平成 21 年度戦略的基盤技術高度化支援事業

（平成 21 年度補正予算事業）

「画像・動画処理用 C 言語の LSI 化の支援システム開発」

研究開発成果等報告書

平成 22 年 3 月

委託者　中国経済産業局

委託先　株式会社プライムゲート

ま え が き

本成果報告書は、平成 21 年度戦略的基盤技術高度化支援事業（平成 21 年度補正予算事業）における「画像・動画処理用 C 言語の LSI 化の支援システム開発」の研究結果についてその成果報告を行う物である。

目　次

第1章 研究開発の概要

1-1　研究開発の背景・研究目的及び目標

1）研究の目的

現在、高度に変化する製品の機能への要望に対して LSI 設計技術者不足となっている現状があり、技術者のいる特定企業への回路設計が集中することで、納期の長期化や工数不足によりコストの増大が起きている。

本研究開発では、新製品開発における動作制御部分 LSI の開発において開発期間の短縮と工数低減を目的とした C 言語を用いた汎用性のあるフレームワークを開発し、川下企業の新製品開発の開発時間を 1/10 に削減、また、生産性を 30％向上させることにより、コスト削減を目指す。

2）研究の概要

【1】生産性向上技術の確立

フレームワーク、画像圧縮用ユニットおよび高位合成結果を全て連結した場合に、最終的な生産性向上目標値を現在の 10 倍以上とする。（これを 100%として、子項目【2】以降の目標値を記述）

具体的な評価は以下の項目で行う。

1. 設計総工数：従来の設計総工数との比較
2. 設計記述量：従来のプログラム記述量との比較
3. 設計記述修正量：プログラムの改変数および改変量の比較

総合的目標値の 10 倍以上の生産性向上のうち、各研究項目での目標値は以下の通りに振り分ける。

【2】画像・動画技術開発

画像圧縮用途向けのソフトウェア開発手法として、アルゴリズムレベルから実装レベルのソフトウェアへの開発工程の詳細な検討によって、ユニット部分としての生産性向上率 40％を達成する。

