // namespace tlm
```

7.1.3 ルール

a) デフォルトコンストラクタ tlm_phase はフェーズの値を 0 に設定するものとする。対応する列挙体は文字通り UNINITIALIZED_PHASE である。

b) メソッドの tlm_phase(unsigned int)、operator= と operator unsigned int は対応する unsigned int もしくは enum を使ってフェーズの値を得たり設定したりするものになる。

c) 機能オペレータ << はフェーズの名前に対応する文字列を与えられた出力ストリームに書くものとする。たとえば 「BEGIN_REQ」である。
d) DECLARE_EXTENDED_PHASE(arg)マクロは tlm_phase から拡張された tlm_phase_arg という新しい単独のクラスを創設する。tlm_phase は public メソッドの get_phase を持っていて static object のリファレンスを返す。マクロの引数はオペレータ<<によって書かれた文字列として、指示された対応するフェーズが使用される。
e) static const name_arg によって指示されたオブジェクトは拡張フェーズを表し、それがフェーズ引数として nb_transport に渡されることが意図されている。
f) もし、受け取った拡張フェーズがコンポーネントで無視できないなら、アプリケーションは、関連するソケットをインスタンスする時、 新しいプロトコル・タイプのクラスを定義して、テンプレート引数としてそのクラスの名前を使用するべきである。 これは、非互換なプロトコルのソケットとのバインディングを防ぐ。
g) 無視可能フェーズへのトランジションはどんな受取コンポーネントにも単に無視される場合もある。 nb_transport をコールする場合、呼ばれる側 (callee) がフェーズへのトランジションを無視した時、値として TLM_ACCEPTED を返すべきである。
例

```
DECLARE_EXTENDED_PHASE(ignore_me); // Declare two extended phases
DECLARE_EXTENDED_PHASE(internal_ph); // Only used within target
struct Initiator: sc_module
{ ...
  { ...
    phase = tlm::BEGIN_REQ;
    delay = sc_time(10, SC_NS);
    socket->nb_transport_fw( trans, phase, delay ); // Send phase BEGIN_REQ to target
    phase = ignore_me; // Set phase variable to the extended phase
    delay = sc_time(12, SC_NS);
    socket->nb_transport_fw( trans, phase, delay ); // Send the extended phase 2ns later
    ...

struct Target: sc_module
{
  ...
  SC_CTOR(Target)
  : m_peq("m_peq", this, &Target::peq_cb) {} // Register callback with PEQ
  virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload& trans,
  tlm::tlm_phase& phase, sc_time& delay ) {
    cout << "Phase = " << phase << endl; // use overloaded operator<< to print phase
    m_peq.notify(trans, phase, delay); // Move transaction to internal queue
    return tlm::TLM_ACCEPTED;
  }
  void peq_cb(tlm::tlm_generic_payload& trans, const tlm::tlm_phase& phase)
  { // PEQ callback
    sc_time delay;
    tlm::tlm_phase phase_out;
    if (phase == tlm::BEGIN_REQ) { // Received BEGIN_REQ from initiator
      phase_out = tlm::END_REQ;
      delay = sc_time(10, SC_NS);
      socket->nb_transport_bw(trans, phase_out, delay); // Send END_REQ back to initiator
      phase_out = internal_ph; // Use extended phase to signal internal event
      delay = sc_time(15, SC_NS);
      m_peq.notify(trans, phase_out, delay); // Put internal event into PEQ
    }
    else if (phase == internal_ph) // Received internal event
    {
      phase_out = tlm::BEGIN_RESP;
      delay = sc_time(10, SC_NS);
      socket->nb_transport_bw(trans, phase_out, delay); // Send BEGIN_RESP back to initiator
    }
  } // Ignore phase ignore_me from initiator
```

```
    tlm_utils::peq_with_cb_and_phase<Target, tlm::tlm_base_protocol_types> m_peq;
  };
```

7.2 基本プロトコル

7.2.1 イントロダクション

- 基本プロトコルは、メモリ・マップド・バスへのインタフェースを持つコンポーネントのトランザクション・レベル・モデル間の最大限の相互利用性を確実にするために 1 セットの規則から成る。 基本プロトコルは以下の使用を必要とする。
  1. TLM-2 コアトランスポート、ダイレクト・メモリ、およびデバッグトランスポートインタフェース
  2. ソケットのクラスの tlm_initiator_socket と tlm_target_socket(または、これらから派生したクラス)
  3. 汎用ペイロードのクラス tlm_generic_payload
  4. フェーズのクラス tlm_phase
  5. 以下で定義される更なる 1 セットの規則
- もし、無視可能なフェーズ拡張であれば、基本プロトコル規則は汎用ペイロードと、フェーズの拡張を可能にする。 無視可能でない拡張は新しいプロトコル・タイプのクラスの定義を必要とする。
- 基本プロトコルは事前に定義された tlm_base_protocol_types クラスによって表される。しかしながらこのクラスは、2 種類の型だけを定義される。 このクラス(ソケットへのテンプレート引数としての)を使用するすべてのコンポーネントが、基本プロトコルの規則を尊重するのをコンベンションによって強いられる。

7.2.2 クラス定義

```
namespace tlm {
  struct tlm_base_protocol_types
  {
    typedef tlm_generic_payload tlm_payload_type;
    typedef tlm_phase tlm_phase_type;
  };
} // namespace tlm
```

7.2.3 Base protocol phase sequences

a) この項は、基本プロトコルに特定しているが、他のプロトコルをモデル化するのにノンブロッキング・トランスポート・インタフェースを使用するときのガイドとして使用されるかもしれない。他のプロトコルをモデル化する時、他のフェーズを定義することが必要であるが、その定義による基本プロトコルに相互運用性の損失の影響が出て来る可能性もある。

b) 基本プロトコルはブロッキング・トランスポート・インタフェース、ノンブロッキング・トランスポート・インタフェース、または両方同時使用を可能にする。ブロッキング・トランスポート・インタフェースはフェーズ情報を運ばない。 基本プロトコルと共に使用される場合、nb_transport を呼ぶ順番の強い規制管理がある。しかし、b_transport を呼び出す順番を管理するそのような規制は無い。したがって、 nb_transport は AT 記述に最適で、 b_transport は LT 記述に最適でる。

c) phase transition の完全なシーケンスは以下の通りである。

BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP

d) BEGIN_REQ と END_RESP はイニシエータ・ソケットのみ送信可能である。END_REQ と BEGIN_RESP はターゲット・ソケットのみ送信可能である。

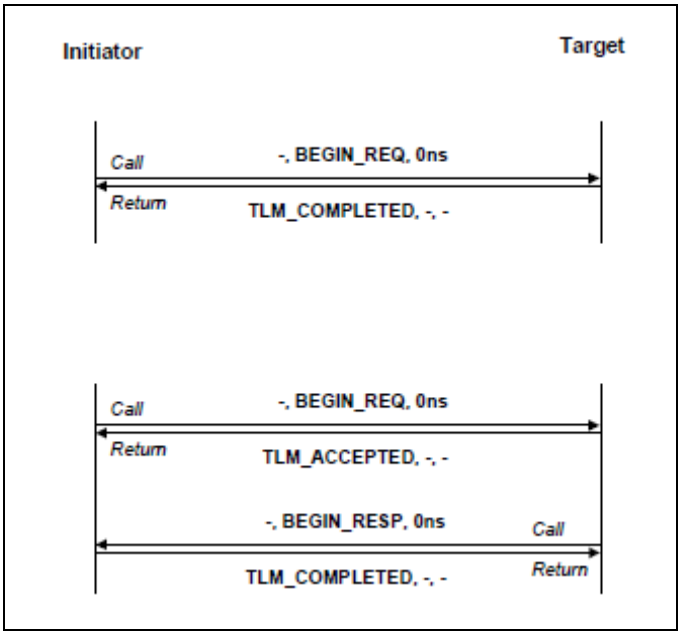
e) ブロッキング・トランスポート・インタフェースの場合、一つのトランザクション・インスタンスの全ライフタイムは b_transport へのコールとその応答のリターンになる。また、b_transport と BEGIN_REQ へのコール、および、 b_transport と BEGIN_RESP からのリターン、どの通信も純粋に概念的である。

f) 基本プロトコルには、TLM_UPDATED の値を持つ nb_transport への各コールと nb_transport からの各リターンはフェーズ・トランジションとする。つまり、同じトランザクションのための nb_transport への 2 つの連続した呼び出しには、フェーズ引数に違う値を持つ。 無視可能なフェーズ拡張は受入れられる。その場合、拡張フェーズの挿入がこの規則の目的のために（フェーズが無視されても）phase transition とみなすものとなる。

g) nb_transport に TLM_COMPLETED の値を返させることによって、短く phase sequence を切るこ

とができる。 TLM_COMPLETED のリターン値はトランザクションの終わりを示す、その場合、フェーズ引数が無視されるべきである。（第 4.1.2.7 項 The tlm_sync_enum return value を参照）。TLM_COMPLETED が成功した終了を意味しないので、イニシエータは成否の確認のためにトランザクションの応答状態をチェックするべきである。 また、フェーズ END_RESP へのトランジションはトランザクションの終わりを示すものとする。その場合、呼ばれる側(callee)が TLM_COMPLETED の値を返す義務は無い。

h) もし、イニシエータにおいて、END_REQ が最初に受信されていなくてターゲットから BEGIN_RESP を受ける場合、 イニシエータはすぐに、BEGIN_RESP に先行する暗黙の END_REQ を仮定するものとする。



Examples of early completion

i) 前述のすべての規則を纏めると、許された phase transition sequences の組み合わせは以下の通りになる。経路(forward, backward or return)が括弧内に示されている。 無視可能なフェーズ拡張は任意な点で挿入されるかもしれない。 その都度、トランザクションはうまくいったりいかなかったりする場合もある。

BEGIN_REQ(fw) (target returns TLM_COMPLETED)
BEGIN_REQ(fw) → END_REQ(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → BEGIN_RESP(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → END_REQ(rtn/bw) → BEGIN_RESP(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → BEGIN_RESP(rtn/bw) → END_RESP(rtn/fw)
BEGIN_REQ(fw) → END_REQ(rtn/bw) → BEGIN_RESP(bw) → END_RESP(rtn/fw)

j) コンポーネントがイリーガルか out-of-order フェーズ・トランジションを受け取った場合、これは送信側の誤りである。 受信側の振舞いは未定義である。 これは、実行時にエラーが発生するかもしれないことを意味している。

7.2.4 Base protocol timing parameters and flow control

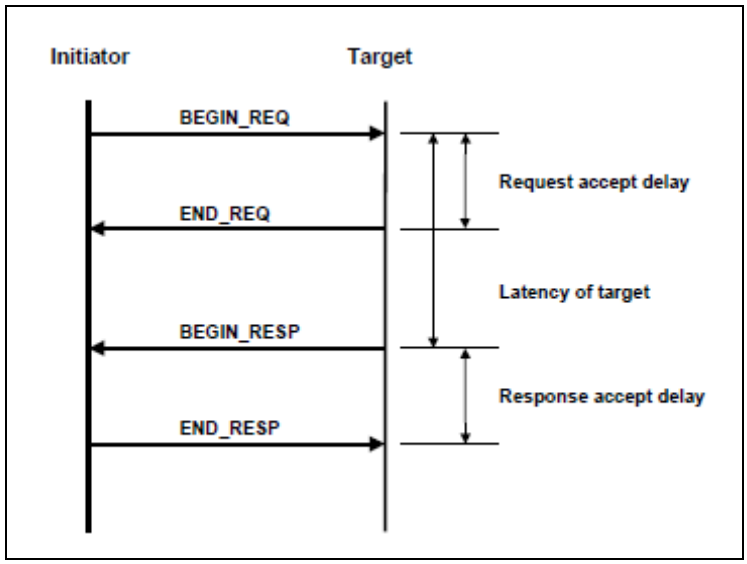
a) 4 つのフェーズの場合、 request accept delay (or minimum initiation interval between sending successive transactions) 、 request accept delay、 response accept delay をモデル化できる。 AT 記述には、この種類のタイミング精度は適切である。

b) 基本プロトコルの場合、 イニシエータは BEGIN_REQ フェーズを使って新トランザクションを始まる前にすぐ前のトランザクションから END_REQ か BEGIN_RESP を受けとるか、またはターゲットが一つ前のトランザクションの終了を TLM_COMPLETED という値を nb_transport_fw から受けとるかで決まる。

c) 基本プロトコルの場合、ターゲットは BEGIN_RESP フェーズを使って新トランザクションを始まる前にすぐ前のトランザクションから END_RESP を受けとるか、または イニシエータが一つ前のトランザクションの終了を TLM_COMPLETED という値を nb_transport_bw から受けとるかで決まる。

d) 与えられたソケットを通して non-blocking トランスポート・インタフェースを使用すること

で、送られた連続したトランザクションはパイプライン化することができる。各 BEGIN_REQ(または、BEGIN_RESP) を END_REQ(または、END_RESP)で応じることによって、相互接続コンポーネントは、同時に、いろいろなトランザクション・オブジェクトを発行可能にすることができる。 すぐに END_REQ(または、END_RESP)で応じないことによって、相互接続コンポーネントはイニシエータ(または、ターゲット)から来るトランザクション・オブジェクトの流れにフロー制御を及ぼすことができる。



Approximately-timed timing parameters

e) 与えられたソケットを通して 2 個のアウトスタンディング要求か応答の可能性を除くこの規則は、 non-blocking トランスポート・インタフェースに当てはまるだけで、 b_transport への呼び出しに直接影響を与えられない。(規則は b_transport 自身が nb_transport_fw とコールする場合、b_transport への呼び出しに間接的に影響しているかもしれない。)

7.2.5 Base protocol transaction ordering rules

a) この項の規則は基本プロトコル、および単体の基本プロトコルに適用される。他のプロトコル・タイプのクラスによって表された特定のプロトコルは、それら自身に必要な規則を持つ。

b) 同じトランザクション・オブジェクトか異なったトランザクション・オブジェクトにかかわらず連続した b_transport 呼び出しのタイミング・アノテートには決まった規制が全く無い。したがって、loosely-timed コード化スタイルに、ブロッキング・トランスポート・インタフェースは適切である。

c) 与えられたトランザクション・オブジェクトのための nb_transport への連続した呼び出しのタイミング・アノテートには強く決められた規制がある。したがって、approximately-timed コード化スタイルに、non-blocking トランスポート・インタフェースは適切である。

d) b_transport コールはリエントラントである。 b_transport の実装は、wait をさせることができる、そして、その間、同じトランザクション・オブジェクト (または、違うトランザクション・オブジェクト) のためにタイミング・アノテートの規制のない同じソケットを通して、b_transport への別の call をかけることができる。

e) 与えられたトランザクション・オブジェクトのための与えられたソケットを通した nb_transport への連続した呼び出しには、増加するタイミング・アノテートがあるものとする。すなわち、t が時間引数である nb_transport の表現 sc_time_stamp() + t から計算された値のシーケンスは減少不可である。これは forward と backward パスで同じく適用される。

f) 異なったトランザクション・オブジェクトのための与えられたソケットを通した連続した nb_transport 呼び出しのタイミング・アノテートには決まった規制が全く無い。
approximately-timed シミュレーションのために、タイミング・アノテート命令は通常、減少不可である。しかしながら、ブロッキングとノンブロッキング・トランスポート呼び出しが複雑であった場合では、異なったトランザクションのための nb_transport 呼び出しは out-of-order に現れることができる。

g) 各イニシエータが、減少不可のタイミング・アノテートで b_transport を呼ぶことが勧められる。(そして、減少不可のタイミング・アノテートで nb_transport と呼ぶのが強いられる。)二つの異なるイニシエータが loosely-timed の場合、起きたトランザクション・ストリームの収束時

に Out-of-order タイミングアノテートは必要になる。
h) 与えられたソケットに関しては、イニシエータはブロッキング、ノンブロッキング・トランスポート・インタフェース、ダイレクト・メモリ・インタフェース、およびトランスポート・デバッグ・インタフェースを通して一般的なペイロードのメモリ管理規則を条件として同じトランザクション・オブジェクトにパスすることができる。第 6.5 項の汎用ペイロード・メモリ管理を参照すること。
i) 与えられたソケットに関しては、イニシエータが異なったトランザクション・オブジェクトのためにブロッキングとノンブロッキング・トランスポート・インタフェースを切り換えることが許可されている。あらゆるターゲットがブロッキングとノンブロッキング・トランスポート・インタフェースの両方をサポートして、どのような内部状態情報も保守するのが強いられるので、それは両方のインタフェースからアクセスしやすくなっている。意図としては、イニシエータが loosely-timed と approximately-timed シミュレーションモードの間で切り替えのスイッチを作ることを許可することである。 b_transport と nb_transport_fw をインターリーブで呼び出すイニシエータは、タイミング精度に関して低い期待を持つべきである。
j) b_transport と nb_transport_fw のどちらか 1 つを実装するのが必要なベース・プロトコル・ターゲットがブロッキングとノンブロッキング・トランスポート・インタフェースの両方をサポートできるように、便利ソケット simple_target_socket を提供する。第 5.3.2 項の Simple ソケットを参照すること。
k) 与えられたトランザクション・オブジェクトに関しては、イニシエータはトランザクションのライフタイムの中頃でブロッキングとノンブロッキング・トランスポート・インタフェースを切り換えないものとする。言い換えれば、b_transport が戻る前にイニシエータは、nb_transport_fw と呼ばないものとする。または、未完了の BEGIN_REQ があるときには b_transport を call すること。
l) 同時に、複数の平行なソケット、または、複数の平行な経路で与えられたトランザクション・オブジェクトを送らないものとする。それぞれのトランザクション・インスタンスはトランザクション・インスタンスのライフタイムが残っている間、1 セットのコンポーネントとソケットを通してユニークな明確な経路を取るものとする。そして、それはトランスポート、ダイレクト・メモリ、およびデバッグ・トランスポート・インタフェースに共通である。もちろん、与えられたソケットを通して送られた異なったトランザクションは、異なった経路を取るかもしれない。すなわち、それらが異なって送られるかもしれない。
m) Write トランザクション(TLM_WRITE_COMMAND)のために、TLM_OK_RESPONSE の応答状態は、write コマンドがターゲットで完了した状態で持っているのを示すものとする。ターゲットはトランジションで BEGIN_RESP フェーズに応答状態を設定するのが強いられる。言い換えると、相互接続コンポーネントがターゲットからうまくいった完了の確認を持っていなくて Write コマンドトランザクションを終了するのが許容されていない。この規則の意図は、ターゲット・シミュレーション・モデルの中でストレージのコヒーレンシーを保証することである。
n) Read トランザクション(TLM_READ_COMMAND)のために、 TLM_OK_RESPONSE の状態は、Read コマンドが終了し、汎用ペイロード・データ配列がターゲットによって変更されたことを示すものとする。ターゲットがトランジションで BEGIN_RESP フェーズに応答状態を設定するのが強いられる。

7.2.6 Summary of obligations on base protocol components

- これは、本書の他の場所でより完全に提示されたいくつかの規則の簡潔な再表示であり、単に便利さを提供しているものである。

7.2.6.1 Obligations on an initiator

基本プロトコルを使用するとき、これはイニシエータの義務の概要である:
a) 各接続にクラス tlm_initiator_socket(または、派生しているクラス)の 1 個のイニシエータ・ソケットをメモリ・マップド・バスに使用すること。
b) デフォルト・テンプレート・タイプ引数 tlm_base_protocol_types を tlm_initiator_socket に使用すること。
c) メソッドの nb_transport_bw と invalidate_direct_mem_ptr を実装すること。
d) 呼び出しの前に忘れずに応答状態と DMI ヒント属性を特にいつもリセットして、引数として b_transport か nb_transport_fw にそれを通過する前にそれぞれの汎用ペイロード・トランザクション・オブジェクトのあらゆる属性を設定すること。
e) トランザクションが、extend される必要があるなら、汎用ペイロードの拡張メカニズムだけ

を使用すること。そして、どんな拡張も無視可能であることをターゲットとあらゆるインターコネクトで許可すること。
f) 引数として b_transport、nb_transport_fw または nb_transport_bw に通過されたあらゆるタイミング・アノテートを受け取ること。
g) トランザクション(または BEGIN_RESP を受けた後に)の完了のときに、応答状態属性の値をチェックすること。

7.2.6.2 Obligations on an initiator using nb_transport

a) 引数としてトランザクションを nb_transport_fw に渡す前に、トランザクション・オブジェクトにメモリ・マネージャを設定すること。そして、トランザクションのアクワイアー・メソッドを call すること。トランザクションが完了するときにはリリース・メソッドを call すること。
b) nb_transport_fw と呼ぶときには、トランザクションの状態に従って、BEGIN_REQ か END_RESP にフェーズ引数を設定すること。前のトランザクションのための END_REQ を受けるまで(または、暗示するまで) BEGIN_REQ を送らないこと。
c) 与えられたトランザクションのために nb_transport_fw と呼ぶときには、現在のシミュレーション時間に加えられるタイミング・アノテートが、値の非減少シーケンスを形成するのを確実にすること。
d) メソッド nb_transport_bw の実装では、適切に入って来るフェーズ値の END_REQ と BEGIN_RESP に応じること。BEGIN_REQ と END_RESP として入って来るフェーズ値は不正である。他のすべての入って来るフェーズ値は無視可能なものとして扱うこと。

7.2.6.3 Obligations on a target

基本プロトコルを使用するとき、これはターゲット上の義務の概要である:
a) 各接続にクラス tlm_target_socket(または、派生しているクラス)の 1 個のターゲット・ソケットをメモリ・マップド・バスに使用すること。
b) デフォルト・テンプレート・タイプ引数 tlm_base_protocol_types を tlm_target_socket に使用すること。
c) メソッドの b_transport、nb_transport_fw、get_direct_mem_ptr、および transport_dbg を実装すること。(ターゲットは便利ソケット simple_target_socket を使用することによって明らかにあらゆるメソッドを実装する必要性を避けることができる。)
d) メソッドの b_transport と nb_transport_fw の実装では、応答状態の例外、DMI ヒント、および各種拡張を含む、汎用ペイロードのあらゆる属性の値を点検して、作用すること。汎用ペイロードの完全な機能性を実装するよりむしろ、ターゲットは、誤り応答を発生させることによって与えられた属性に応じるのを選べるようにすること。応答状態属性の値にトランザクションの成否を示すように設定すること。
e) 引数として b_transport、nb_transport_fw または nb_transport_bw にパスされたあらゆるタイミング・アノテートを受け取ること。
f) get_direct_mem_ptr の実装では、値の false を返すか、コマンドの値と汎用ペイロードのアドレス属性を点検して、作用すること、そして、適切に DMI 記述子のすべての属性を設定すること。(クラス tlm_dmi)
g) transport_dbg の実装では、値 0 を返すか、汎用ペイロードのコマンド、アドレス、データの長さ、およびデータ・ポインタ属性の値を点検して、作用すること。
h) 各インタフェースに関しては、ターゲットは、汎用ペイロードにおけるどんな無視可能な拡張も点検して、作用するかもしれないが、そうすることは強制されない。

7.2.6.4 Obligations on a target using nb_transport

a) nb_transport_bw と呼ぶときには、トランザクションの状態に従って、END_REQ か BEGIN_RESP にフェーズ引数を設定すること。前のトランザクションのために END_RESP を受けるまで(または、暗示されるまで) BEGIN_RESP を送らないこと。
b) 与えられたトランザクションのための nb_transport_bw と呼ぶときには、現在のシミュレーション時間に加えられる値の非減少しているシーケンスを形成するタイミング・アノテーションを確実にすること。
c) メソッド nb_transport_fw の実装では、適切に入って来るフェーズ値の BEGIN_REQ と END_RESP に応じること。 END_REQ と BEGIN_RESP が入って来るフェーズ値は不正である。他のすべての入って来るフェーズ値を無視可能であるとして扱うこと。
d) nb_transport_fw の実装で、return を超えてトランザクション・オブジェクトのポインタか参

照を保つのが必要であるときには、トランザクションのアクワイアー・メソッドを call すること。トランザクション・オブジェクトが使い終わられたらリリース・メソッドを call すること。

7.2.6.5 Obligations on an interconnect component

a) 各接続にクラス tlm_initiator_socket か tlm_target_socket(または、派生されたクラス)の 1 個のイニシエータかターゲット・ソケットをメモリ・マップド・バスに使用すること。

b) 各ソケットにデフォルト・テンプレート・タイプ引数 tlm_base_protocol_types を使用すること。

c) それぞれのイニシエータ・ソケットのためにメソッドの nb_transport_bw と invalidate_direct_mem_ptr を実装して、それぞれのターゲット・ソケットのためにメソッドの b_transport、nb_transport_fw、get_direct_mem_ptr、および transport_dbg を実装すること。(便利ソケットを使用することによって、明らかにあらゆるメソッドを実装する必要性を避けることができる。)

d) 両方の forward と backward パスの適切なソケットを通してあらゆる入って来るトランザクション・オブジェクトを伝えること。唯一の例外は get_direct_mem_ptr と transport_dbg メソッドの実装である。(トランザクション・オブジェクトを進めないで、それぞれ値 false と 0 を返すかもしれない)。

e) トランスポート・インタフェースの実装では、インターコネクト・コンポーネントで修正できる唯一の汎用ペイロード属性は、アドレスと、DMI ヒントと、拡張である。いかなる他の属性も変更しないこと。いかなる他の属性も変更する必要があるコンポーネントは、新しいトランザクション・オブジェクトを組み立てて、その結果、それ自体でイニシエータになるはずである。

f) トランスポート・インタフェースの実装では、イニシエータとターゲットのために上で説明されるようにフェーズとタイミング・アノテートのための基本プロトコル規則を受け取ること。

g) フォワード・パスの汎用ペイロードアドレス属性を解読すること、そして、必要なら、ターゲットの位置に従って、システム・メモリ・マップでアドレス属性を変更すること。これはトランスポート、ダイレクト・メモリ、およびデバッグトランスポートインタフェースに適用される。

h) get_direct_mem_ptr の実装では、フォワード・パスのどんな DMI 記述子属性も変更しないこと。 DMI ポインタ、DMI 開始アドレス、および終わりのアドレスを変更すること。そうすれば、DMI はリターン・パスで適切に属性にアクセスする。

i) invalidate_direct_mem_ptr の実装では、バックワード・パスに沿って呼び出しを通過する前に、アドレスレンジ引数を変更すること。

j) nb_transport_fw の実現で、機能からのリターンを超えてトランザクション・オブジェクトのポインタか参照を保つのが必要であるときは、トランザクションのアクワイアー・メソッドを call すること。トランザクション・オブジェクトが使い終わったらリリース・メソッドを call すること。

k) 各インタフェースに関して、インターコネクトは、汎用ペイロードにおけるどんな無視可能拡張も検査して、作用するかもしれないが、そうすることを強制するわけではない。トランザクションが、さらに拡張される必要があるなら、汎用ペイロード拡張メカニズムだけを使用すること、そして、どんな拡張もターゲットや相互接続で無視可能な拡張を許可すること。拡張のための汎用ペイロード・メモリ管理規則を守ること。

8. その他のクラス

8.1 グローバル・クォンタムとクォンタム・キーパー

8.1.1 イントロダクション

テンポラル・デカップリング機能は、SystemC のプロセスに対して、クォンタムと呼ばれる時間を実際のシミュレーション時刻より先に進めるもので、LT コーディング・スタイルと関連付けて用いられる。テンポラル・デカップリング機能を用いると、スレッドのコンテクスト・スイッチとイベントを減らすことにより非常に高速なシミュレーション速度が実現できる。

テンポラル・デカップリング機能を用いる時には、b_transport()および nb_transport()の引数によって渡される時間は、クォンタム時間の開始（通常、sc_time_stamp()の戻り値である現在のシミュレーション時刻）からの相対時間として定義される。

グローバル・クォンタム時間の値は、唯一の tlm_global_quantum クラスによって管理される。各プロセスはグローバル・クォンタム時間を使うべきだが、1 つのプロセスは、自身のローカル・クォンタム時間の計算を行うことだけが許可されている。

tlm_quantumkeeper クラスには、クォンタム時間に対して制御するためのメソッドが用意されて

いる。
テンポラル・デカップリング機能を用いる時には、一定のコーディング・スタイルを保つために、クォンタム・キーパーを用いることを強く推奨する。しかしながら、SystemC 記述の中にこの機能を実装するということも可能である。tlm_quantumkeeper クラスの使用有無に関わらず、テンポラル・デカップリング機能を用いる全てのモデルは、tlm_global_quantum クラスによって管理されるグローバル・クォンタムを参照すべきである。
tlm_global_quantum クラスのネームスペースは tlm で、tlm_quantumkeeper クラスのネームスペースは tlm_utils である。
テンポラル・デカップリングの詳細については、3.3.2 章 LT コーディング・スタイルとテンポラル・デカップリングも参照のこと。
タイミング・アノテーションに関しては、4.1.3 章トランスポート・インタフェースを用いたタイミング・アノテーションも参照のこと。

8.1.2 クラス定義

```
namespace tlm {
  class tlm_global_quantum
  {
  public:
    static tlm_global_quantum& instance();
    virtual ~tlm_global_quantum();
    void set( const sc_core::sc_time& );
    const sc_core::sc_time& get() const;
    sc_core::sc_time compute_local_quantum();
    protected:
    tlm_global_quantum();
  };
} // namespace tlm

namespace tlm_utils {
  class tlm_quantumkeeper
  {
  public:
    static void set_global_quantum( const sc_core::sc_time& );
    static const sc_core::sc_time& get_global_quantum();
    tlm_quantumkeeper();
    virtual ~tlm_quantumkeeper();
    virtual void inc( const sc_core::sc_time& );
    virtual void set( const sc_core::sc_time& );
    virtual sc_core::sc_time get_current_time() const;
    virtual sc_core::sc_time get_local_time();
    virtual bool need_sync() const;
    virtual void reset();
    virtual void sync();
  protected:
    virtual sc_core::sc_time compute_local_quantum();
  };
} // namespace tlm_utils
```

8.1.3 テンポラル・デカップリングの使い方に関する一般ルール

a) 最大のシミュレーション速度を実現するためには、全てのイニシエータ・モデルがテンポラル・デカップリングを用い、他の SystemC プロセスは全く存在しないか、必要最小限に抑える必要がある。
b) 理想的には、テンポラル・デカップリングを持つイニシエータ・モデルのみに、SystemC プロセスがあり、それぞれの SystemC プロセスにおいては、時間を先に進めておき、クォンタム量を超えた時のみシミュレーション・カーネルに戻る(Yield)ようにする。
c) クォンタム時間は、イニシエータ間の一般的な間隔よりも小さくなるよう設定すべきである。
d) Yield とは、SC_THREAD または SC_CTHREAD の場合には、wait()を実行すること、SC_METHOD の場合には、関数から return することを意味する。
e) テンポラル・デカップリングといえども、あくまでも標準的な SystemC カーネル上で動作するものであるので、イベントはスケジュールできるし、プロセスは停止、再開することもでき、LT

コーディング・スタイルは他のコーディング・スタイルと混在できる。
f) すべてのイニシエータがテンポラル・デカップリングを使用しなければいけないというわけではない。使用しているプロセスと使用していないプロセスは混在可能である。しかし、そうすると、シミュレーション速度が落ちてしまう。
g) テンポラル・デカップリングを行う各イニシエータは、本章で説明するローカル・オフセット時間変数を用いて、ローカルな処理時間と通信時間を計算する。クォンタム・キーパーがローカル・オフセット時間を管理することを強く推奨する。
h) sc_time_stamp()を呼んだときにはあくまでも現在のクォンタムが開始した実シミュレーション時間を返す。
i) ローカル・オフセット時間は、SystemC スケジューラからは不明である。トランスポート・インタフェースを使用する場合は、b_transport()あるいは nb_transport()関数の引数にローカル・オフセット時間を渡すべきである。
j) nb_transport()関数でテンポラル・デカップリングやクォンタム・キーパーを使用することは、ルールの範囲外である。しかし、AT コーディング・スタイル固有のプロセス間通信のオーバーヘッドが大きく、テンポラル・デカップリングによる速度の利点が無効になる可能性があるため、常に有利に働くとは限らない。
k) ローカルでないオブジェクトの値を読み出した場合には自他のプロセスで変更しない限り、あくまでもクォンタムの開始した実際の時間における値を返す。とくに、sc_signal の値はアップデートされない。

8.1.4 tlm_global_quantum クラス

a) tlm_global_quantum クラスによって単一のグローバルなクォンタム時間(同期間隔)が定義される。これがクォンタム時間のデフォルトとなる。テンポラル・デカップリングをサポートしているイニシエータ・モデルは、おおよそこのクォンタム時間が来るごとに実際のシミュレーション時間との間で同期が図られる。ただし、ターゲットの要求によってはそれよりも頻繁に同期が実行される場合もある。
b) それぞれのイニシエータ・モデルはそれぞれ別のクォンタム時間を持つことも可能であるが、一般的には全てのイニシエータ・モデルはグローバルなクォンタム時間を持つ。同期の回数が少なくてもすむイニシエータは他よりクォンタム時間を長く設定できるわけではあるが、一般的には、同期の回数が多く必要なモデルがもっともシミュレーション時間に大きく影響してしまうからである。
c) instance()メソッドは、唯一のグローバル・クォンタム・オブジェクトのリファレンスを返す。
d) set()メソッドは、引数で受け取った値をグローバル・クォンタム値として設定する。
e) get()メソッドは、グローバル・クォンタムの値を返す。
f) compute_local_quantum()は、唯一のグローバル・クォンタムを元に、ローカル・クォンタムの値を計算して返す。グローバル・クォンタムの次の倍数から、sc_time_stamp の値を引くことにより計算する。compute_local_quantum()がグローバル・クォンタムの整数倍のシミュレーション時間でコールされる場合は、ローカル・クォンタムはグローバル・クォンタムと等しい。そうでない場合は、ローカル・クォンタムはグローバル・クォンタムよりも小さい。

8.1.5 tlm_quantumkeeper クラス

a) コンストラクタはローカル・タイムのオフセットを SC_TIME_ZERO に設定するが、仮想関数の compute_local_quantum()はコールしない。これは、コンストラクタでローカル・クォンタムを計算せずに、アプリケーションがクォンタム・キーパー・オブジェクトを生成直後に、reset()メソッドをコールからである。
b) tlm_quantum_keeper クラスの実装では、sc_time クラスの静的オブジェクトは生成せずに、コンストラクタが sc_time クラスのオブジェクトを生成してもよい。これは、アプリケーションが最初のクォンタム・キーパー・オブジェクトを生成する前に、sc_core::sc_set_time_resolution()関数をコールしてもよいことを意味している。
c) set_global_quantum()メソッドは、グローバル・クォンタム時間を引数によって設定する。しかしローカルのクォンタム時間は変更されない。get_global_quantum()メソッドはグローバル・クォンタム時間を返す。set_global_quantum()を呼んだ後には、reset()メソッドを呼び出し、ローカル・クォンタム時間を再計算させることが望ましい。
d) get_local_time()メソッドはローカル・オフセット時間を返す。
e) get_current_time()メソッドは、ローカル時間、すなわち、sc_time_stamp()+ローカル・オフ

セット時間を返す。
f) inc()メソッドは、ローカル・オフセット時間を引数によって渡しローカル時間を実時間よりも先に進める。
g) set()メソッドは、引数で渡された値をローカル・オフセット時間に設定する。
h) need_sync()メソッドは、現在のローカル・オフセット時間がローカル・クォンタムより大きい時に true を返す。
i) sync()メソッドは、wait(ローカル・オフセット時間)を呼び出し、実際のシミュレーション時間とローカルの時間との同期をとり、reset()メソッドを呼び出す。
j) reset()メソッドは、compute_local_quantum()を呼び出し、ローカル・オフセット時間を 0 に戻す。
k) tlm_quantumkeeper クラスの compute_local_quantum()メソッドは、tlm_global_quantum クラスの compute_local_quantum メソッドを呼び出す。
l) tlm_quantum_keeper クラスはクォンタム・キーパーのデフォルトの実装となり、このクラスを継承したクラスを作り、compute_local_quantum メソッドを書き換えることが可能ではあるが、普通ではない。
m) ローカル・オフセット時間がローカル・クォンタム以上になった場合には、プロセスを yield させる必要があるが、自分で wait()を呼び出さないで、sync()メソッドを呼び出すことを強く推奨する。
n) ローカル・クォンタム量を超えた場合に同期を自動的に実行するようなしかけはない。したがって、イニシエータは自分で need_sync メソッドを呼び出し、必要に応じて sync()を呼び出す必要がある。
o) b_transport()メソッドは、そのコール前後でグローバル時間が変化していた場合に、自身で yield するかもしれない。ローカル・オフセット時間やタイミング・アノテーションの値は、常にグローバル時刻からの相対値で表現される。b_transport()あるいは nb_transport_fw()からの戻りでは、イニシエータが set()メソッドのコールによって、クォンタム・キーパーのローカル・オフセット時間を設定し、need_sync()メソッドのコールによって同期チェックを行う必要がある。
p) イニシエータがローカル・クォンタム量を超えるよりも前に実行をサスペンドして、実時間を先に進んでしまっているローカル時間に追いつかせる必要が生じた場合、sync()を呼び出すか、あるいは明示的に別のイベントを wait させてもいい。途中で同期させるこの方法を、sync-on-demand(要求に応じた同期)と呼ぶ。
q) sync()を頻繁にコールし過ぎると、テンポラル・デカップリングの効果が減ってしまう
例

```
struct Initiator: sc_module // Loosely-timed initiator
{
  tlm_utils::simple_initiator_socket<Initiator> init_socket;
  tlm_utils::tlm_quantumkeeper m_qk; // The quantum keeper
  SC_CTOR(Initiator) : init_socket("init_socket") {
    SC_THREAD(thread); // The initiator process
    ...
    m_qk.set_global_quantum( sc_time(1, SC_US) ); // Replace the global quantum
    m_qk.reset(); // Re-calculate the local quantum
  }
  void thread() {
    tlm::tlm_generic_payload trans;
    sc_time delay;
    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    for (int i = 0; i < RUN_LENGTH; i += 4) {
      int word = i;
      trans.set_address(i);
      trans.set_data_ptr( (unsigned char*)(&word) );
      delay = m_qk.get_local_time(); // Annotate b_transport with local time
      init_socket->b_transport(trans, delay );
      qk.set( delay ); // Update qk with time consumed by target
      m_qk.inc( sc_time(100, SC_NS) ); // Further time consumed by initiator
      if ( m_qk.need_sync() ) m_qk.sync(); // Check local time against quantum
    }
```

```
    }
    ...
};
```

8.2 ペイロード・イベント・キュー

8.2.1 イントロダクション

ペイロード･イベント･キュー(PEQ)は､トランザクション･オブジェクトに関連付けられたSystemCのイベント通知用のキューである｡それぞれのトランザクションは遅延と共にPEQ内に格納され､現在のシミュレーション時間と設定された遅延時間の合計時間になった時点でPEQから出力される｡

２つのペイロード・イベント・キューがユーティリティとして提供されている。PEQは便利なだけではなく、ATコーディング・スタイルにおけるタイミング・アノテーションの仕組みを理解する上でも概念的に関連がある。

しかし、ここで説明するペイロード・イベント・キューを使わなくても、アプロキシメイトリー・タイムド・モデルの実装は可能である。アプロキシメイトリー・タイムド・モデルでは、nb_transport()によって遅延時間を持つトランザクションを受け取る時、PEQへの格納が適している。PEQは、正確なシミュレーション時間に発生するnb_transport()コールに関するタイミング・ポイントをスケジューリングする。

PEQのnotify()メソッドを、引数にディレイを指定してコールすることにより、PEQへトランザクションを格納する。引数を取らずに、すぐにスケジューラへ通知を行うnotify()メソッドも存在している。

遅延時間は現在のシミュレーション時間(sc_time_stamp)との加算に使われ、トランザクションがPEQの最後から出力する時間となる。イベントのスケジューリングは、内部的にsc_eventクラスを利用し、SystemCの時間イベント通知によって管理される。待ち状態のイベント通知がある時にnotify()メソッドがコールされた場合は、他の通知がキャンセルまでの間、最も早いシミュレーション時間における通知が残ったままとなる。

PEQには2種類があって、それぞれトランザクションの取り出し方法が異なる。peq_with_getの場合は、get_event()メソッドをコールすると、取り出しが可能な状態のトランザクションに関連するイベントを返す。get_next_transaction()メソッドは、あるタイミングで有効になっているトランザクションの１つを取り出すために、繰り返しコールされるメソッドである。評価フェーズにおいてイベント通知が発生した時にPEQからトランザクションを取り出すことができない場合には、その後のタイミングで再度get_next_taransaction()コールによって取り出すまでトランザクションは有効なまま継続する。受け取れるトランザクションが１つもない場合は、get_next_transaction()はNULLポインタを返す。

peq_with_cb_and_phaseの場合は、コールバック・メソッドをコンストラクタ引数に登録することで､各トランザクションを取り出す際にそのコールバック・メソッドがコールされる。このPEQを用いる場合には、通知ごとにトランザクション・オブジェクトとフェーズ・オブジェクトの両方が、コールバック関数の引数として渡される。

トランザクションはPEQに追加された順番ではなく、シミュレーション時間とディレイ引数の合計で決まるスケジュール時間順に基づいて取り出される。同じ時間に取り出し可能なトランザクションが複数ある場合には、同一のデルタ・サイクルで、追加された順番に従って１つずつトランザクションを取り出せる。トランザクションが失われたりキャンセルされることはない。

8.2.2 クラス定義

```
namespace tlm_utils {
  template <class PAYLOAD>
  class peq_with_get : public sc_core::sc_object
  {
  public:
    typedef PAYLOAD transaction_type;
    peq_with_get(const char* name);
    void notify(transaction_type& trans, sc_core::sc_time& t);
    void notify(transaction_type& trans);
    transaction_type* get_next_transaction();
    sc_core::sc_event& get_event();
  };
```

```
template<typename OWNER, typename TYPES=tlm::tlm_base_protocol_types>
class peq_with_cb_and_phase : public sc_core::sc_object
{
public:
  typedef typename TYPES::tlm_payload_type tlm_payload_type;
  typedef typename TYPES::tlm_phase_type tlm_phase_type;
  typedef void (OWNER::*cb)(tlm_payload_type&, const tlm_phase_type&);
  peq_with_cb_and_phase(OWNER* , cb );
  peq_with_cb_and_phase(const char* , OWNER* , cb);
  ~peq_with_cb_and_phase();
  void notify (tlm_payload_type& , tlm_phase_type& , const sc_core::sc_time& );
  void notify (tlm_payload_type& , tlm_phase_type& );
};