【3】高位合成ツール向けライブラリや IP の開発

高位合成向けかつ画像圧縮用のライブラリや IP（LSI を構成するために必要な機能ブロック）の整備によって生産性向上率 30％を達成する。

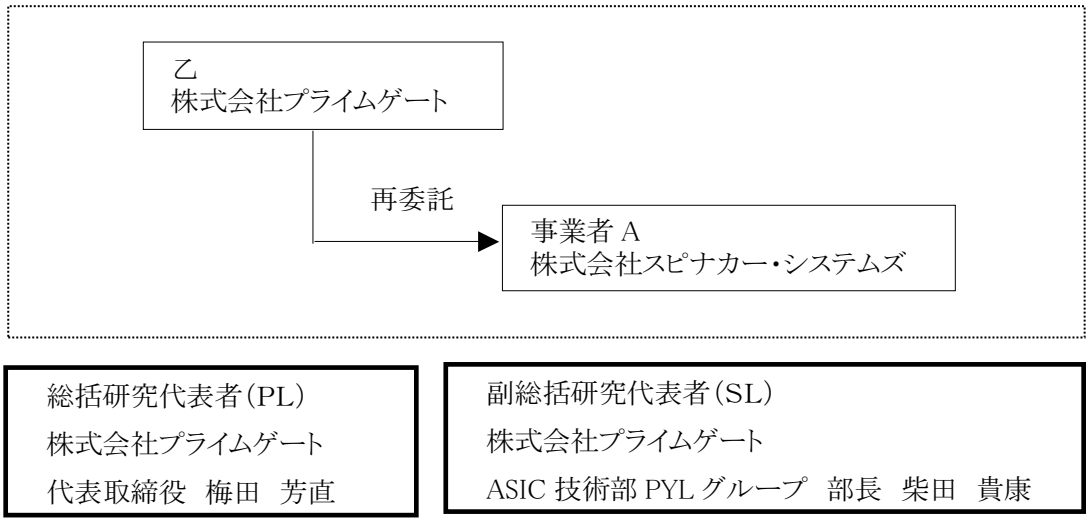
【4】ソフトウェアを LSI 化する設計手法の生産性向上方法の開発

実装レベルでの高位合成ツール自体の問題点をアルゴリズムレベルで解決可能なソフトウェアを開発して、生産性向上率 30％を達成する。

1-2　研究体制

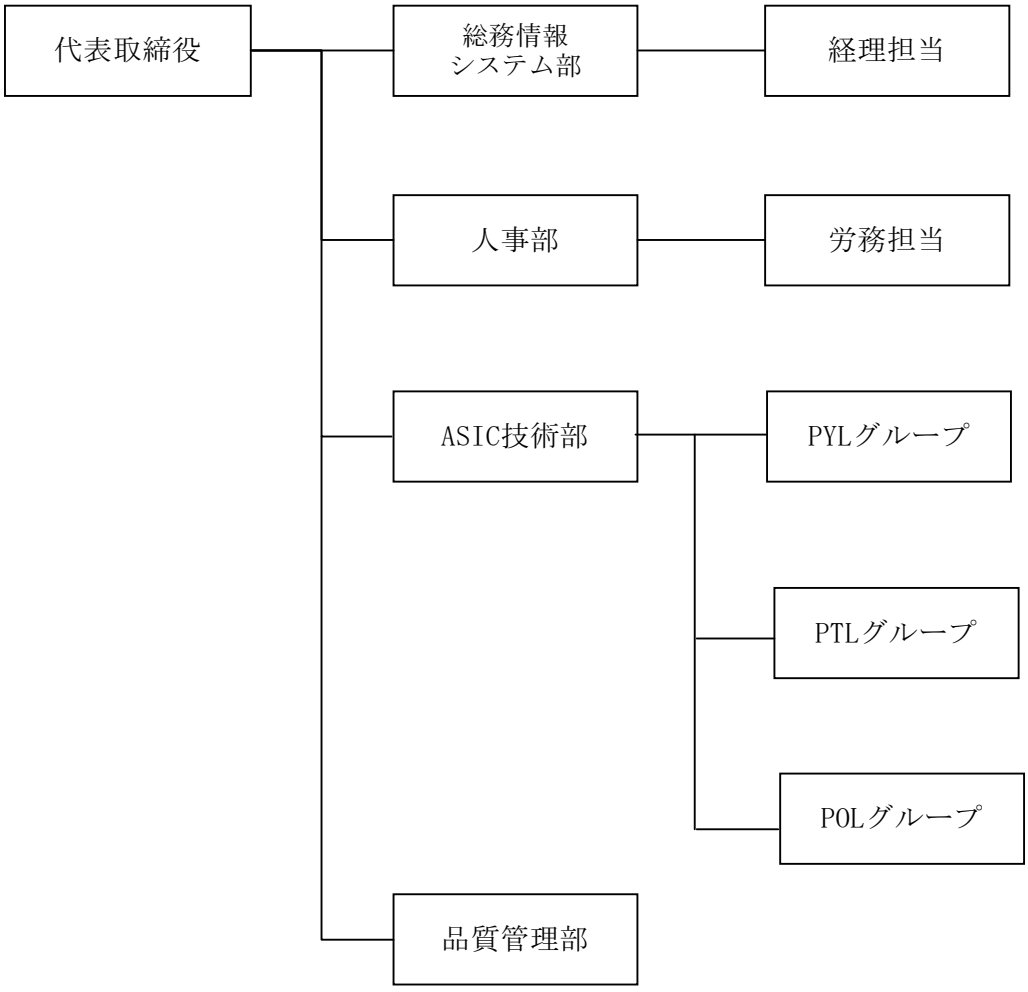
（1）研究組織及び管理体制

1）研究組織（全体）



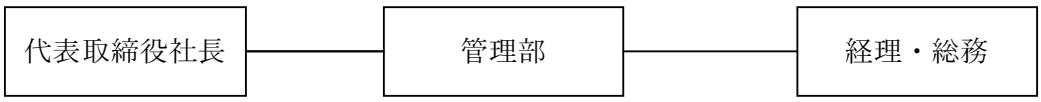
2）管理体制

①事業管理者 ［株式会社プライムゲート］



②(再委託先)

[株式会社スピナカー・システムズ]



(2)管理員及び研究員

【事業管理者】株式会社プライムゲート

①管理員

氏　名	所属・役職	実施内容(番号)
梅田　芳直	代表取締役	【5】
堀本　美穂	総務情報システム部　部長	【5】
金重　俊志	人事部兼品質管理部　部長代理	【5】
森重　浩子	総務情報システム部　主任	【5】
木村　梢	総務情報システム部兼人事部	【5】

②研究員

氏　名	所属・役職	実施内容(番号)
梅田 芳直(再)	代表取締役	【1－1】
柴田　貴康	ASIC 技術部 PYL グループ　部長	【1－1】、【2－1】、 【2－2】、【2－3】、 【2－4】、【3－1】、 【3－2】、【4－1】
松岡　秀樹	ASIC 技術部 PYL グループ　課長	【2－4】、【3－2】
池本　博朗	ASIC 技術部 PYL グループ　課長代理	【1－1】、【4－1】
明石　紳吾	ASIC 技術部 PTL グループ　課長代理	【2－1】、【2－2】、 【2－3】
岩井　祥悟	ASIC 技術部 POL グループ　課長代理	【3－1】、【4－1】
永島　正康	ASIC 技術部 PYL グループ　主任	【2－4】
原　武史	ASIC 技術部 POL グループ　主任	【3－1】
大澤　智志	ASIC 技術部 PYL グループ　主任	【4－1】
瀧野　頼光	ASIC 技術部 PYL グループ　主任	【3－2】
立川　源	ASIC 技術部 PTL グループ　主任	【2－3】
石村　直樹	ASIC 技術部 PTL グループ　主任	【2－1】
北村　修治	ASIC 技術部 PYL グループ　主任	【1－1】
後藤　宏樹	ASIC 技術部 PYL グループ　リーダー	【1－1】
岡本　典暁	ASIC 技術部 POL グループ　リーダー	【3－1】
大田　直也	ASIC 技術部 PTL グループ　リーダー	【2－2】
池田　誠	ASIC 技術部 PYL グループ　担当	【2－4】
廣下　淳	ASIC 技術部 POL グループ　担当	【3－1】
石川　裕尚	ASIC 技術部 PTL グループ　担当	【2－3】
黒川　剛	ASIC 技術部 PTL グループ　担当	【2－1】
木南　研吾	ASIC 技術部 POL グループ　担当	【4－1】
戸塚　秀樹	ASIC 技術部 PTL グループ　担当	【2－2】
園田　昌之	ASIC 技術部 PYL グループ　担当	【2－4】
伊波　真哉	ASIC 技術部 PTL グループ　担当	【2－1】
古巻　和裕	ASIC 技術部 POL グループ　担当	【3－1】
仙田　眞之	ASIC 技術部 PYL グループ　担当	【1－1】
髙田　豊	ASIC 技術部 PTL グループ　担当	【2－3】
田村　康博	ASIC 技術部 PYL グループ　担当	【3－2】
矢野　功人	ASIC 技術部 PTL グループ　担当	【2－2】