} // namespace tlm_utils
```

8.3 アナリシス・インタフェースとアナリシス・ポート

アナリシス・ポートは、複数のコンポーネントにまたがるトランザクションに対する解析機能、例えば、機能の正確性をチェックするとか機能カバレージ情報を取得するなどを実現するために使用される。

アナリシス・ポートのもっとも大切な点は、一つのアナリシス・ポートが複数のチャネル(サブスクライバー)にバインド可能であり、アナリシス・ポートに対して write()をコールしたときにそれぞれのサブスクライバーに対して同一の write()をコールさせるようにすることである。アナリシス・ポートは 0 以上複数のサブスクライバーに対してバインドすることができ、バインドしなくても構わない。

それぞれのサブスクライバーは、tlm_analysis_if の write()メソッドを実装する。このメソッドは const reference によってトランザクションに渡され、サブスクライバーはただちに処理を実行する。もし、サブスクライバーがトランザクションのライフタイムを延長しておきたいような場合には、自分でトランザクション(generic_payload)に対する deep_copy()メソッドを行い、サブスクライバー自身がコピーして作成したトランザクションの持ち主になり自分でメモリ管理を行うことになる。

アナリシス・ポートはモデルの動作そのものに関わるようなことに対して用いてはならず、その他の解析等の側面的な解析機能のみで使用しなければならない。 tlm_analysis_if インタフェースは、tlm_write_if を継承したクラスである。この tlm_write_if は、アナリシスの目的だけではなく、他の目的にも使用可能である。例えば、8.2 の Payload event queue においても用いられている。

TLM2 キットには、tlm_analysis_fifo という、段数が無限の tlm_fifo に対して tlm_alansysis_if を用いてトランザクションを write できるようにするものが含まれている。tlm_fifo は、tlm_analysis_triple というトランザクションと、開始時間、終了時間を含むものもサポートしている。

8.3.1 クラス定義

```
namespace tlm {
  // Write interface
  template <typename T>
  class tlm_write_if : public virtual sc_core::sc_interface {
  public:
    virtual void write( const T& ) = 0;
  };

  template <typename T>
  class tlm_delayed_write_if : public virtual sc_core::sc_interface {
  public:
    virtual void write( const T& , const sc_core::sc_time& ) = 0;
  };

  // Analysis interface
  template < typename T >
  class tlm_analysis_if : public virtual tlm_write_if<T>
  {
```

```
  };

  template < typename T >
  class tlm_delayed_analysis_if : public virtual tlm_delayed_write_if<T>
  {
  };

  // Analysis port
  template < typename T>
  class tlm_analysis_port : public sc_core::sc_object , public virtual tlm_analysis_if< T >
  {
  public:
    tlm_analysis_port();
    tlm_analysis_port( const char * );
    // bind and () work for both interfaces and analysis ports, since analysis ports implement the analysis
interface
    void bind( tlm_analysis_if<T> & );
    void operator() ( tlm_analysis_if<T> & );
    bool unbind( tlm_analysis_if<T> & );
    void write( const T & );
  };

  // Analysis triple
  template< typename T>
  struct tlm_analysis_triple {
    sc_core::sc_time start_time;
    T transaction;
    sc_core::sc_time end_time;
    // Constructors
    tlm_analysis_triple();
    tlm_analysis_triple( const tlm_analysis_triple &triple );
    tlm_analysis_triple( const T &t );
    operator T() { return transaction; }
    operator const T& () const { return transaction; }
  };

  // Analysis fifo - an unbounded tlm_fifo
  template< typename T >
  class tlm_analysis_fifo :
  public tlm_fifo< T > ,
  public virtual tlm_analysis_if< T > ,
  public virtual tlm_analysis_if< tlm_analysis_triple< T > > {
  public:
    tlm_analysis_fifo( const char *nm ) : tlm_fifo<T>( nm, -16 ) {}
    tlm_analysis_fifo() : tlm_fifo<T>( -16 ) {}
    void write( const tlm_analysis_triple<T> &t ) { nb_put( t ); }
    void write( const T &t ) { nb_put( t ); }
  };
} // namespace tlm
```

8.3.2 ルール

a) tlm_write_if と tlm_analysis_if は、単方向、ネゴシエーション無し、ノンブロッキングのインタフェースであり、引数で渡されるトランザクションに対して即座に応答しなければならない。

b) tlm_analysis_port のコンストラクタは、ポートのインスタンス名の文字列を引数として使用する。これは、ベースクラスの sc_object に渡される。

c) bind メソッドは、サブスクライバーのアナリシス・ポートのインスタンスを登録し、write メソッドが呼ばれたときに登録されたサブスクライバー内の write メソッドが呼ばれる。複数のサブスクライバーを単一のアナリシスポートインスタンスに登録することも可能。

d) operator() は、bind()メソッドと同一。

e) サブスクライバーの数が 0 であっても構わない。その場合には、write()メソッドは伝達され

ない。
f) unbind()メソッドは、bind()の逆であり、サブスクライバーのリストから削除される。
g) tlm_analysys_port の write()メソッドが呼ばれた場合、そのポートに登録されたすべてのサブスクライバーに対して、write()メソッドが const リファレンスの引数と共に呼ばれる。
h) write()メソッドはノンブロッキングであり、その実装の中に wait()を含んではいけない。
i) write()メソッドは、const reference によって渡されたトランザクションの内容や、その中のポインタ先のデータ(data,バイトイネーブル配列)を書き換えてはいけない。
j) もし、write()メソッドが return する前にそのトランザクション・オブジェクトに対しての処理を完了することができない場合には、新しいトランザクション・オブジェクトを deep_copy()メソッドによってコピーしてから実行する。コピーされたオブジェクトのメモリ管理はサブスクライバーが責任を負う。
k) tlm_analysys_fifo クラスのコンストラクタは、bound されてない tlm_fifo を構成する。
l) tlm_analysys_fifo クラスの write()メソッドは tlm_fifo ベースクラスの nb_put()を同一の引数とともに呼び出す。
例

```
struct Trans // Analysis transaction class
{
  int i;
};

struct Subscriber: sc_object, tlm::tlm_analysis_if<Trans>
{
  Subscriber(const char* n) : sc_object(n) {}
  virtual void write(const Trans& t)
  {
    cout << "Hello, got " << t.i << "¥n"; // Implementation of the write method
  }
};

SC_MODULE(Child)
{
  tlm::tlm_analysis_port<Trans> ap;
  SC_CTOR(Child) : ap("ap")
  {
    SC_THREAD(thread);
  }
  void thread()
  {
    Trans t = {999};
    ap.write(t); // Interface method call to the write method of the analysis port
  }
};

SC_MODULE(Parent)
{
  tlm::tlm_analysis_port<Trans> ap;
  Child* child;
  SC_CTOR(Parent) : ap("ap")
  {
    child = new Child("child");
    child->ap.bind(ap); // Bind analysis port of child to analysis port of parent
  }
};

SC_MODULE(Top)
{
  Parent* parent;
  Subscriber* subscriber1;
  Subscriber* subscriber2;
```

```
  SC_CTOR(Top)
  {
    parent = new Parent("parent");
    subscriber1 = new Subscriber("subscriber1");
    subscriber2 = new Subscriber("subscriber2");
    parent->ap.bind( *subscriber1 ); // Bind analysis port to two separate subscribers
    parent->ap.bind( *subscriber2 ); // This is the key feature of analysis ports
  }
};
```

9. TLM1 レガシー

以下の TLM1 のコア・インタフェース及び tlm_fifo チャネルは依然として TLM 2.0 標準であるが、このドキュメントでは詳しく触れない。

9.1 TLM1 コア・インタフェース

シグネチャ付きのトランスポート・メソッドである transport(const REQ& , RSP&) は TLM 1.0 に含まれていなかったが、TLM 2.0 において付け加えられた。

9.2 TLM1 fifo インタフェース

（省略）

9.3 tlm_fifo

（省略）

1. TLM 2.0 Glossary (用語集)

注：青字はSystemC LRMにて定義・使用されている用語

Words	Descriptions	訳語	説明
adaptor	A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two transaction level interfaces, often at different abstraction levels. Typically, an adaptor is used to convert between two transaction-level interfaces of different types. See transactor.	アダプタ	トランザクション・レベル・インタフェースとピン・レベル・インタフェース間、もしくは、異なる抽象度のトランザクション・レベル・インタフェース間を接続するモジュール。異なるタイプの 2 つのトランザクション・レベル・インタフェースを変換するために用いられる。
approximately timed (AT)	A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. See cycle approximate.	アプロキシメイトリー・タイムド	外部から観測可能な状態と対応した詳細リファレンス・モデルの状態に一対一対応が付き、その対応が状態遷移を保つ（正確なタイミングは保たない）ようなモデリング・スタイル。タイミング精度は定義されない。
attribute (of a transaction)	Data that is part of and carried with the transaction and is implemented as a member of the transaction object. These may include attributes inherent in the bus or protocol being modeled, and attributes that are artefacts of the simulation model (a timestamp, for example).	アトリビュート	トランザクションで運ばれる、トランザクション・オブジェクトのメンバーとして実装されるデータ。モデル化された BUS もしくはプロトコルで継承されたアトリビュート、タイムスタンプ等とシミュレーション・モデルの成果物も含む。
automatic deletion	A generic payload extension marked for automatic deletion will be deleted at the end of the transaction lifetime, that is, when the transaction reference count reaches 0.	自動削除	自動削除のためにマークされた汎用ペイロード拡張はトランザクション・ライフタイムの終わり、すなわちトランザクション参照カウントが 0 に達すると、削除される。
backward path	The calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator.	バックワード・パス	ターゲットかインターコネクト・コンポーネントが別のインターコネクト・コンポーネントかイニシエータの向きにインタフェース・メソッド・コールバックする際に使用するパス。

© Copyright 2009 JEITA, All rights reserved

base protocol	A protocol types class consisting of the generic payload and tlm_phase types, together with an associated set of protocol rules which together ensure maximal interoperability between transaction-level models	基本プロトコル	汎用ペイロードとtlm_phase タイプからなるプロトコル・タイプのクラスであり、トランザクション・レベル・モデルの間の最大限度の相互利用性を確実にするため関連セットのプロトコル規則を持つ.
bidirectional interface	A TLM 1.0 transaction level interface in which a pair of transaction objects, the request and the response, are passed in opposite directions, each being passed according to the rules of the unidirectional interface. For each transaction object, the transaction attributes are strictly readonly in the period between the first timing point and the end of the transaction lifetime.	双方向インタフェース	TLM 1.0 のトランザクション・レベル・インタフェース。トランザクション・オブジェクトのペア、リクエストとレスポンスが、それぞれ反対方向に、単方向インタフェースのルールに従って送信される。各トランザクション・オブジェクトのトランザクション・アトリビュートはトランザクション存続期間中は厳密に読み出しのみ。(値を書き換えてはいけない。)
blocking	Permitted to call the wait method. A blocking function may consume simulation time or perform a context switch, and therefore shall not be called from a method process. A blocking interface defines only blocking functions.	ブロッキング	wait 関数を呼び出すことが許されている。ブロッキング関数はシミュレーション時間を消費する。また、コンテクスト・スイッチする。したがってメソッド・プロセスから呼び出してはいけない。ブロッキング・インタフェースはブロッキング関数のみ定義する。
blocking transport interface	A blocking interface of the TLM-2 standard which contains a single method b_transport. Beware that there still exists a blocking transport method named transport, part of TLM-1.0.	ブロッキング・トランスポート・インタフェース	TLM-2 標準におけるブロッキングインフェースであり、ただ一つのメソッド b_transport からなる。TLM-1.0 標準の一部である transport という名のブロッキング・トランスポート・メソッドは依然として存在する事に注意されたい。
bridge	A module that connects together two similar or dissimilar transaction-level interfaces, each representing a memory-mapped bus or other protocol, usually at the same abstraction level. A bus bridge is a device that connects two similar or dissimilar buses together. A communication bridge is a device that connects network segments on the data link layer of a network. See transactor.	ブリッジ	2 つのトランザクション・レベル・インタフェース間を接続するモジュール。それぞれはメモリ・マップド・バスもしくは他のプロトコルを表すもので、通常は同じ抽象度のもの。バス・ブリッジは 2 つのバスを接続するデバイス。コミュニケーション・ブリッジはネットワークのデータ・リンク層のネットワーク・セグメントを接続するデバイス。

© Copyright 2009 JEITA, All rights reserved

caller	In a function call, the sequence of statements from which the given function is called. The referent of the term may be a function, a process, or a module. This term is used in preference to initiator to refer to the caller of a function as opposed to the initiator of a transaction.	呼び出し元関数	ある関数を呼び出している関数、プロセス、モジュール。この用語は、トランザクションのイニシエータとしてではなく、関数の呼び出し元としてイニシエータを指すときによく使われる。
callee	In a function call, the function that is called by the caller. This term is used in preference to target to refer to the function body as opposed to the target of a transaction.	呼び出し先関数	呼び出し元関数から呼び出される関数。この用語は、トランザクションのターゲットとしてではなく、呼び出される関数本体としてターゲットを指すときによく使われる。
channel	A class that implements one or more interfaces or an instance of such a class. A channel may be a hierarchical channel or a primitive channel or, if neither of these, it is strongly recommended that a channel at least be derived from class sc_object. Channels serve to encapsulate the definition of a communication mechanism or protocol. (SystemC term)	チャネル	1 つ以上のインタフェースを実装したクラス、もしくは、そのクラスのインスタンス。チャネルは階層チャネル、プリミティブ・チャネル、そうでなければ、sc_object クラスを継承したチャネルが強く推薦される。チャネルはコミュニケーション機構もしくはプロトコルをの定義を隠蔽するのに使われる。(SystemC 用語)
child	An instance that is within a given module. Module A is a child of module B if module A is within module B. (SystemC Term)	子	あるモジュールの中にインスタンスされたもの。 モジュールBの中にモジュールAがあるなら、モジュール A はモジュールBの子である。
combined interfaces	Pre-defined groups of core interfaces used to parameterize the socket classes. There are four combined interfaces: the blocking and non-blocking forward and backward interfaces.	統合インタフェース	ソケットクラスをパラメタライズするためあらかじめ定義されたコア・インタフェースのグループ。次の 4 つの統合インタフェースがある: ブロッキングとノンブロッキング、フォワードとバックワード(を統合した)インタフェース。
convenience socket	A socket class, derived from tlm_initiator_socket or tlm_target_socket, that implements some additional functionality and is provided for convenience. Several convenience sockets are provided as utilities	便利ソケット	tlm_initiator_socket か tlm_target_socket から派生したソケットクラスであり、何らかの追加機能を実装して、利便性のため提供される。 ユーティリティとして数個の便利ソケットが提供される。

© Copyright 2009 JEITA, All rights reserved

core interface	One of the [N] specific transaction level interfaces defined in this standard, not derived from any other interface, and with a template parameter representing a transaction type,. Each core interface is an interface proper. The core interfaces are distinct from the generic payload API.	コア・インタフェース	この標準で定義されるトランザクション・レベル・インタフェースで、他のいかなるインタフェースも継承しておらず、トランザクション・タイプをテンプレート・パラメータとして持つ。コア・インタフェースはインタフェース・プロパである。コア・インタフェースは汎用ペイロード API とは違う。
cycle accurate	A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly reevaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.	サイクル精度	モデルの外部境界で、あるサイクルでのモデルの状態の予測ができる、従って、モデルの状態とそれに対応する RTL モデルの外部観測可能な状態との対応付けが毎サイクルできるモデリング・スタイル。ただし、毎サイクルでモデル全体の状態の再評価、全ての外部ピン、内部レジスタの状態を表すことは要求されない。この用語は、サイクルの概念を持つモデルにのみ使われる。
cycle approximate	A model for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding cycle accurate model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. This term is only applicable to models that have a notion of cycles.	サイクル近似	モデルの外部観測可能な状態とそのモデルに対応するサイクル精度モデルの状態間に 1 対 1 の対応付けが可能なモデル。その対応付けは、状態遷移を保持するが、正確なタイミングは保持しない。タイミングの精度は定義されない。この用語は、サイクルの概念を持つモデルにのみ使われる。

© Copyright 2009 JEITA, All rights reserved

cycle count accurate, cycle count accurate at transaction boundaries	A modeling style in which it is possible to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model as sampled at the timing points marking the boundaries of a transaction. A cycle count accurate model is not required to be cycle accurate in every cycle, but is required to accurately predict both the functional state and the number of cycles at certain key timing points as defined by the boundaries of the transactions through which the model communicates with other models.	サイクル数精度、トランザクション境界でのサイクル数精度	モデルの状態とそのモデルに対応する RTL モデルの外部観測可能な状態との間で、トランザクション境界を表すタイミング・ポイントで状態をサンプルすることで 1 対 1 の対応付けが可能なモデリング・スタイル。サイクル数精度モデルは全てのサイクルでサイクル精度である必要はないが、機能的状態とトランザクション境界で定義されるキーとなるタイミング・ポイントでサイクル数が正確に予測されることが求められる。
declaration	A C++ language construct that introduces a name into a C++ program and specifies how the C++ compiler is to interpret that name. Not all declarations are definitions. For example, a class declaration specifies the name of the class but not the class members, while a function declaration specifies the function parameters but not the function body. (See definition.) (C++ term)	宣言	C++プログラムにある名前を導入し、C++コンパイラがその名前をいかに解釈すべきかを指定する C++の構成概念。宣言は定義とは限らない。例えば、クラス宣言はクラス名を規定するが、クラス・メンバーは規定しない。一方、関数宣言は、関数のパラメータと関数本体を規定する。(C++用語)
definition	The complete specification of a variable, function, type, or template. For example, a class definition specifies the class name and the class members, and a function definition specifies the function parameters and the function body. (See declaration.) (C++ term)	定義	変数、関数、型、テンプレートの完全な規定。例えば、クラス定義はクラス名とクラス・メンバーを規定する。関数定義は、関数のパラメータと関数本体を規定する。(C++用語)
extension	A user-defined object added to and carried around with a generic payload transaction object, or a user-defined class that extends the set of values that are assignment compatible with the tlm_phase type. An ignorable extension may be used with the base protocol, but a mandatory extension requires the definition of a new protocol types class.	拡張	汎用ペイロード・トランザクション・オブジェクトに付け加えられ、かつ一緒に転送されるユーザ定義オブジェクト、あるいは tlm_phase タイプに代入コンパチブルな値のセットを拡張するユーザ定義クラス。無視可能拡張は基本プロトコルと一緒に使用してもよいが、必須拡張は新しいプロトコル・タイプのクラスの定義を必要とする。

© Copyright 2009 JEITA, All rights reserved

forward path	The calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target.	フォワード・パス	イニシエータかインターコネクト・コンポーネントが別のインターコネクト・コンポーネントかターゲットの向きにインタフェース・メソッド・コールする際に使用するパス。
generic payload	A specific set of transaction attributes and their semantics together defining a transaction level protocol which may be used to achieve a degree of interoperability between untimed, loosely timed and approximately timed models for components communicating over a memory-mapped bus. The attributes are partitioned into those applicable for all models, and those only applicable for approximately timed models	汎用ペイロード	トランザクション・アトリビュートのセットとメモリ・マップド・バスを介して通信するコンポーネントのアンタイムド、ルーズリー・タイムド、アプロキシメイトリー・タイムド・モデル間のある程度の相互利用を達成するためのセマンティクス。アトリビュートは、全てのモデルに使えるものとアプロキシメイトリー・タイムド・モデルにのみ使えるものとに分けられる。
global quantum	The default time quantum used by every quantum keeper and temporally decoupled initiator. The intent is that all temporally decoupled initiators should typically synchronize on integer multiples of the global quantum, or more frequently on demand.	グローバル・クォンタム	すべてのクォンタム・キーパーとテンポラル・デカップリングされたイニシエータが使用する、規定のタイム・クォンタム。 意図はすべてのテンポラル・デカップリングされたイニシエータが通常はグローバル・クォンタムの整数倍、または、要求に応じてより頻繁に同期を取るべきであるということ。

© Copyright 2009 JEITA, All rights reserved

initiator	A module that can initiate transactions. The initiator is responsible for initializing the state of the transaction object, and for deleting or reusing the transaction object at the end of the transaction's lifetime. An initiator is usually a master and a master an initiator, but the term initiator means that a component can initiate transactions, whereas the term master means that a component can take control of a bus. In the case of the TLM 1.0 interfaces, the term initiator as defined here may not be strictly applicable, so the terms caller and callee may be used instead for clarity.	イニシエータ	″トランザクションを開始できるモジュール。イニシエータはトランザクションのライフタイムの終わりに、トランザクション・オブジェクトの状態を初期化すること、そして、トランザクション・オブジェクトを削除するか、または再利用する責任がある。イニシエータは通常マスタであり、マスタは通常イニシエータである。ただし、イニシエータと言う用語はトランザクションを開始することのできるコンポーネントを意味し、マスタと言う用語はバスの制御を取ることのできるコンポーネントを意味する。TLM 1.0 インタフェースにおいては、イニシエータと言う用語は厳密には適切でないため、代わりに呼び出し元関数、呼び出し先関数といった用語を使って明確にするのがよい。″
initiator socket	A class containing a port for interface method calls on the forward path and an export for interface method calls on the backward path. A socket also overloads the SystemC binding operators to bind both port and export	イニシエータ・ソケット	フォワード・パス上の、インタフェース・メソッド・コールのためのポートを含むクラス、及び、バックワード・パス上のインタフェース・メソッド・コールのためのエクスポート。 また、このソケットはポートとエクスポートの両方をバインドするために SystemC のバインド演算子をオーバーロードする。
interconnect component	A module that accesses a transaction object, but does act as an initiator or a target with respect to that transaction. An interconnect component may or may not be permitted to modify the attributes of the transaction object, depending on the rules of the payload. An arbiter or a router would typically be modeled as an interconnect component, the alternative being to model it as a target for one transaction and an initiator for a separate transaction.	インターコネクト・コンポーネント	トランザクション・オブジェクトにアクセスするが、そのトランザクションに関してイニシエータあるいはターゲットとして作用するモジュール。ペイロードのルールに従って、インターコネクト・コンポーネントがトランザクション・オブジェクトのアトリビュートを変更することを許可または禁止される。

© Copyright 2009 JEITA, All rights reserved

interface	A class derived from class sc_interface. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper. (SystemC term)	インタフェース	sc_interface を継承したクラス。インタフェース・プロパはインタフェース。オブジェクト指向の意味では、チャネルもインタフェース。ただし、チャネルはインタフェース・プロパではない。(SystemC 用語)
Interface Method Call (IMC)	A call to an interface method. An interface method is a member function declared within an interface. The IMC paradigm provides a level of indirection between a method call and the implementation of the method within a channel such that one channel can be substituted with another without affecting the caller. (SystemC term)	インタフェース・メソッド・コール	インタフェース・メソッドの呼び出し。インタフェース・メソッドとはインタフェース内で宣言されたメンバー関数。IMC の枠組みは、メソッド呼び出しとメソッドの実装の分離を提供する。例えば、呼び出し元に影響を与えることなく、チャネルの置き換えが可能となる。(SystemC 用語)
interface proper	An abstract class derived from class sc_interface but not derived from class sc_object. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically contains no function definitions and no data members. (SystemC term)	インタフェース・プロパ	sc_interface を継承した抽象クラス（ただし、sc_object は継承しない）。インタフェース・プロパはチャネル内で実装されるメソッドを宣言し、それらはポートを介して呼び出される。インタフェース・プロパは純粋バーチャル関数の宣言を持つが、通常は、関数定義、データ・メンバは持たない。(SystemC 用語)
interoperability	The ability of two or more transaction level models from diverse sources to exchange information using the interfaces defined in this standard. The intent is that models that implement common memory-mapped bus protocols in the programmers view use case should be interoperable without the need for explicit adaptors. Furthermore, the intent is to reduce the amount of engineering effort needed to achieve interoperability for models of divergent protocols or use cases, although it is expected that adaptors will be required in general.	相互利用性	この標準で定義するインタフェースを使うことにより、異なった出所のトランザクション・レベル・モデルで情報を交換できるようになること。この意味は、PV ユースケースで共通メモリ・マップド・バスプロトコルを実装したモデルアダプター無しで相互利用できるはずだということ。さらに、異なったプロトコル、ユースケースのモデルについても、一般にはアダプタが要求されるが、相互利用のための手間を削減を意味する。

© Copyright 2009 JEITA, All rights reserved

lifetime (of an object)	The lifetime of an object starts when storage is allocated and the constructor call has completed, if any. The lifetime of an object ends when storage is released or immediately before the destructor is called, if any. (C++ term)	(オブジェクトの) ライフタイム	オブジェクトのライフタイムは、ストレージがある場合アロケートされ、コンストラクタ呼び出しが完了した時に開始され、そしてストレージがリリースされた時、もしくはデストラクタが呼び出される直前に終了する。(C++用語)
lifetime (of a transaction)	The period of time that starts when the transaction becomes valid and ends when the transaction becomes invalid. Because it is possible to pool or re-use transaction objects, the lifetime of a transaction object may be longer than the lifetime of the corresponding transaction. For example, a transaction object could be a stack variable passed as an argument to multiple put calls of the TLM1.0 interface.	(トランザクションの) ライフタイム	トランザクションが有効になった時に始まり、無効になるまでの時間の周期を指す。トランザクション・オブジェクトはプールまたは再利用される可能性がある為、そのトランザクション・オブジェクトのライフタイムは、対応するトランザクション自身のライフタイムより長い場合がある。例えば、トランザクション・オブジェクトは、TLM1.0インタフェースの呼び出し時のように、複数の引数スタックされる事もある。
local quantum	The amount of simulation time remaining before the initiator is required to synchronize. Typically, the local quantum equals the current simulation time subtracted from the next largest integer multiple of the global quantum, but this calculation can be overridden for a given quantum keeper.	ローカル・クォンタム	イニシエータが同期を要求される前に残っているシミュレーション時間の量。通常、ローカル・クォンタムはグローバル・クォンタムの次の最も大きい整数倍から現在のシミュレーション時刻を差し引いたものと等しいが、クォンタム・キーパーを与えることで、この計算をオーバーライドすることができる。

© Copyright 2009 JEITA, All rights reserved

loosely timed	A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.	ルーズリー・タイムド	オペレーティング・システムのブートと、複数スレッド間の明示的な同期管理に必要とされる機能をサポートする為に、最適で十分な最小タイミング情報を与えたモデリング・スタイル。ルーズリー・タイムド・モデルには、タイマー・モジュール、または調停間隔や実行スロット長が含まれる場合がある。ユーザによっては、それらプロトコルの頑健性をテストするにあたり、ルーズリー・タイムドの記述にランダム遅延を入れる場合があるが、しかしこれによってモデリング・スタイルの基本的な性質を変える事はない。
master	This term has no precise technical definition in this standard, but is used to mean a module or port that can take control of a memory-mapped bus in order to initiate bus traffic, or a component that can execute an autonomous software thread and thus initiate other system activity. Generally, a bus master would be an initiator.	マスタ	この用語には明確な標準の技術定義は無いが、バス・トラフィックを開始する為にメモリ・マップド・バス制御を行う事の出来るモジュール、またはポート、また自律的なソフトウェア・スレッドを実行する事が出来るものや、その他システムを起動するコンポーネントを指す。一般的には、バス・マスタはイニシエータになりうる。
method	A function that implements the behavior of a class. This term is synonymous with the C++ term member function. In SystemC, the term method is used in the context of an interface method call. Throughout this standard, the term member function is used when defining C++ classes (for conformance to the C++ standard), and the term method is used in more informal contexts and when discussing interface method calls. (SystemC term)	メソッド	クラスの動作を実装した関数。この用語はC++用語のメンバー関数と同じ事を表している。SystemCでメソッドとは、インタフェース・メソッド呼び出しの文脈の中で用いられる。一般的にメンバー関数はC++クラス定義（C++標準に準拠した）される時に使用され、メソッドはより略式的な文脈か、またはインタフェース・メソッド呼び出しを論じている時に使用される。（SystemC用語）

© Copyright 2009 JEITA, All rights reserved

multi-socket	One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. A multi-initiator socket can be bound to more than one target socket, and more than one initiator socket can be bound to a single multi-target socket. When calling interface methods through multisockets, the destinations are distinguished using the subscript operator.	マルチソケット	他のコンポーネントに属する複数のソケットにバインドすることができる便利ソケットのひとつ。マルチイニシエータ・ソケットに複数のターゲット・ソケットをバインドすることができ、また、単一のマルチ・ターゲット・ソケットに複数のイニシエータ・ソケットをバインドすることができる。マルチソケットを通してインタフェース・メソッド・コールをする際、ディスティネーションは、添字演算子を使用して識別される。
nb_transport	The nb_transport_fw and nb_transport_bw methods. In this document, the italicised term nb_transport is used to describe both methods in situations where there is no need to distinguish between them.	nb_transport	nb_transport_fw と nb_transport_bw メソッド。本書では、nb_transport というイタリック体で示した用語は、それらを区別する必要がない状況にいて両方のメソッドを指すのに使用されている。
non-blocking	Not permitted to call the wait method. A non-blocking function is guaranteed to return without consuming simulation time or performing a context switch, and therefore may be called from a thread process or from a method process. A non-blocking interface defines only non-blocking functions.	ノンブロッキング	wait の呼び出しが許されない。ノンブロッキング関数はシミュレーション時間の消費、コンテクスト・スイッチ無しでリターンすることが保証されている。そのため、ノンブロッキング関数はスレッド・プロセスもしくはメソッド・プロセスから呼び出される。ノンブロッキング・インタフェースはノンブロッキング関数のみ定義する。
non-blocking transport interface	A non-blocking interface of the TLM-2 standard. There a two such interfaces, containing methods named nb_transport_fw and nb_transport_bw.	ノンブロッキング・トランスポート・インタフェース	TLM-2 標準におけるノンブロッキング・インタフェース。nb_transport_fw と nb_transport_bw と命名されたメソッドを含んだ、2 種類のインタフェースがある。
object	A region of storage. Every object has a type and a lifetime. An object created by a definition has a name, whereas an object created by a new expression is anonymous. (C++ term)	オブジェクト	ストレージの領域。あらゆるオブジェクトは型とライフタイムを持つ。定義によって作成されたオブジェクトは名前を持つが、new 構文によって作成されたオブジェクトは匿名である。(C++ 用語)

© Copyright 2009 JEITA, All rights reserved

parent	The inverse relationship to child. Module A is the parent of module B if module B is a child of module A. (SystemC term)	親	子とは逆の関係。もしモジュール A がモジュール B の親であるならば、B は A の子である。(SystemC 用語)
payload event queue (PEQ)	A class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Transactions are put into the queue annotated with a delay, and each transaction pops out of the back of queue at the time it was put in plus the given delay. Useful when combining the non-blocking interface with the approximately-timed coding style.	ペイロード・イベント・キュー	SystemC イベント通知のキューを保持するクラスであり、各通知が関連トランザクション・オブジェクトを運ぶ。トランザクションは遅延時間をアノテートしてキューに積まれる。そして、各トランザクションはそれらが積まれた時刻＋与えられた遅延時間後にキューの後ろから出てくる。アプロキシメイトリー・タイムド・コーディング・スタイルとノンブロッキング・インタフェースを組み合わせると、便利である。
phase	The period in the lifetime of a transaction occurring between successive timing points.	フェーズ	継続的なタイミング・ポイント間で発生しているトランザクションが有効な期間を指す。
programmers view (PV)	The use case of the software programmer who requires a functionally accurate, loosely timed model of the hardware platform for booting an operating system and running application software.	プログラマーズ・ビュー (PV)	機能要求に忠実なソフトウェア・プログラマ向けの使用法であり、オペレーティング・システムのブートとアプリケーション・ソフトウェアの実行を実施するハードウェアのルーズリー・タイムド・モデルを指す。
protocol types class	A class containing a typedef for the type of the transaction object and the phase type, which is used to parameterize the combined interfaces, and effectively defines a unique type for a protocol.	プロトコル・タイプ・クラス	トランザクション・オブジェクトとフェーズのタイプのための typedef を含むクラスであり、統合したインタフェースをパラメタライズするのに使用され、また、プロトコルのためにユニークなタイプを効率良く定義することができる。
quantum	In temporal decoupling, the amount a process is permitted to run ahead of the current simulation time.	クォンタム	テンポラル・デカップリングにおいて、現在のシミュレーション時刻よりさらに進めることが許されている処理の量。
quantum keeper	A utility class used to store the local time offset from the current simulation time, which it checks against a local quantum.	クォンタム・キーパー	現在のシミュレーション時刻からのローカル時間へのオフセットを保持するためのユーティリティ・クラスであり、ローカル・クォンタムに対してチェックされる。

© Copyright 2009 JEITA, All rights reserved

return path	The control path by which the call stack of a set of interface method calls is unwound along either the forward path or the backward path. The return path for the forward path can carry information from target to initiator, and the return path for the backward path can carry information from initiator to target.	リターン・パス	一組のインタフェース・メソッド・コールのスタックが呼ぶコントロールパスであり、フォワード・パスかバックワード・パスのどちらかに沿って巻き戻される。フォワード・パスへのリターン・パスはターゲットからイニシエータまで情報を運ぶことができ、そして、バックワード・パスへのリターン・パスはイニシエータからターゲットまで情報を運ぶことができる。
simple socket	One of a family of convenience sockets that are simple to use because they allows callback methods to be registered directly with the socket object rather than the socket having to be bound to another object that implements the required interfaces. The simple target socket avoids the need for a target to implement both blocking and non-blocking transport interfaces by providing automatic conversion between the two.	シンプル・ソケット	直接制限されなければならないソケットよりむしろ必要なインタフェースを実装する別のオブジェクトへのソケット・オブジェクトに登録されるべきコールバック・メソッドを許容するので使用するのが簡単な便利ソケットのひとつ。シンプル・ターゲット・ソケットは、自動変換を 2 つの間に提供することによってブロッキングとノンブロッキングの両方のトランスポート・インタフェースをターゲットに実装する必要性を排除する。
slave	This term has no precise technical definition in this standard, but is used to mean a reactive module or port on a memory-mapped bus that is able to respond to commands from bus masters, but is not able itself to initiate bus traffic. Generally, a slave would be modeled as an OSCI target.	スレーブ	この用語は、この標準の中では明確な技術的定義を持たないが、バス・マスタからのコマンドに応答する事が出来る反応的なメモリ・マップド・バス上のモジュールまたはポートを意味する際に用いられる。しかしそれ自身は、バス・トラフィックを起こす事が出来ない。一般的にスレーブは、OSCI ターゲットのようにモデルされる。
socket	See initiator socket and target socket	ソケット	イニシエータ・ソケットとターゲット・ソケットを参照のこと。

© Copyright 2009 JEITA, All rights reserved

standard error response	The behavior prescribed by this standard for a generic payload target that is unable to execute a transaction successfully. A target should either a) execute the transaction successfully or b) set the response status attribute to an error response or c) call the SystemC report handler.	標準エラー応答	汎用ペイロードのターゲットがトランザクションをうまく完了できない時の、この標準において定義された動作。ターゲットは、次のいずれかを行う。a) トランザクションを完了させる。b) レスポンス・ステータス・アトリビュートに、エラーレスポンスをセットする。c) SystemC のリポートハンドラをコールする。
sticky extension	A generic payload extension object that will not be automatically deleted when the reference count of the transaction object reaches 0. Sticky extensions are not deleted by the memory manager.	スティッキー拡張	トランザクション・オブジェクトの参照カウントが 0 に達しても自動的に削除されない汎用ペイロードの拡張オブジェクト。 スティッキー拡張はメモリ・マネージャによって削除されない。
synchronize	To yield such that other processes may run, or when using temporal decoupling, to yield and wait until the end of the current time quantum.	同期	他のプロセスが走るのを待つ。あるいは、テンポラル・デカップリングを使用する際、または現在のタイム・クォンタムの終わりまで待つ。
synchronization -on-demand	An indication from the nb_transport method back to its caller that it was unwilling or unable to fulfill a request to effectively execute a transaction at a future time (temporal decoupling), and therefore that the caller must yield control back to the SystemC scheduler so that simulation time may advance and other processes run.	オンデマンド同期	意図していないか、あるいは将来の時刻(すなわちテンポラル・デカップリング)にトランザクションを効果的に実行するという要求を実現させることができない、という内容の nb_transport メソッドから呼び出し元への表示。従って、関数呼び出し元が SystemC スケジューラにコントロールを譲り返さなければならないので、シミュレーション時刻は進むし、他のプロセスは走る。
tagged socket	One of a family of convenience sockets that add an int id tag to every incoming interface method call in order to identify the socket (or element of a multi-socket) through which the transaction arrived.	タグ付きソケット	トランザクションが到着したソケット(または、マルチソケットの要素)を特定するためにあらゆる入って来るインタフェース・メソッド・コールに int id tag を加える便利ソケットのひとつ。

© Copyright 2009 JEITA, All rights reserved

target	A module that represents the final destination of a transaction, able to respond to transactions generated by an initiator, but not itself initiate new transactions. For a write operation, data is copied from the initiator to one or more targets. For a read operation, data is copied from one target to the initiator. A target may read or modify the state of the transaction object. In the case of the TLM 1.0 interfaces, the term target as defined here may not be strictly applicable, so the terms caller and callee may be used instead for clarity.	ターゲット	トランザクションの最終的な到達先を表すモジュールで、イニシエータによって生成されるトランザクションに応答する事が出来る。しかしそれ自身は新しいトランザクションを開始する事は出来ない。書き込み操作においては、データがイニシエータから 1 個以上のターゲットまでコピーされる。読み出し操作においては、データはあるターゲットからイニシエータまでコピーされる。 ターゲットは、トランザクション・オブジェクトの状態を読む事や、変更する事が可能である。TLM 1.0 インタフェースにおいては、ターゲットと言う用語は厳密には適切でないため、代わりに呼び出し元関数、呼び出し先関数といった用語を使って明確にするのがよい。
target socket	A class containing a port for interface method calls on the backward path and an export for interface method calls on the forward path. A socket also overloads the SystemC binding operators to bind both port and export.	ターゲット・ソケット	バックワード・パス上の、インタフェース・メソッド・コールのためのポートを含むクラス、及び、フォワード・パス上のインタフェース・メソッド・コールのためのエクスポート。 また、このソケットはポートとエクスポートの両方をバインドするために SystemC のバインド演算子をオーバーロードする。
temporal decoupling	The ability to allow one or more masters to run ahead of the current simulation time in order to reduce context switching and thus increase simulation speed.	テンポラル・デカップリング	コンテクスト・スイッチを抑えてシミュレーション速度を向上させる目的で、シミュレーション時間を進めるマスタを一つ以上許容する機能を指す。
timing point	A point in time at which the processes that are interacting through a transaction either transfer control or are synchronized. Certain timing points are implemented as function calls or returns, others as event notifications. Timing points mark the boundaries between the phases of a transaction.	タイミング・ポイント	トランザクションを通じて相互に影響しているプロセスにおける、転送制御や同期される時点のタイミングを指す。一定のタイミング・ポイントは、関数の呼び出しや関数からの復帰のように実装され、外部にイベントを通達する。タイミング・ポイントはトランザクションのフェーズ間のを範囲を示す。

© Copyright 2009 JEITA, All rights reserved

TLM-1	The first major version of the OSCI Transaction Level Modeling standard. TLM-1.0 was released in 2005.	TLM-1	OSCI ＴＬＭ標準の最初のメジャーバージョン。TLM 1.0 は 2005 年にリリースされた。
TLM-2	The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM-2.0.	TLM-2	OSCI ＴＬＭ標準の２つ目のメジャーバージョン。本ドキュメントは TLM 2.0 について記載している。
transaction	An abstraction for an interaction or communication between two or more concurrent processes. A transaction carries a set of attributes and is bounded in time, meaning that the attributes are only valid within a specific time window. The timing associated with the transaction is limited to a specific set of timing points, depending on the type of the transaction. Processes may be permitted to read, write, or make themselves sensitive to attributes of the transaction. A transaction may be an atomic transaction or a complex transaction.	トランザクション	二つ以上の並列プロセスにおける、相互作用や通信を行う為の抽象化。トランザクションは時間範囲とアトリビュートのセットを伝える。アトリビュートは、特定の時間範囲内で有効である事を意味する。トランザクションにおける時間調整は、トランザクションの型に依存した特定のタイミング・ポイントのセットに限定される。
transaction object	"The object that stores the attributes associated with an OSCI transaction. The type of the transaction object is passed as a template argument to the core interfaces."	トランザクション・オブジェクト	トランザクションに含まれるアトリビュートを保持するオブジェクト。
transaction level (TL)	The abstraction level at which communication between concurrent processes is abstracted away from pin wiggling to transactions. This term does not imply any particular level of granularity with respect to the abstraction of time, structure, or behavior.	トランザクション・レベル (TL)	並列プロセス間通信で、ピンレベルの信号動作からトランザクションとして抽出される際の抽象度。この用語には、時間、構造、動作の抽象化についての詳細な粒度は含まない。
transaction level model, transaction level modeling (TLM)	A model at the transaction level and the act of creating such a model, respectively. Transaction level models typically communicate using function calls, as opposed to the style of setting events on individual pins or nets as used by RTL models.	トランザクション・レベル・モデル、トランザクション・レベル・モデリング (TLM)	トランザクション・レベルによるモデルと、そのようなモデルを作成する行動を表す。トランザクション・レベル・モデルは一般的に、RTL モデルによって使用されるような各ピンやネット上にイベントを設定するようなスタイルとは異なり、関数呼び出しを使用して通信を行う。

© Copyright 2009 JEITA, All rights reserved

transactor	A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two or more transaction level interfaces, often at different abstraction levels. In the typical case, the first transaction level interface represents a memory mapped bus or other protocol, the second interface represents the implementation of that protocol at a lower abstraction level. However, a single transactor may have multiple transaction level or pin level interfaces. See adaptor, bridge.	トランザクタ	トランザクション・レベル・インタフェースからピン・レベル・インタフェース（一般的なワード・インタフェースの意味）への接続や、二つ以上のトランザクション・レベル・インタフェースの集合による接続がなされるような、異なる抽象度を扱う際に頻繁に使用されるモジュール。一般的には、初めのトランザクション・レベル・インタフェースはメモリ・マップド・バスやその他プロトコルを表し、次のレベルのインタフェースは、低抽象度のプロトコルの実装を表すが、一つのトランザクタには、複数のトランザクション・レベルや、ピン・レベル・インタフェースを持つ可能性がある。アダプタ・ブリッジを参照。
transport interface	The one and only bidirectional core interface in TLM-1. The transport interface passes a request transaction object from caller to callee, and returns a response transaction object from callee to caller. TLM-2 adds separate blocking and non-blocking transport interfaces.	トランスポート・インタフェース	″TLM-1 標準では一つしかない双方向のコア・インタフェース。トランスポート・インタフェースは呼び出し元関数から呼び出し先関数へリクエスト・トランザクション・オブジェクトを送り、呼び出し先関数から呼び出し元関数へリクエスト・トランザクション・オブジェクトを返す。 TLM-2 標準ではさらにブロッキング・トランスポート・インタフェースとノンブロッキング・トランスポート・インタフェースの２つに分割される。″

© Copyright 2009 JEITA, All rights reserved

unidirectional interface	A TLM 1.0 transaction level interface in which the attributes of the transaction object are strictly readonly in the period between the first timing point and the end of the transaction lifetime. Effectively, the information represented by the transaction object is strictly passed in one direction either from caller to callee or from callee to caller. In the case of void put(const T& t), the first timing point is marked by the function call. In the case of void get(T& t), the first timing point is marked by the return from the function. In the case of T get(), strictly speaking there are two separate transaction objects, and the return from the function marks the degenerate end-of-life of the first object and the first timing point of the second.	単方向インタフェース	トランザクションのアトリビュートがトランザクション・ライフタイムの最初のタイミング・ポイントから最後までの期間で厳密に読み出しのみに限定されたTLM1.0 のトランザクション・レベル・インタフェース。トランザクション・オブジェクトで表される情報は、呼び出し元関数から呼び出し先関数、または呼び出し先関数から呼び出し元関数へ送られる。void put(const T&t)の場合では、最初のタイミング・ポイントは関数呼び出しによって表される。T valid get(T& t)の場合では、この関数からのリターンによって表される。T get()の場合、厳密に言うと二つのトランザクション・オブジェクトが関わっている。この関数からのリターンは 1 つ目のトランザクション・オブジェクトの終わりと 2 つ目のトランザクション・オブジェクトの最初のタイミング・ポイントを表す。
untimed	A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.	アンタイムド	時間やサイクルを明示的に言及しないモデリング・スタイル。しかしオペレーションの並列性や順序は含まれる。時間概念は無いが、複数並列スレッド間のオペレーションの順序性は、イベント、ミューテックス、ブロッキングFIFO等の同期プリミティブを使って、複数並列スレッド間のオペレーションの順序性を実現しなければならない。ユーザによっては、プロトコルの頑健性をテストするため、アンタイムドな記述にランダム遅延を入れる場合があるが、この方法によってモデリング・スタイルの基本的な性質が変わることはない。

© Copyright 2009 JEITA, All rights reserved

valid	The state of [...] an object returned from a function by pointer or by reference, during any period in which the [...] object is not deleted and its value or behavior remains accessible to the application. [...] (SystemC term)	バリッド	ポインタまたは参照で関数からリターンされるオブジェクトの状態で、そのオブジェクトが削除されておらず、その値や動作をアプリケーションからアクセスが可能な状態である期間。(SystemC 用語)
within	The relationship that exists between an instance and a module if the constructor of the instance is called from the constructor of the module, and also provided that the instance is not within a nested module. (SystemC term)	包含	インスタンスのコンストラクタがモジュールのコンストラクタから呼ばれる場合にインスタンスとモジュールの間に存在する関係。あるいは、入れ子にされたモジュールの中にインスタンスがなければ、提供される。(SystemC 用語)
yield	Return control to the SystemC scheduler. For a thread process, to yield is to call wait. For a method process, to yield is to return from the function.	(制御を) 譲渡する	SystemC スケジューラにコントロールを返すこと。スレッド・プロセスにおいて、yield は、call wait である。メソッド・プロセスにおいては、yield は、関数から戻ることである。

© Copyright 2009 JEITA, All rights reserved

4.2.3 TLM 2.0 補足解説

4.2.3.1 TLM-2.0 Draft から正式版への変更点

本節は、OSCI の TLM2 USER MANUAL Software version: TLM-2.0、Document version: JA22 における TLM2 USER MANUAL Software version: TLM 2.0 Draft 2、Document version: 1.0.0 からの変更点を解説したものである。

3. イントロダクション

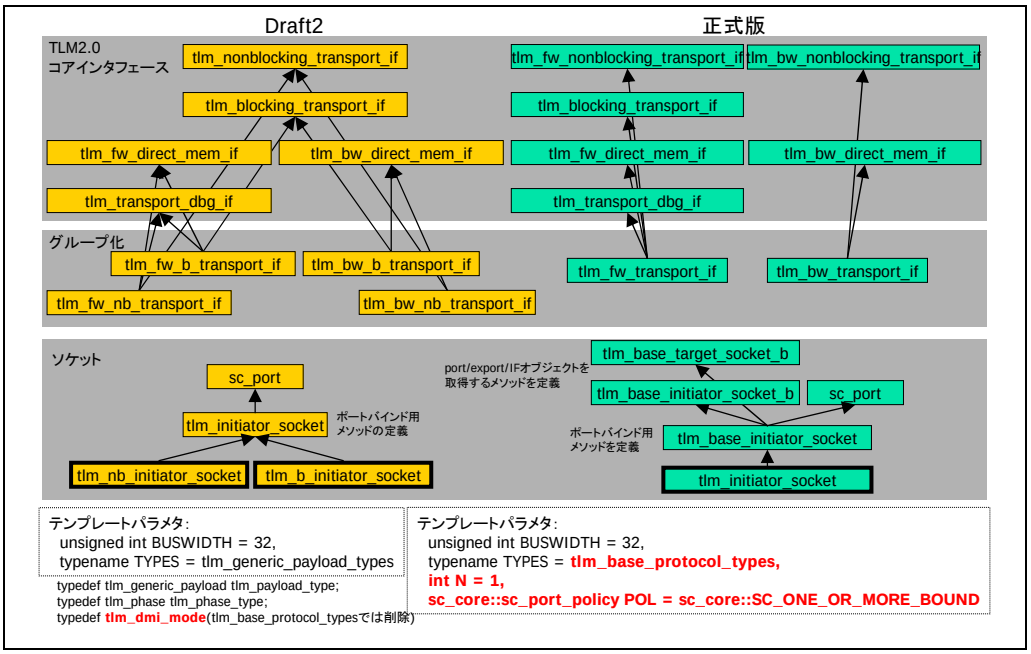
- セクション 3.3.3 が変更。
  Draft2: Untimed versus loosely-timed modeling ⇒ 正式版: Synchronization in loosely-timed models
- セクション追加。
  3.6 Combined interfaces and sockets

4. TLM-2 コア・インタフェース

- ノン･ブロッキング・トランスポート・インタフェースの用途
  Draft2: アプロキシメイトリー・タイムドとルーズリー・タイムドの両方
  正式版: アプロキシメイトリー・タイムド向け。ルーズリー・タイムドのモデリングには推奨されない。
  - ブロッキングでタイミング・アノテーションがサポートされ、ノンブロッキングをルーズリー・タイムドで使用する必要がなくなった為。
- tlm_phase の変更
  Draft2: enum tlm_phase {BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP };
  正式版: enum tlm_phase_enum と class tlm_phase
  - ユーザによってフェーズ拡張が可能なように、tlm_phase がクラスに、元の tlm_phase が tlm_phase_enum として変更されている。
- tlm_sync_enum の変更
  Draft2: enum tlm_sync_enum { TLM_REJECTED = 0, TLM_ACCEPTED = 1, TLM_UPDATED = 2, TLM_COMPLETED = 3 };
  正式版: enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };
  - リジェクトは TLM2.0 の範疇でないと判断された。LRM リリース時に再度理由を確認する。
- nb_transport メソッドの変更
  Draft2: nb_transport メソッド
  正式版: nb_transport_fw、nb_transport_bw
  - フォワードパス、バックワードパスで呼ばれるメソッドが見た目で分かり辛い為に名前が変更された。
- メッセージ・シーケンス・チャートに２つの図が追加された。
  トランザクションの早期終了
  タイミング・アノテーション
- sc_time 引数の説明が詳細になった。
- DMI/Debug ともにトランザクションとして汎用ペイロード(tlm_generic_payload)がデフォルトが採用され，DMI は拡張機構で実現された。
  - トランザクション形式を一般の形に統一。
- transport が b_transport に変更された。
- debug transaction interface から debug transport interface に名称変更された。
- DMI 関連のクラス定義が変更・削除された。
  - get_direct_mem_ptr メソッドの引数が大きく変更され，adress の代わりに tlm_generic_payload が追加され，tlm_dmi_mode クラスが tlm_dmi クラスに統合された
- メソッドの説明文が追加。
  - 特に tlm_dmi クラスについての説明項目が 12 個も追加（16 個→28 個）
- tlm_transport_dbg_if クラス定義が変更された。
- 注釈“helper function 開発中”が削除。

5. 結合インタフェースとソケット

- クラスの継承関係が大幅に変更。
- I/F クラスのグルーピングの仕方が変更。
- blocking 用ソケットと non-blocking 用ソケットを 1 つにマージ。
- ソケットの複数接続が可能になった(従来は 1 対 1)。



イニシエータのクラスの継承関係

6. 汎用ペイロード

- 6.10 Data pointer attribute: 全体として、言語マニュアル的表現からユーザーマニュアル表現になり、使い方の説明が加わり分かりやすくなった
- 6.11 Data length attribute: データ長はトランザクション･オブジェクトがインタフェースメソッドコールで渡される前に明示的に指定されるべきなので、デフォルト値が 1 から 0 に変更された。
- 6.12 Byte enable pointer attribute: Byte Enable pointer の値を、イネーブル/ディスエーブルのマスクとして使えるように値を決めた。
- 6.13 Byte enable length attribute: デフォルト値が、 1 から 0 (null)に変更された
- 6.14 Streaming width attribute:
  - ・ ストリーミングの扱いについての説明が追加された。
  - ・ 0 は通常のデータ転送として扱われるとなっていたのが、無効に変更された。
- 6.16 Response status attribute: response status attribute の扱い方の具体的な解説が追加された
- 6.20 汎用ペイロードの拡張:
  - ・ 9.19 Extension mechanism → 6.20 Generic payload extension
  - ・ 6.21 Instance-specific extensions の章が追加された。
  - ・ 安全使用のガイドラインと規則が規則に 1 本化された。
  - ・ tlm_extension_base クラスに copy_from 純粋仮想関数と free 仮想関数が追加された。デストラクタが public から protected に変更された。

8. その他のクラス

- 8.1 グローバル・クォンタムとクォンタム・キーパー: 機能自体に変更はないが、クラス・API 体系に若干の変更あり
  - クラス・ルールの変更
    - ・ Draft2 では tlm_quantumkeeper クラスだけであったが、グローバル・クォンタムに関連する機能だけ、tlm_global_quantum クラスとして切り出された。

・Draft2 では 29 個のルール、正式版では 34 個のルールがある。大部分のルールは共通で変更点は瑣末。
- 仕様の変更
・nb_transport 関数(AT コーディングスタイル)でのテンポラル・デカップリングやクォンタム・キーパーの使用はルールの対象外と明記された。

- 8.2 ペイロード・イベント・キュー: 機能自体に大きな変更はないが、クラス・API 体系に若干の修正あり
- クラス・API の変更
・Draft2 では tlm_peq クラスだけであったが、正式版では peq_with_get クラス、peq_with_cb_and_phase クラスの２つがある。
- 仕様の変更
・PEQ へのトランザクションの格納・取り出し API が変更になった。
tlm_peq::delayed_write_export → peq_with_get::notify
tlm_peq::write_port → peq_with_get::get_next_transaction
・peq_with_cb_and_phase クラスを使用する場合は、トランザクションが取り出し可能になった際、自動的に呼ばれるコールバック・メソッドの登録が可能。
またフェーズ情報をセットにした格納が可能になった。

4.2.3.2 TLM2.0 拡張方法

TLM2.0 では、MMB(メモリ・マップド・バス)をモデリングするための基本的なプロトコルが定義されています。しかしながら、TLM2.0 では、どのバスにでもあるような必要最小限のデータ構造(Generic Payload)と、必要最小限の通信手段(Base Protocol)が定義されているにすぎず、実際の何らかのバスを厳密に定義しようとすると、いくつかの拡張をする必要があります。
TLM2.0 の中心部分の定義は、どんなバスにでも適用可能な、全ての機能を盛り込んだ汎用的なものというよりは、どのようなバスにでもある共通的なもののみのサポートになっているわけです。このように TLM2.0 を汎用的なものではなく、共通的なものに絞りこんだ理由としては、SystemC TLM メソドロジをサポートした IP の普及のために必要な相互利用性の確保を最大の目標にしていることがあげられます。
特に、ソフトウエア/ハードウエア協調設計環境の構築においては、バスの詳細定義よりも、接続の容易さが必要になりますので、TLM2.0 は、タイミング精度、個別のバスによって異なる仕様を詳細に定義するというよりは、高速性と一般化による汎用性を重視していると言えます。
とはいっても、世の中にあるバスモデルの詳細仕様が TLM2.0 ベースで全くモデリングできないことになってしまうと、TLM2.0 の適用範囲が狭まってしまいます。そこで考え出されたのが、いくつかの拡張手段です。
TLM2.0 には、大きく分けて、転送する情報を追加する方法と、詳細なタイミング・ポイントとその通信プロトコルを定義するための、Phase を追加する二つの拡張手段があります。これらの二つを利用して、詳細なプロトコルを定義することが可能になっています。

1. Generic Payload 拡張
Generic Payload に、ユーザが定義したクラスのポインターを追加して転送することによる拡張手段です。標準ではアドレス、データ、データ長、バースト長等の基本情報がサポートされていますが、これ以外の情報、例えば、ロック、バスマスターID 等のプロトコル依存の情報、トランザクション ID 等、解析機能のための情報等をユーザ定義のクラスのメンバ変数に設定し、標準の Generic Payload と一緒に転送します。

2. フェーズ拡張
TLM2.0 では、通信の手段として、ブロッキング転送である、b_transport()と、ノンブロッキング転送である、nb_transport()がサポートされています。このうち、nb_transport()には、BEGIN_REQ、END_REQ、BEGIN_RESP、END_RESP の 4 つの Phase が定義されており、この Phase をイニシエータとターゲットにより交換することにより、基本的なタイミング・ポイントを定義できるようになっています。しかし現実のバス・プロトコルでは、4 つでは足りず、6 つ必要な場合もあります。こういった詳細なバス・プロトコルを定義するために、ユーザが Phase を追加できるようになっています。

3. 無視可能な拡張とそうでない拡張

それでは、あるIPベンダーがGeneric Payloadの拡張やPhase拡張をしてモデルを開発した場合、そのモデルは他のベンダーのTLM2.0準拠のモデルと一緒に動かすことができるのでしょうか？つまり、拡張をおこなうことによって互換性がどこまで保たれるのでしょうか？

残念というべきか当然ですが、あるベンダーが独自に拡張してそのプロトコルを自分で定義した場合、そのモデルは、TLM2.0をベースにしているとはいえ、そのプロトコルを知らないモデルとの間で正しく動かすことはできません。

しかしながら、そこには一定のルールがあります。Generic Payloadの拡張にしろ、フェーズ拡張にしろ、「無視できる拡張」と「無視できない拡張」があります。「無視できる拡張」とは、相手がそのプロトコルを知らなかったとしても、基本的なTLM2.0の動作に影響を与えない拡張のことです。

「無視できない拡張」とは、プロトコル依存の拡張をした場合で、相手がその内容を知らない場合には、基本的なTLM2.0の動作にまで影響を与えてしまう拡張のことです。

例えば、バスのパフォーマンス解析のためにそれぞれのトランザクションIDとか、イニシエータのIDとかをGeneric Payloadの拡張クラスに定義したとします。こういったツール依存の拡張は、ツールによっては必要な場合もありますが、TLM2.0としての基本的な機能そのものには影響を与えません。つまり必須ではありませんので、無視できるわけです。このようなものは、独自にイニシエータに追加しても、ターゲットに問題なく渡すことができます。

例えば、バスモデルのみ特定ベンダーから受け取ったIPを使用し、イニシエータとターゲット・モデルを自分で作成する場合を考えてみます。この場合、イニシエータ側で情報を追加し、ターゲットでその情報を取得することができます。バスモデルは、それらの追加情報のヘッダファイル等がなくても、ソースが公開されていないモデルであったとしても、ユーザが追加した情報をそのまま素通りさせることができます。つまり、バスモデルベンダーに特別なエンハンスメント要求等をしなくても、ユーザ独自の情報をイニシエータ側とターゲット側でやりとりできるような仕組みになっているわけです。

それでは、ターゲット・モデルのみを開発する場合においで、イニシエータが何らかの拡張情報を追加している場合はどうなるでしょうか？

この場合、イニシエータが勝手に追加した情報のヘッダファイルがなくても無視すればいいだけです。もし、そのヘッダファイルを入手することができれば、初めて、その追加情報にアクセスすることができるようになります。

逆に、バスモデルを開発するときには、そういった拡張情報を消してはいけません。知らない拡張情報に関しては、変更を加えず素通りさせることが明確にルールで決められています。また、イニシエータではなく、バスモデルがイニシエータから渡されたトランザクションに拡張情報を追加してターゲットに渡すこともできますが、この場合にはターゲットから戻ってきた時点において自分で追加した情報を削除する必要があります。

このような無視できる情報に対して、モデルの動作そのものに影響を与えるようなものを追加してしまうと、もはや、その拡張情報を知るもののみがそのIPに対応したモデルを動かすことができるようになってしまいます。

問題になるのは、あるTLM2.0に対応したモデルが無視できる拡張しか持たないのかそうでないのかをどう判断させるかです。仕様書を参照してほしいという曖昧な方法はここでは通用しません。TLM2.0では、このために、無視できない拡張をした場合には、新しい「Traitsクラス」を別途定義して、それを使用したソケットを使用することが定められています。

つまり、無視できない特定プロトコルをサポートしたソケットを接続する場合には、特定のクラスを用いてソケットを定義し、それら同士のみが接続できるように強制させているのです。もし違うもの同士を接続したとしても、コンパイルエラーとなります。このことが重要なのです。エラボレーション時ではなく、コンパイル時点で、異なる意味をもつプロトコル同士の接続をさせないようにしているということです。

この新しい「Traitsクラス」を使用する義務があるというのは、何も自分で拡張構造を定義したり拡張フェーズを定義したときだけではありません。意味論として、少しでもTLM2.0のルールからはずれたモデルを作成してしまうと、その時点で互換性が無くなってしまうので、Traitsクラスを使用する必要があるのです。

ある意味では、TLM2.0のベース・プロトコルとほとんど同等で、少しだけ違うといったものが簡単に作れるということになります。そのIPを使うユーザ同士がそのことを認識して使えばいいと

いうことになります。

4. 新しいソケットを定義する具体的方法

例えば、Generic Payload も、Phase も拡張しないが、ルールだけを変えたい場合に具体的にどうすればいいかをみてみましょう。
以下のようなモデルがあるとします。

```
SC_MODULE (target1_pv_base),tlm::tlm_fw_transport_if<>
{
  typedef tlm::tlm_target_socket<32> target_socket_type;
  target_socket_type socket1;
  ...
```

ここでは、template 内でデフォルト値を使用しています。もう少し省略しないで書くと次のようになります。

```
SC_MODULE(target1_pv_base),tlm::tlm_fw_transport_if<tlm::tlm_base_protocol_types>
{
  typedef tlm::tlm_target_socket<32,tlm::tlm_base_protocol_types>
target_socket_type;
  target_socket_type socket1;
  ...
```

この tlm::tlm_base_protocol_types は、次のように TLM2.0 で定義されています。

```
struct tlm_base_protocol_types
{
 typedef tlm_generic_payload tlm_payload_type;
 typedef tlm_phase tlm_phase_type;
};
```

これをそのままコピーして、次のように自分のプロトコル（「Traits クラス」）を定義します。中身の generic payload や、phase はそのままでも別のものでも構いません。

```
struct my_protocol_types
{
 typedef tlm_generic_payload tlm_payload_type;
 typedef tlm_phase tlm_phase_type;
};
```

あとは、上記の tlm_base_protocol_types の部分を、my_protocol_types に置き換えるだけです。

```
SC_MODULE(target1_pv_base),tlm::tlm_fw_transport_if<my_protocol_types>
{
  typedef tlm::tlm_target_socket<32,my_protocol_types> target_socket_type;
  target_socket_type socket1;
```

これだけで、後はいっさい変更する必要はありません。これで、my_protocol_types を持ったソケットを、TLM2.0 のベース・プロトコル(tlm::tlm_base_protocol_types)につなげようとすると、コンパイル時にエラーがでるようになり、間違った接続であることがわかります。
なお、モデル内で typedef によりソケットタイプ等を使用するようにしておいた方が後での変更に柔軟に対応できるようになります。

5. Generic Payload の拡張方法

tlm_generic_payload にないメンバが必要な場合には、tlm_generic_payload を継承するのではなく、別途必要なメンバ変数を保持するクラスを定義して、generic_payload に対して set_extension()関数を用いてそのクラスを登録します。
例:

```
class my_payload : public tlm::tlm_extension<my_payload> {
  public:
    virtual tlm::tlm_extension_base* clone() const {
       my_payload* t = new my_payload;
          t->m_lock = this->m_lock;
       return t;
```

```
    }
    my_payload(bool lock) : m_lock(bool lock) {}
  public:
    bool m_lock;
};
```

上記は、m_lock という変数を持ったクラスでこれを generic_payload にセットします。
追加方法(イニシエータ側)

```
tlm_generic_payload* trans = new tlm_generic_payload;
my_payload* my_p = new my_payload(true);
trans->set_extension(my_p);
```

取得方法(ターゲット側)

```
b_transport(tlm_generic_payload  &trans,sc_time& time) {
my_payload* my_p; = NULL;
trans->get_extention(my_p);
if(!my_p) {
   // No such extension, Initiator is not my IP.
} else {
   bool lock = my_p->m_lock; // Oh, this is my IP.
    ...
```

拡張したクラスのヘッダファイルさえあれば、そのクラスの拡張があるかどうかを調べて、あった場合にはそれを利用するという使い方ができます。

6. Phase の拡張方法

Phase を拡張するには次のようにします。DECLARE_EXTEND_PHASE とは TLM2.0 で定められたマクロです。

```
DECLARE_EXTENDED_PHASE (BEGIN_DATA);
DECLARE_EXTENDED_PHASE (END_DATA);
```

イニシエータ側

```
phase = BEGIN_DATA;
sync = i_socket->nb_transport_fw(trans,phase,time);
```

ターゲット側

```
tlm::tlm_sync_enum nb_transport_fw(transaction_type &trans, phase_type &phase,
sc_time &time)
{
  cout << "Target: received phase[" << phase << "]" << endl;
```

Phase 拡張は、主にプロトコルを拡張するために使われることから、通常ユーザ定義の「Traits クラス」を用います。もし、自分の知らない phase が来た場合には無視します。つまり何も来なかったことにします。エラーにしてはいけません。
ベース・プロトコル(つまり Traits クラスを定義しない場合)のまま使用する場合、無視可能なフェーズ拡張を受信した場合には、TLM_ACCEPTED のみを返す仕様になっています。そのため、関数の戻り値を判別して、ターゲットが無視したのか、理解されたのかを、イニシエータが簡単に知る手段はありません。

7. ベース・プロトコルとは何か？

TLM2.0 には、tlm_fw_transport_if、tlm_bw_transport_if という二つのインタフェース・クラスとそのクラスに定義されている b_transport, nb_transport があり、これが中心になっています。これらを使わなかったら、TLM2.0 とは呼べません。
このインタフェースに渡すべき、tlm_generic_payload、tlm_phase は、TLM2.0 の中の、ベース・プロトコルとして定義されている部分であり、これらを拡張することも、全く異なるものを使用することもできるようになっています。
互換性を最大限追求するのであれば、このベース・プロトコルの仕様に基づいたものにするべきであるといえます。詳細なバス・プロトコルを使用する場合には、バス・プロトコル専用のソケットクラス(「Traits クラス」) を使用してベース・プロトコルとは別ものであるということをは

っきりさせる必要があるということになります。

4.2.3.3 TLMのタイミング精度

TLM2.0では、タイミング精度と高速性の2つの相反する要求を満たすためにいくつかのコーディングスタイルを提唱しています。また、必要なタイミング精度を確保するためには、Phase拡張などを実装する場合もあります。ここでは、どのようなタイミング精度が実際のデザインにおいて必要になり、それぞれのタイミング精度に対して、TLM2.0がどのような手法を用意しているかについてまとめます。

1. タイミング・ポイント

トランザクション・レベル・モデリングを行う際には、そのトランザクションが開始されて、何らかの処理を行い、終了するまでの時間軸において、重要な時刻に着目したモデリングを行うことにより、高速化なシミュレーション環境を構築します。
ゲートレベルのシミュレーション環境においては、それぞれのゲートの遅延時間までモデリングしてシミュレーションするのに対し、RTL記述においては、ゲートの遅延時間を排除して、クロックにのみ着目してモデリングします。これと同様、TLMにおいては、クロックよりも粒度の荒いタイミングに着目して、クロックを排除することにより高速化させます。
TLM2.0では、このタイミング・ポイントの取り方によって、2タイミング・ポイント、3タイミング・ポイント、4タイミング・ポイント、それ以上のタイミング・ポイントを表現することができます。
TLM2.0の基本インタフェースには、b_transport() (ブロッキング)と、nb_transport() (ノンブロッキング)の2つの関数が用意されており、b_transport()においては、2つのタイミング・ポイント、nb_transport()においては、2,3,4,それ以上のタイミング・ポイントを定義することができます。
通常、b_transport()の2つのタイミング・ポイントを用いたコーディングスタイルを、LTコーディングスタイル(Loosely (粗い)Timed)コーディングスタイルと呼び、それ以上のタイミング・ポイントを用いたコーディングスタイルを、AT(Approximately Timed)コーディングスタイルと呼んでいます。

2. 2つのタイミング・ポイント

もっとも簡単なタイミング精度は、開始と終了の2つのタイミング・ポイントのみを持つものです。
CPUなど、データの送受信を要求する側(イニシエータ)がトランザクションの要求をした時点が開始時刻となり、要求された側(ターゲット)が処理をおこない、その終了後送信側に制御を戻した時点が、トランザクションの終了時刻になります。この場合、開始時刻と終了時刻との差がレイテンシとなり、通常はターゲット側において、そのレイテンシを決めることになります。
イニシエータとターゲットが直結している場合には、ターゲットの処理時間のみがトータルの処理時間を左右することになります。しかし、間にバス等のインターコネクトモデルが含まれている場合には、バス内の時間をそれぞれ追加することになるので、最大3つの時間を定義可能です。

レイテンシ = (バスの前処理時間) + (ターゲットのレイテンシ) + (バスの後処理時間)

2 つのタイミング・ポイント：直結の場合

```
イニシエータ          ターゲット
  要求開始        ----> 処理開始
                        (処理)
  終了          <---- 処理終了
```

2 つのタイミング・ポイント：バスを含む場合

```
イニシエータ          バス              ターゲット
  要求開始        ----> 事前処理開始
                        (処理)
                        要求開始 --------->処理開始
                           (処理)
                        事後処理開始<-----処理終了
                       (処理)
  終了          <---- 事後処理終了
```

3. 3 つのタイミング・ポイントが必要になる例

しかしながら、2 つのタイミング・ポイントではモデリングできず、どうしても 3 つのタイミング・ポイントが必要な場合があります。

例えば、ターゲットがイニシエータにデータを戻す際に、イニシエータ側の受信スピードが遅く、その時間をイニシエータ側で制御したいとします。この場合、2 つのタイミング・ポイントでは、ターゲット側の処理が終わってから、イニシエータ側に戻った後、イニシエータ側で wait()を行っても、ターゲット側はすでに処理を終わってしまっているため、ターゲット側での処理時間が全体として実際よりも短くなってしまいます。

2 つのタイミング・ポイント

```
イニシエータ          ターゲット
  要求開始        ----> 処理開始
                        (処理)
  受信処理開始    <---- 処理終了
    (処理)            本来はここもビジーのはず。

  受信処理終了
```

3 つのタイミング・ポイント

```
イニシエータ          ターゲット
  要求開始        ----> 処理開始
                        (処理)
  受信処理開始    <---- 受信処理開始
    (処理)

  受信処理終了    ----> 受信処理終了
```

このため、イニシエータとターゲット間では、開始時刻と終了時刻の間にもう 1 つのタイミング・ポイントをもうける必要があります。

TLM2.0 のベース・プロトコルでは、こういった途中の処理状態を表すために、4 つの Phase を定義しています。

```
イニシエータ ターゲット
BEGIN_REQ                   リクエスト開始
             END_REQ        リクエスト終了
             BEGIN_RESP     レスポンス開始
END_RESP                    レスポンス終了
```

上記の 3 つのタイミング・ポイントでは、このうち、END_REQ は使わずに、BEGIN_REQ、BEGIN_RESP、END_RESP の 3 つの Phase を用いることにより、2 つの時間を設定できます。

```
イニシエータ        ターゲット
 BEGIN_REQ        ---> 処理開始
                      (処理)
```

```
受信処理開始     <--- BEGIN_RESP
（処理）
END_RESP        ---> 処理終了
```

つまり、イニシエータがターゲットに BEGIN_REQ を送ることによりターゲットが処理を開始し、ターゲットのレスポンス送信が可能になった時点で BEGIN_RESP をイニシエータ側に通知して、イニシエータ側がすべての処理を終了した時点で、ターゲット側に END_RESP を通知することにより、両者が同時に終了させることができるわけです。

4. 4つのタイミング・ポイントが必要になる例

それでは、BEGIN_REQ の後の END_REQ は、どういうケースで必要になるのでしょうか？
イニシエータ側は、BEGIN_REQ により要求をターゲットに開始しています。ターゲットの処理が終わるのを待つだけであれば、END_REQ という要求受付という情報を受け取らなくても良さそうです。
これがどうしても必要になるケースは、1 つのトランザクション処理が終了する前に、別のトランザクションを開始したい場合です。つまり、ターゲットに対して始めの要求をしたら、その処理が終了する前に 2 番目の要求を開始したい場合です。2 番目の要求を開始できるタイミングは少なくとも 1 番目の要求が完了した時点になりますので、どうしても、END_REQ という Phase をターゲットから受け取る必要が出てくるのです。

```
BEGIN_REQ1   --->

             <---  END_REQ1
BEGIN_REQ2   ---->

             <---  END_REQ2
......
             <---  BEGIN_RESP1
END_RESP1    --->
             <---  BEGIN_RESP2
END_RESP2    --->
```

このように処理待ちのトランザクションのことを、アウトスタンディング・トランザクションと呼びます。 （処理の形態としてはパイプライン処理）

5. 何故アウトスタンディング・トランザクションが必要になるか？

ここで、少し本題から離れて、どういう場合にアウトスタンディング・トランザクションが必要になるかを考えてみます。
アウトスタンディング・トランザクションをサポートしていないバスの場合、1 つのトランザクションが開始してから終了するまでの間、バスそのものはずっと占有し続けます。しかし、例えば SDRAM のようにページのミスヒットによっては、レイテンシが非常に長くなる場合もあり、メモリの実際の読み書きをしている時間中ずっとバスを占有し続けるのは非常に無駄になります。
そこで、ターゲット側のコントローラーに、FIFO を用意して、とりあえずアドレス等の要求のみを受け付けて、バスを解放してしまいます。つまり、BEGIN_REQ から、END_REQ までの時間は、要求を受け付けるだけの時間になります。通常、READ 命令の場合には、アドレスを送るだけの処理となり、WRITE 命令の場合には、アドレスと実際のデータを送る処理になります。その後、END_REQ から BEGIN_RESP の間に実際のメモリの読み書き処理が行われ、それが終了した BEGIN_RESP と END_RESP の間に、READ 命令の場合には、実際のデータを転送する処理、WRITE 命令の場合には終了通知をイニシエータにおこなうことになります。
WRITE 命令の場合

```
BEGIN_REQ --->   アドレス送信
                 データ送信

          <---   END_REQ 受付処理終了、バス解放
                 (メモリ書き込み処理)
          <---   BEGIN_RESP (終了通知)
END_RESP  --->
```

READ 命令の場合

```
BEGIN_REQ --->   アドレス送信
          <---   END_REQ 受付処理終了、バス解放
                 (メモリ読み込み処理)
          <---   BEGIN_RESP (データ送信)
データ受信

END_RESP  --->   データ受信終了通知
```

なお実際には、ターゲットが BEGIN_REQ を受け取った時点で FIFO が空であれば、すぐにメモリの読み書きを始めることができますが、それはあくまでも FIFO の出口側の話で、FIFO の入り口でアドレスなりデータの受信を終了した時点で、END_REQ をイニシエータに通知してバスを解放できます。FIFO がフルであったとすると、FIFO が空くまで、END_REQ を返すことはできなくなります。レスポンス側でも同様のことが起こり、イニシエータ側のデータの受け取り作業が遅れると、ターゲット側の出力側の FIFO にデータがたまり続けます。こういったシステムではデッドロックを起こす危険性もあります。
いずれにせよ、アウトスタンディング・トランザクションをサポートする高機能なバスを使用し、バスの占有率を取得したいケースでは、少なくとも 4 つのタイミング・ポイントをサポートしたモデリングスタイルが必要になります。

6. 4 つ以上のタイミング・ポイントが必要になるケース

通常は、上記の 4 つのタイミング・ポイントがあればほとんどの場合モデリングできるわけですが、バス・プロトコルによっては、Phase を追加して、6 つのタイミング・ポイントが必要になる場合があります。
例えば、AXI プロトコルが良い例です。AXI の場合、アドレスチャネルとデータ送信チャネル、データ受信チャネルがすべて独立して、それぞれイニシエータとターゲット間にてハンドシェイクしています。
例えば、WRITE 命令の場合、通常はアドレスを送信した後、ただちにデータを送信し始めるため、その間に別の処理をするというのはありません。しかし AXI プロトコルの場合、アドレスだけを複数送っておいて、あとからデータを別途送るということができるわけです。そうすると、アドレス開始、アドレス終了、データ送信開始、データ送信終了、レスポンス開始、レスポンス終了という 6 つのタイミング・ポイントが必要になってきます。
TLM2.0 では、ベース・プロトコルとして互換性を最大限確保する Phase としては、4 つが定義されていますが、こういったバス独自の機能をサポートするために、Phase を追加することができるようになっています。この Phase 拡張の機能を使用すれば、AXI プロトコルの複雑なバスにも完全に対応することができるわけです。
ただし、Phase 拡張を用いた場合には、通常イニシエータ側とターゲット側にて TLM2.0 のベース・プロトコルを逸脱したプロトコルを双方で守って送受信する必要がでてきますので、当然のことながら、互換性は損なわれることになります。

```
BEGIN_REQ    -->  アドレス送信
             <--  END_REQ 受付終了

BEGIN_AXI_DT -->  データ送信
             <--  END_AXI_DT

             <--  BEGIN_RESP
END_RESP     -->
```

注：BEGIN_AXI_DT、END_AXI_DT は、このドキュメント内で便宜的に定義しているもの
注意したいのは、AXI を使用することと、6 つの Phase が必要になるということが同義語ではないことです。
AXI はバスの定義ではなく、Point-to-Point でのプロトコルを定義しているにすぎず、6 つの Phase が本当に必要になるかどうかは、AXI のどの機能を用いてどのようにバスを構成しているかに依存するのです。たとえ、アドレスチャネルとデータチャネルが独立してハンドシェイクしていたとしても、常にアドレスを送った直後にデータを送るようにしているのであれば、BEGIN_REQ とその後の Phase を分ける必要はなく、TLM2.0 のベース・プロトコルで十分だということになります。これは、OCP-IP 等のプロトコルでも同じことがいえます。

7. イニシエータ側のモデリング方法

イニシエータ側におけるモデリング方法は、アウトスタンディング・トランザクション/パイプラインを使用しているかどうかによって大きく異なってきます。
例えば、2 つのタイミング・ポイントを実現する b_transport()を用いてイニシエータ側のモデリングをする場合は非常に簡単です。
何故かというと、b_transport()の場合、その関数が戻ってきたときには、必ず全ての処理が終了しているため、連続的に b_transport()をコールし、簡単にそこで取得した値を利用することができるからです。
例

```
socket.b_transport(Read,アドレス１、&データ１);  (模式的に表現しています)
cout << データ１ << endl;
sccket.b_transport(Read,アドレス２、&データ２);
cout << データ２ << endl;
```

これは、b_transport()を使用している時だけでなく、たとえ、nb_transport()を用いて 4 つの Phase を用いた場合でも、複数のトランザクションを同時にスタートしなくても良い場合には、同じようなコーディングをおこなうことができます。