【再委託先】※研究員のみ

株式会社スピナカー・システムズ

氏　名	所属・役職	実施内容（番号）
片桐　徹	代表取締役社長	【3－1】

(3) 経理担当者及び業務管理者の所属、氏名

（事業管理者）

株式会社プライムゲート

（経理担当者）　総務情報システム部部長　堀本美穂

（業務管理者）　人事部兼品質管理部部長代理　金重俊志

（業務管理者）　総務情報システム部兼人事部主任　森重浩子

（再委託先）

株式会社スピナカー・システムズ

（業務管理者）　代表取締役社長　片桐徹

（経理担当者）　管理部　立見則夫

(4)他からの指導・協力者

氏　名	所属・役職	備考
渡邊　孝博	早稲田大学 大学院情報生産システム研究科 教授	アドバイザー

(5)知的財産権の帰属

知的財産権は全てコンソーシアム内の事業管理者及び再委託先に帰属することを希望。

(6)その他

なし

1-3　成果概要

【1】生産性向上技術の確立

フレームワーク、画像圧縮用ユニットおよび高位合成結果を全て連結した場合に、最終的な生産性向上目標値を現在の 10 倍以上とした。

この目標値に対し研究結果として

- 本テーマで開発したフレームワーク CDEF を使用することで 1.4 倍生産性向上した。さらに人的リソース支援により 1.9 倍の生産性向上した。
- 同じく本テーマで開発した FGen, ライブラリを使用することで、5.8 倍の生産性向上した。 IP のみにおいては最大 75 倍の生産性向上した。
- 従来開発手法と C 言語ベース開発手法との予測工数の比較を行った結果 2.7 倍生産性向上した。

という結果が得られた。

本研究目標値の 10 倍には届かなかったが、全て連結した場合は生産性 2 倍以上という成果が得られた。また、IP 再利用という点においては生産性 75 倍という大変生産性の高い値を得られた。

【2】画像・動画技術開発

画像圧縮用途向けのソフトウェア開発手法として、アルゴリズムレベルから実装レベルのソフトウェアへの開発工程の詳細な検討によって、ユニット部分としての生産性向上率 40％を達成する。

この目標値に対し研究結果として

- 各種 IP/ライブラリを作成し、それを適用した場合に検証環境構築工数が 95h から 80h へと 16%削減され、1.2 倍の生産性向上した。
- 前開発で得た成果を手順書にフィードバックすることで、今後の同様の開発において設計の生産性が 1.5 倍向上し、検証の生産性は 1.3 倍向上した。
- 画像入出力機能を持つ共通の SystemC 検証環境のベースを作成した。これを再利用することで、今後の同様の開発において検証環境立ち上げ工程の生産性を 1.8 倍向上した。
- 手設計と高位合成設計の比較を行い設計および検証での工程で 3 倍の効率化を達成した。
- 検討した手順に準拠して検証用ソフトウェア IP の作成を行うことで、工数が約 25％短縮され、生産性は約 1.3 倍向上した。
- 手順に準拠して SoC 検証環境の作成を行うことで、工数が約 60％短縮され、生産性は約 2.5 倍向上した。

という結果が得られた。

生産性においてはそれぞれの項目でほぼ 120%～300%となっており、目標値の 10 倍を 100%とした場合の 40%にはやや達成しなかった。

【3】高位合成ツール向けライブラリや IP の開発

高位合成向けかつ画像圧縮用のライブラリや IP（LSI を構成するために必要な機能ブロック）の整備によって生産性向上率 30％を達成する。

この目標値に対し研究結果として

- 従来の HDL 作成手順をもとに新たなハードウェア IP 作成手順を作成し、その手順が生産性向上の手

法として問題がないか検証を行った。工数が約 50%短縮され生産性は約 2 倍の向上となった。

- SystemVerilog を導入することが生産性向上の手法として問題がないか検証を行った。「C 言語との親和性」、「記述量」、「検証実行時間」を総合的に判断した結果、生産性は約 1.5 倍向上した。
- ソフトウェアIP作成フローによりソフトウェアIP作成の工数を、43.3%削減し生産性は 2.32 倍向上した。

という結果がえられた。

生産性向上においてはそれぞれ 150%～232%となっており、目標値の 10 倍を 100%とした場合の 30%にはやや達成しなかった。

【4】ソフトウェアを LSI 化する設計手法の生産性向上方法の開発

実装レベルでの高位合成ツール自体の問題点をアルゴリズムレベルで解決可能なソフトウェアを開発して、生産性向上率 30%を達成する。

この目標値に対し研究結果として

- 分離・設計・検証の各作業を手順化し、記述フォーマットおよび検証環境ひな形を作成した。また、手順の効果として各手順の有無による作業時間の予測を算出し最大で 3.0 倍(67%削減)の生産性向上という予測結果を得た。

という結果がえられた。

生産性向上においては 300%以上となっており、目標値の 10 倍を 100%とした場合の 30%を十分達成することができた。

1-4　該当研究の連絡窓口

住所:山口県宇部市床波一丁目 6 番 13 号
名称:株式会社プライムゲート

代表者役職・氏名: 代表取締役　梅田　芳直
Tel:0836-54-0016　　Fax:0836-51-4989
E-mail:ML_SOUMU@primegate.co.jp