```
socket.my_transport(Read,アドレス１、&データ１);  (模式的に表現しています)
cout << データ１ << endl;
sccket.my_transport(Read,アドレス２、&データ２);
cout << データ２ << endl;
```

** my_transport() の中で、nb_transport()、レスポンス処理を待つ等の処理をインプリメントしておけばいい。
しかし、アウトスタンディング・トランザクションやパイプライン処理をしている場合にはこうはいきません。1 つめの送信処理が終了した時点で 2 つ目の送信処理をしたのでは遅すぎるからです。1 つめの送信処理をしている途中で、2 つ目の送信処理を開始しなければいけなくなります。こういった場合には、複数の Thread を SystemC で用意して、それぞれ送信処理と受信処理を別々の Thread にて実行するようにモデリングする必要があります。これは考えてみれば当たり前のことで、RTL においても、複数トランザクションのサポートが必要なければ、1 つのステートマシンで実現できるのに対し、こういった場合には、複数のステートマシンで実現することが必要になり、TLM でも同じことがいえるわけです。
ただし、アウトスタンディング・トランザクションとして使用できる数にはたいてい制限があり、むやみに送信要求をするということはあり得ません。したがって、個別の送信では上記のようなモデリングはできなくても、ある程度まとまった送受信を行うという処理にしてしまえば、全体としては 1 つの送受信処理と見なすようなモデリングは可能です。

8. どのタイミング・ポイントを使うべきか

それでは、実際問題として、どのようなタイミング・ポイントを用いてモデリングするべきでしょうか？　これはトランザクション・レベル・モデリングを利用する目的によって変わってきます。例えば、ハードウエアーソフトウエア協調シミュレーション等を行うだけであれば、たとえバスとして AXI を使用していたとしても、2 つのタイミング・ポイントで十分のはずです。これに対し、バスの占有率を調査したいようなケースにおいては、やはり詳細なタイミング・ポイントまで表現しておかないと、現実の数値とかけ離れてしまいます。
できれば、複数の利用目的にかなうように、タイミング精度を切り替えられるようなコーディングスタイルにしておくのがいいのではないかと考えられます。あるいは、機能とタイミングを分離したコーディングスタイルを利用するという手もあります。
いずれにしても、TLM2.0 では様々な目的に利用できるようなコーディングスタイルを提唱していますので、用途と目的に応じたモデリングをおこなうことにより、新しい設計環境に対応できるようになっています。

4.2.4 OSCI TLM2.0 へのフィードバック

Recommendation for OSCI/TLM2.0 document by JEITA SystemC Working Group (v1.0)

JEITA SystemC working group has been working to review TLM2.0 official document and we are briefly translating it to Japanese so that TLM2.0 user in Japan can understand it easily.

We heard that OSCI TLMWG are working to write LRM targeting to IEEE standard, So we recommend some documentation issues so that LRM will be easier to be understood.

1) Base protocol concept and each extension have to be clarify at beginning of the standard

Explanation of base protocol and extension for phase and generic_payload and new traits class related issues are described in several different sections.
We think base protocol concept should be discussed at very beginning of the standard and need to be clarified together with real world bus protocol and interoperability issue. Because interoperability of TLM2.0 and usage model of TLM2.0 depend on base protocol and extensions.
Also, more explanation about what is covered by base protocol, what is covered by ignorable extension, or by non ignorable extension should be discussed clearly, maybe using some graphics.

2) Require phase transition table to explain nb_transport phase

Base protocol define 4 phases for nb_transport, but it is very difficult to start writing code based on this document.
For example, when a user try to implement nb_transport_bw() function, he or she needs to cover every cases for each transition. By this document, it is very difficult to understand which phase he or she needs to implement, and what should be done if he or she receives erroneous phase.
We strongly recommend to add some kind of table to address these issues.

3) 5.3.2.4 get_direct_mem_ptr should be invalidate_direct_mem_ptr.

5.3.2.4
j) An initiator is not obliged to register an invalidate_direct_mem_ptr callback with a simple initiator or socket, in which case an incoming get_direct_mem_ptr call shall be ignored.

This get_direct_mem_ptr should be invalidate_direct_mem_ptr
We think this is just a document typo.

4) 5.3.2.4 Simple target socket b/nb conversion figure is not clear

5.3.2.4 has 2 figures to explain nb/b adaptor and b/nb adaptor (figure 12/figure 13).
But it is difficult to understand how it works.
We recommend to add which process calling "wait()" in the figure.

5) Simple target socket b/nb conversion examples need more scenarios

5.3.2.4 shows how b/nb conversion works, but it uses only one scenario. Other scenarios such as using END_RESP phase or directory jumping to BEGIN_RESP are required.

6) 8.2 Payload event queue explanation is not enough, need more examples

8.2 describes 2 classes, peq_with_get, peq_with_cb_and_phase, but it does not have any example how to use these classes. We recommend to add examples and better explanation.

7) TLM1.0 positioning

TLM1.0 positioning is not clear in the document. Please clarify which part of TLM1.0 is standard in the LRM.

4.2.5 SystemC推奨設計メソドロジ 2008年度版

© Copyright 2004-2009 JEITA, All rights reserved

はじめに

- 目的
  - SystemCの特徴を生かした設計メソドロジ（スタイル）の提案を行い、設計メソドロジの共通認識を広める
  - 設計メソドロジの議論の土台を作る
  - 設計メソドロジ案を公開し、それに対応してもらうことで各種ツールの親和性を高めたい
- 対象
  - 主にデジタル信号処理系を対象とする
  - アルゴリズムをHWとSWに分割し、実装及び検証するまでの手法を検討する

© Copyright 2004-2009 JEITA, All rights reserved

目次

- 1. 背景
- 2. システム設計
- 3. SW設計
- 4. HW設計
- 5. 検証
- 6. モデリング(高速化)
- 7. その他

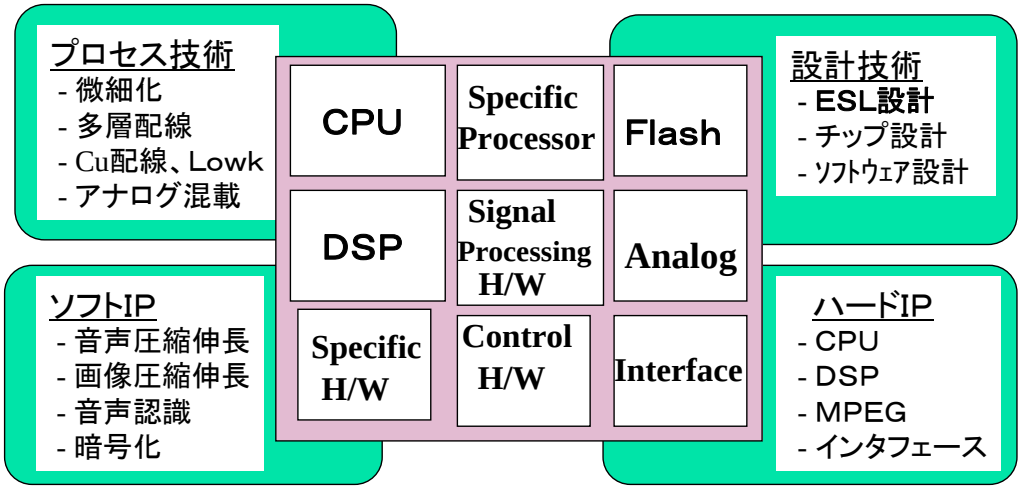
© Copyright 2004-2009 JEITA, All rights reserved

1. 背景

© Copyright 2004-2009 JEITA, All rights reserved

1-1.システムLSIの状況

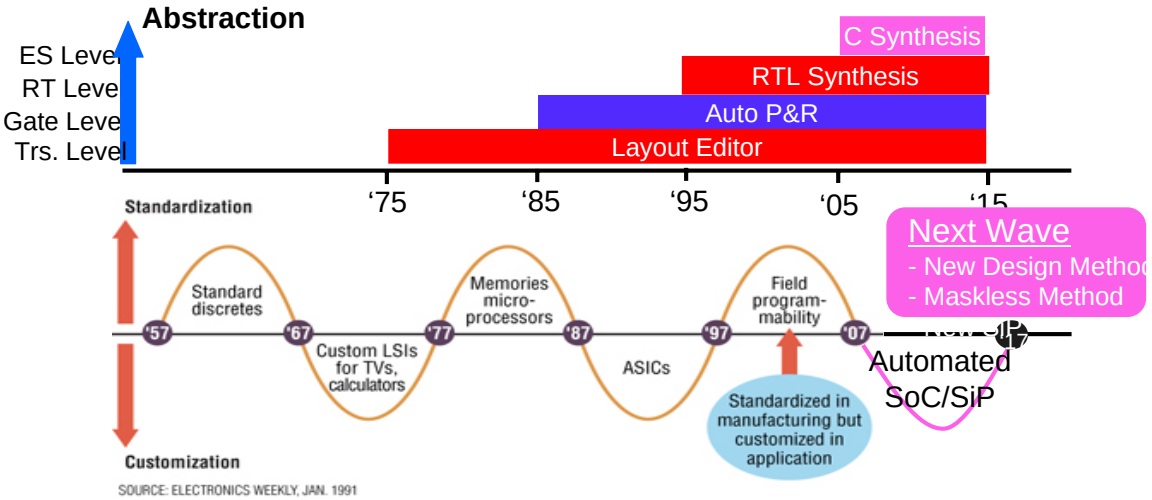
- システム技術が1チップに統合 （コンピュータ技術、無線技術、ネットワーク技術、デジタル信号処理等）
- 主な用途は、デジタルAV機器用、携帯電話用と車載用
- 開発期間は、HWとSW共、6ヶ月から長くて1年
- 「ESL設計技術」の育つ土壌



© Copyright 2004-2009 JEITA, All rights reserved

1-2.「牧本ウェーブ」 と EDA設計技術

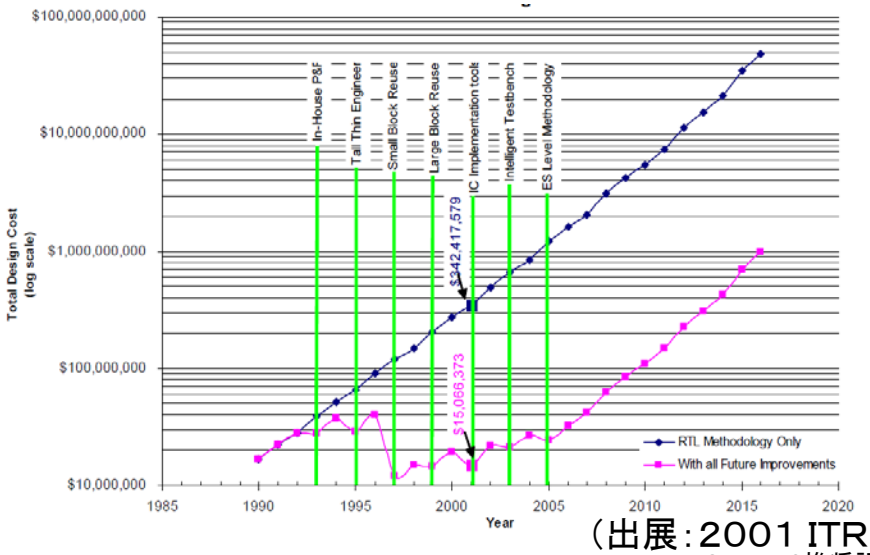
- 1991年に英国の新聞が命名（牧本氏、テクノビジョン代表）。「半導体産業は、標準化指向とカスタム指向が10年毎に入れ替わる。」
- EDA設計技術革新がパラダイムシフトをドライブした。
- ASICとFPGAは、RTL検証と合成がビジネスを可能にした。



© Copyright 2004-2009 JEITA, All rights reserved

1-3.設計危機

- 低消費電力SoC(ITRSシステムドライバー)
  - 2001年での設計コスト:15M$
  - 過去12年設計技術の改善なし:342M$
- 今後も、設計コストを抑止するための設計技術の改善努力が必要



(出展:2001 ITRS)

© Copyright 2004-2009 JEITA, All rights reserved

JEITA

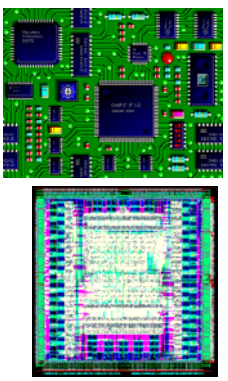
1-4. EDAとは ?

- Electronic Design Automation (EDA)
  - 広義では、電子機器システムの構想設計、実装設計と設計検証を可能にする「設計技術(DT: Design Technology)」
  - 狭義では、エレクトロニクス製品の設計ツール
- 設計技術は、設計ツール、設計ライブラリと設計方法論
- 設計技術は、システム設計者の設計構想と設計目標をコスト低減を考慮し、製造可能で検査可能な形式に変換する技術

システム設計者

EDA

- 設計ツール
- 設計ライブラリ
- 設計方法論

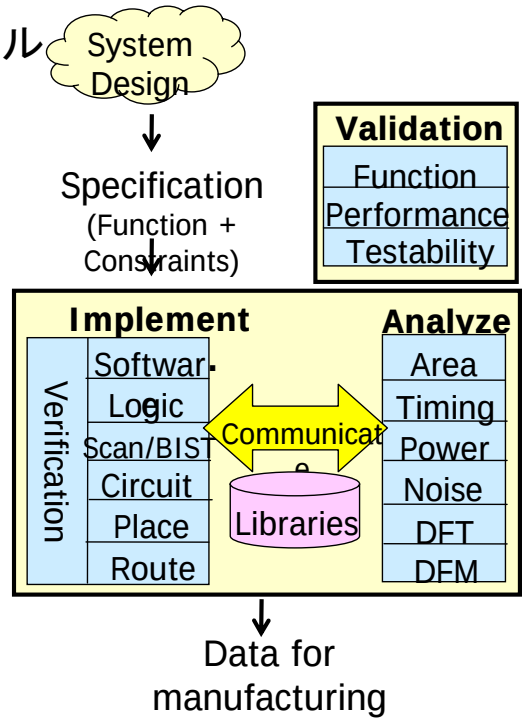


© Copyright 2004-2009 JEITA, All rights reserved

JEITA

1-5.設計システムとEDA標準の位置付け

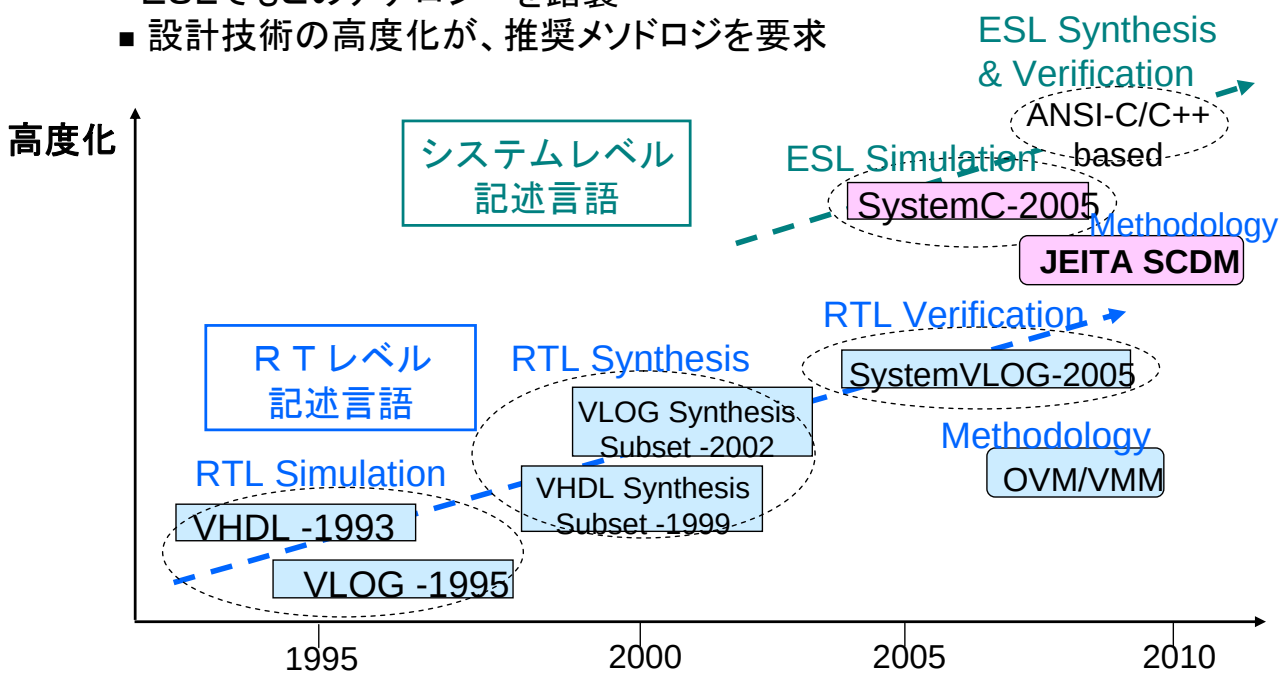
- 設計システムは、多種多様なツールを統合化し、仕様検討から製造可能データ作成を実現
- EDA標準は、設計システムへの共通セマンティックスのための機構を提供
- EDA標準の分類
  - Language
  - Format
  - Interoperability
  - Standard for new standards development



© Copyright 2004-2009 JEITA, All rights reserved

1-6. 設計言語標準化の歴史

- RTLでは、シミュレーション→合成→検証 と進化
  ESLでもこのアナロジーを踏襲
- 設計技術の高度化が、推奨メソドロジを要求



© Copyright 2004-2009 JEITA, All rights reserved

2. システム設計

© Copyright 2004-2009 JEITA, All rights reserved

2-1.概要

SystemCは、LSIのシステムレベルの設計に使われます。LSIシステム全体を高速にシミュレーションし、システムアーキテクチャ(CPUやメモリ、周辺ハードウエア、CPUバスなどの配置や接続)の最適解を探索します。または、周辺ハードウエアの詳細アーキテクチャを決め、ハードウエアに実装する(合成する)ために使われています。

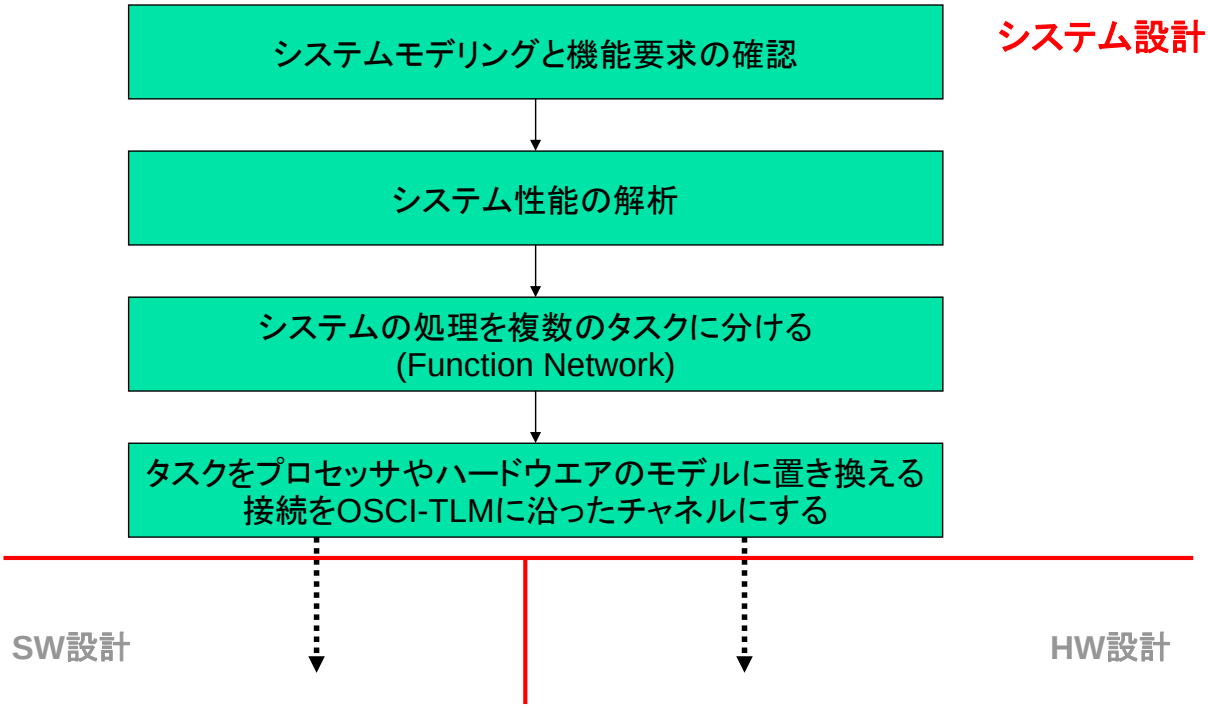
2005年にSystemCはIEEE-1666として標準化されました。2006年にTLM 1.0がOSCIより公開され、また合成サブセットがレビューのため公開されています。2008年にはTLM 2.0が正式リリースされる予定です。EDAツールもこれらの標準に対応していくと思われます。

ハードウエアの設計においても、これら標準化の流れに従って方法論を構築できれば、システムLSIの設計全体の流れをスムースにします。しかしながら、これら標準においてTLMと合成の橋渡しに関する議論がまだ十分行われていません。OSCIの合成サブセットにおいてSystemCの特徴を生かす合成方法について十分に述べられていません。

ここでは、システムLSI設計の方法論について、主に合成の立場から1つの筋の通る骨組みを提案します。

© Copyright 2004-2009 JEITA, All rights reserved

2-2.システムLSIの開発ステップ



© Copyright 2004-2009 JEITA, All rights reserved

2-3.システムLSIの開発ステップ

システムモデリングと機能要求の確認

システム全体または一定単位（ユニット）をC/C++/SystemCによって、Untimedな機能検討用モデリングを行い、機能要求が満たされている事を確認する。

システム性能の解析

機能検討モデルを用いたシステム性能検討C/C++/SystemCシミュレーション、ソフトウェアエミュレーションで実施する。

システムの処理を複数のタスクに分ける (Function Network)

システム性能確認に基づいた各機能単位を分割し、それら分割単位と接続からシステムのデータフローを明確にする。

タスクをプロセッサやハードウエアのモデルに置き換える 接続をOSCI-TLMに沿ったチャネルにする

各ファンクションユニットをCPU、SW、HWのモデルに置き換え、バス周辺のアーキテクチャ検討やSWとHWの検討を行う。

© Copyright 2004-2009 JEITA, All rights reserved

2-4.本ステップにおけるSystemCユースケース

	データ型	並列性	時間	チャネル
システムモデリングと機能要求の確認	△	×	×	×
システム性能の解析	△	×	×	×
システムの処理を複数のタスクに分ける	△	○	×	○
タスクをプロセッサやハードウエアのモデルに置き換える 接続をOSCI-TLMに沿ったチャネルにする	○	○	△	○

© Copyright 2004-2009 JEITA, All rights reserved

2-5.システムモデリングと機能要求の確認

- **このフェーズにおける目的**
  - C/C++を用いて、コンセプトモデルとなる機能検討用モデルの作成を行う。
    - 機能検討を効率的に行う為に出来るだけ詳細化を行わない。
    - ここでのモデルで使用するアルゴリズムは既に検討済みとする。
  - 機能検討に関わる構文に限定してSystemCを使用する。
    - タイミング、構造的な詳細化は避ける。
- **ゴール**
  - パフォーマンスを可能な限り損なわない機能検討モデルを構築する。
  - 機能が要求に満たす事を確認する。

© Copyright 2004-2009 JEITA, All rights reserved

2-6.機能検討モデリングにおけるSystemC要件

- ここでのSystemCの使用は、機能検討に関わる構文のみに限定する。
  - タイミング、構造的な詳細化は避ける。
    - シミュレーション速度の低下。
    - 機能ユニット分割後の検討でタイミングや構造は修正の可能性がある為、機能分割に影響を及ぼさない追加が望ましい。
  - 必要に応じてハードウェア化を検討する際にSystemCを利用する。
    - SystemCデータ型
    - SC_FIXED等
    - シミュレーション速度やソフトウェア化を考慮する場合、#ifdefやtypedefによるSystemCとCの構文切り替えを用いる。

© Copyright 2004-2009 JEITA, All rights reserved

2-7.機能検討モデリングにおけるSystemC要件

- 後段の実装手法を意識したコード記述をする事が望ましい。
  - 例えば動作合成によるRTL実装を行う場合に避ける事が望ましい記述例
    - 再帰呼び出し
    - 動的メモリアロケーション
    - STLの使用等

© Copyright 2004-2009 JEITA, All rights reserved

2-8.システム性能要求の確認

- このフェーズにおける目的
  - モデルの性能解析を行う。
  - 機能分割要件の明確化。
- ゴール
  - モデルの性能要件を満たす事を確認する。
  - 機能分割要件の情報をまとめる。

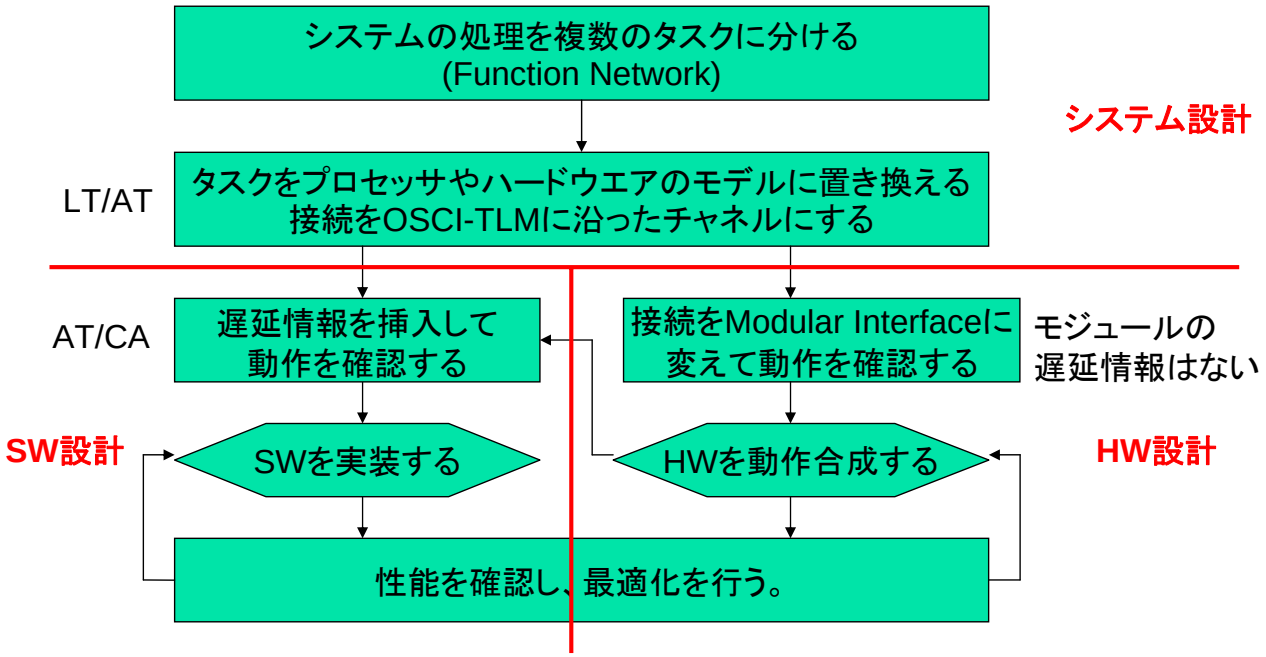
© Copyright 2004-2009 JEITA, All rights reserved

2-9.システム性能の検討と確認

- モデルの性能評価
- ファンクション・ネットワーク構築向けの機能分割の為の性能検討
  - 単純に機能単位に分割
  - ブロック間のデータ通信量を観測
  - 各機能の動作率等プロファイリングを行う

© Copyright 2004-2009 JEITA, All rights reserved

2-10.システムLSIの開発ステップ



LT:Loosely Timed， AT:Approximately Timed， CA: Cycle Accurate

© Copyright 2004-2009 JEITA, All rights reserved

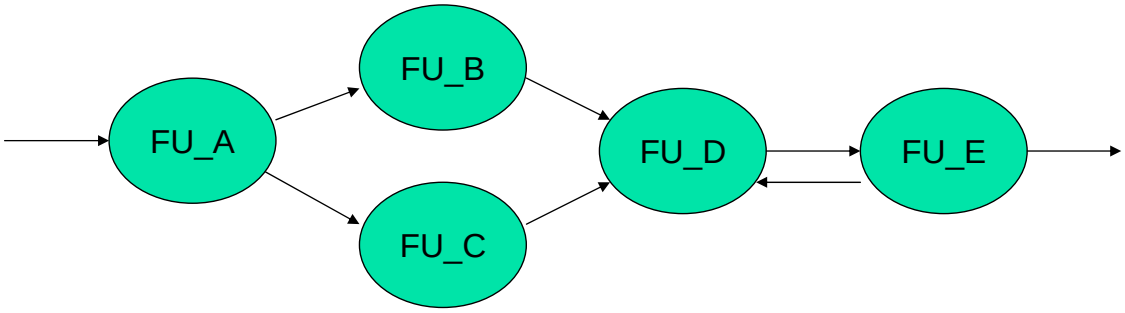
2-11.システム処理を複数タスクに分ける

- **このフェーズにおける目的**
  - アルゴリズムの分割と処理手順の明確化を行う。
    - ファンクション・ネットワークを構築し、TLM等を用いてデータのフローや並列性をモデルする。
    - SW/HW分割の為の基本情報を見つける。
- **ゴール**
  - 処理フローから並列処理、順次処理、パイプライン化の可能性が判明する。
    - 並列処理やパイプライン化のためには複数のハードウエアかプロセッサが必要である
    - 順次処理が多い場合は、プロセッサを用いるのが最適である可能性が高い

© Copyright 2004-2009 JEITA, All rights reserved

2-12.Function Network

- 機能ユニット(Function Unit)がデータの流れに応じて接続(Network)される。



図からわかること

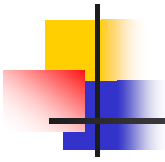
•FU_BとFU_Cは並列に動作可能。
•A->B/C->Dの処理はパイプライン化することもシーケンシャル処理することもできる。
•D->Eの処理はフィードバックがあるためにシーケンシャルにしか処理できない。

© Copyright 2004-2009 JEITA, All rights reserved

2-13.システムの検討項目

- Function Unit
  - 機能役割をモデル化
  - ファンクション(function)とデータ(data)
- Network
  - データフローをモデル化
  - 端子(port)と接続(interface)
- Structure
  - データ構造をモデル化
  - クラス(class)とアクセスメソッド(member function)
- Process
  - 計算過程をモデル化
  - 状態遷移(state machine)とイベント(event)

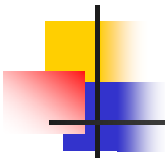
© Copyright 2004-2009 JEITA, All rights reserved



2-14.TLMの設計場面(use case)

① SWアプリケーション開発とSW/HW実装
② SWパフォーマンス解析
③ SWアーキテクチャ解析
④ HWアーキテクチャ解析
⑤ HWパフォーマンス検証
⑥ HW改良と実装
HWはBL、テストベンチがTL
⑦ HW機能検証
HWはRTL,テストベンチがTL

© Copyright 2004-2009 JEITA, All rights reserved



2-15.ソフトウエアとハードウエア

- ソフトウエアの利点
  - 実装後も変更が容易
  - 開発期間が短い（RTLと比べた場合、BLとはどうか）
- ハードウエアの利点
  - 高速性能が得やすい
  - 低消費電力化の可能性が高い

© Copyright 2004-2009 JEITA, All rights reserved

2-16. アーキテクチャ設計

CPUを含むシステムでのアーキテクチャ設計では、CPUの速度、メモリのサイズ、周辺ハードウエアの仕様、CPUバスについて検討します。CPUに着目した場合、その上で動作するソフトウエアのサイズ、処理能力を確認します。メモリは、データ格納場所の割り当て、インタフェース速度が十分であるかを確認します。

周辺ハードウエアは、IPなどを使用して仕様が固定されている場合と新規に作成するために詳細設計が必要な場合があります。新規のハード上の回路規模が大きい場合には、さらにそれ単体で細かくアーキテクチャ設計を行っていくことになります。CPUを含まないシステムの場合には、データの流れが決まっていることが多く、データは個々のモジュール間で直接渡されるようになります。

CPUやメモリ、CPUバスなど既存のものをシステムに組み込む場合は、シミュレーションの高速化のために主にデータ転送のモデリングを工夫します。一般に用いられるのがTransaction Level Modeling(TLM)と呼ばれるものです。モデリングのためにはシミュレーションの速度を落とすことなく遅延情報などを詳細に記述することに重きを置きます。一方、ハードウエア設計ではリソースの詳細を明確にする必要があります。

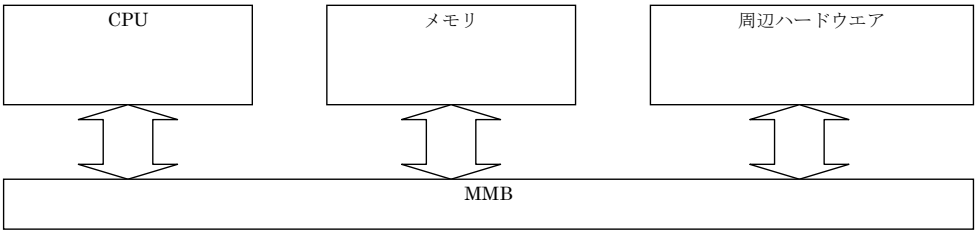
© Copyright 2004-2009 JEITA, All rights reserved

2-17.タスクをプロセッサやハードウエアのモデルに置換える

- このフェーズにおける目的
  - タスクをSW/HWを考慮したモデルに置き換える。
    - TLM等を用いて検証速度を考慮したモデルを行う。
    - SW/HW分割したモデルの詳細な検討を行う。
- ゴール
  - CPUやバス周辺、メモリ等システムのアーキテクチャが要求に満たす事を確認する。
    - モデルは各SW/HW設計に渡される。

© Copyright 2004-2009 JEITA, All rights reserved

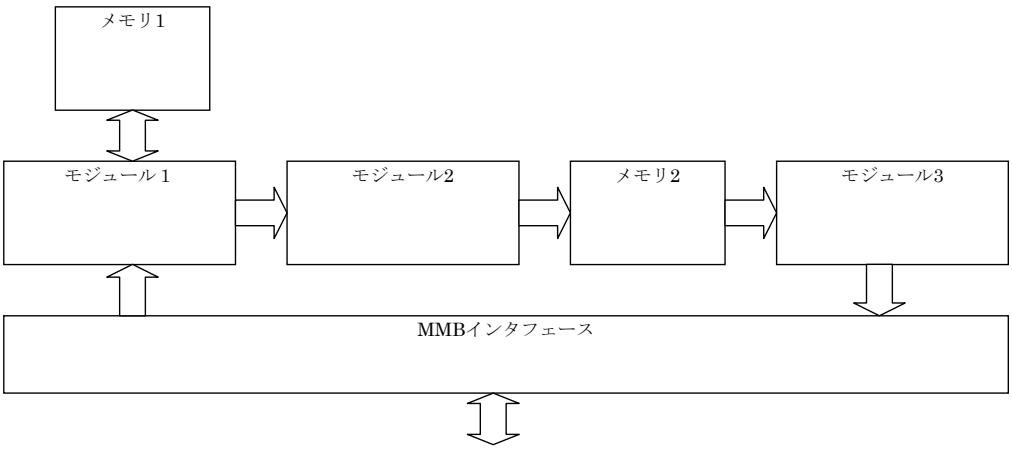
2-18. CPUを中心としたアーキテクチャ



- 1つのCPUはファンクションの1個か複数個を実現する。
- ファンクションを接続するネットワークは、MMB上に実現される。ネットワークを実現するSWのライブラリが必要である。

© Copyright 2004-2009 JEITA, All rights reserved

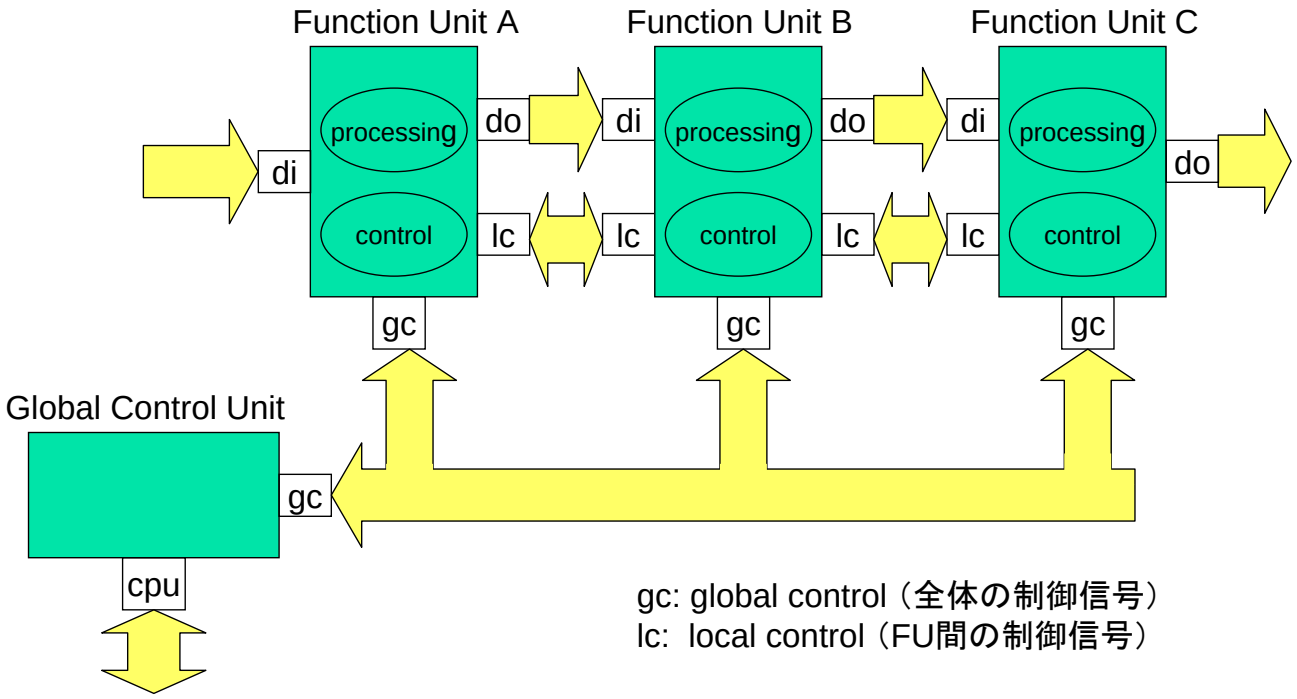
2-19 周辺ハードウエアのアーキテクチャ



- 一般にモジュールは単機能。
- ファンクションを接続するネットワークは、HWライブラリとして用意する。

© Copyright 2004-2009 JEITA, All rights reserved

2-20.システム



© Copyright 2004-2009 JEITA, All rights reserved

3.HW/SW協調検証

© Copyright 2004-2009 JEITA, All rights reserved

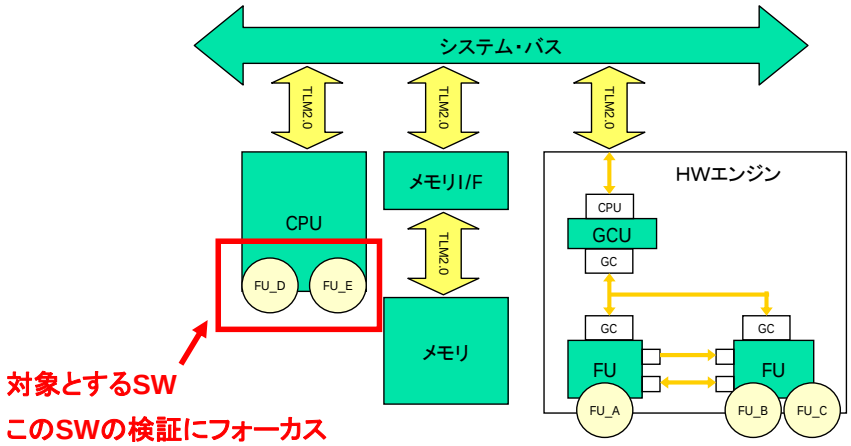
3-1.内容

- HW/SW協調検証編の位置づけ
- HW/SW協調検証の4つのPhase
- Phase1-4の目的と抽象度
- 各Phaseの詳細説明
  - Phase1の詳細説明
  - Phase2の詳細説明
  - Phase3の詳細説明
  - Phase4の詳細説明

© Copyright 2004-2009 JEITA, All rights reserved

3-2.HW/SW協調検証編の位置づけ

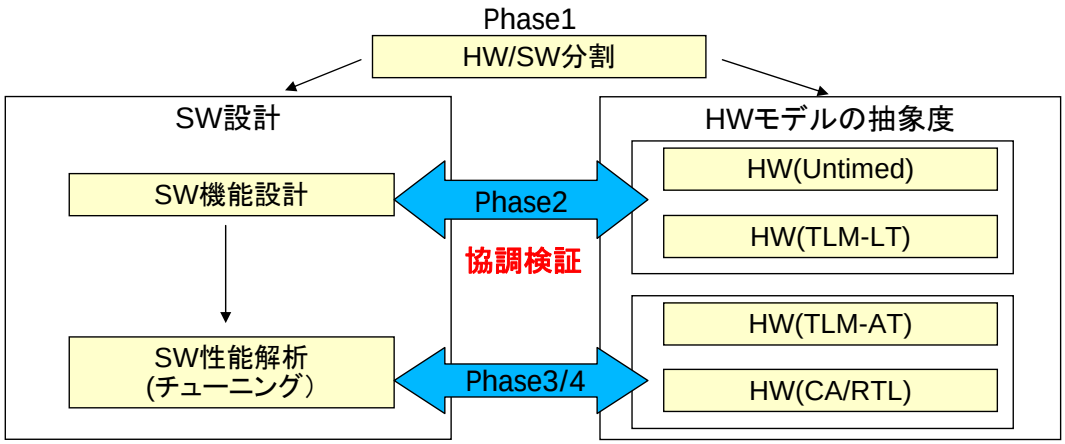
HW/SW協調検証環境



- HW/SW協調検証編では，Function Networkにおいて HW/SW分割でSWに割り当てられたFunction Unitの検証にフォーカス
- 目的に応じたSW検証に必要なHWモデルの抽象度を定義

© Copyright 2004-2009 JEITA, All rights reserved

3−3. HW/SW協調検証の4つのPhase



- Phase1：システム全体の機能設計，HW/SW分割見積り
- Phase2：SW機能設計・検証，機能的なHWとSWのIF検証
- Phase3：システム性能の再確認，SWチューニング(HWとSWのタイミング)
- Phase4：HW性能の再確認

© Copyright 2004-2009 JEITA, All rights reserved

3−4. Phase1-4の目的と抽象度

	目的	SW抽象度（言語）	CPUモデル抽象度	HWモデル抽象度（CPU除く）	相対必要速度
Phase1	システム全体の機能設計 HW/SW分割見積り	SystemC(UT), C, C++	Host Native	Untimed	数Gcps
		SW環境選定	**CPU選定**		
Phase2	SW機能設計・検証 機能的なHWとSWのIF検証	実際のSW （RTOSとデバイスドライバの開発）	ISS	TLM-LT	数Mcps
Phase3	システム性能の再確認 SWチューニング (HWとSWのタイミング)	実際のSW （RTOS上で、アルゴリズムを動作させる）	ISS	TLM-AT	数 100Kcps
Phase4	HW性能の再確認	実際のSW （HWとのIFを確認する）	サイクル精度ISS/RTL	CA/RTL	数Kcps

3-5.Phase1の詳細説明

- 目的
  - SystemC推奨設計メソドロジ（システム設計編）に従い
    - システムモデリングと機能要求の確認
    - システム性能の解析
- モデリング（HW・SW)
  - 各Function unitにはHW/SWの区別なくアルゴリズムを表現する機能のみをモデル化
  - なるべく高い抽象度モデル
    - C,C++, SystemC(UT)など
- 結果（出力）
  - Function Networkのモデル
  - HW/SWの分割結果（各FunctionがSWかHWか）
  - CPUとSW開発環境（ソフトウェアスタック）の選定

© Copyright 2004-2009 JEITA, All rights reserved

3-6.Phase2の詳細説明

- 目的
  - SW機能設計・検証
    - デバイスドライバを開発する
  - 機能的なHWとSWのIF検証
    - HWに依存しているデバイスドライバの機能設計
  - システム性能の再確認
    - RTOSなどのソフトウェアスタックの実装
- SWの開発
  - IF検証用としてHWモデル内のレジスタのR/Wプログラム
  - RTOSなどのソフトウェアスタックの実装
    - デバイスドライバ（HW非依存部とHW依存部の両方）
- HWのモデリング
  - システム・バスのプロトコルに従ったTLMモデル（TLM2.0に準拠）
  - ブロッキングモデル（TLM-LTレベル）
    - タイミング精度より，機能確認を優先
- 結果（出力）
  - 実際のSW（ソフトウェアスタック）
    - RTOS、ファームウェア、ライブラリ（OpenGLなど）
  - HWエンジンモデル（TLM-LT)

© Copyright 2004-2009 JEITA, All rights reserved

3-7.Phase3の詳細説明

- 目的
  - SWのチューニング（HWとSWのタイミング）
    - 最適なHWとSWのI/F（並列化など）を確認
- SWの開発
  - SW全体をチューニング
    - アルゴリズム（演算の並列化やデータの固まりの見直し）
    - RTOS（タスク・スレッドのスケジューリングや優先順位）
    - デバイスドライバ（割込みの優先順位やレジスタ本数）
- HWのモデリング
  - 検証対象に準じて、モデルの詳細化を実施する
  - ノンブロッキングモデル（TLM-ATレベル）
    - タイミングクリティカルなHW-SW間プロトコルをTLM-ATへ変更
- 結果（出力）
  - 実際のSW
    - チューニングされたSW（ソフトウェアスタック全体）
  - HWエンジンモデル（TLM-AT）

© Copyright 2004-2009 JEITA, All rights reserved

3-8.Phase4の詳細説明

- 目的
  - HW性能の再確認
    - SWを含めた正確なタイミングでのHWの性能解析とチューニング
- SWの開発
  - サイクル精度レベルでの最終確認
  - 必要ならチューニングも実施
- HWのモデリング
  - サイクル精度のモデルを作成する
    - 動作合成結果のRTLやサイクル精度のSystemCモデルを利用．動作合成しないモデルについては別途サイクル精度のモデルを用意（例えば既存RTLなど）
- 結果（出力）
  - 実際のSW
    - サイクル精度でチューニングが必要な場合
  - HWエンジンモデル（CAまたはRTL）

© Copyright 2004-2009 JEITA, All rights reserved

4. HW設計

© Copyright 2008 JEITA, All rights reserved

4-1.はじめに

- 設計対象のシステム
  - データ加工を行うシステム
    - デジタルフィルタ・サーボ
    - 音声・画像・動画圧縮伸長
    - 誤り訂正符合生成・再生
    - 暗号化・復元
    - ・・・
- 対象外
  - アナログシステム
  - メモリやペリフェラルなどの単純なモジュール単体
  - 制御が主体のシステム

© Copyright 2004-2009 JEITA, All rights reserved

4-2.FNDM2

Function Network Design Methodology
version 2.0

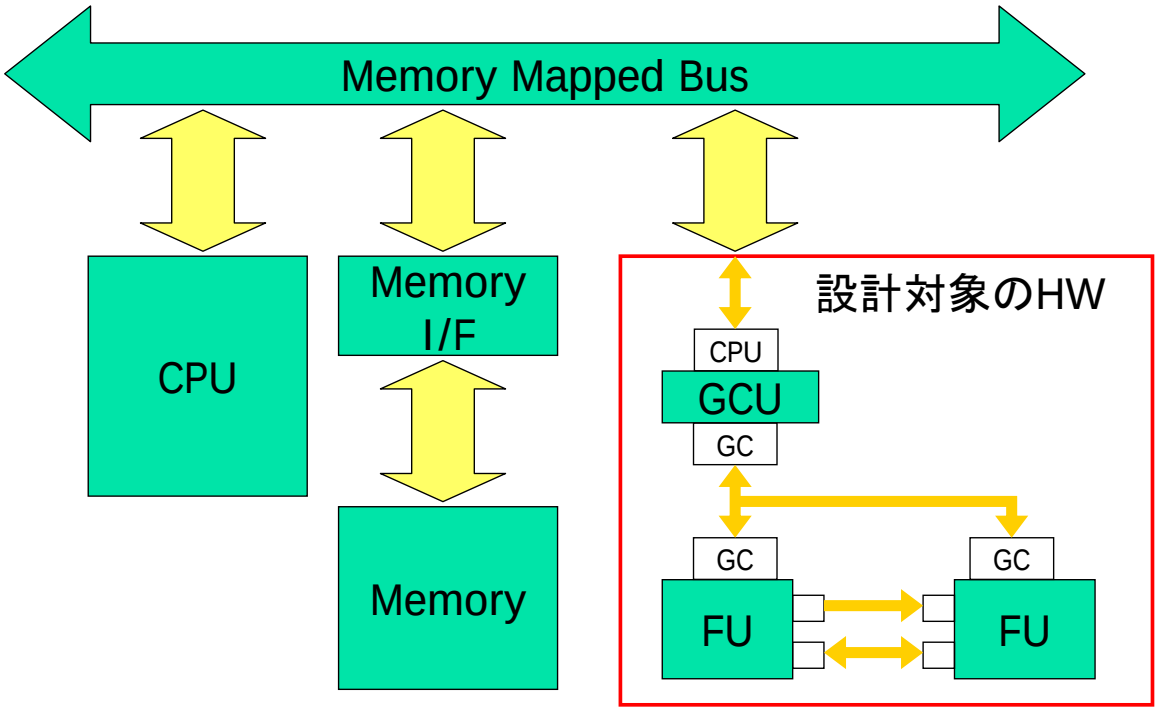
© Copyright 2004-2009 JEITA, All rights reserved

4-2.FNDM2について

- 2007年度に「SystemC推奨設計メソドロジ・合成編」として作成されたものを改良。
- 2007年度はHWの動作合成を前提にしたFNDM(Function Network Design Methodology)と呼ぶ設計手法を提案した。
- そこで、2007年度版をFNDM1とし、新たに提案するものをFNDM2とする。
- FNDM2は設計対象範囲をデータ処理するHWだけでなく、HWと組み合わせられるデータ処理を行うSWにまで広げている。

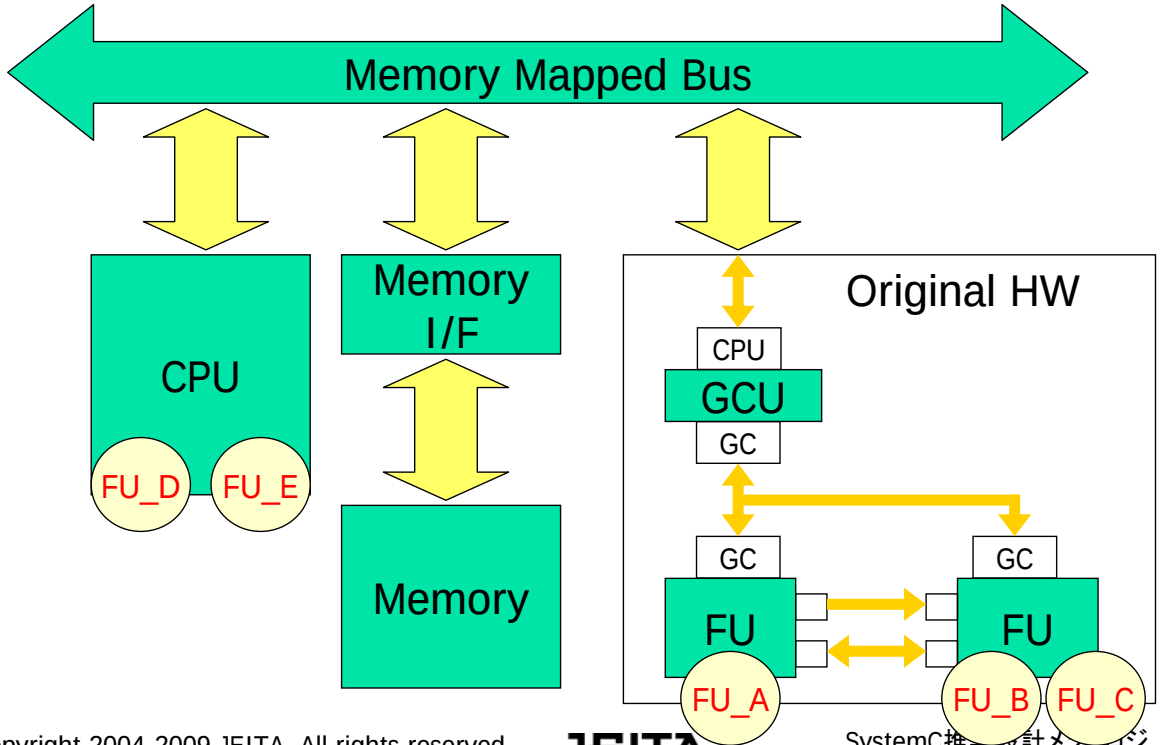
© Copyright 2004-2009 JEITA, All rights reserved

4-2.FNDM1で設計対象としたHW



© Copyright 2004-2009 JEITA, All rights reserved

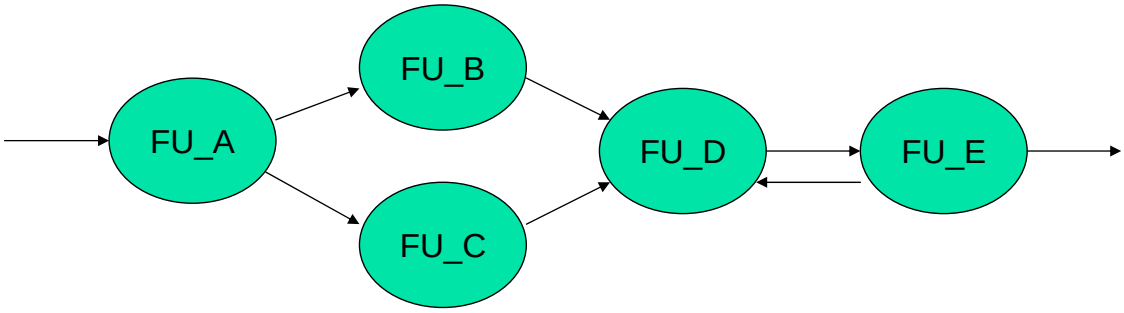
4-2.FNDM2で設計対象とするHWとSW



© Copyright 2004-2009 JEITA, All rights reserved

4−2.Function Network

- 機能ユニット(Function Unit)がデータの流れに応じて接続(Network)される。

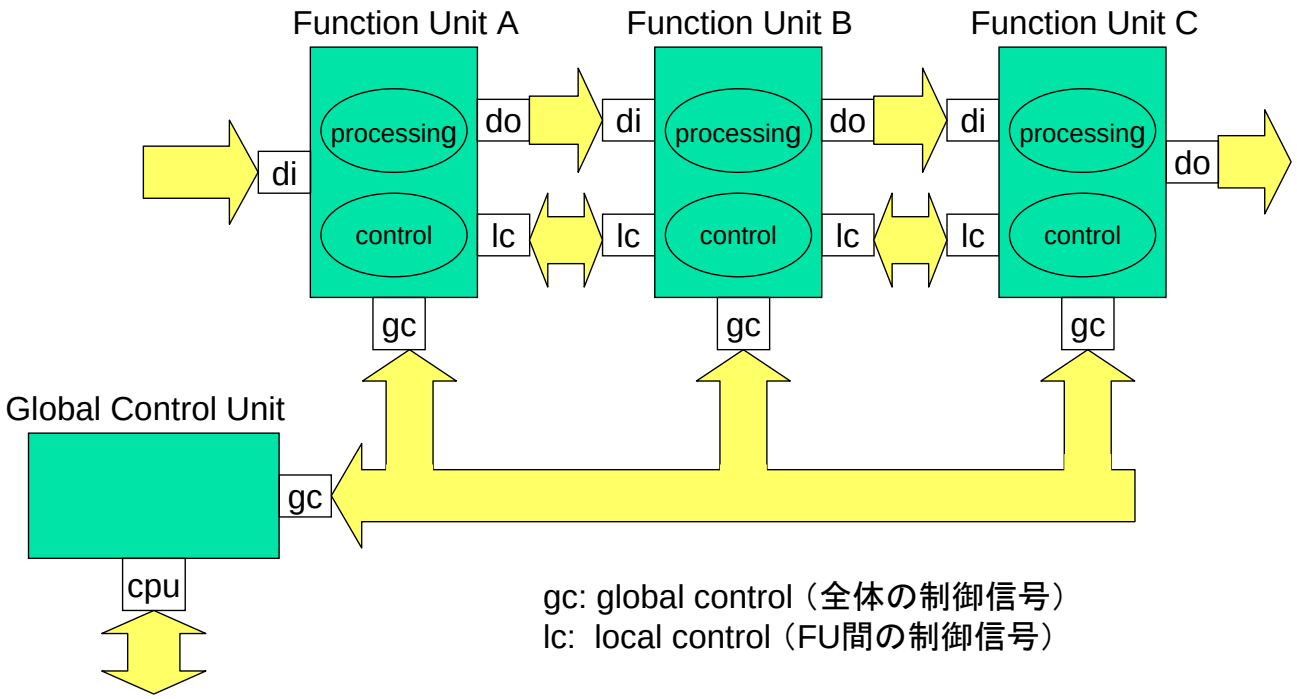


図からわかること

- FU_BとFU_Cは並列に動作可能。
- FU_A->FU_B/FU_C->FU_Dの処理はパイプライン化することもシーケンシャル処理することもできる。
- FU_D->FU_Eの処理はフィードバックがあるためにシーケンシャルにしか処理できない。

© Copyright 2004-2009 JEITA, All rights reserved

4−2.Function Network を実装するシステムの構成



gc: global control（全体の制御信号）
lc: local control（FU間の制御信号）

© Copyright 2004-2009 JEITA, All rights reserved

4-2.信号の分類

- データ転送
  片方向のハンドシェークを使ったデータ転送が多い。
  モジュラーインタフェースを使う。
- ローカル制御
  データ転送の相手との制御情報の交換する(データのハンドシェーク信号はデータ転送に含まれる)
- グローバル制御
  GCU (Global Control Unit)と制御信号を交換する。
  パラメータのような静的な制御信号が多い。

© Copyright 2004-2009 JEITA, All rights reserved

4-2.Function Unit

- データの処理を行うユニット
- データの演算、加工はprocessingプロセスで行う。
- データに依存して処理が変わる場合、processingプロセスから取り出されたその制御信号はcontrolプロセスで扱う。
- プロセスは、processingプロセスかcontrolプロセスに分類できる。その数に制約はない。
- データの入出力ポート数に制約はない。
- gcポートは1つでなければならない。

© Copyright 2004-2009 JEITA, All rights reserved

4−2.Global Control Unit

- FU (Function Unit)の制御を行う。
- 通常はシステムに1つ。
- 将棋倒し型の場合は、CPUインタフェースと制御データの保持を行う。
- 中央制御型の場合は、将棋倒し型の機能に加えて各FUの動作タイミングを制御する。

© Copyright 2004-2009 JEITA, All rights reserved

4−2.FUの動作タイミング

- 将棋倒し型

  データの入力側のモジュールから順に入力データを受け取り準備が整ったところから処理を始める。

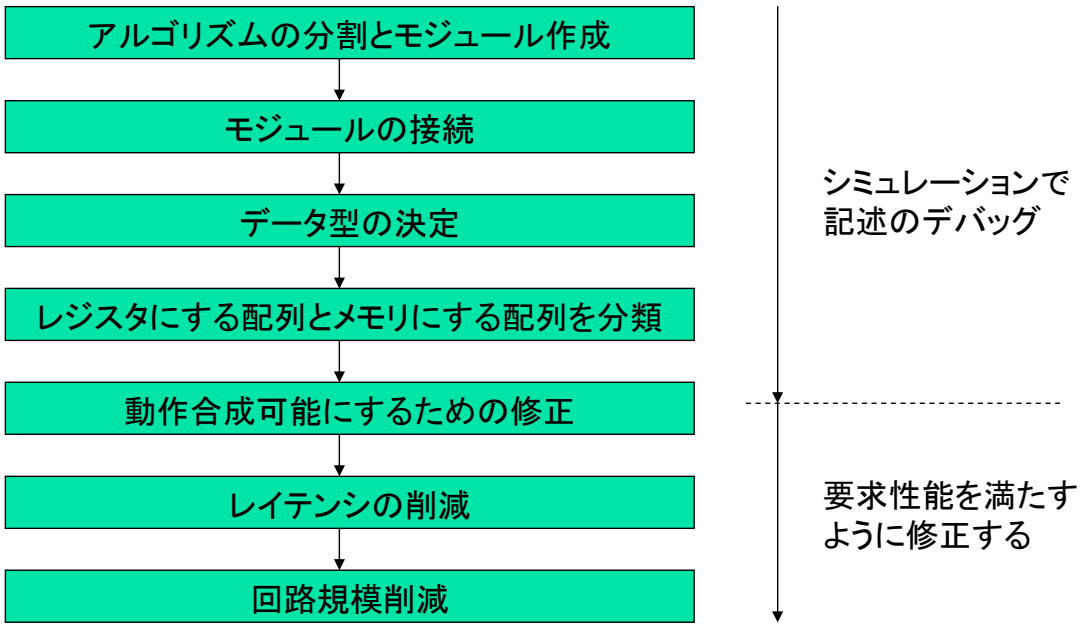
  出力が滞り、内部バッファに余裕がなくなった場合は、データ入力を停止する。

- 中央制御型

  GCUの合図に基づいて各モジュールは動作を行う。

© Copyright 2004-2009 JEITA, All rights reserved

4-2.HW設計の流れ



© Copyright 2004-2009 JEITA, All rights reserved

4-2.動作記述の特徴を生かす

下記の4つを独立して設計を行う。

ライブラリ

- インタフェース （モジュラーインタフェース）
- メモリアクセス （モジュラーインタフェース）
- 演算系 （ユーザ定義型）

システム

- アルゴリズムとアーキテクチャ
  （モジュールと階層構造）

© Copyright 2004-2009 JEITA, All rights reserved

4-2.基本ルール

- モジュールの役割をFU(Function Unit)かGCU(Global Control Unit)のどちらかに定める。
- モジュールはそれぞれの用意された雛形を拡張して記述する。
- インタフェース、メモリアクセス、演算系はライブラリを利用する。
- ライブラリを拡張したり、新規に作成する場合は、メンバ関数名、引数の統一化を行う。

© Copyright 2004-2009 JEITA, All rights reserved

4-3.インタフェース

© Copyright 2004-2009 JEITA, All rights reserved

4-3.出力インタフェースの公開メンバ関数

- ブロッキング関数（データの送受信が終了するまで戻らない）
  void put(const TD &data, const bool &vld=true)
- ブロッキング関数（データの送受信を1回だけ試み成否を返す）
  bool ns_put(const TD &data, const bool &vld=true)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  bool nb_can_put(const bool &vld=true)
  実行
  bool nb_put(const TD &data)
  後処理
  bool nb_clr_put()
- ポート接続
  void bind(T channel)
  void operator ()(T channel)
- インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-3.シングル入力インタフェースの公開メンバ関数

- ブロッキング関数（データの受信が終了するまで戻らない）
  void get(TD &data, const bool &en=true)
  T get(const bool &en=true)
- ブロッキング関数（データの受信を1回だけ試み成否を返す）
  bool ns_get(TD &data, const bool &en=true)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  bool nb_can_get(const bool &en=true)
  実行
  bool nb_get(TD &data)
  後処理
  bool nb_clr_get()
- ポート接続
  void bind(T channel)
  void operator ()(T channel)
- インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-3.インタフェース・チャネルの公開メンバ関数とポート

- チャネルのトレース

  friend sc_trace(sc_trace_file*, const T&, const std::string&)

- FIFOなどの内部にレジスタを持つチャネルでは、クロックやリセットが必要になるが、その場合は、以下のポート名を使うものとする。

  clk　クロック

  rst　リセット（負論理）

  ※解析用に信号が必要な場合については検討中である。

© Copyright 2004-2009 JEITA, All rights reserved

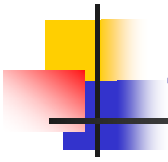
4-3.put関数

void put(const TD &data, const bool &vld=true)

vldが真のときにdataを送信する。送信が終了するまで次の処理に進まない。

vldが偽のときは、送信を行わない。vldが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

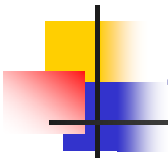


4−3.ns_put関数

bool ns_put(const TD &data, const bool &vld=true)

- vldが真のときにdataの送信を1回試みる。成功した場合はtrueが、失敗した場合はfalseが戻る。
- vldが偽のときは、送信を行わない。vldが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

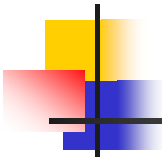


4−3.nb_can_put関数

bool nb_can_put(const bool &vld=true)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- vldが偽の場合は常に偽が戻される。
- vldが真でかつ送信がすぐ可能な場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



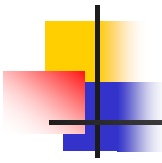
4-3.nb_put関数

bool nb_put(const TD &data)

ノンブロッキング関数であり、クロックを含まずに処理する。

データ送信が行われる場合に真が戻り、行われないと判断できる場合に偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



4-3.nb_clr_put関数

bool nb_clr_put()

ノンブロッキング関数であり、クロックを含まずに処理する。

nb_put()を実行した次のクロックサイクルでデータ送信を終了する場合に必ず実行する。
nb_can_put(false)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−3.put()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void put(const TD &data, const bool &vld=true){
  if(vld){
    while(!nb_can_put()) wait();
    while(!nb_put(data)) wait();
    do{wait();}
    while(nb_clr_put());
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

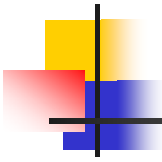
4−3.get関数

void get(TD &data, const bool &en=true)

T get(const bool &en=true)

- enが真のときにデータを受信する。受信が終了するまで次の処理に進まない。
- enが偽のときは、受信を行わない。enが真のときに必要な最小クロック数だけ待って次の処理に進む。
- get()関数には2通りの形態があり、dataを引数として渡してそこに受信値を受け取るか、戻り値として受信値として受け取る。

© Copyright 2004-2009 JEITA, All rights reserved

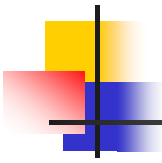


4−3.ns_get関数

bool ns_get(TD &data, const bool &en=true)

- enが真のときにdataの受信を1回試みる。成功した場合はtrueが、失敗した場合はfalseが戻る。
- enが偽のときは、受信を行わない。enが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

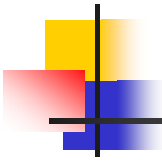


4−3.nb_can_get関数

bool nb_can_get(const bool &en=true)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- enが偽の場合は常に偽が戻される。
- enが真でかつ受信の予約ができた場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



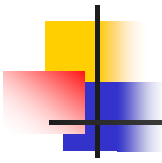
4-3.nb_get関数

bool nb_get(TD &data)

ノンブロッキング関数であり、クロックを含まずに処理する。

データ受信が行われる場合に真が戻り、行われない場合に偽が戻る。nb_can_get()が真になったあと一定数のクロック後に1クロックだけ真になる。このタイミングを逃すとnb_get()で真の戻り値を得ることができない。

© Copyright 2004-2009 JEITA, All rights reserved



4-3.nb_clr_get関数

bool nb_clr_get()

ノンブロッキング関数であり、クロックを含まずに処理する。

nb_can_get()を実行した次のクロックサイクルで次のデータ受信を続けない場合に必ず実行する。nb_can_get(false)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−3.get()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void get(TD &data, const bool &en=true){
  if(en){
    while(!nb_can_get()) wait();
    do{wait();}
    while(!nb_get(data));
    while(!nb_clr_get()) wait();
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

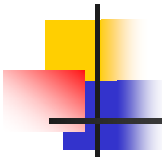
4−3.bind関数

void bind(T channel)

void operator ()(T channel)

モジュールのポートとそのモジュールの外部のチャネルとを接続する。そのモジュールの上位モジュールのポートとの接続も可能。

© Copyright 2004-2009 JEITA, All rights reserved

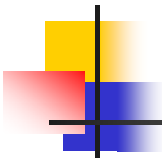


4−3.reset関数

void reset()

インタフェースポートに関連したレジスタを初期化する。

© Copyright 2004-2009 JEITA, All rights reserved



4−3.sc_trace関数

friend sc_trace(sc_trace_file*, const T&, const std::string&)

vcdファイルにトレースを記録する場合に使用する。

© Copyright 2004-2009 JEITA, All rights reserved

4−3.blocking関数の使い方

blocking関数の記述は簡単。

```
while(1){
  input.get(d_in);
  d_out = calc(d_in);
  output.put(d_dout);
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4−3.Non-blocking関数の使い方

nb_clr_get()とnb_clr_putをnb_can_get()とnb_can_put()に続くwait()の直後におく。条件判定に直接記述するとどちらか一方のみしか起動できない場合があるので、check_stall()関数を用意する。

```
bool en、vld, is_ds;  T data;
en=true;  vld=false;  is_ds=false;
while(1){
  input.nb_can_get(en);
  if(is_ds)
    is_ds = false;
  else
    vld = input.nb_get(data);

  if(vld)  d_out = calc(d_in);

  if(vld)
    if(en=output.nb_can_put())  output.nb_put(d_dout);

  do{wait();}
  while(check_stall(data, is_ds));
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4-3.check_stall関数

- ノンブロッキング関数を利用するとき、複数のclr関数がある場合に用意する。
- 出力の都合でストールする場合、入力データを取りこぼす可能性があるので対処をしておく。

```
bool check_stall(T &ds, bool &is_ds ){
    bool stall = !input.nb_clr_get();
    stall = stall || !output.nb_clr_put();
    if(stall&&!is_ds)
      is_ds = input.nb_get(ds);
    return stall;
  }
```

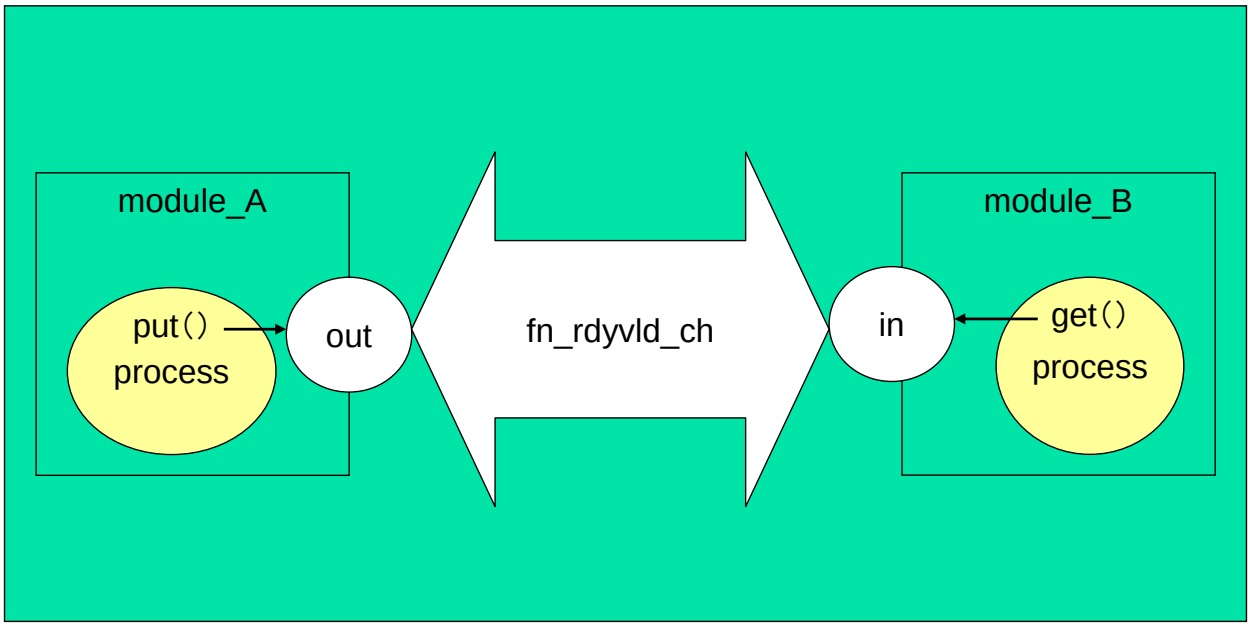
© Copyright 2004-2009 JEITA, All rights reserved

4-3.インタフェース・ライブラリ

- モジュラーインタフェースで用意される。
- mi::mi_get_if<TD>クラスかmi::mi_put_if<TD>クラスを継承している。
- ハンドシェークのみでバッファされないデータ転送
  fn_rdyvld_ch<TD>　チャネル
  fn_rdyvld_ch<TD>::in　データ入力ポート
  fn_rduvld_ch<TD>::out　データ出力ポート
- レジスタで構成されたFIFOを持つデータ転送
  fn_fifo_ch<TD,FL>　チャネル
  fn_fifo_ch<TD,FL>::in　データ入力ポート
  fn_fifo_ch<TD,FL>::out　データ出力ポート
- SRAMで構成されたFIFOを持つデータ転送
  fn_fifo_sram_ch<TMW>　チャネル
  fn_fifo_sram_ch<TMW>::in　データ入力ポート
  fn_fifo_sram_ch<TMW>::out　データ出力ポート
- FIFOのチャネルにはclk（立ち上がり動作）とrst(負論理）のポートがあり、クロックとリセットの供給が必要である。

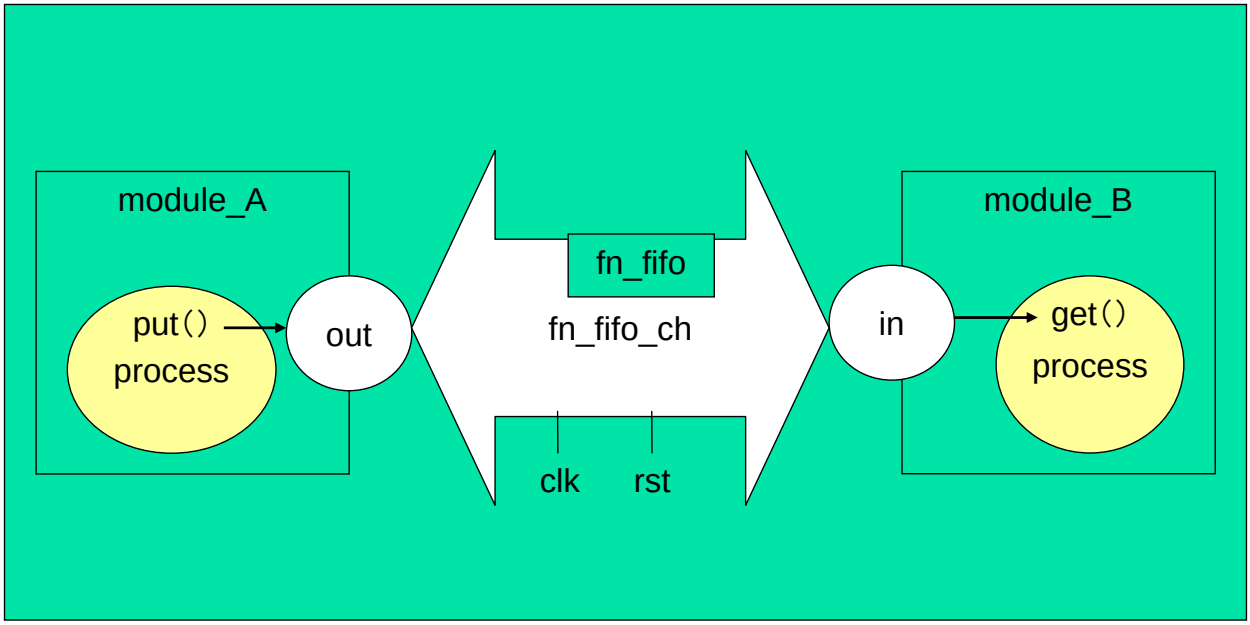
© Copyright 2004-2009 JEITA, All rights reserved

4-3.ハンドシェーク (fn_rdyvld_ch)



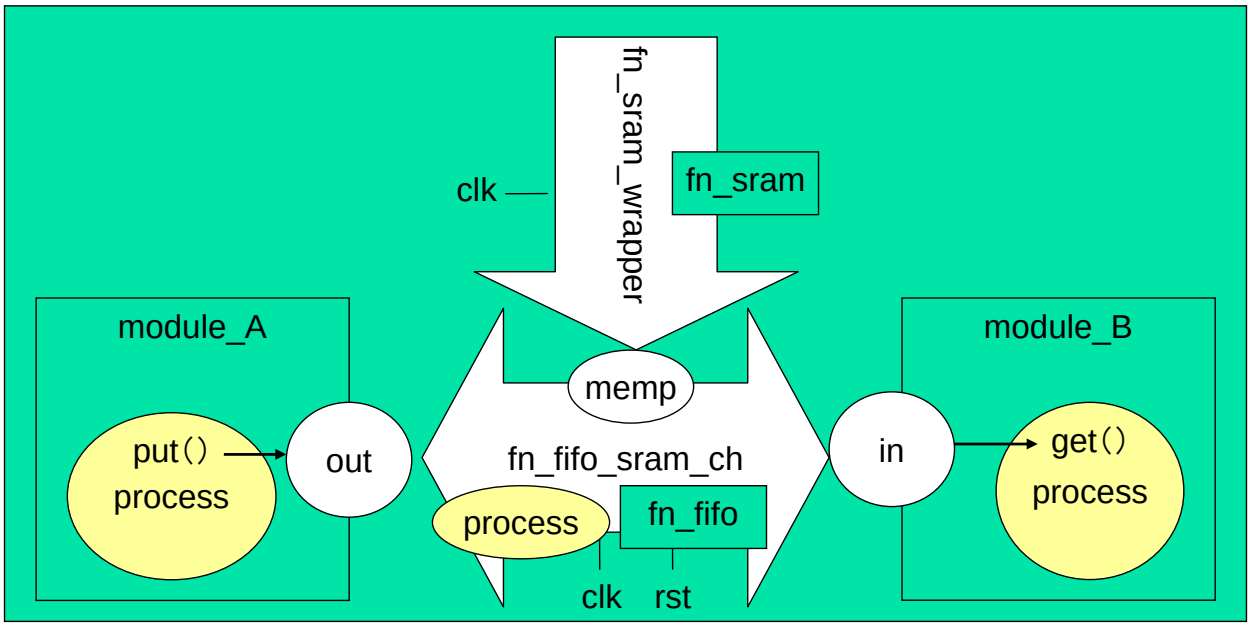
© Copyright 2004-2009 JEITA, All rights reserved

4-3.FIFO (fn_fifo_ch)



© Copyright 2004-2009 JEITA, All rights reserved

4.3.SRAMを用いたFIFO (fn_fifo_sram_ch)



© Copyright 2004-2009 JEITA, All rights reserved

4−3.マルチ入力インタフェース

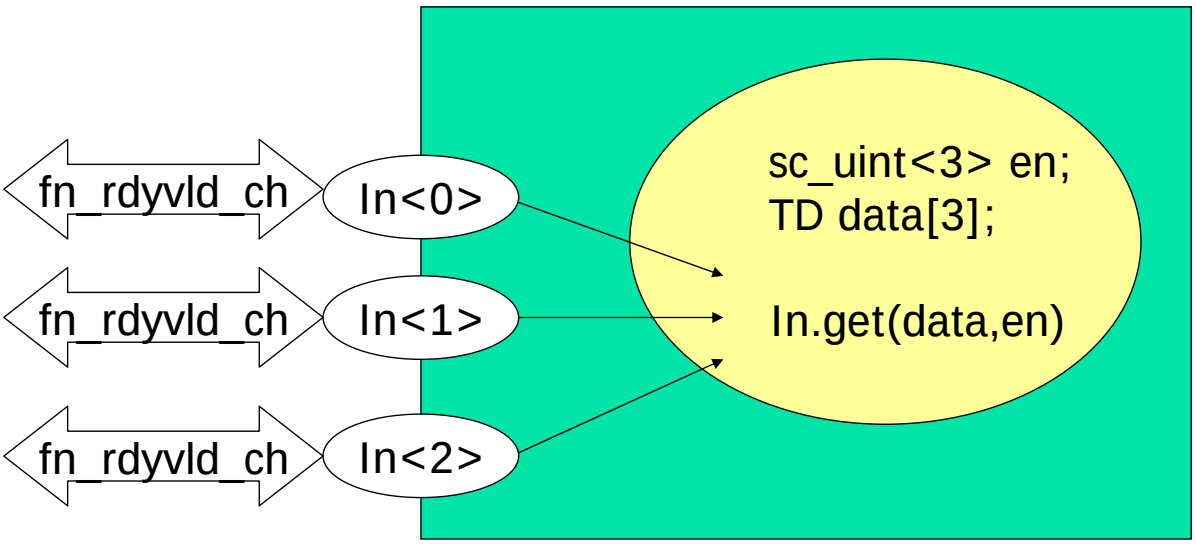
© Copyright 2004-2009 JEITA, All rights reserved

4-3.マルチ入力インタフェースの概要

- 1つのモジュールに同一種類の入力インタフェースが複数あるときに使用することができる。
- 複数の入力を一度にアクセスするためのメンバ関数が用意されている。
- 同一種類の入力インタフェースが複数ある場合でも、独立して入力する場合は、シングル入力インタフェースを用いた方が良い。
- ポートは入力の数だけ存在し、シングル入力インタフェースと同じチャネルを接続する。

© Copyright 2004-2009 JEITA, All rights reserved

4-3.マルチ入力インタフェースの例



© Copyright 2004-2009 JEITA, All rights reserved

4-3.マルチ入力インタフェースの公開メンバ関数

- ブロッキング関数（データの受信が終了するまで戻らない）
  void get(TD data[num_of_port], const sc_uint<num_of_port> &en)
- ブロッキング関数（データの受信を1回だけ試み成否を返す）
  sc_uint<num_of_port> ns_get(TD data[num_of_port], const sc_uint<num_of_port> &en)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  sc_uint<num_of_port> nb_can_get(const sc_uint<num_of_port> &en)
  実行
  sc_uint<num_of_port> nb_get(TD &data, const sc_uint<num_of_port> &en)
  後処理
  sc_uint<num_of_port> nb_clr_get(const sc_uint<num_of_port> &en)
- ポート接続
  void bind<num>(T channel)
- 全インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-3.get関数

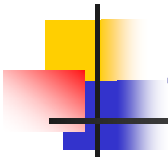
void get(TD data[num_of_port], const sc_uint<num_of_port> &en)

enのビットが1のポートのデータを受信する。指定した全受信が終了するまで次の処理に進まない。

enが0のときは、受信を行わない。enが0でないときに必要な最小クロック数だけ待って次の処理に進む。

num_of_portは、ポートの数。

© Copyright 2004-2009 JEITA, All rights reserved

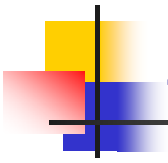


4-3.ns_get関数

sc_uint<num_of_port> ns_get(TD data[num_of_port], const sc_uint<num_of_port> &en)

- enと戻り値は各ビットが各ポートの動作を示す。
- enのビットが1のポートのdataの受信を1回試みる。成功したポートのビットは1が、失敗したポートのビットは0が戻る。
- enのビットが0のポートは、受信を行わない。enが0の場合は、0でないときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

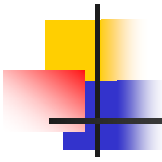


4-3.nb_can_get関数

sc_uint<num_of_port> nb_can_get(const sc_uint<num_of_port> &en)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- enが偽の場合は常に偽が戻される。
- enが真でかつ受信の予約ができた場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



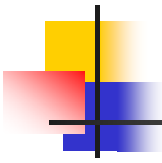
4-3.nb_get関数

sc_uint<num_of_port> nb_get(TD &data, const sc_uint<num_of_port> &en)

ノンブロッキング関数であり、クロックを含まずに処理する。

データ受信が行われる場合に真が戻り、行われない場合に偽が戻る。nb_can_get()が真になったあと一定数のクロック後に1クロックだけ真になる。このタイミングを逃すとnb_get()で真の戻り値を得ることができない。

© Copyright 2004-2009 JEITA, All rights reserved



4-3.nb_clr_get関数

sc_uint<num_of_port> nb_clr_get(const sc_uint<num_of_port> &en)

ノンブロッキング関数であり、クロックを含まずに処理する。

nb_can_get()を実行した次のクロックサイクルで次のデータ受信を続けない場合に必ず実行する。nb_can_get(0)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−3.get()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void get(TD data[num_of_port], const sc_uint<num_of_port>
    &en){
  if(en){
    sc_uint<num_of_port> abits = en;
    while(nb_can_get(en)!=en) wait();
    do{ wait(); }
    while((abits &= ~nb_get(abits))!=0);
    while(nb_clr_get(en)!=en) wait();
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

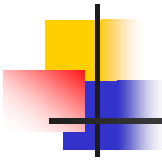
4−3.bind関数

template<unsigned int num>

void bind(T channel)

モジュールのポートとそのモジュールの外部のチャネルとを接続する。そのモジュールの上位モジュールのポートとの接続も可能。numはポート番号。

© Copyright 2004-2009 JEITA, All rights reserved

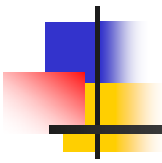


4-3.reset関数

void reset()

インタフェースポートに関連したレジスタを初期化する。

© Copyright 2004-2009 JEITA, All rights reserved



4-4.マルチ出力インタフェース

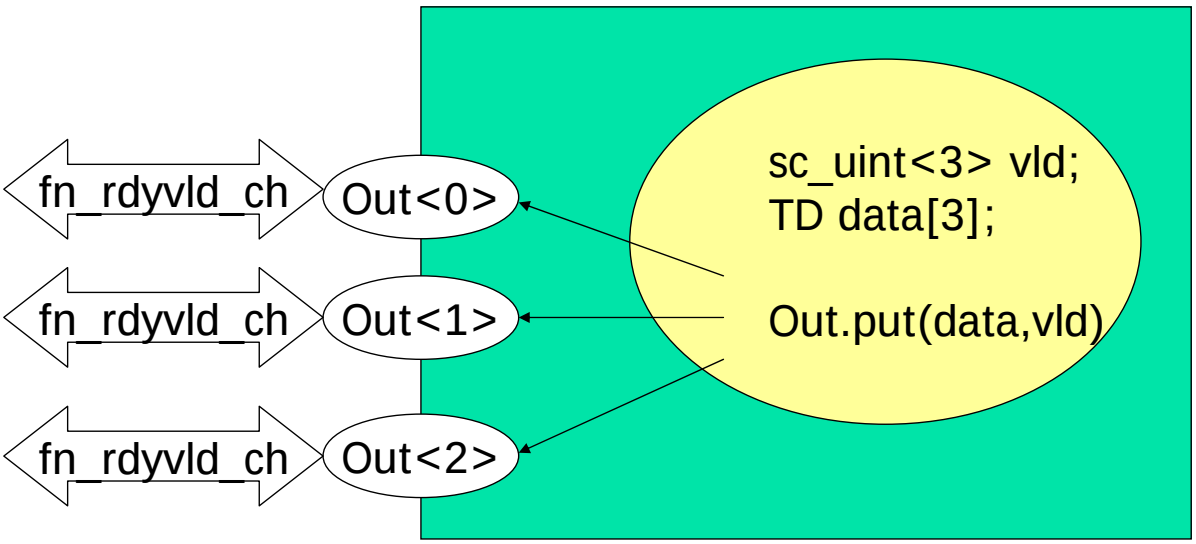
© Copyright 2004-2009 JEITA, All rights reserved

4-4.マルチ出力インタフェースの概要

- 1つのモジュールに同一種類の出力インタフェースが複数あるときに使用することができる。
- 複数の出力を一度にアクセスするためのメンバ関数が用意されている。
- 同一種類の出力インタフェースが複数ある場合でも、独立して出力する場合は、シングル出力インタフェースを用いた方が良い。
- ポートは出力の数だけ存在し、シングル出力インタフェースと同じチャネルを接続する。

© Copyright 2004-2009 JEITA, All rights reserved

4-4.マルチ出力インタフェースの例



© Copyright 2004-2009 JEITA, All rights reserved

4-4.マルチ出力インタフェースの公開メンバ関数

- ブロッキング関数（データの送受信が終了するまで戻らない）
  void put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)
- ブロッキング関数（データの受信を1回だけ試み成否を返す）
  sc_uint<num_of_port> ns_put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  sc_uint<num_of_port> nb_can_put(const sc_uint<num_of_port> &vld)
  実行
  sc_uint<num_of_port> nb_put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)
  後処理
  sc_uint<num_of_port> nb_clr_put(const sc_uint<num_of_port> &vld)
- ポート接続
  void bind<num>(T channel)
- 全インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-4.put関数

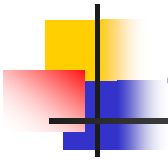
void put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)

vldのビットが1のポートのデータを送信する。指定した全送信が受信されるまで次の処理に進まない。

vldが0のときは、送信を行わない。vldが0でないときに必要な最小クロック数だけ待って次の処理に進む。

num_of_portは、ポートの数。

© Copyright 2004-2009 JEITA, All rights reserved

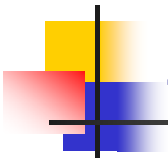


4−4.ns_put関数

sc_uint<num_of_port> ns_put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)

- vldと戻り値は各ビットが各ポートの動作を示す。
- vldのビットが1のポートのdataの送信を1回試みる。成功したポートのビットは1が、失敗したポートのビットは0が戻る。
- vldのビットが0のポートは、送信を行わない。vldが0の場合は、0でないときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

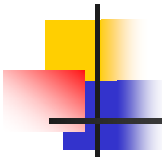


4−4.nb_can_put関数

sc_uint<num_of_port> nb_can_put(const sc_uint<num_of_port> &vld)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- vldが偽の場合は常に偽が戻される。
- vldが真でかつ送信がすぐ可能な場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved

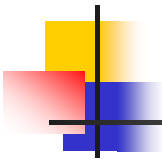


4−4.nb_put関数

sc_uint<num_of_port> nb_put(const TD data[num_of_port], const sc_uint<num_of_port> &vld)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- データ送信が行われる場合に真が戻り、行われないと判断できる場合に偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



4−4.nb_clr_put関数

sc_uint<num_of_port> nb_clr_put(const sc_uint<num_of_port> &vld)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- nb_put()を実行した次のクロックサイクルでデータ送信を終了する場合に必ず実行する。nb_can_put(0)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−4.put()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void put(const TD data[num_of_port], const
    sc_uint<num_of_port> &vld){
  if(vld!=0){
    sc_uint<num_of_port> abits = vld;
    while(nb_can_put(vld)!=vld) wait();
    while((abits &= ~nb_put(data,abits))!=0) wait();
    do{wait();}
    while(nb_clr_put(vld)!=vld) wait();
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

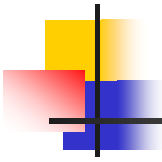
4−4.bind関数

template<unsigned int num>

void bind(T channel)

モジュールのポートとそのモジュールの外部のチャネルとを接続する。そのモジュールの上位モジュールのポートとの接続も可能。numはポート番号。

© Copyright 2004-2009 JEITA, All rights reserved

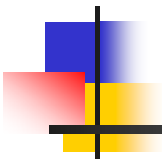


4−4.reset関数

void reset()

インタフェースポートに関連したレジスタを初期化する。

© Copyright 2004-2009 JEITA, All rights reserved



4−5.メモリアクセス

© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリ書き込みの公開メンバ関数

- ブロッキング関数（メモリの書き込みが終了するまで戻らない）
  void put(const TA &address, const TD &data, const bool &vld=true)
- ブロッキング関数（メモリの書き込みを1回だけ試み成否を返す）
  bool ns_put(const TD &data, const bool &vld=true)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  bool nb_can_put(const TA &address, const bool &vld=true)
  実行
  bool nb_put(const TD &data)
  後処理
  bool nb_clr_put()
- ポート接続
  void bind(T channel)
  void operator ()(T channel)
- インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリ読み出しの公開メンバ関数

- ブロッキング関数（メモリの読み出しが終了するまで戻らない）
  void get(const TA &address, TD &data, const bool &en=true)
  T get(const TA &address, const bool &en=true)
- ブロッキング関数（メモリの読み出しを1回だけ試み成否を返す）
  bool ns_get(TD &data, const bool &en=true)
- ノンブロッキング関数（毎サイクル実行される必要がある）
  前準備
  bool nb_can_get(const TA &address, const bool &en=true)
  実行
  bool nb_get(TD &data)
  後処理
  bool nb_clr_get()
- ポート接続
  void bind(T channel)
  void operator ()(T channel)
- インタフェースポートの初期化
  void reset()

© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリアクセス・チャネルの公開メンバ関数とポート

- チャネルのトレース

  friend sc_trace(sc_trace_file*, const T&, const std::string&)

- ポート

  clk　クロック

  rst　リセット（負論理）

  // 解析用信号（検討中）

© Copyright 2004-2009 JEITA, All rights reserved

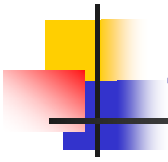
4-5.put関数

void put(const TA &address, const TD &data, const bool &vld=true)

vldが真のときにdataを書き込む。書き込みが終了するまで次の処理に進まない。

vldが偽のときは、書き込みを行わない。vldが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

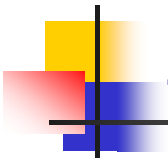


4−5.ns_put関数

bool ns_put(const TA &address, const TD &data, const bool &vld=true)

- vldが真のときにdataの書き込みを1回試みる。成功した場合はtrueが、失敗した場合はfalseが戻る。
- vldが偽のときは、書き込みを行わない。vldが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

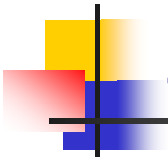


4−5.nb_can_put関数

bool nb_can_put(const TA &address, const bool &vld=true)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- vldが偽の場合は常に偽が戻される。
- vldが真でかつ書き込みがすぐ可能な場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



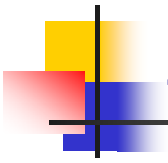
4-5.nb_put関数

bool nb_put(const TD &data)

ノンブロッキング関数であり、クロックを含まずに処理する。

データ書き込みが行われる場合に真が戻り、行われないと判断できる場合に偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



4-5.nb_clr_put関数

bool nb_clr_put()

ノンブロッキング関数であり、クロックを含まずに処理する。

nb_put()を実行した次のクロックサイクルでデータ書き込みを終了する場合に必ず実行する。nb_can_put(false)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−5.put()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void put( const TA &address, const TD &data, const
   bool &vld=true){
  if(vld){
    while(!nb_can_put(address)) wait();
    while(!nb_put(data)) wait();
    do{wait():}
    while(nb_clr_put());
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

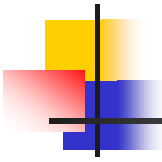
4−5.get関数

void get(const TA &address, TD &data, const bool &en=true)

T get(const TA &address, const bool &en=true)

- enが真のときにデータを読み出す。読み出しが終了するまで次の処理に進まない。
- enが偽のときは、読み出しを行わない。enが真のときに必要な最小クロック数だけ待って次の処理に進む。
- get()関数には2通りの形態があり、dataを引数として渡してそこに受信値を受け取るか、戻り値として受信値として受け取る。

© Copyright 2004-2009 JEITA, All rights reserved

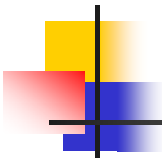


4−5.ns_get関数

bool ns_get(const TA &address, TD &data, const bool &en=true)

- enが真のときにdataの読み出しを1回試みる。成功した場合はtrueが、失敗した場合はfalseが戻る。
- enが偽のときは、読み出しを行わない。enが真のときに必要な最小クロック数だけ待って次の処理に進む。

© Copyright 2004-2009 JEITA, All rights reserved

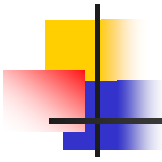


4−5.nb_can_get関数

bool nb_can_get(const bool &en=true)

- ノンブロッキング関数であり、クロックを含まずに処理する。
- enが偽の場合は常に偽が戻される。
- enが真でかつデータ読み出しの予約ができた場合に真が戻り、それ以外の場合は偽が戻る。

© Copyright 2004-2009 JEITA, All rights reserved



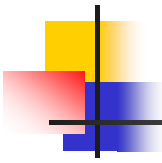
4−5.nb_get関数

bool nb_get(TD &data)

ノンブロッキング関数であり、クロックを含まずに処理する。

データ読み出しが行われる場合に真が戻り、行われない場合に偽が戻る。nb_can_get()が真になったあと一定数のクロック後に1クロックだけ真になる。このタイミングを逃すとnb_get()で真の戻り値を得ることができない。

© Copyright 2004-2009 JEITA, All rights reserved



4−5.nb_clr_get関数

bool nb_clr_get()

ノンブロッキング関数であり、クロックを含まずに処理する。

nb_can_get()を実行した次のクロックサイクルで次のデータ読み出しを続けない場合に必ず実行する。nb_can_get(false)で代用させる事ができる。

© Copyright 2004-2009 JEITA, All rights reserved

4−5.get()のnon-blocking関数による定義

non-blocking関数は、以下の使用方法でblocking関数が作れるように実装すること。