連絡担当者所属役職・氏名:株式会社プライムゲート　総務情報システム部　部長　堀本　美穂
総務情報システム部　担当　木村　梢
Tel: 0836-54-0016　　Fax: 0836-51-4989
E-mail: ML_SOUMU@primegate.co.jp

第2章 本論・支援ツールのフレームワーク開発

2-1 【1-1】支援ツールのフレームワーク開発

2-1-1 概要・目的

研究所ベースのアルゴリズムレベルのソフトウェアから事業所ベースの高位合成可能なソフトウェアへ工程が進む中で、今までの実績を元に生産性に影響をもたらす部分を検討し抽出する。

生産性に影響のある要素の中で相互作用を考慮した各工程でのソフトウェアにおける記述量削減、記述作業工数削減を実現できる様に支援ツールの仕様を決定する。

上記仕様を元に、汎用性を持ちながら画像・動画圧縮の特徴を十分にカバーできるフレームワーク設計を行う。フレームワークは以下の 3 要素で構成し、その開発を行う。

1) マニュアル、規定、手順書および標準規格類
   C ベース設計マニュアル類、C ベース工程管理手順書および C ベーステンプレート（ひな形)によるフィードバックによる学習機能を兼ね備えたマニュアル、規定、手順書および標準規格類を文書データベースとして開発する。
2) 統合置換ソフトウェア
   ソフトウェアとハードウェアの回路設計部分を全てソフトウェア状態からハードウェア状態へ変更または置換する支援ツールを開発する。
   また、高位合成ソフトは複数種類存在し、これに対応したソフトウェア記述の制約は同様に複数種類存在するので、このフォーマットの違いを統合置換ソフトウェアによって自動変換する。
3) 各種特定用途用ユニットとインターフェース部
   上記 2 項目を含む特定用途向けのユニットと、そのユニットを外部ソフトウェア等に繋ぐインターフェース部を開発する。本研究開発では、特定用途向けのユニットとして画像・動画圧縮処理に特定した規定フォーマットライブラリや IP データベース等を開発する。

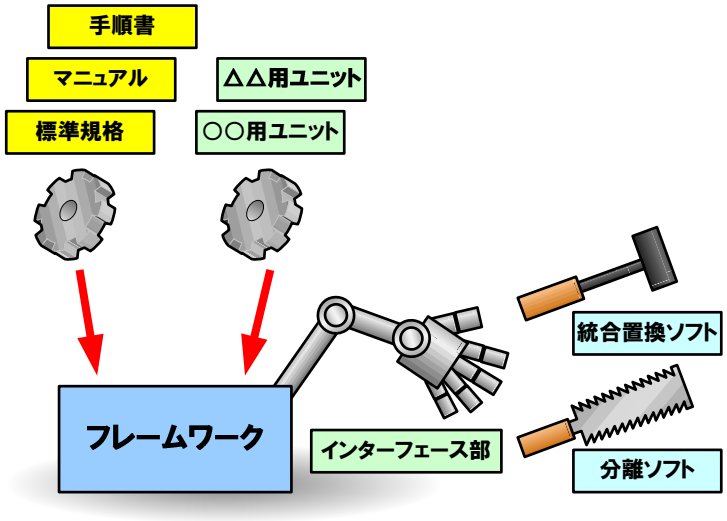
実施計画書にて定義されている「2) C 言語ベースの分離ソフトウェア」については、テーマ「【4-1】ソフトウェア記述の標準化」との関連が深いため、テーマ【4-1】にて検討することとした。

2-1-2　内容

本テーマでは、他テーマで開発する手順書・マニュアル・IP・テンプレートなどに沿った効率的な開発環境を提供するために、フレームワークを開発する。

フレームワークは先述の 3 要素で構成されるが、マニュアル・手順書類と特定用途向けユニット/インターフェースは共にフレームワークの部品として使用されるデータとして考えることができる。また、本テーマではこれらを同類の要素として取り扱うことができるシステムとなるよう検討する。統合置換ソフトウェアはフレームワークが道具として使用する外部の支援ツールであると言える。