```
void get(const TA &address, TD &data, const bool
   &en=true){
  if(en){
    while(!nb_can_get(address)) wait();
   do{wait();}
    while(!nb_clr_get());
    while(!nb_get(data)) wait();
  }else
    wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

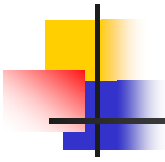
4−5.bind関数

void bind(T channel)

void operator ()(T channel)

モジュールのポートとそのモジュールの外部のチャネルとを接続する。そのモジュールの上位モジュールのポートとの接続も可能。

© Copyright 2004-2009 JEITA, All rights reserved

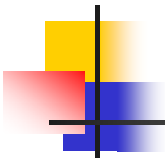


4−5.reset関数

void reset()

メモリアクセスポートに関連したレジスタを初期化する。

© Copyright 2004-2009 JEITA, All rights reserved



4−5.sc_trace関数

friend sc_trace(sc_trace_file*, const T&, const std::string&)

vcdファイルにトレースを記録する場合に使用する。

© Copyright 2004-2009 JEITA, All rights reserved

4−5.blocking関数の使い方

blocking関数の記述は簡単。
forループの中で使われる場合が多い

```
sum = 0;
for(address = address_min; address <
   address_max; ++address){
  input.get(address, d_in);
  sum+=din;
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4−5.Non-blocking関数の使い方

Non-blocking関数は毎サイクル実行されなければならない。
メモリアクセスのnb_can_getとnb_getは並列に実行する。
forループの中で使われる場合が多い。

```
sum=0;  addr=addr_min;
for(sc_uint<4> i=0; i<8;){
  if(mem.nb_can_get(addr, addr<addr_max)) addr++;
  vld = mem.nb_get(din);
  if(vld){
    ++i;
    sum+= din;
  }
  wait();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4−5.Non-blocking関数の使い方2

nb_can_getの第2引数enableを使用するのではなく、nb_clr_get関数によって処理を継続しないことを示した方がわかりやすいかもしれない。

```
sum=0;  addr=addr_min;
for(sc_uint<4> i=0; i<8;){
  if(addr<addr_max)
    if(mem.nb_can_get(addr)) addr++;
  vld = mem.nb_get(din);
  if(vld){
    ++i;  sum+= din;
  }
  wait();
  mem.nb_clr_get();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4−5.Non-blocking関数の使い方3

メモリアクセスのnb_can_putとnb_putの使い方は、インタフェースの場合に似ている。
これもforループの中で使われることが多い。

```
en=true;  vld=false;
for(address = address_min; address < address_max; ){
  if(input.nb_can_get(en))  vld = input.nb_get(d_in);
  else  vld = !en;
  if(vld)  d_out = calc(d_in);

  if(mem.nb_can_put(address, vld)){
    en = mem.nb_put(d_dout);
    ++address;
  }else  en = !vld;
  wait();
  mem.nb_clr_put();
}
```

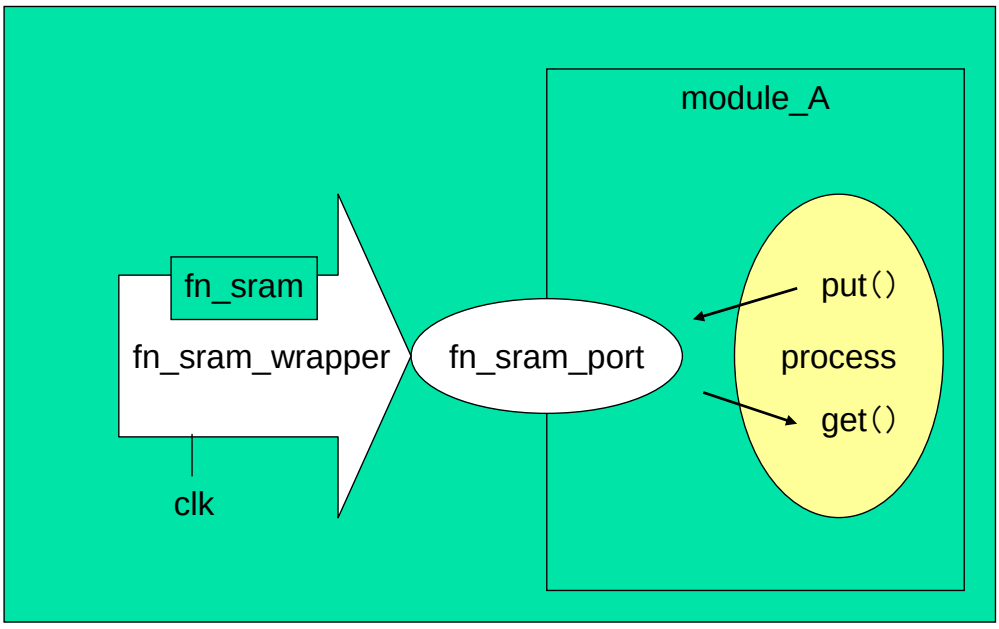
© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリアクセス・ライブラリ

- モジュラーインタフェースで用意される。
- mi::mi_mem_get_if<TA,TD>クラスかmi::mi_mem_put_if<TA,TD>クラスを継承している。
- メモリはモジュールの外部に配置する。
- 1つのプロセスからgetとputをする場合
  fn_sram_port<TMW> SRAMアクセスポート
- getするプロセスとputするプロセスが異なる場合
  fn_sram_ch<TMW>　チャネル
  fn_sram_ch<TMW>::mem　SRAMアクセスポート
- getするモジュールとputするモジュールが異なる場合
  fn_sram_wr_ch<TMW>　チャネル
  fn_sram_wr_ch<TMW>::in　SRAMリードアクセスポート
  fn_sram_wr_ch<TMW>::out　SRAMライトアクセスポート
- **getとputをするモジュールが複数ある場合**
  fn_sram_multi_ch<TMW, num_of_port> チャネル
  fn_sram_multi_if<TMW, port_number> SRAMアクセスポート
- getとputが別のチャネルにはclk（立ち上がり動作）とrst(負論理）のポートがあり、クロックとリセットの供給が必要である。

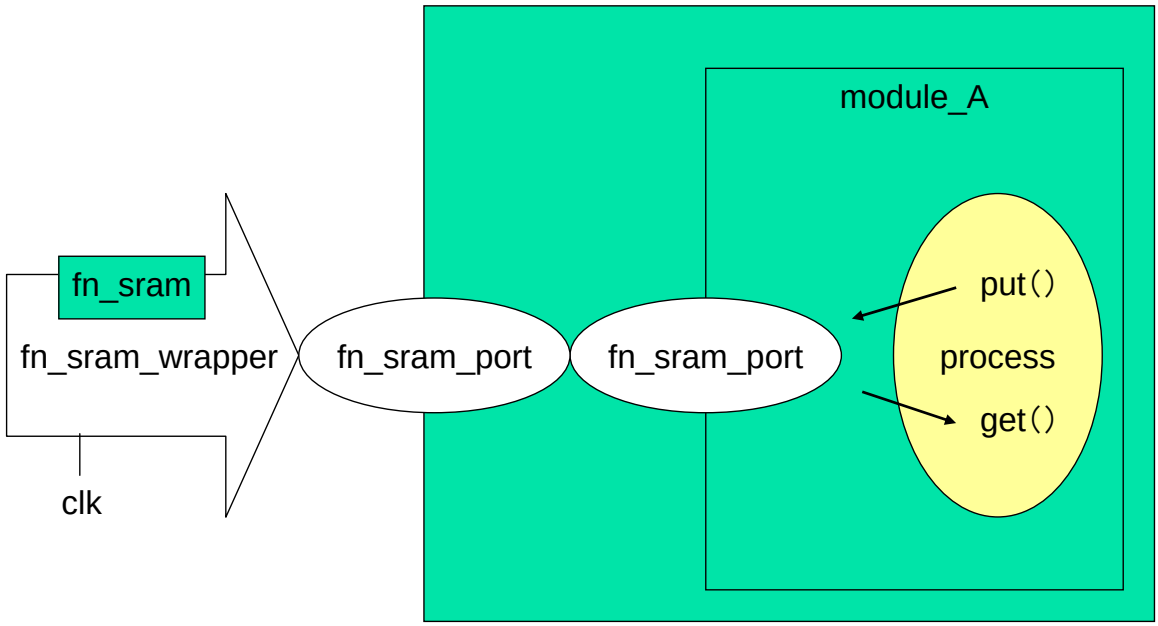
© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリアクセス(fn_sram_port)



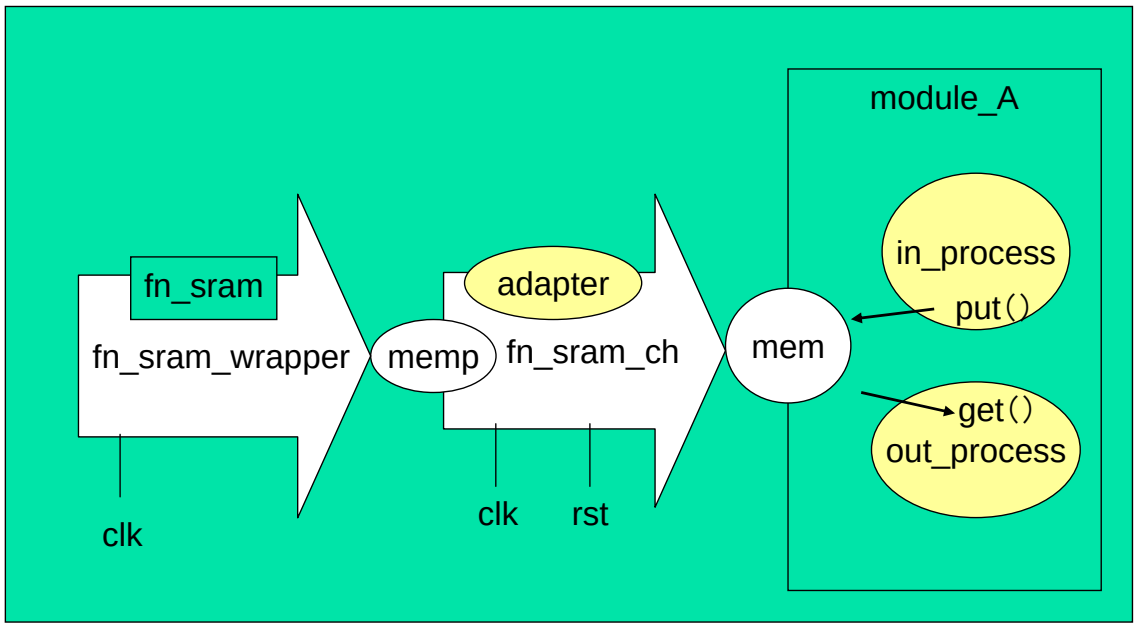
© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリアクセス(fn_sram_port)



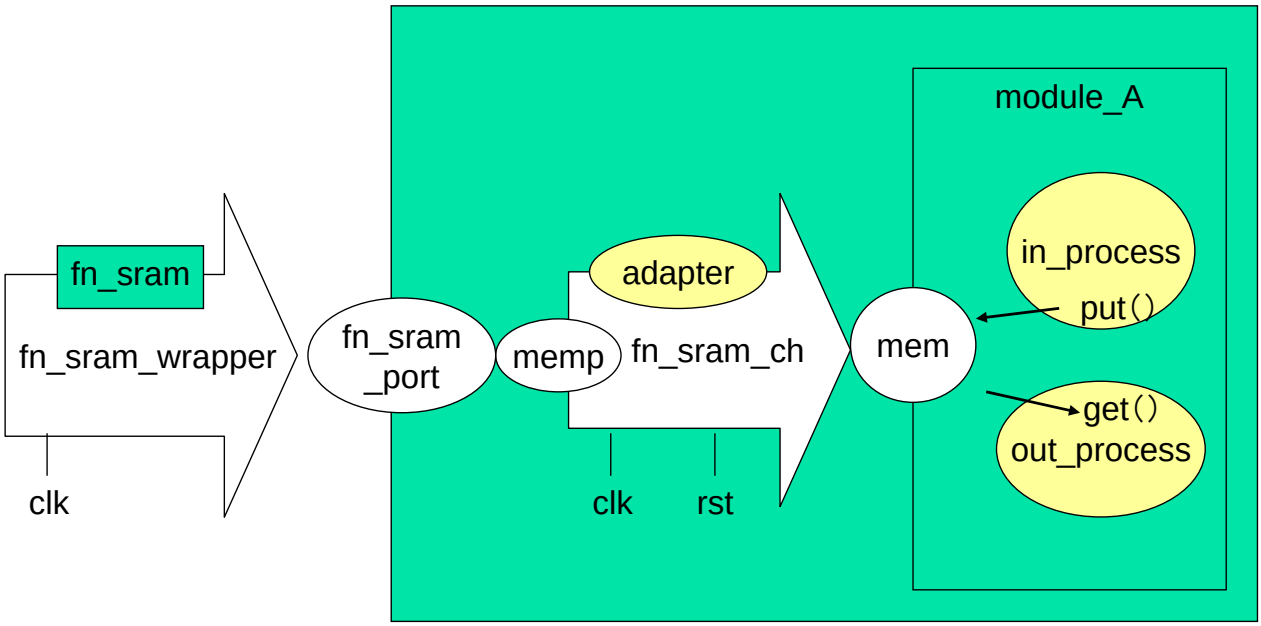
© Copyright 2004-2009 JEITA, All rights reserved

4-5.メモリアクセス(fn_sram_ch)



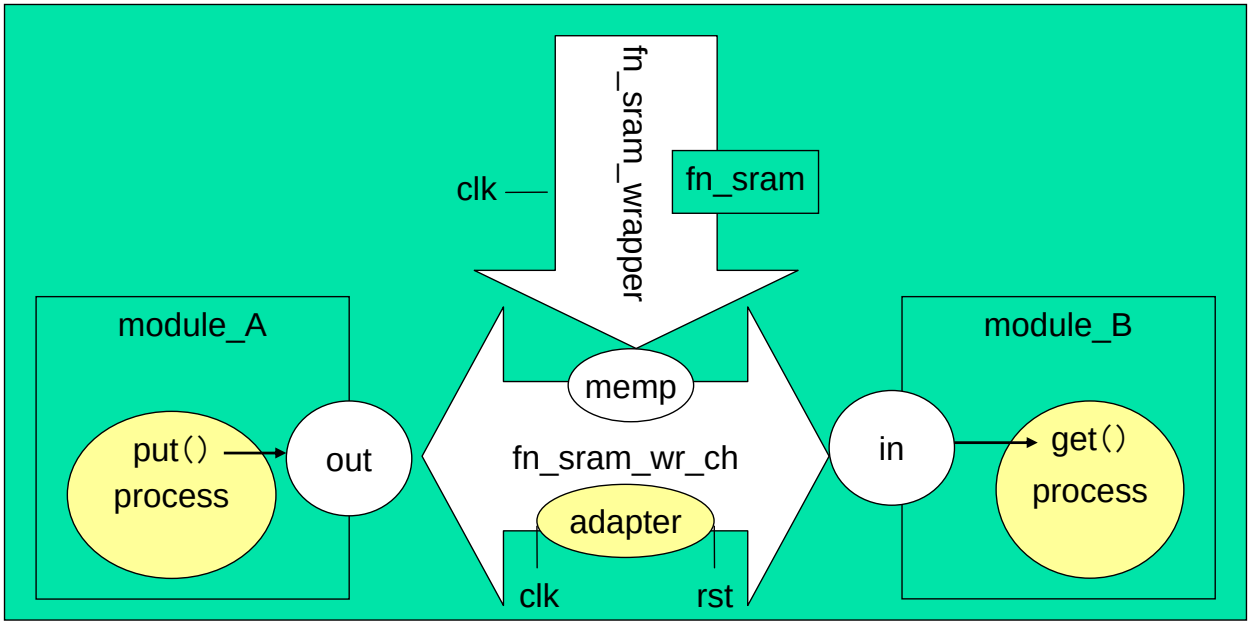
© Copyright 2004-2009 JEITA, All rights reserved

4−5.メモリアクセス(fn_sram_ch)



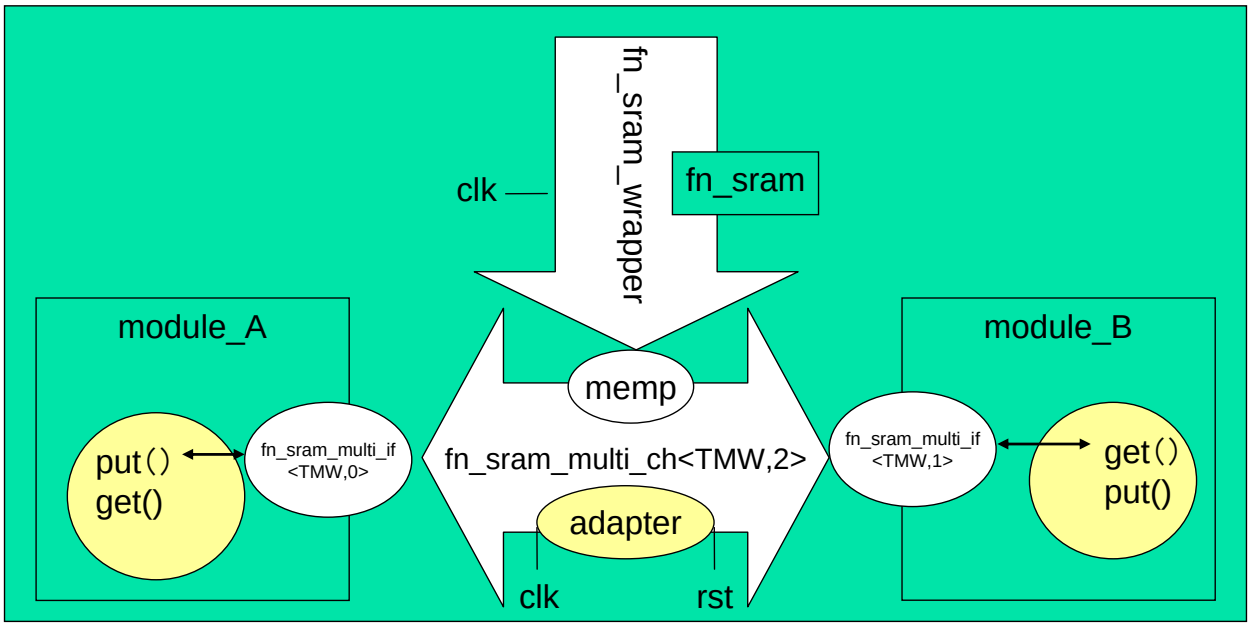
© Copyright 2004-2009 JEITA, All rights reserved

4−5.メモリアクセス(fn_sram_wr_ch)



© Copyright 2004-2009 JEITA, All rights reserved

4−5.メモリアクセス(fn_sram_multi_ch)



© Copyright 2004-2009 JEITA, All rights reserved

4−6.演算系

© Copyright 2004-2009 JEITA, All rights reserved

4-6.演算系ライブラリ

- 次の定数と演算子と関数は必ず用意する。

static const unsigned int raw_bits_len

T(const sc_uint<raw_bits_len>&)

void set_raw_bits(T&, const sc_uint<raw_bits_len>&)

ビット列を与えて値を初期化

sc_uint<raw_bits_len> raw_bits(const T&)

値をビット列として取り出し

// sc_uintを使わずにfn_raw_bits<ram_bits_len>クラスを用意した方が良いが、そうするとクラスの名前まで統一化することになる。

== 等値の評価

<< 外部出力

const std::string to_string()

sc_trace(sc_trace_file*, const T&, const std::string&)

© Copyright 2004-2009 JEITA, All rights reserved

4-6.値のビット列による初期化と取り出し

演算系の値をsc_uint<raw_bits_len>との間で相互変換可能にする。この値は、演算系がその値を示す場合に最低限必要なビット列である。たとえば、浮動小数点であれば、指数と仮数のデータを並べたものである。並べ方はライブラリ依存とする。

データ保存などの演算を行わないデータ処理が共有可能になる。

合成不可能な型（たとえばdouble型など）の定数は、あらかじめビット列に変換しておいて使用する。

© Copyright 2004-2009 JEITA, All rights reserved

4-7.アルゴリズムとアーキテクチャ

© Copyright 2004-2009 JEITA, All rights reserved

4-7.モジュールGCUとFUの雛形

- モジュールGCUとFUの接続は、モジュラーインタフェースgcのみとする。
- モジュールFUは、GCUとの接続gcと必要な数だけのデータ入出力とローカルコントロール入出力がある。これらは、すべてモジュラーインタフェースである。基本的にモジュールFUは、これ以外にクロックポートclkとリセットポートrstだけを持つ。

© Copyright 2004-2009 JEITA, All rights reserved

4-7.共有ヘッダファイル

```
// common.hh
#ifndef _COMMON_HH_
#define _COMMON_HH_ 1
#if SYSTEMC_VERSION <= 20050714  // 2.1.v1
#include <systemc.h>
#else
#include <systemc>
using namespace sc_core;
using namespace sc_dt;
#endif
#include “fn.h”

typedef sc_int<16> data_t;
typedef fn_rdyvld_ch<data_t> data_ch_t;
typedef data_ch_t::in data_in_if_t;
typedef data_ch_t::out data_out_if_t;
#endif
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.GCUのヘッダファイル

```
// gcu.hh
#ifndef _GCU_HH_
#define _GCU_HH_ 1
class gcu : public sc_module
{
public:
  sc_in<bool> clk;
  sc_in<bool> rst;
  // CPU interface のポートを羅列する。
    ...
  gc_ch::gcu gc;
  void processing1();
  SC_HAS_PROCESS(gcu);
  gcu(const sc_module_name& name=“gcu”);
};
#endif
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.GCUのソースファイル

```
// gcu.cc
#include “common.hh”
#include “gc_ch.hh”
#include “gcu.hh”
void gcu::processing1(){
  if(!rst){
    // 初期化処理
  }else{
    // 動作
  }
}

gcu::gcu(const sc_module_name& name):
 sc_module(name),clk(“clk”),rst(“rst”), ..., gc(“gc”){
  SC_METHOD(processing1); // processing1はRTL
  sensitive << clk.pos() << rst.neg();
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.FUのヘッダファイル

```
// fxxx.hh
#ifndef _FXXX_HH_
#define _FXXX_HH_ 1
class fxxx : public sc_module
{
  public:
  sc_in<bool> clk;
  sc_in<bool> rst;
  gc_ch::ful gc;
  lc_ch::out lc;
  data_in_if_t din;
  data_out_if_t dout;
  void processing1();
  SC_HAS_PROCESS(fxxx);
  fxxx(const sc_module_name& name=“fxxx”);
};
#endif
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.FUのソースファイル

```
// fxxx.cc
#include “common.hh”
#include “fxxx.hh”
void fxxx::processing1(){
  // 初期化処理
  wait();
  while(true){
    // 動作
  }
}

fxxx::fxxx(const sc_module_name& name) :
 sc_module(name), clk(“clk”), rst(“rst”),
 gc(“gc”), lc(“lc”), din(“din”), dout(“dout”) {
  SC_CTHREAD(processing1, clk.pos()); // processing1はBL
  reset_signal_is(rst, false);
}
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.モジュラインタフェースgcまたはlc

```
// gc_ch.hh
#ifndef _GC_CH_HH_
#define _GC_CH_HH_ 1
#include “common.hh”
class gc_gcu_port;
class gc_fu1_port;
class gc_fu2_port;
class gc_ch : public sc_object
{
public:
  sc_signal<ttt1> sss1; // tttnは型名で、sssnは信号名で置き換える。
  sc_signal<ttt2> sss2;
  sc_signal<ttt3> sss3;
  typedef gc_gcu_port gcu;
  typedef gc_fu1_port fu1;
  typedef gc_fu2_port fu2;
  friend void sc_trace(sc_trace_file* tf, const gc_ch &ch, const std::string& str){
    sc_trace(tf, ch.sss1, str+”.sss1”);
    sc_trace(tf, ch.sss2, str+”.sss2”);
    sc_trace(tf, ch.sss3, str+”.sss3”);
  }
  gc_ch(const char* name=“gc_ch”):sc_object(name),
  sss1(“sss1”),sss2(“sss2”),sss3(“sss3”){};
};
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.モジュラインタフェースgcまたはlc (続き)

```
class gc_gcu_port : public sc_object
{
public:
  sc_out<ttt1> sss1;
  sc_in<ttt2> sss2;
  sc_in<ttt3> sss3;
  void bind(gc_ch &ch){
    sss1(ch.sss1);
    sss2(ch.sss2);
    sss3(ch.sss3);
  }
  void operator ()(original_gc_ch &ch){ bind(ch); }
  gc_gcu_port(const char* name=“gc_gcu_port”) :
   sc_object(name),
    sss1(“sss1”), sss2(“sss2”), sss3(“sss3”){};
};
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.モジュラインタフェースgcまたはlc (続き)

```
class gc_ful_port : public sc_object
{
public:
  sc_in<ttt1> sss1;
  sc_out<ttt2> sss2;
  void bind(gc_ch &ch){
    sss1(ch.sss1);
    sss2(ch.sss2);
  }
  void operator ()(gc_ch &ch){ bind(ch); }
  gc_ful_port(const char* name=“gc_ful_port”) :
    sc_object(name),
    sss1(“sss1”), sss2(“sss2”){};
};
```

© Copyright 2004-2009 JEITA, All rights reserved

4-7.モジュラインタフェースgcまたはlc (続き)

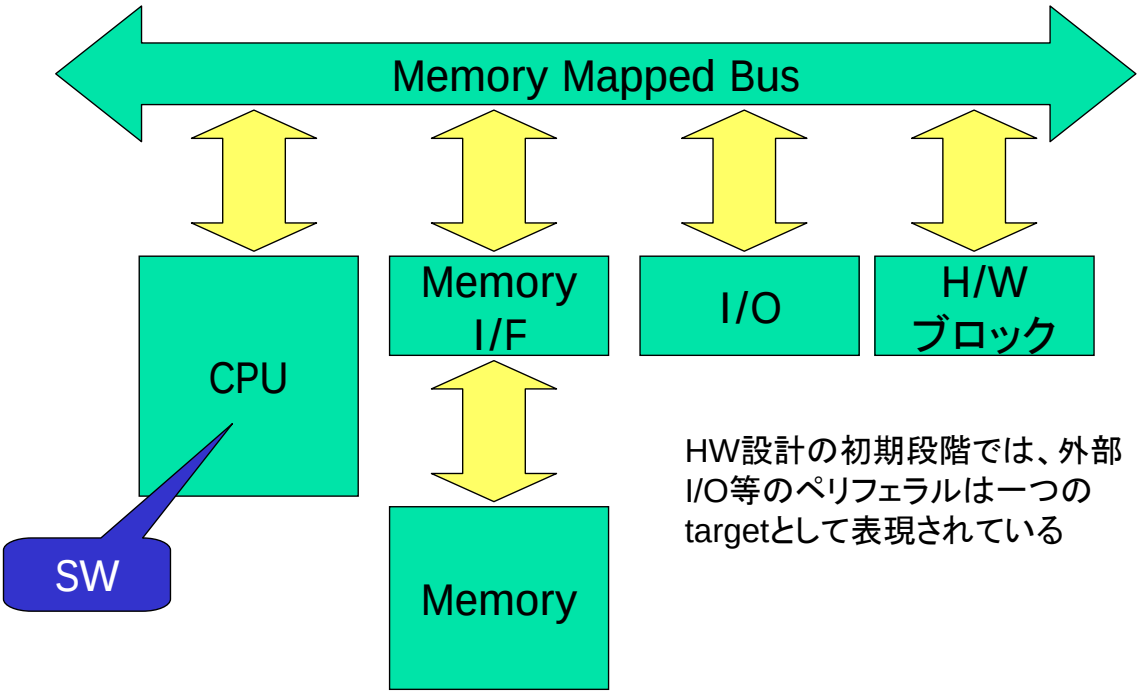
```
class gc_fu2_port : public sc_object
{
public:
  sc_in<ttt1> sss1;
  sc_out<ttt3> sss3;
  void bind(gc_ch &ch){
    sss1(ch.sss1);
    sss3(ch.sss3);
  }
  void operator ()(gc_ch &ch){ bind(ch); }
  gc_fu2_port(const char* name=“gc_fu2_port”) :
    sc_object(name),
    sss1(“sss1”), sss3(“sss3”){};
};
#endif
```

© Copyright 2004-2009 JEITA, All rights reserved

4-8.その他のHW設計について

© Copyright 2004-2009 JEITA, All rights reserved

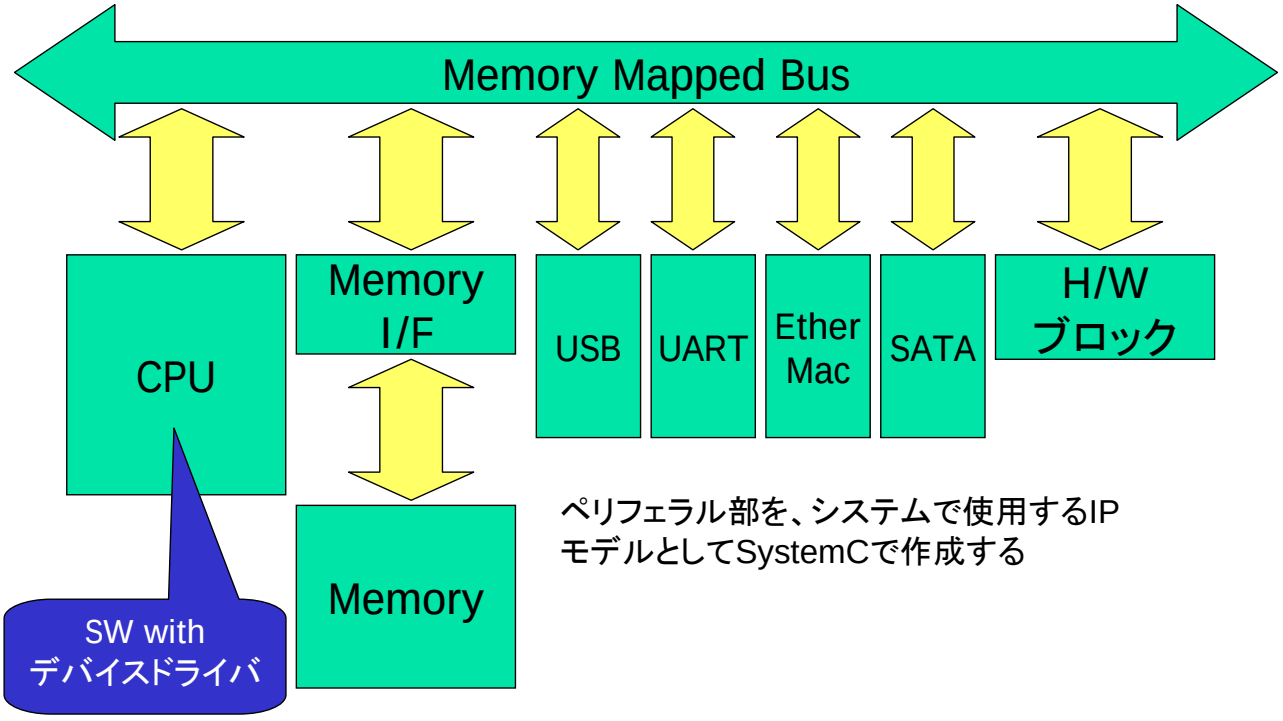
4-8.HW/SW分割後のシステム



HW設計の初期段階では、外部I/O等のペリフェラルは一つのtargetとして表現されている

© Copyright 2004-2009 JEITA, All rights reserved

4-8.HWプラットフォームの詳細化



ペリフェラル部を、システムで使用するIPモデルとしてSystemCで作成する

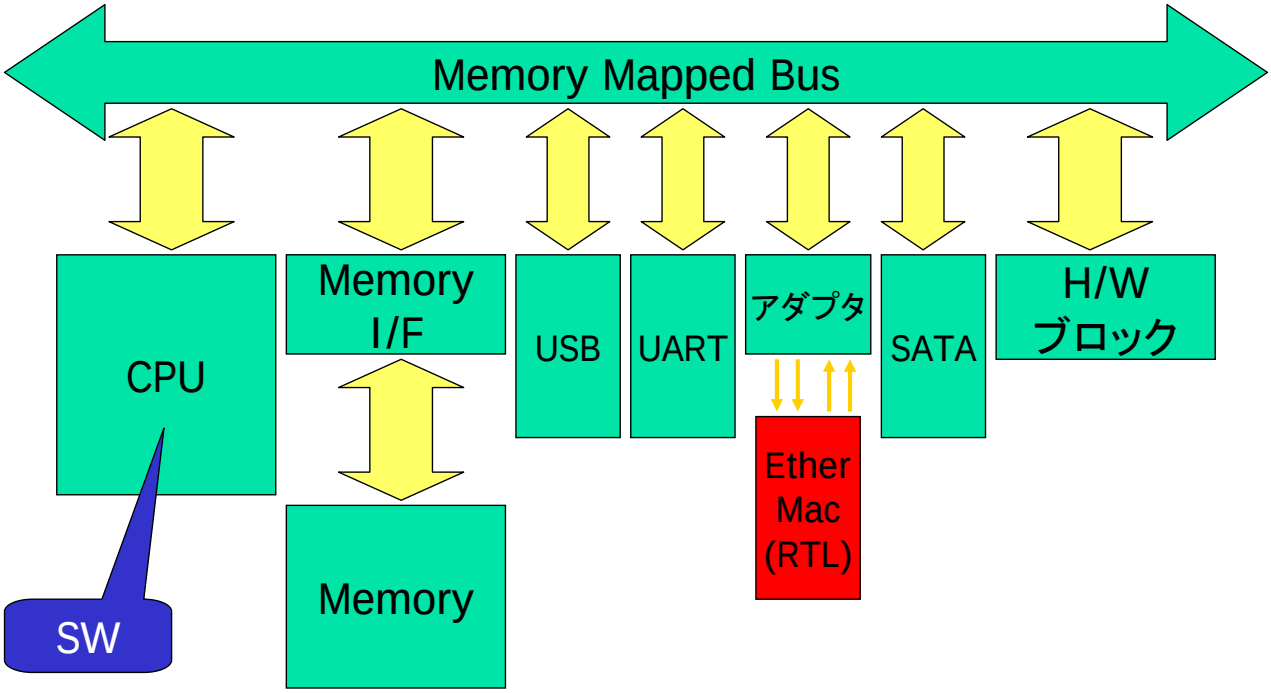
© Copyright 2004-2009 JEITA, All rights reserved

4-8.HWプラットフォームの詳細化

- 外部I/O等のペリフェラルは、HW設計初期段階では、一つのTargetモジュールとして表現されている
- HWを詳細化を進めるにあたり、一つのTargetモジュールを、具体的なインタフェースIPのモデルに置き換えていく
  - 各IPのアドレス空間（レジスタ割付）を行う
  - 想定されるレイテンシの設定を設定する
  - （必要であれば）IP内部でのデータ処理を実装する
- これらのIPモデルは主に低レベルSW（デバイスドライバ開発）用のプラットフォームに用いられる
- これらのIPモデルは、最終的に等価なRTLのIPと置き換えることを想定している
  - 動作合成の対象外になる

© Copyright 2004-2009 JEITA, All rights reserved

4-8.ペリフェラルのRTL検証環境



© Copyright 2004-2009 JEITA, All rights reserved

4-8.ペリフェラルのRTL検証環境

- 動作合成により作られるブロック、及びペリフェラルIP部の検証環境として、詳細化されたHWプラットフォームを流用できる
- RTLを検証する場合、HWプラットフォームとRTLを接続する為のアダプタを用意する
- またSystemCで作成した各IPモデルは、RTL単体検証時のリファレンスモデルとしても流用できる
  - その場合はモデルの詳細な機能実装が必要

© Copyright 2004-2009 JEITA, All rights reserved

5.検証

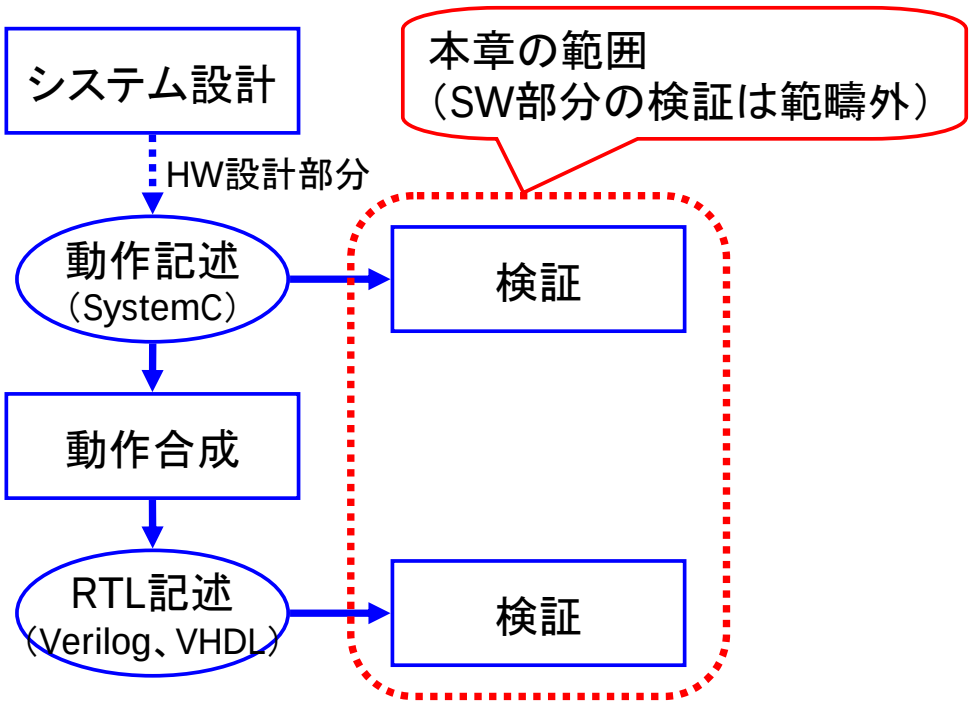
© Copyright 2004-2009 JEITA, All rights reserved

5-1. 内容

1. 内容
2. 本章の位置付け
3. 検証対象
4. 検証内容
5. 検証方法1
6. 検証方法2
7. 今後の検討事項・課題

© Copyright 2004-2009 JEITA, All rights reserved

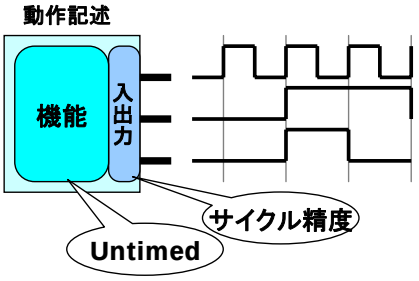
5-2. 本章の位置付け



© Copyright 2004-2009 JEITA, All rights reserved

5-3. 検証対象

- 新規にHW設計する部分
- HW設計の手法としては、動作合成利用を想定
  - 動作記述（SystemC）
    - 通信はサイクル精度
    - 機能はUntimed
  - RTL記述（Verilog、VHDL）
    - 動作合成で合成したもの



© Copyright 2004-2009 JEITA, All rights reserved

5-4. 検証内容

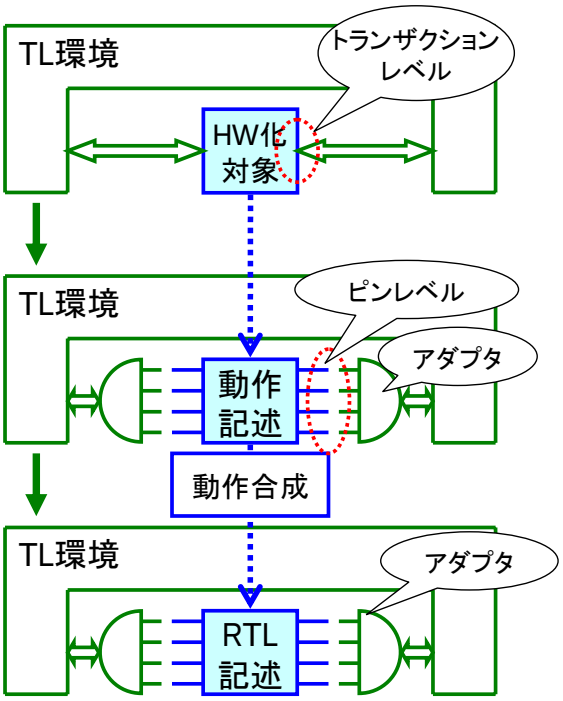
- 動作記述を十分に検証し、RTLでのみ検証可能な項目のみRTL記述を検証する。
  - RTLでは動作確認が基本。

検証内容	動作記述	RTL記述（追加検証）	
インタフェース	○	○	パイプライン・ストール等動作記述で表現できないもの
メモリアクセス	○		
演算系（ユーザ定義型）	○		
アルゴリズム	○		
階層構造	○		
性能制約		○	レイテンシ、スループット

© Copyright 2004-2009 JEITA, All rights reserved

5-5. 検証方法

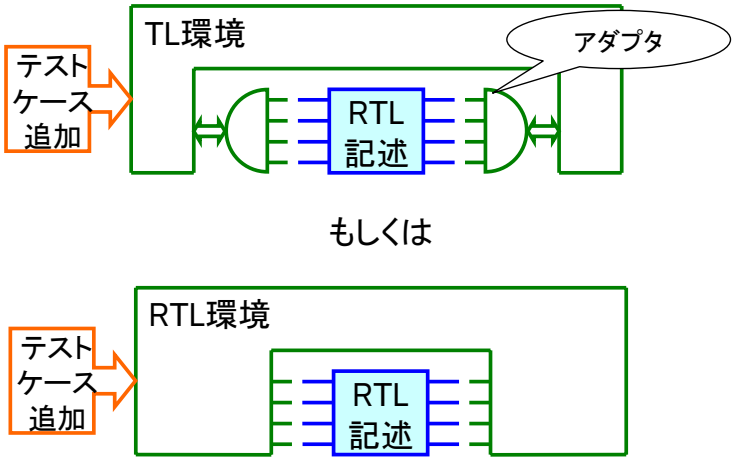
- シミュレーション
  - 動作記述
    - SystemCシミュレーション
  - RTL記述
    - SystemC + Verilog/VHDL 混在シミュレーション
- テスト環境の再利用
  - TL環境の利用
  - ファームウェアの利用



© Copyright 2004-2009 JEITA, All rights reserved

5-6. 検証方法

- RTL記述の検証
  - 抽象度の詳細化に伴いテストケース追加



© Copyright 2004-2009 JEITA, All rights reserved

5-7. 今後の検討事項・課題

- 複数モジュールに分割実装後のテスト
  - 動作合成利用時はアルゴリズム記述を複数モジュールに分割するケースが多い。

  ⇒ モジュール毎のテストケースと期待値の準備に手間が掛る
- 検証の品質
  - RTL検証での品質指標の一つにカバレッジがある。

  ⇒ 高位設計でも良い指標となるか?
- 等価性検証
  - RTL 対 ゲートレベル では実用化

  ⇒ SystemC 対 RTL では適用範囲がまだ狭い

© Copyright 2004-2009 JEITA, All rights reserved

6.モデリング(高速化)

© Copyright 2004-2009 JEITA, All rights reserved

6-1. モデリング(高速化)のポイント

- 必要となる(実装すべき)機能の明確化
  - 機能検証？性能検証？動作合成？
- 適切なモデル抽象度
  - 時間概念必要？ サイクル精度必要？
- I/Fのトランザクション化
  - OSCI TLM2.0の活用
- イベントドリブン(処理のポーリングは行わない)
  - コンテキストスイッチを極力減らすモデリング
- 動的な抽象度の切り替え
  - 観測ポイントに達するまで高速シミュレーション
- 処理が重くなるコーディングを避ける
  - コピーでなくポインタ使用、動的アロケーションを減らす

© Copyright 2004-2009 JEITA, All rights reserved

6-2.必要となる(実装すべき)機能の明確化

- モデルの使われる用途/開発フェーズに応じ実装対象を検討
  - 機能検証用途
    - 時間の概念は組込まない
    - 計算で概略時間を求める
  - 性能検証用途
    - タイミングだけを合わせ込んで、機能の実装を端折る
    - トラフィックジェネレータなど
  - 動作合成用途
    - モデリングの高速化を行う上ではターゲット外

© Copyright 2004-2009 JEITA, All rights reserved

6-3. 適切なモデル抽象度

- 用途に応じ適切な抽象度を選ぶ

（STARCのTLモデリングガイド参照）

  - UTTR (Unitimed + Transaction)
    - 機能モデル設計・検証、通信量・計算量見積り
    - 先行SW機能設計・検証
  - ATTR/ATBP (Approximately-timed + Transaction/Bus Phase)
    - 共有リソース競合検証、HW/SW分割・検証
    - 先行SWタイミング設計・検証
    - プロセッサ必要性能検証、バス・メモリアーキ設計・検証
  - CABC (Cycle-accurate + Bus Cycle)
    - HW機能モデルのサイクル精度レベルでの設計・検証

© Copyright 2004-2009 JEITA, All rights reserved

6-4. I/Fのトランザクション化

- データ転送単位、I/Fの単純化
  - OSCI TLM2.0の活用
    - 2つのコーディングスタイル
      LT(Loosely-timed)では、Blocking APIを使用
      → b_transport
      AT(Approximately-timed)では、Non-blocking APIを使用
      → nb_transport_fw、nb_transport_bw
    - Generic Payload
      バスを介したモデル間で通信を行うトランザクションの共通定義
    - Generic Phase
      BEGIN_REQ、END_REQ、BEGIN_RESP、END_RESPの4フェーズ
    - Temporal Decoupling
      モデルがローカル時間を持ち、グローバル時間に先行して実行する
      グローバル時間への同期回数を減らすことでSim速度を高速化する
    - DMI (direct memory interface)
      メモリへのポインタにダイレクトにアクセスし、Sim速度を高速化する
    - Debug Transaction Interface

© Copyright 2004-2009 JEITA, All rights reserved

6-5. イベントドリブン（処理のポーリングは行わない）

- 無駄にコンテキストスイッチを起こさず、最小限のイベントドリブンに留める（処理のポーリングは行わない）

TLM1.0を使ったクロックベースでのポーリング

```
void Target::get_process() {
  while (1) {
    while (!port->nb_get(req) == false) {
      wait(clk.posedge_event());
    }

    if (req.get_addr() == REG1) {
      ・・・
```

TLM2.0のPEQを利用

```
Target::Target(sc_module_name name)
  : sc_module(name) {
  SC_THREAD(get_process);
  sensitive << m_peq.get_event();
  dont_initialize();
  ・・・
}

void Target::get_process() {
  while (1) {
    ptr = m_peq.get_next_transaction();
    ・・・
    sock->nb_transport_bw(ptr, phase, delay);
    ・・・
```

© Copyright 2004-2009 JEITA, All rights reserved

6-6. 動的な抽象度の切り替え

- OSのブートが完了するまでは抽象度の高い（時間精度の低い）高速シミュレーションを行い、観測したいポイントに達してからは抽象度の低い（時間精度の高い）低速シミュレーションに切り替える
  - TLM2.0では、シミュレーション中に動的に抽象レベルの切り替えが可能な枠組みを提供

LT（高速・抽象レベル高・時間精度低）

AT（低速・抽象レベル低・時間精度高）

© Copyright 2004-2009 JEITA, All rights reserved

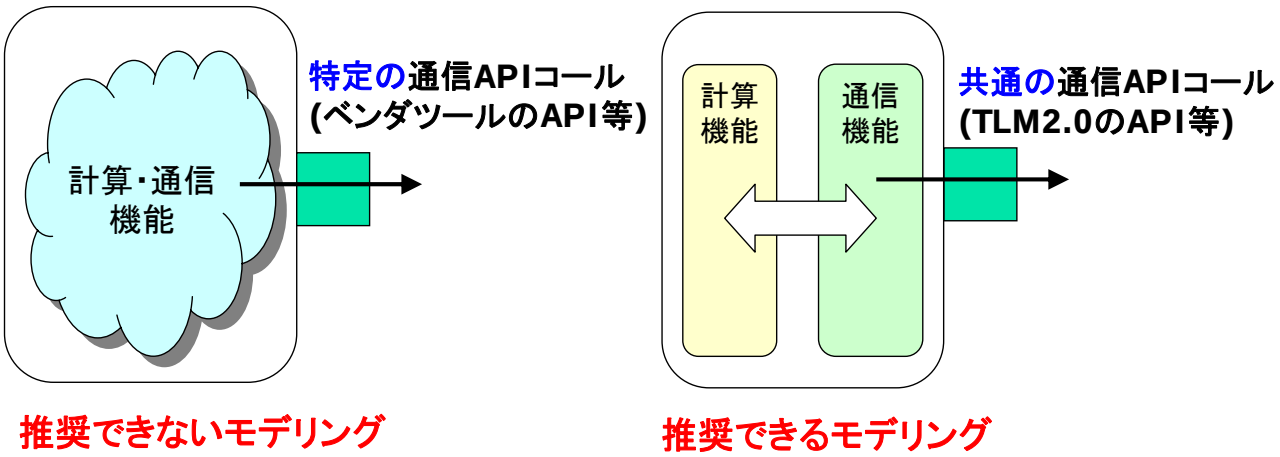
6-7. 処理が重くなるコーディングを避ける

- 処理速度の高速化ノウハウ
  - sc_intなどのSystemC型は使わずに、int型などのANSI C整数型を使用
  - データコピーをせず、ポインタ・参照渡しを用いる
  - メモリ領域の動的な生成・解放を減らす
  - オペレータのオーバーロードを用いない
  - 処理の重い条件式は後置にする
- コンパイル速度の高速化ノウハウ
  - ヘッダファイルのインクルードを途中で切る
  - テンプレートを多用しない

© Copyright 2004-2009 JEITA, All rights reserved

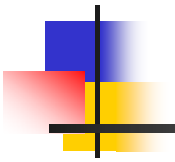
6-8. より良いモデリングを目指して

- 再利用性向上 (STARCのTLモデリングガイドより)
  - 通信インタフェースの共通化
  - 計算機能と通信機能を明確に分離したモデル構成



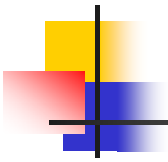
推奨できないモデリング　　推奨できるモデリング

© Copyright 2004-2009 JEITA, All rights reserved



7. その他

© Copyright 2004-2009 JEITA, All rights reserved



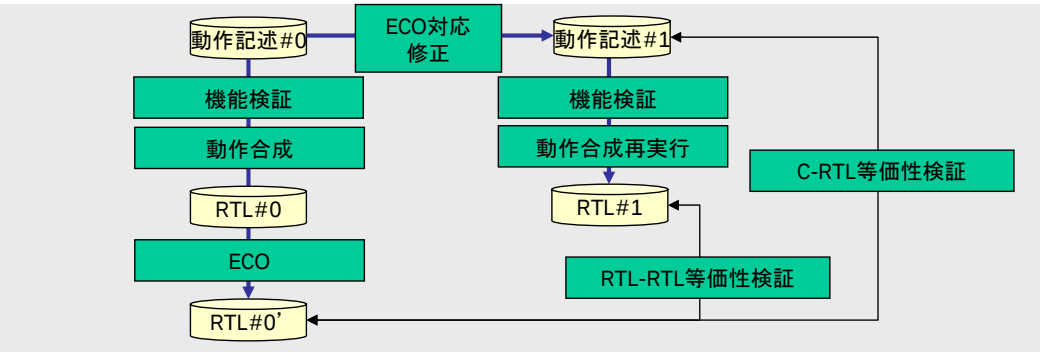
7-1.概要

- ここでは、主に設計工程にまたがって考慮するべきトピックス、他のカテゴリに当てはまらないトピックスを集めた。
- 具体的には以下について説明を行っている。
  - ECO
  - 配線性の考慮
  - フォルスパス

© Copyright 2004-2009 JEITA, All rights reserved

7-2.ECO

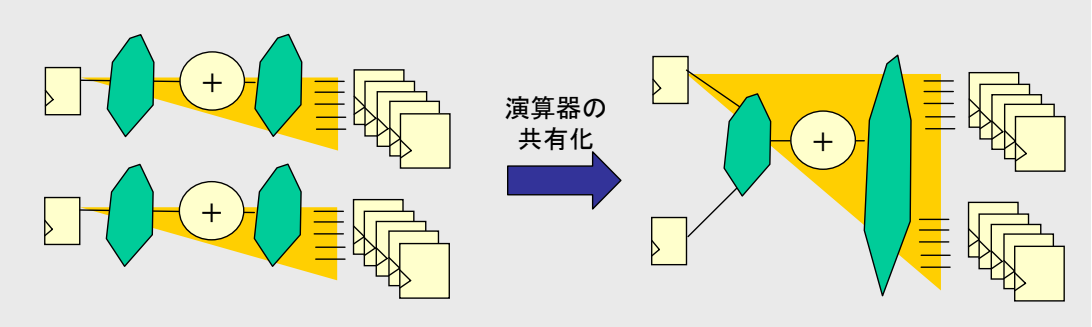
- 動作合成を前提としたECOの課題
  - RTLの可読性が低く、人手修正が困難。
  - 動作記述とRTLの対応が1対1ではない。
  - 動作記述の修正が微少であっても生成されるRTLの差分が大きい。
- 対応
  - マイナーなタイミング修正を除いては、基本的にECOは、動作記述の修正で行う。
    → 問題を把握しやすく、修正ミスを防ぐことができる。
    → 動作記述を設計資産とする。
  - 等価性検証を使用する場合は、検証を行ったコードをリファレンスとする。



© Copyright 2004-2009 JEITA, All rights reserved

7-3.配線性の考慮

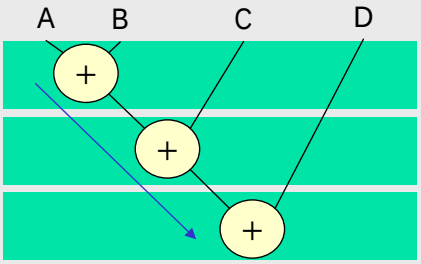
- 動作合成を前提とした配線性の課題
  - 自動リソースシェアリングの結果、ロジックコーンが巨大化し、レイアウト時に問題になる場合がある。
- 対応
  - 動作合成ツールのレポートや、RTLチェックツール等により、ロジックコーンの大きさを確認しておく。
  - 問題がある場合は、動作合成実行時に制約を指定するか、コードの修正を行う。

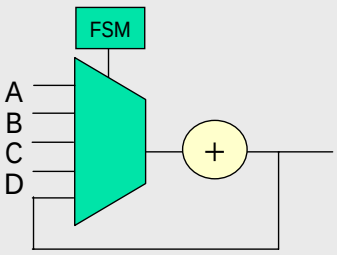


© Copyright 2004-2009 JEITA, All rights reserved

7-4.フォルスパス

- 動作合成を前提とした設計おけるフォルスパスの課題
  - 動作合成ツールのリソースシェアリングの結果、False Pathが発生する場合があり、後工程で問題になる場合がある。
- 対応
  - 要求性能を満たす範囲で、フォルスパスが発生しないよう、動作合成ツールに対する制約を設定する。
  - フォルスパス・スクリプトを利用する場合は、別途妥当性をチェックするためのツールを利用し、確認を行うこと。





演算器の共有によるFalse Pathの例

© Copyright 2004-2009 JEITA, All rights reserved

4.2.6 SystemCユーザフォーラム2009 開催報告

© Copyright 2009 JEITA, All rights reserved JEITA

SystemCユーザフォーラム2009の概要

- 主催：JEITA EDA技術専門委員会
- 協賛：OSCI
- 日時：2009年1月23日　10:00～12:00
- 会場：パシフィコ横浜アネックスホール (定員200名)
- 参加者数：123名(申し込み者数：127名)
- 講演内容：
  - 司会
    - 長谷川 隆 氏 (SystemC-WG主査/富士通マイクロエレクトロニクス(株))
  - SystemCコミュニティの最新状況
    - Stanley.J.Krolikoski 氏 (OSCI)
  - SystemC推奨設計メソドロジの紹介
    - SystemCワーキング・グループ
  - STARC TLモデリングガイドの紹介
    - 吉永 和弘 氏((株)半導体理工学研究センター)
  - 事例#1 画像処理IP開発への高位設計適用事例
    - 浅野 哲也 氏 ((株)ルネサス テクノロジ)
  - 事例#2 TLM2.0を利用した大規模回路設計
    - 長谷 川裕恭 氏 ((株)エッチ・ディー・ラボ)

© Copyright 2009 JEITA, All rights reserved JEITA

SystemCユーザフォーラム2009開催の目的

- 本年度のSystemCユーザフォーラムは、SystemCの普及を目的として以下の方針の下で実施した。
  - SystemCユーザへのOSCI活動状況を報告する事。
  - JEITA SystemC WG活動内容を紹介し、活動実績を発表する事。
  - SystemCに関する国内の活動、適用事例を紹介し、ユーザへ普及の現状をアピールする事。

© Copyright 2009 JEITA, All rights reserved

SystemCユーザフォーラム2009サマリー

- 講演について
  - 2時間で5講演を実施。OSCI、JEITA SystemC WG、及びSTARCによる報告、SystemC適用事例2件と充実した内容であったが、時間的にはやや不足気味であった。
  - 質疑応答も活発に行われた(7件)。

- 講演内容:
  - OSCIからは標準化活動の状況について説明がなされた。(20分)。
  - JEITA SystemCからは、SystemC推奨設計メソドロジの概要の紹介、TLM 2.0に関する補足説明を行った(20分)。
  - STARCの吉永氏からは、TLモデリングの概要とその必要性、そして公開されたばかりのTLモデリングガイドライン第2版の紹介があった(20分)。
  - ユーザ事例としては、ルネサステクノロジの浅野氏から、SystemC記述を用いた高位設計環境の紹介があった。(30分)。
  - 同じくユーザ事例としてエッチ・ディー・ラボの長谷川氏からは、同社で実際に行われている SystemCを用いた大規模回路設計について紹介された。(30分)。

© Copyright 2009 JEITA, All rights reserved

SystemCユーザフォーラム2009サマリー

- 会場状況・予稿集について
  - 本年のフォーラム参加人数は昨年の172名から123名と大幅に減少し、目標人数150名を達成できなかった。ただし、厳しい経済状況を考慮すれば、それでも十分に高い数字であり、SystemCへの興味が依然高いことを表していると考えられる。
  - 参加者減少の他の要因としては、昨年度はTLM2.0の発表直後に開発担当者がチュートリアルを行うという、タイムリーな内容であったのに対し、今年度は大きなイベントもなく、アップデート的な内容、事例紹介が中心だった点が挙げられる。
  - 全体の満足度(満足＋やや満足)は昨年よりは低い結果となったが、満足だけみると増加している。参加人数が少ない分、はっきりした目的を持った参加者が多かったのかもしれない。来年度以降開催の際には、フォーラムの位置づけも見直す必要があるものと思われる。

© Copyright 2009 JEITA, All rights reserved

SystemCユーザフォーラム2009サマリー

- アンケートからのフィードバック
  - ユーザフォーラムについて
    - システムレベル言語の動向を知る貴重な機会としていつも参考にしております
    - ぜひ、来年度も開催してほしいです
  - 運営について
    - 後の予定があるので、終了時間はオーバーしないで欲しい
    - セミナー中にスタッフが動き回るな
    - カメラのフラッシュで目が痛い

© Copyright 2009 JEITA, All rights reserved

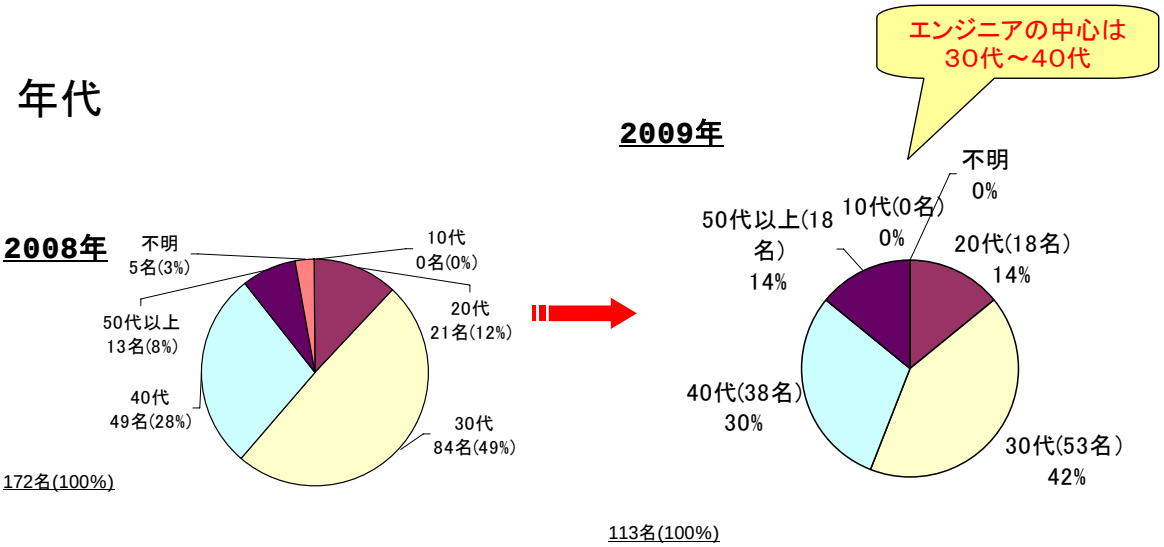
アンケート調査集計結果

- 昨年度とほぼ同様の内容（但し一部新規項目が追加されている）でアンケートを実施し、昨年度までの結果と合わせて聴講者の動向を分析。
  - 今年度はSystemCの利用経験がない方の参加が多かった。
  - 過去のアンケートで、SystemCを利用しない理由としては、「HDLで十分」「言語の完成度が低い」等の回答があったが、今回のアンケートでは、これらが大幅に減少する一方で、「効果が不明」「検討する時間がない」といった理由が多く挙がった。これは、SystemC-WGで開発中の推奨設計メソドロジが貢献できるものと期待している。
  - TLMの標準化が一段落したこともあり、標準化に対する興味よりは、適用事例に期待して参加する方が多かった。
  - 有料開催については、8割の参加者が問題ないとしている(昨年とほぼ同等)。

© Copyright 2009 JEITA, All rights reserved

0. 申込者属性

- 年代

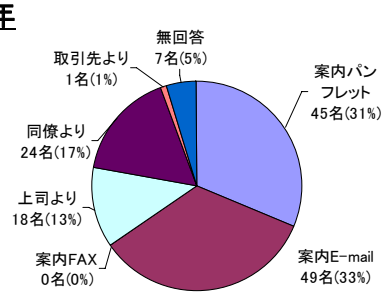


© Copyright 2009 JEITA, All rights reserved

1. 本フォーラムを何で知りましたか？

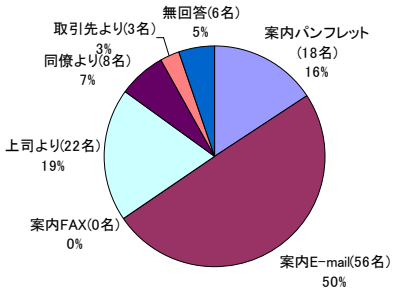
今年度は案内パンフレットが少なかったため、e-mailが増加

2008年



132名(100%)

2009年



113名(100%)

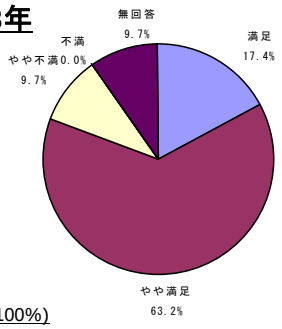
© Copyright 2009 JEITA, All rights reserved

2. 本セッションの内容について(1/2)

満足/やや満足が減少
満足だけみると増加

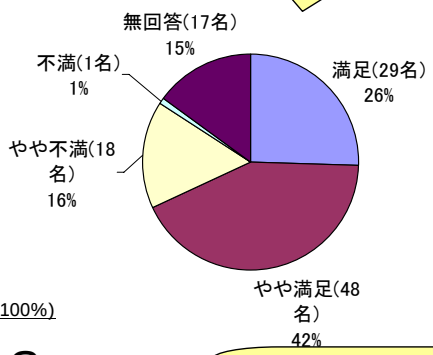
- 内容は期待通りでしたか。

2008年



144名(100%)

2009年

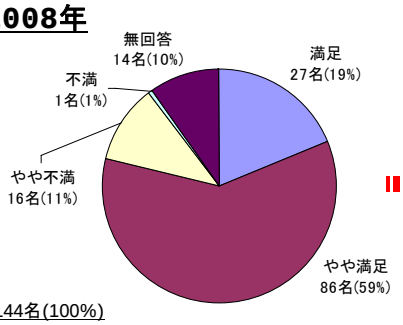


113名(100%)

- 予稿集の内容は期待通りでしたか？

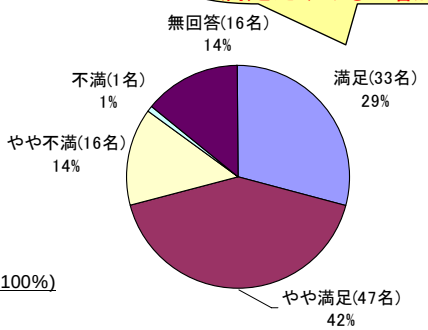
満足/やや満足が減少
満足だけみると増加

2008年



144名(100%)

2009年



113名(100%)

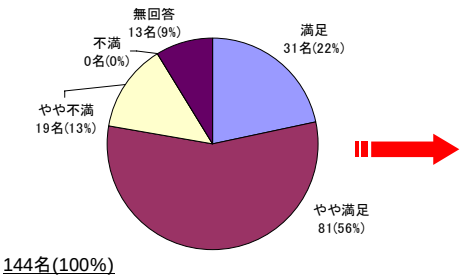
© Copyright 2009 JEITA, All rights reserved

2. 本セッションの内容について(2/2)

チュートリアル中心だった昨年に比べると、今年はUpdate的内容、事例中心であったため、業務への直接的な貢献は減っている。

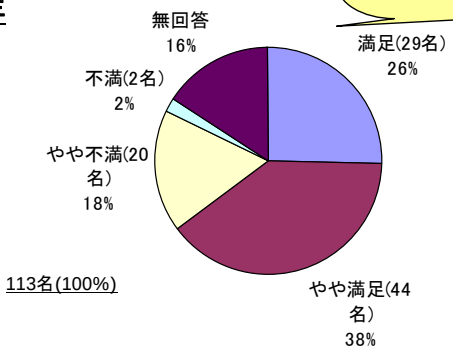
■内容は業務に役立ちますか？

2008年



144名(100%)

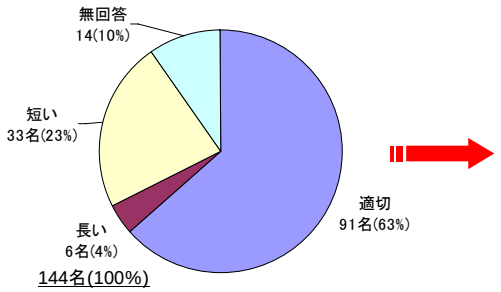
2009年



113名(100%)

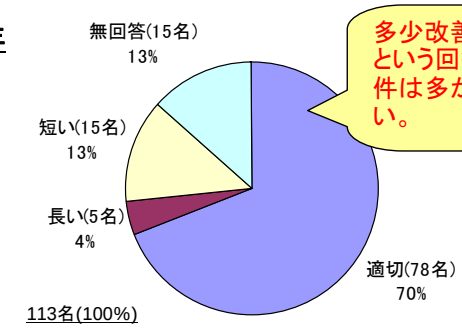
■講演時間

2008年



144名(100%)

2009年



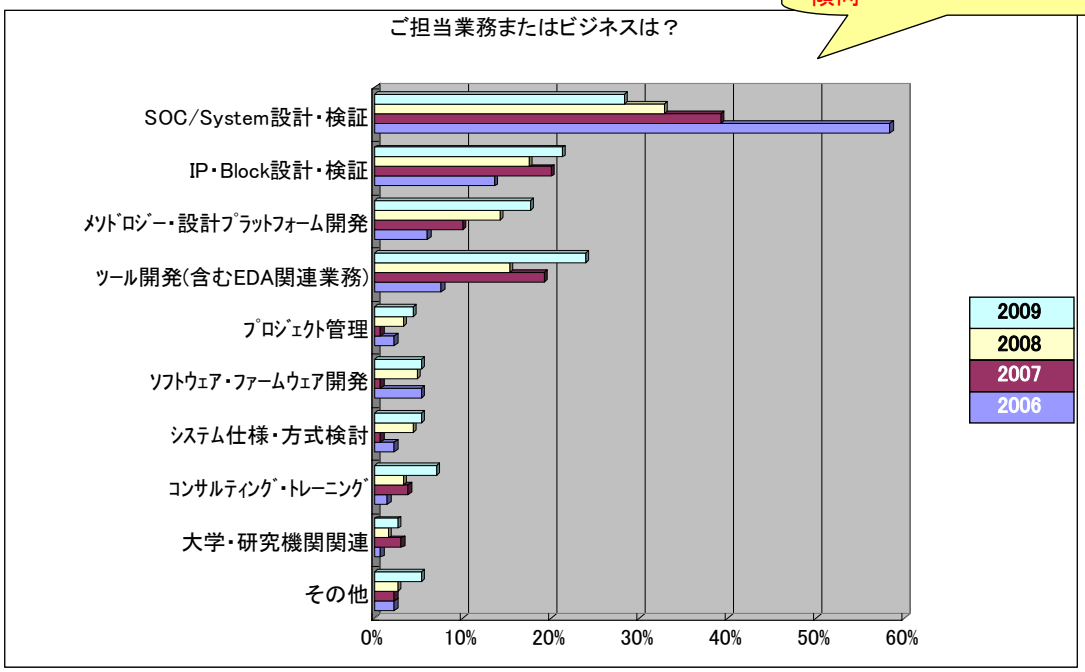
113名(100%)

多少改善されたが、短いという回答が多い。発表5件は多かったかもしれない。

© Copyright 2009 JEITA, All rights reserved

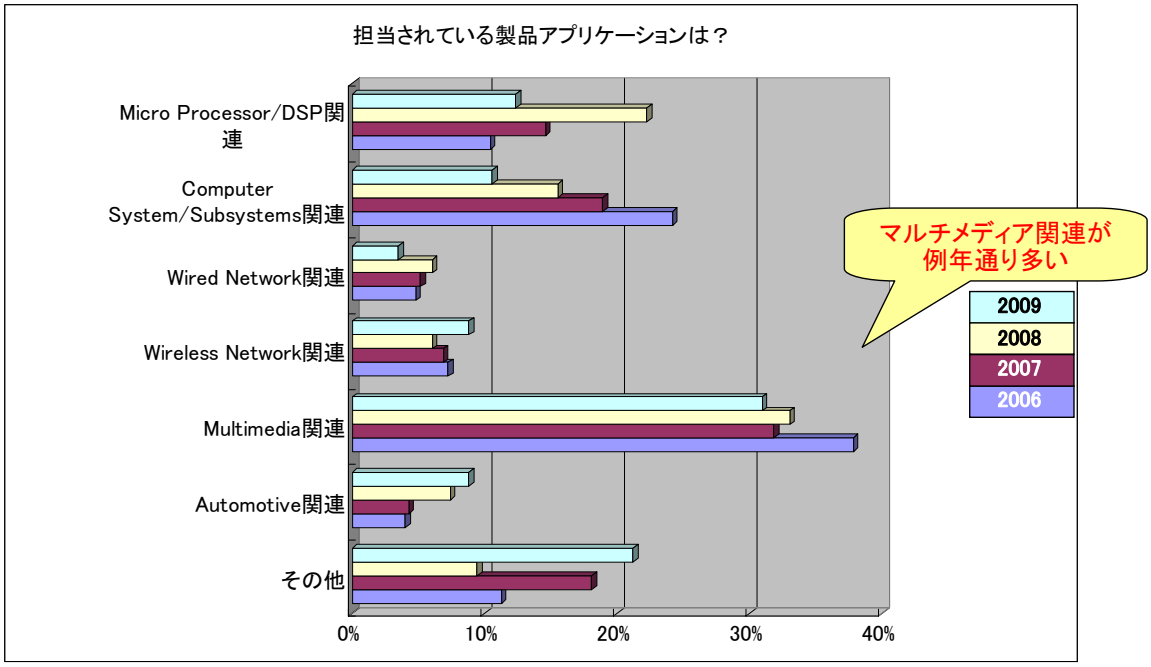
3. ご担当業務またはビジネスは?

SOC/System設計・検証が減少傾向



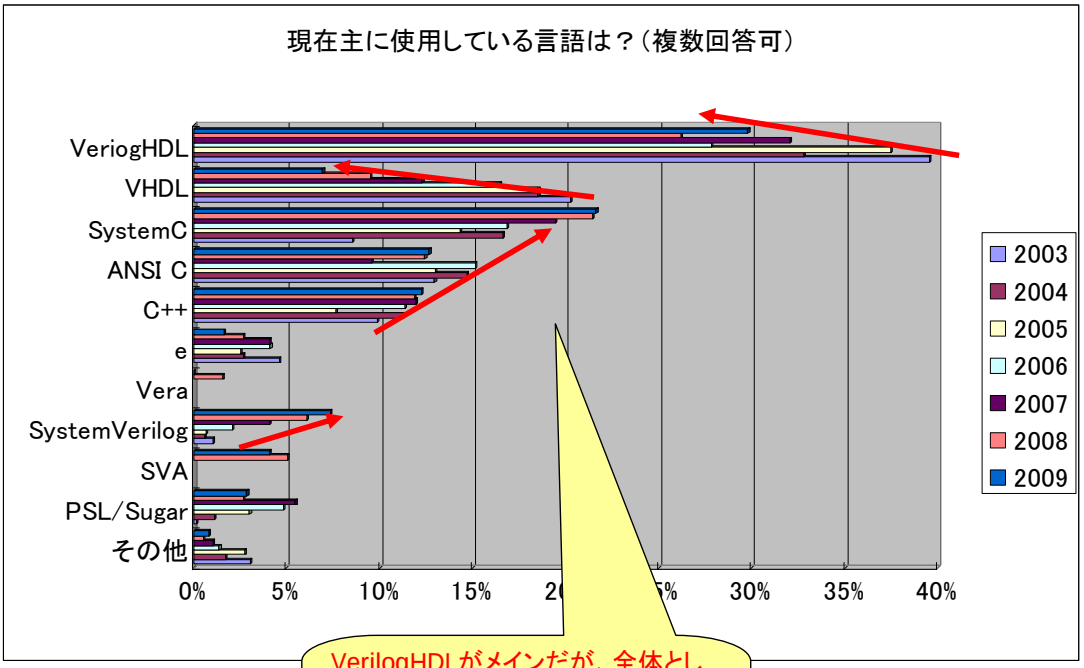
© Copyright 2009 JEITA, All rights reserved

4. 担当されている製品アプリケーションは?



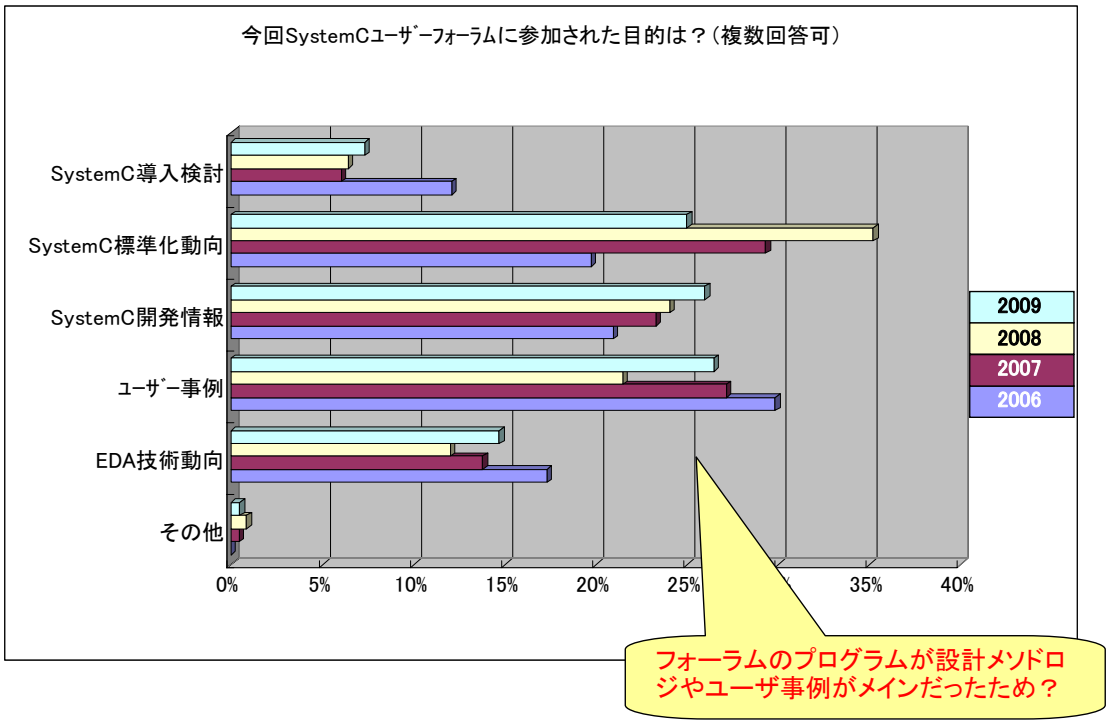
© Copyright 2009 JEITA, All rights reserved

5. 現在主に使用している言語は?