これらの関係をイメージ図に示すと、以下のようになる。



図【1-1】. 1 マニュアル/手順、ユニット、インターフェース、統合置換ソフトウェアの関係イメージ

マニュアル・手順書類 および特定用途向けユニットは、フレームワークで何を行うべきか、インターフェース部を経由して統合置換ソフトウェアなどの「道具」をどのように使うべきか、などの開発業務全般のノウハウが集約された要素である。この要素を如何に高い水準で構築・維持できるか否かによって、生産性が大きく左右される。

設計対象となる回路規模は年々増加しており、従来行われてきた開発手法で対応できる規模の限界に近づきつつある。本研究事業では C 言語設計を取り入れた次世代の設計ノウハウ・開発手順を構築するが、本テーマで開発するフレームワークは、利用者に対して「最新・最先端」の開発手順を「効率よく」提供するという役割を担う。

設計ノウハウの細分化・再利用も生産性に影響を与える要素である。統合置換ソフトウェアなどのフレームワークの「道具」として位置づけられる支援ツールは、開発手順の一部分について充分に細分化された設計ノウハウを、アルゴリズムを明確にした上で実装したものである。

2-1-2-1　開発環境に関する検討

フレームワークの「部品」「道具」を効率よく使用するだけでなく、人的リソースの効率的な使用についてもフレームワークを開発する上で検討を行う。

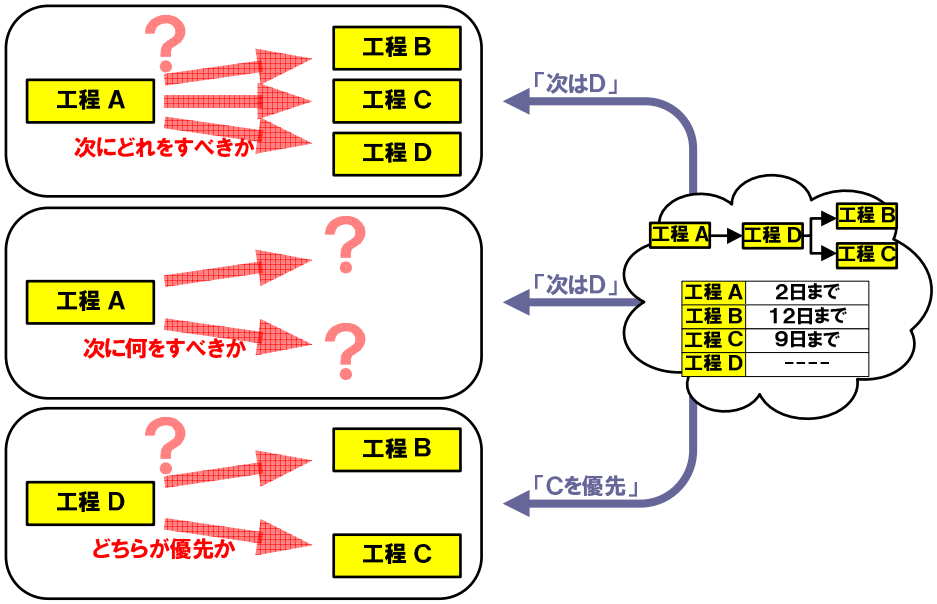
2-1-2-1-1 問題点

従来手法では、各担当者がどの工程を担当するか、工程ごとの優先度や期日といった情報は各担当者、もしくは開発リーダーが人力、もしくは別の外部ソフトウェアを用いて管理する必要があった。このような開発手順外での管理は、時として情報伝達ミスや管理漏れによって重大な納期遅延を引き起こす可能性を抱えている。

また、複数の担当者間で工程の割り当てが適切でないために、特定の担当者に過大または過小な割り当てが発生する可能性も抱えている。前者は開発メンバー全体での生産性の低下を引き起こし、後者は納期遅延や品質低下、安全衛生上の問題が発生する場合がある。