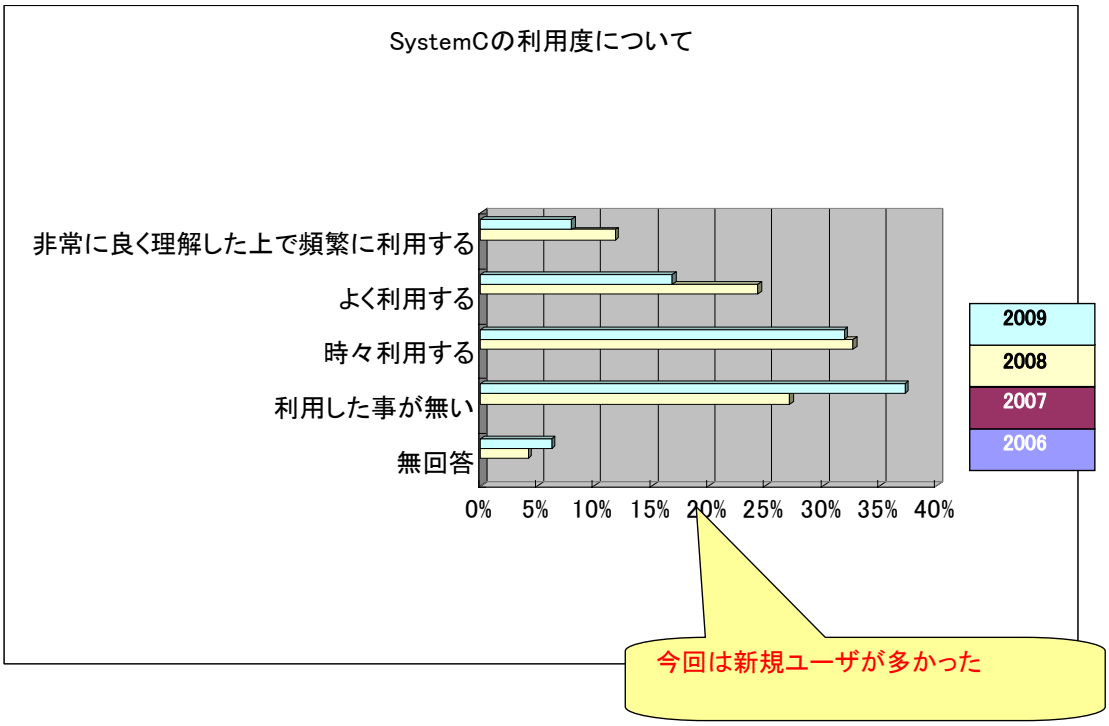
© Copyright 2009 JEITA, All rights reserved

6. SystemCユーザフォーラムに参加された目的は?



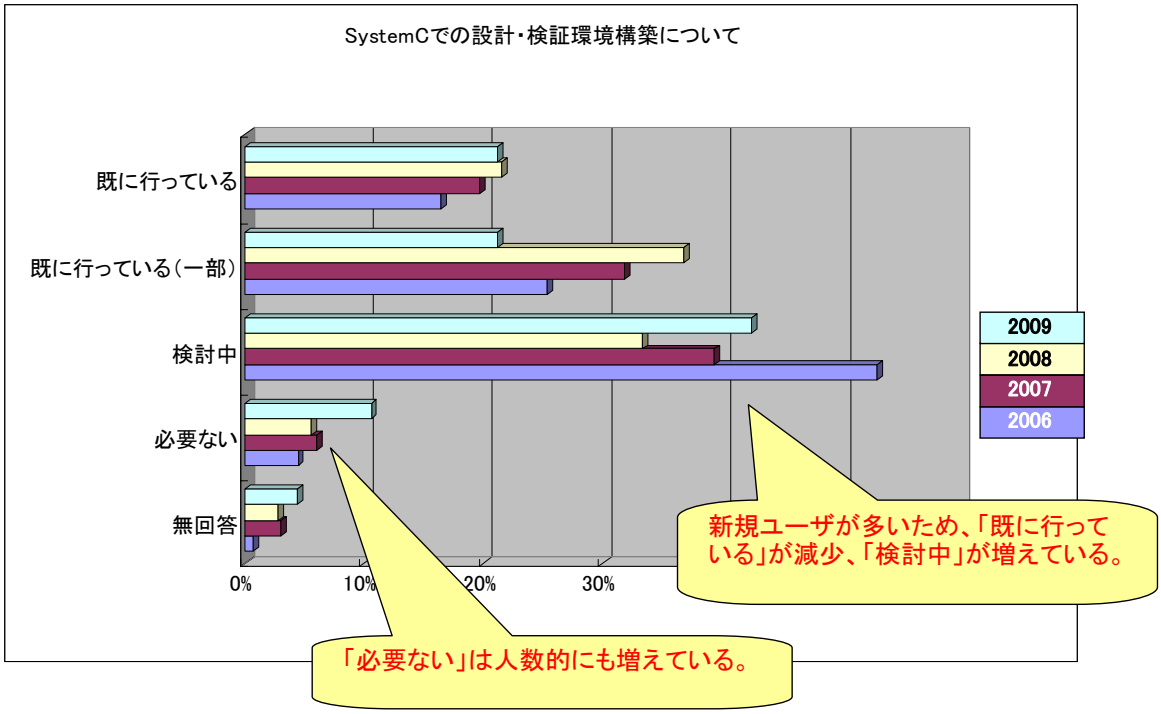
© Copyright 2009 JEITA, All rights reserved
JEITA

7. SystemCの利用度について



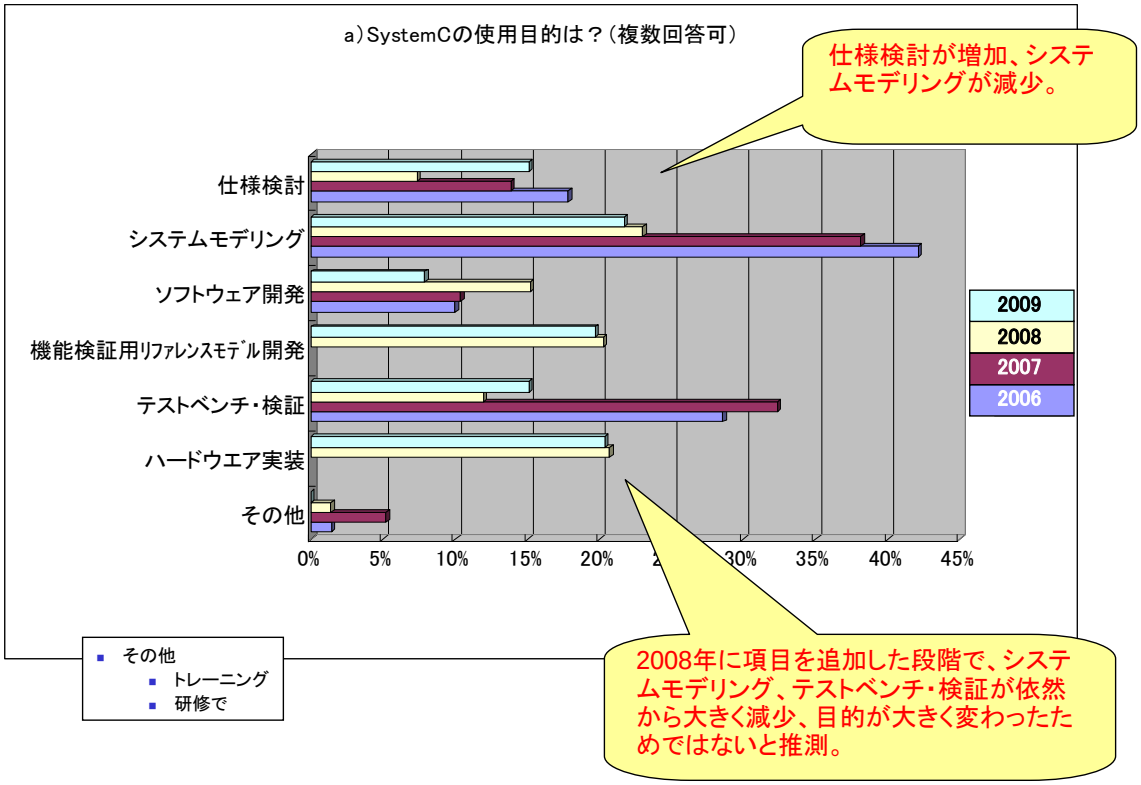
© Copyright 2009 JEITA, All rights reserved
JEITA

8. SystemCでの設計･検証環境構築について



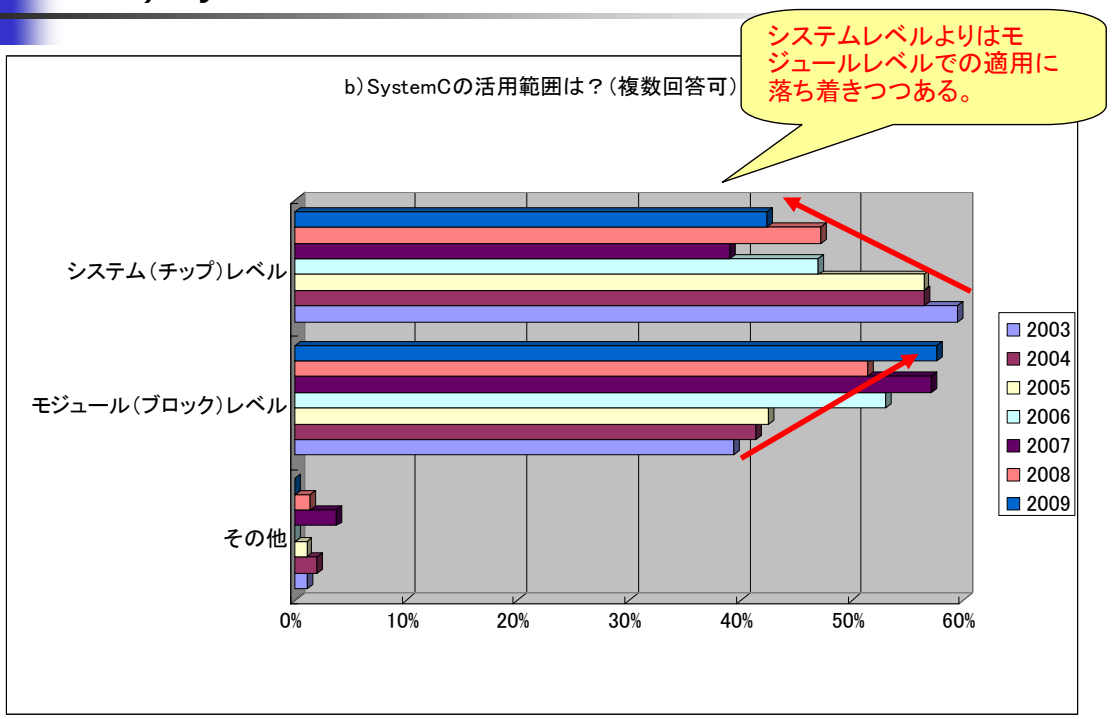
© Copyright 2009 JEITA, All rights reserved

9.「8」で「既に行っている」または「検討中」と回答された方 a) SystemCの使用目的は?



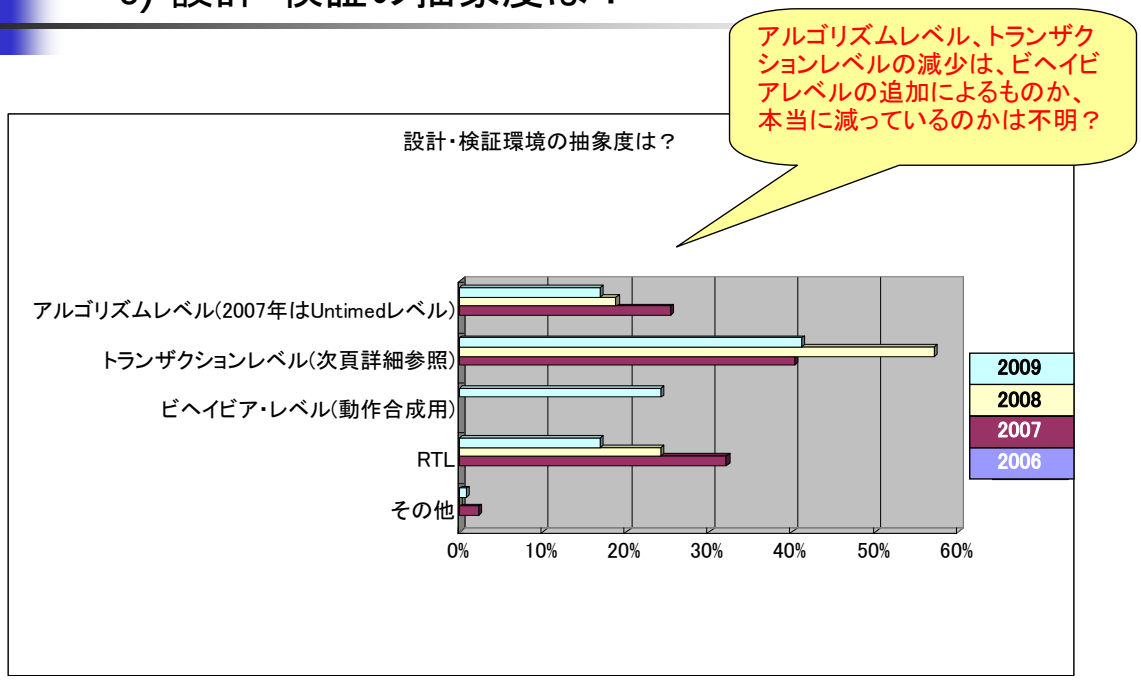
© Copyright 2009 JEITA, All rights reserved

9.「8」で「既に行っている」または「検討中」と回答された方
b) SystemCの活用範囲は?



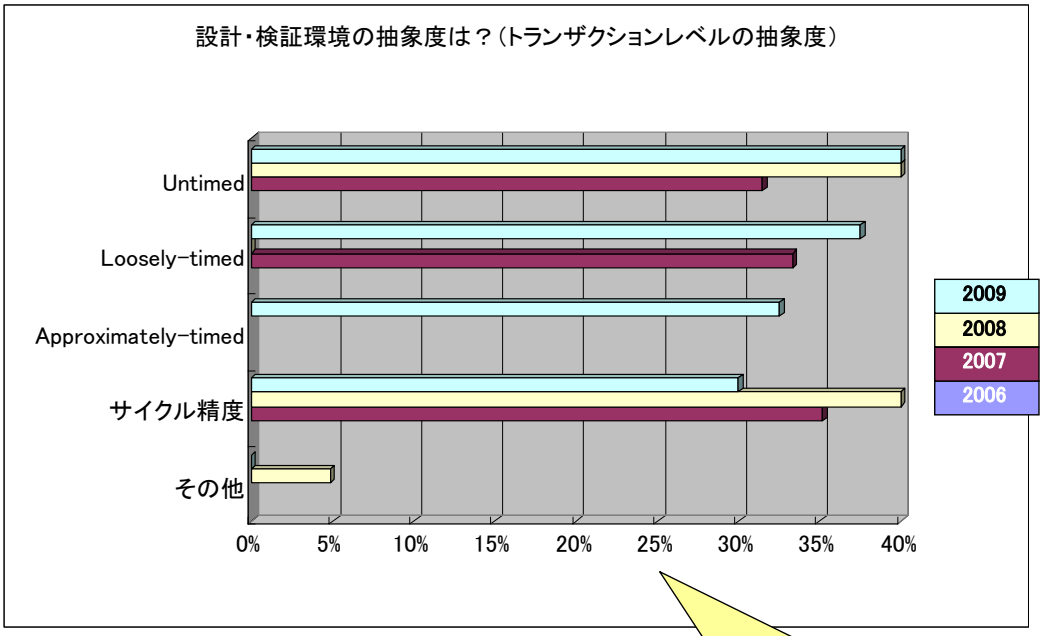
© Copyright 2009 JEITA, All rights reserved

9.「8」で「既に行っている」または「検討中」と回答された方
c) 設計・検証の抽象度は？



© Copyright 2009 JEITA, All rights reserved

9.「8」で「既に行っている」または「検討中」と回答された方
c) 設計・検証の抽象度は(トランザクションレベルの詳細)？

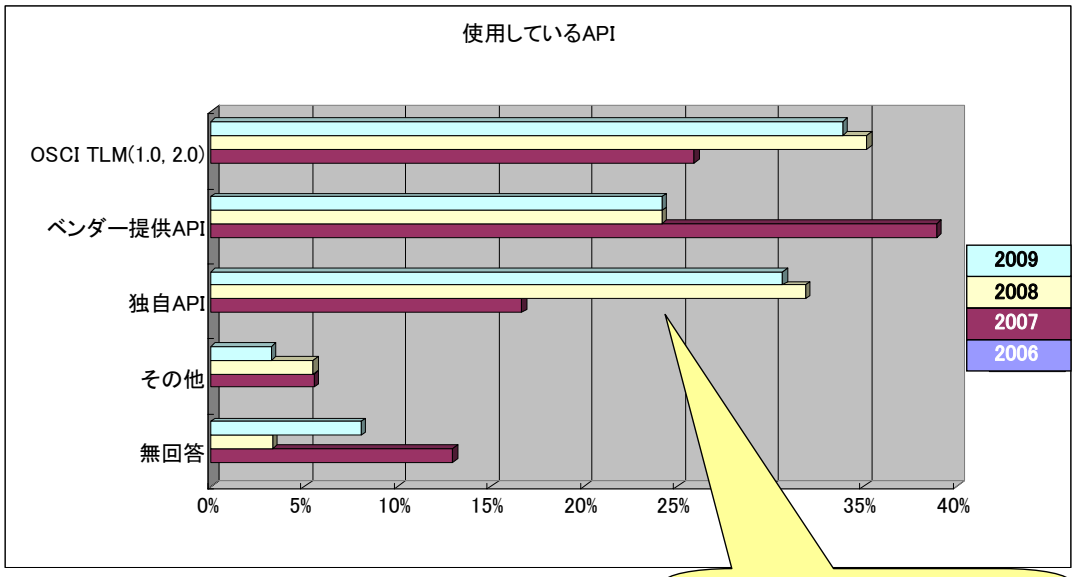


- 2007年度はPV(Untimed)/PVT(Loosely Timed)/CA(サイクル精度)の分類
- 2008年度はUntimed/サイクル精度/その他の分類

メインは抽象度の高いレベルであるが、サイクル精度も多い。

© Copyright 2009 JEITA, All rights reserved

9.「8」で「既に行っている」または「検討中」と回答された方
d) どのようなトランザクションAPIを使用していますか？



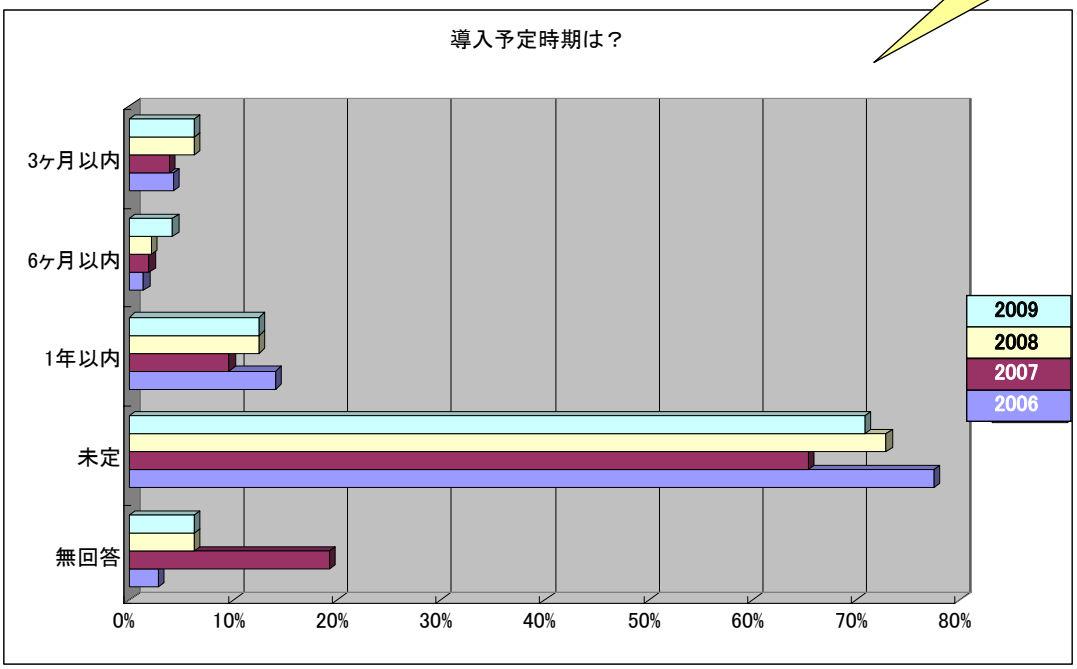
- その他
  - SystemCをツール間のインターフェースとして使用C←→Verilog
  - 検討のみ、使用していない

独自APIが強いのはなぜ？

© Copyright 2009 JEITA, All rights reserved

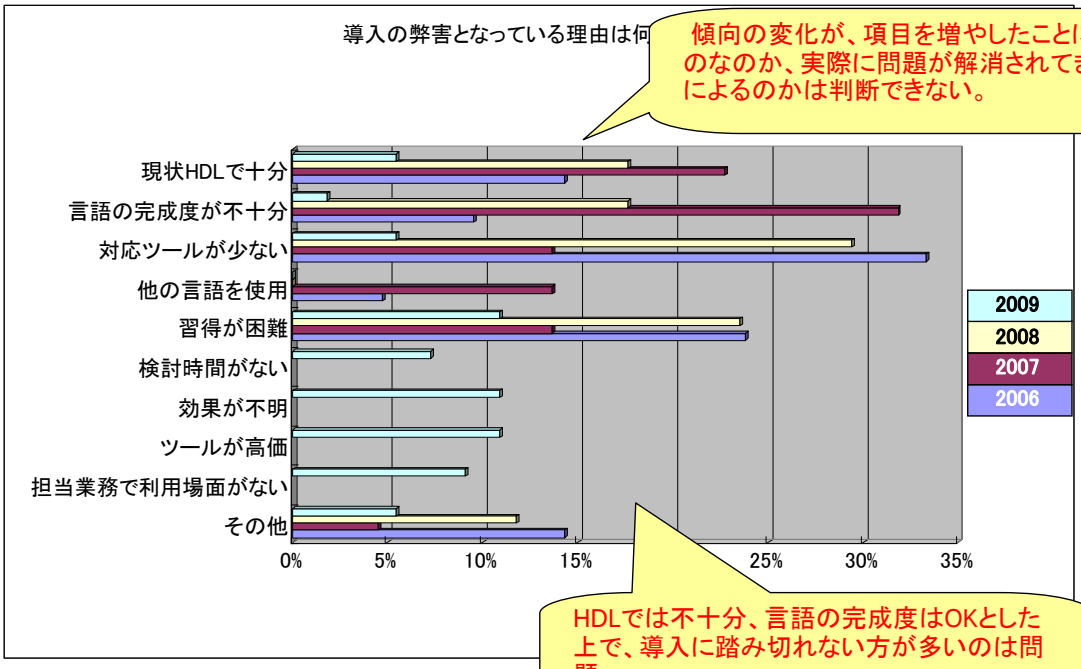
10.「8」で「検討中」と回答された方へ 導入予定時期は?

現状導入が明確な人は少ない



© Copyright 2009 JEITA, All rights reserved
JEITA

11.「8」で「必要ない」、「検討中」と回答された方へ 導入の障害となっている理由は?



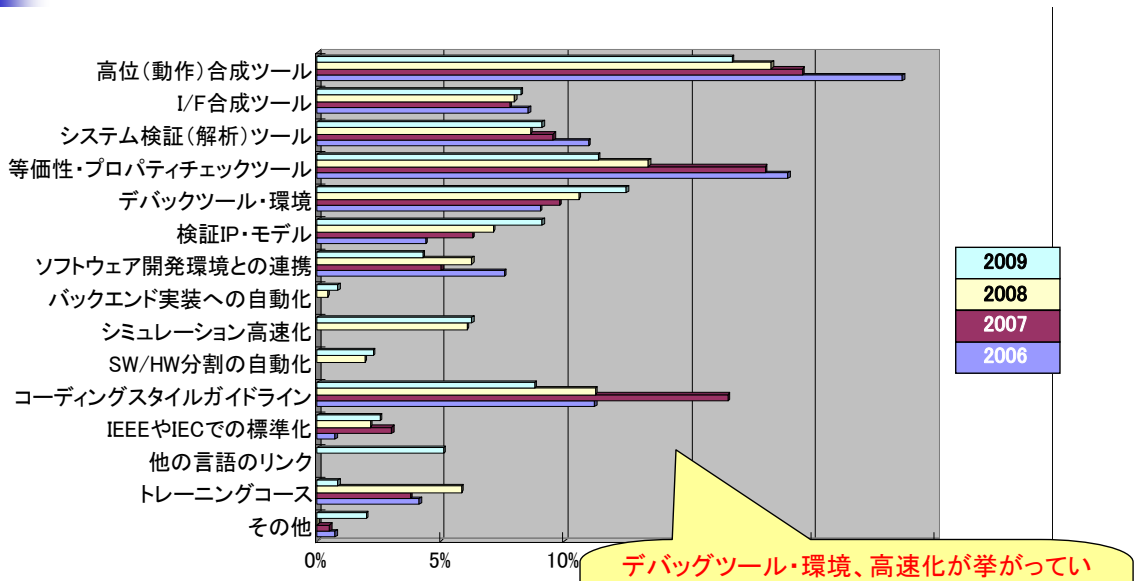
傾向の変化が、項目を増やしたことによるものなのか、実際に問題が解消されてきたことによるのかは判断できない。

HDLでは不十分、言語の完成度はOKとした上で、導入に踏み切れない方が多いのは問題。

- その他
  - MATLABの方が優れている。
  - 営業なので

© Copyright 2009 JEITA, All rights reserved
JEITA

12. SystemCをより活用する為に充実が必要なものは？



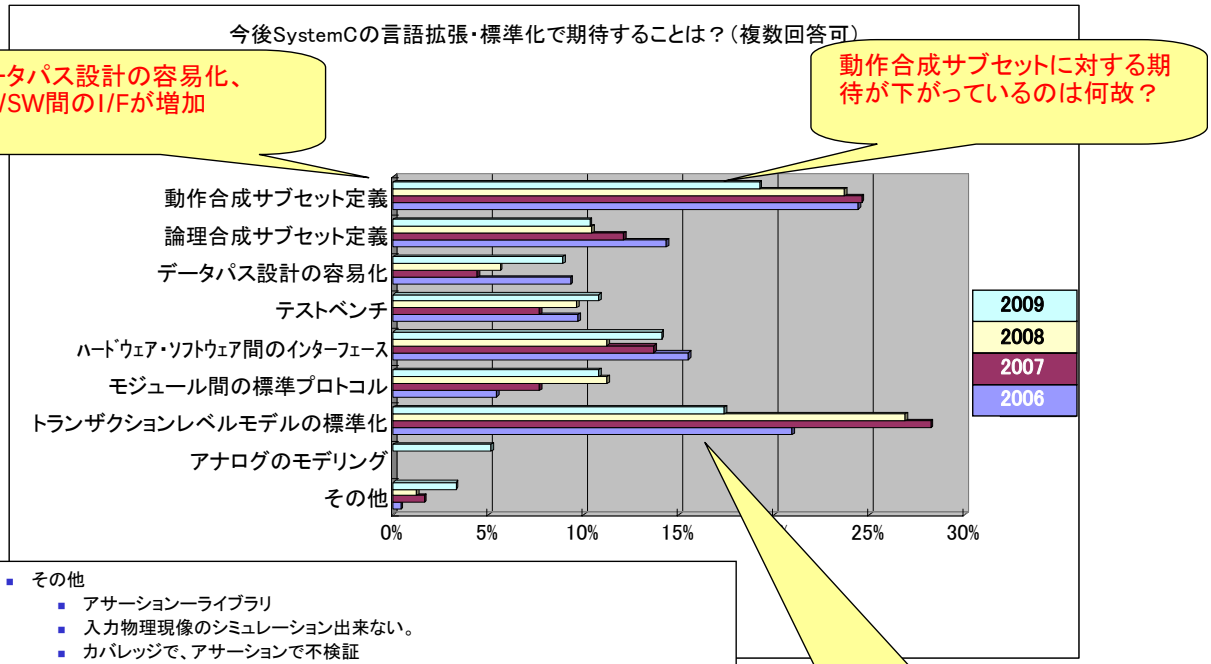
デバッグツール・環境、高速化が挙がっているのは実際に運用されているため？
高位合成、等価性・プロパティチェックが減少傾向なのはなぜ？

- その他
  - 価格削減
  - 導入用（勉強用）のフリーツールの普及
  - Modular I/F記述スタイルの標準化
  - 検証機能
  - 情報共有（ネット上等）
  - コンパイルエラーのメッセージをもっと判りやすくする。
  - ツールコストの削減（工数の削減コスト＞ツール導入コストにならないと難しい）

- 他の言語のリンク
  - VHDL、SystemVerilog、e

© Copyright 2009 JEITA, All rights reserved
JEITA

13. 今後SystemCの言語拡張･標準化で期待することは?



データパス設計の容易化、HW/SW間のI/Fが増加

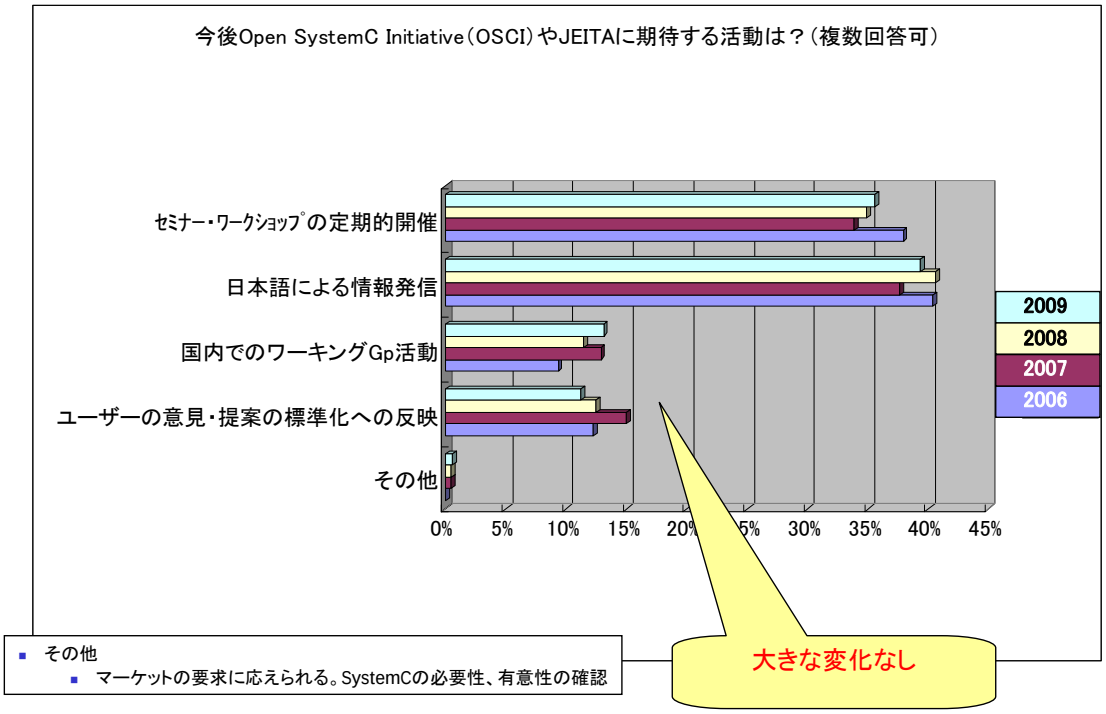
動作合成サブセットに対する期待が下がっているのは何故？

- その他
  - アサーション－ライブラリ
  - 入力物理現像のシミュレーション出来ない。
  - カバレッジで、アサーションで不検証
  - Modular I/F記述スタイルの標準化（HLSとシステム検証のシームレスな接続に必要
  - ツールやモデルの導入容易化
  - Sampleの充実

トランザクションレベルモデリングは、標準化が一段落したため、減少。

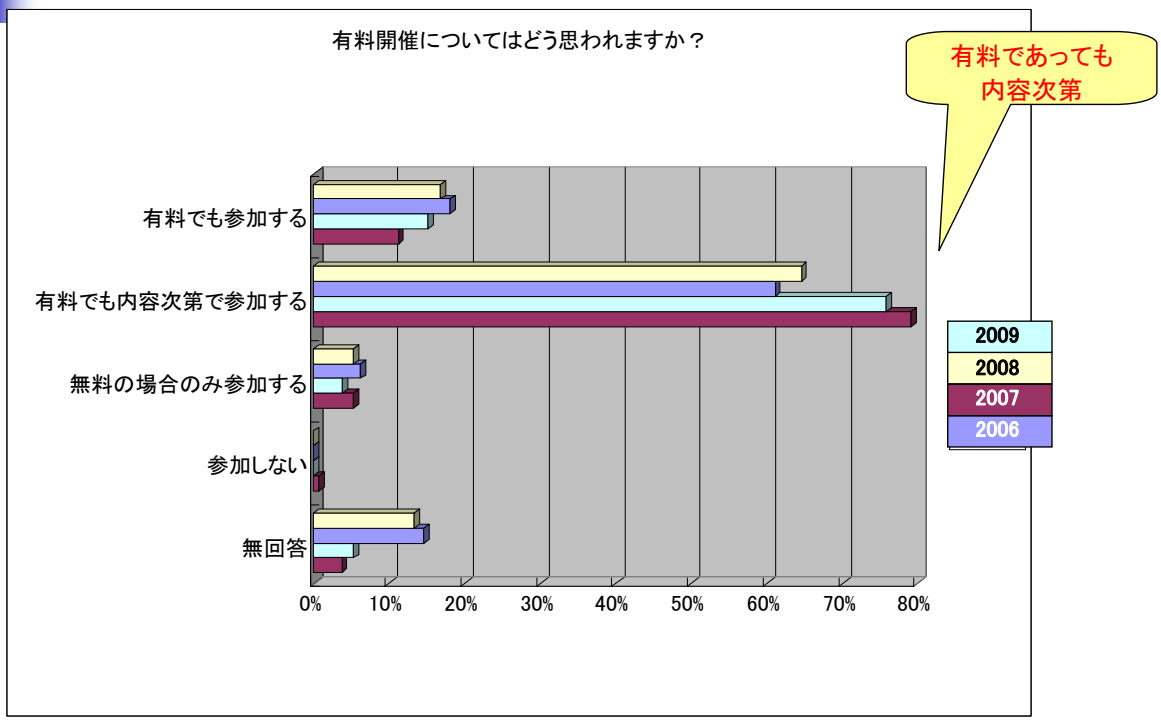
© Copyright 2009 JEITA, All rights reserved
JEITA

14. 今後Open SystemC Initiative（OSCI）やJEITAに期待する活動は？



© Copyright 2009 JEITA, All rights reserved

15.有料開催についてはどう思われますか？



© Copyright 2009 JEITA, All rights reserved

その他コメント

- SystemCの進展は、C設計がさけばれはじめている頃から注意してみていたが、ほとんど進展が見られないように感じる。メリットや、工数削減効果に対して、全体的な検証が不充分ではないか？例えばツール費用が高くて導入が容易にできなかったり、モデルが不充分で、実質的な工数が削減できないなど前から変わってないように思える。

- 等価性（SystemCとRTL）チェック後にもRTLの検証が必要となり、それほど工数が減らないように感じる。そのあたりも今後事例で説明して頂きたい。

- システムレベル設計で、記述したCレベル（SystemCでも）から高位合成を行って早い段階でFPGAの様な物で、評価を行いたい（画像処理のアプリケーションでは、シミュレーションだけでは、アルゴリズム評価が不充なため）、こういったツールのフローが確立されると良いと思う。

- FPGAベンダーのEDAツールにSystemC開発モデル／フローを導入して欲しい。これらのツールが広まれば汎用のSystemC開発ツールも広まっていくと思う。

- OSCIベースのSystemC開発を行うとき、Compile Errorなどを分かりやすくする方法が欲しいと思います。特にRTL設計者が、C＋＋経験が少ない技術者、SystemCを学ぶとき、その部分が、大きなネックとなっています。

© Copyright 2009 JEITA, All rights reserved

4.3 SystemVerilog ワーキンググループ報告 添付資料
SVTG-1 IEEEp1800-2009 Issue List

了承	assigned	editorial/not fixed 再アピール	"Fixed" status is checked

SVTG Issue #	Summary	Assigned ID	Status	JEITA ack
100	"HDL" doesn't seem to be a suitable expression for SystemVerilog. In some cases, SystemVerilog is referred as "HDVL.". Please discuss in TWG and define the terminology and keep consistency in the entire document.	NA	fixed	
101	"Elaboration occurs after parsing the HDL and before simulation" should be revised. The reasons are (1). because Compilation description has been added before this description and compilation has been defined as the pre-process of elaboration (2). not only simulation is the purpose to use SystemVerilog.	1825	not fixed closed	OK as our request makes the scope to change too large
102	"IEEE Std 11364-2005" should be "IEEE Std 1364-2005"	NA	fixed	
103	There is no ""begin_keyword" description in index	NA	fixed	
104	Want to see the keyword list categorized by approved version. (e.g. 1800-2005 keywordA, keywordB; 1800-2008 keywordC, keywordE…;) Please see the attached file for the example.	1826	fixed	We expect this item will be fixed.
105	WindowsNT is shown as the example to describe the difference of binary file operation. Need more updated OS example.	1827	fixed	OK
106	"Syntax 20-22—" should be "Syntax 20-22" because the original contains redundant hyphens.	NA	fixed	

107	"Syntax21-"1 should be "Syntax21-7"	NA	fixed	
108	"Syntax21-2" should be "Syntax21-8"	NA	fixed	
109	"Syntax21-3" should be "Syntax21-9"	NA	fixed	
110	"SDF to Verilog delay value mapping" should be "SDF to SystemVerilog delay value mapping"	NA	fixed	
111	"SDF constructs mapping to Verilog" should be "SDF constructs mapping to SystemVerilog"	NA	fixed	
112	"SDF delay constructs mapping to Verilog declarations" should be "SDF delay constructs mapping to SystemVerilog declarations"	NA	fixed	
113	"Table" are at the top of the charts but "Syntax" are at the bottom of the charts. Using link-jump feature, we can jump to "Table"/"Syntax" but our expectation is to jump to the charts. To remove this inconvenience, please put both at the top of the charts.	NA	fixed	
114	In the definition of "pass_enable_switch_instance", "inout_terminal ,(black comma) inout_terminalred " should be "inout_terminal ,(red comma) inout_terminalred "	NA	fixed	
115	Syntax continues to the subject with "--" but Table continues to the subject with ":". Please unify.	NA	**fixed**	
116	on Page 668, "//from A.3.2" should be "//from A.5.2"	NA	fixed	
117	The hyperlink from Syntax29-5 on page 683 is not active.	NA	**FIXED**	
118	"Software tools can perform" should be "Software tools should perform". Such warnings are mandatory to avoid describing unexpected behavior.	1828	fixed	Our intention was to remove these statements as LRM doesn't have to define how to implement in tools.

119	"assert (req1 || req2);" in the example should be "assert (req1 || req2)"	NA	fixed	
120	"Table 7-2" should be "Table 8-2" on line 2 of page 124.	NA	fixed	
121	"fileId" should be "fileID" on line 6 of page 126.	NA	fixed	
122	we believe that usually logic [31:0] would be little endian.	1829	normal, fixed	Correction: 6.8 -> 11.5.1 As the previous example shows, using up/down-vect would be better for readers than big/little-vect.
123	There are no Sequence methods(ended, triggered, matched) in the BNF	1830	fixed	We strongly expect this item will be fixed.
124	The indentation of Yes/No in table18-29 are broken.	NA	fixed	
125	the title at the article and at the bookmarks in Acrobat is different	NA	fixed	
126	the title at the article and at the bookmarks in Acrobat is different	NA	fixed	
127	the title at the article and at the bookmarks in Acrobat is different	NA	fixed	
128	the title at the article and at the bookmarks in Acrobat is different	NA	fixed	
129	the title at the article and at the bookmarks in Acrobat is different	NA	fixed	
130	"It shall be illegal for a module declaration to mix the port_reference port lists of non- ANSI style module headers with the list_of_port_declarations ports lists of ANSI style headers." should be "It shall be illegal for a module declaration to mix the port_reference port lists of non- ANSI style module headers with the list_of_port_declarations ports lists of ANSI style headers in one module" to explicitly show that the mixture of non-ANSI and ANSI in one module is prohibited.	1831	not fixed close	OK

131	path rule/scope rule is not clearly described here. 22.7 definition would be mentioned here.remove "If $root is not specified, a hierarchical path is ambiguous. For example, A.B.C can mean the local A.B.C or the top-level A.B.C (assuming there is an instance A that contains an instance B at both the top level and in the current module). Verilog addresses that ambiguity The ambiguity is resolved by giving priority to the local scope and thereby preventing access to the top-level path. $root allows explicit access to the top level in those cases in which the name of the top-level module is insufficient to uniquely identify the path."	1832	FIXED	We expect this item will be fixed.
132	"The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code." should be more precisely defined, i.e. the scheduling should be independently defined to avoid racing. We recommend to execute the immediate assertions at "observed".We don't put this to the situation at "$display" which cause the different behavior for each simulator.	1833	fixed as #2005	We expect this item will be fixed.
133	for PLI, tf_ and acc_ should be kept, at least in appendix for legacy code maintenance purpose	1834	not fixed closed	OK
134	Give the name of the superset (summation) of Assertion API, Coverage API, Data read API and VPI. For example, SVPLI.	1835	FIXED	We expect this item will be fixed.

EDAアニュアルレポート　2008

2009年5月発行

禁無断転載

発　　行　　社団法人 電子情報技術産業協会　電子デバイス部
〒101-0065
東京都千代田区西神田3-2-1
千代田ファーストビル南館
電話　03-5275-7258　　FAX　03-5212-8121

作　　成　　三協印刷株式会社
〒152-0002
東京都目黒区目黒本町5-20-7
電話　03-3793-5971　　FAX　03-3793-6242

Copyright 2008 by Japan Electronics and Information Technology Industries Association

本書中に記載の会社名および商標名は、各社の登録商標、商標です。