2-1-2-1-2 解決の糸口

本テーマで開発するフレームワークでは各工程の担当者と工程とを紐付けし、各担当者に対して開発上のフォローを行うことを目標とする。たとえば、開発上の優先度や期日などの情報をフォローする、着手すべき工程とそうでない工程との振り分け、などの機能を持つ。先述の最新・最先端の開発手順を効率よく提供する役割と併せて、以降の工程に必要なマニュアルなどを絞り込み、速やかに次工程に移行できる環境を整えることを目標とする。



図【1-1】. 2　フレームワークによるフォロー

2-1-2-2　開発手順に関する検討

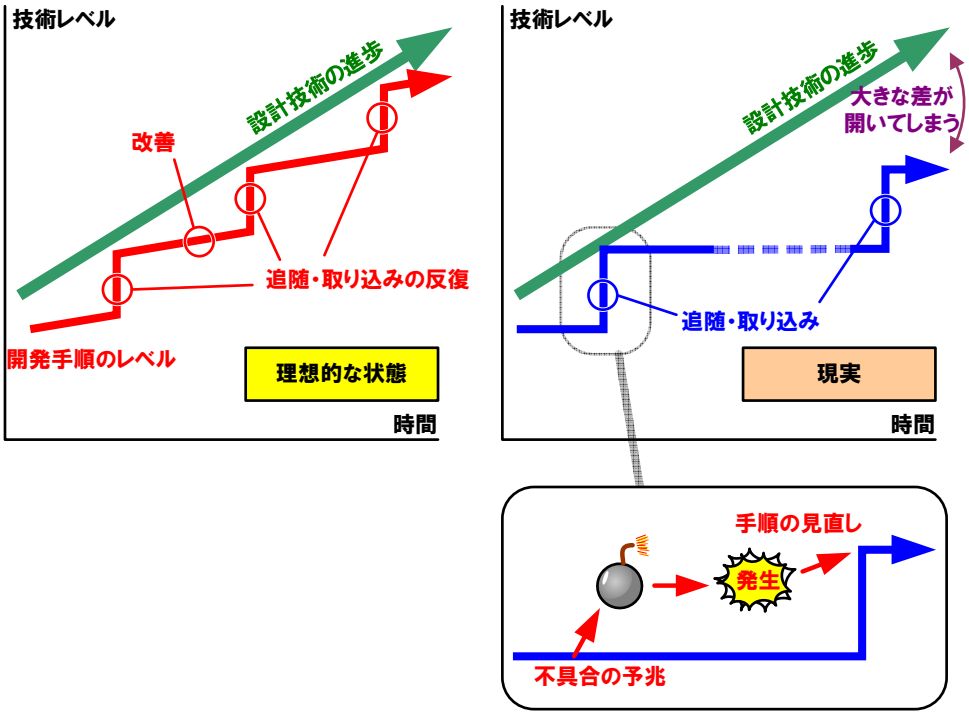
開発手順の改善に関して検討する。

2-1-2-2-1 問題点

先述のとおり、フレームワークでは最新・最先端の開発手順を利用者に提供する役割を担うことになる。

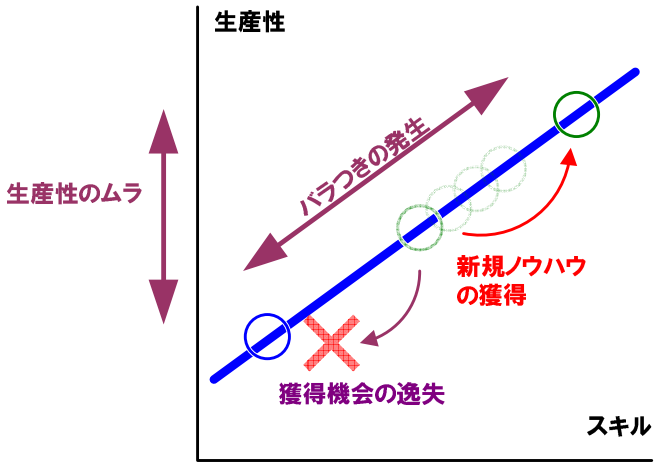
「最新・最先端」とはすなわち、設計技術の進歩に追随し、開発手順に取り込む(フィードバック)ことで到達できるが、このレベルを維持するためには、常にフィードバックを繰り返していかねばならない。また、外部要因からのフィードバックに依存するだけでなく、開発手順の利用者自身による手順の改善も随時行わねばならない。

従来手法においても開発手順の見直しや更新が行われていたが、重大な不具合が発生した際に是正処置的に見直しが行われることがほとんどで、それ以外の予防処置的、または定期的な更新は、積極的には行われていないのが現実である。これは、設計技術の進歩に追随するという視点が抜け落ちた状態であり、時間の経過とともにこれは、従来手法において開発手順の問題を察知することの限界であるとも言える。



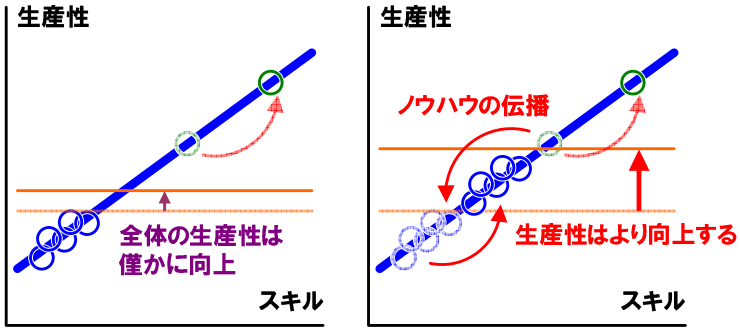
図【1-1】. 3 開発手順への取り込みと改善の必要性

同様に、最新技術を開発業務へ局所的に取り込んだ事例があったとしても、それが即座に開発手順全体に取り込まれることは稀である。その事例によって獲得・蓄積された開発ノウハウは当事者によってのみ伝承され、他の作業者がその開発ノウハウを得る機会は非常に制限された状況となってしまう。これを繰り返した結果として、作業者間で蓄積された開発ノウハウすなわちスキルにバラつきが生じる。スキルのバラつきは生産性の高さのムラとして表面化してくる。



図【1-1】. 4

一部の作業者だけが新たな開発ノウハウを獲得してスキルを高めた場合、作業者全体としてのスキルは僅かしか高められない。つまり全体の生産性も同様に僅かしか向上しない。しかし、獲得した開発ノウハウを他の作業者にも伝搬させることができれば、作業者全体としてのスキルを高め、バラつきの解消に繋げることができる。このとき、全体の生産性はより高いものになるはずである。



図【1-1】. 5

正負両面で挙げた 2 つの例より、開発手順へのフィードバックが高い生産性を維持する上で重要であることがわかる。

2-1-2-2-2 解決の糸口

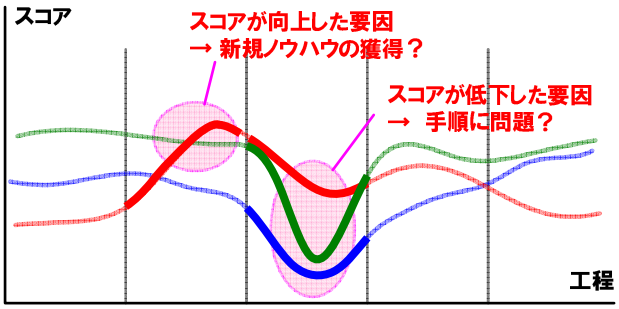
つまり、以下の 2 点を盛り込んだシステム構成とすることが重要である。

- 開発手順の変更を容易に行うことができるシステム
- 開発手順の問題点を早期に察知できるシステム

前者については、先述の通り、マニュアル・手順書等のデータベース化で対応する。また、データベース化するだけではなく、開発業務中に開発手順を常に提示することによって、作業者が意識することなく最新の開発手順を使用できる環境となる。開発手順が適切に維持されていれば、一部の作業者だけが保有していたノウハウは他の作業者も利用することが可能となり、生産性の低下を未然に防ぐことが期待できる。

後者について、ほとんどの見直しが事後に行われているという事実から、従来開発手法および既存システムでは事前に見直しを行うことができないという推理が導かれる。その時に使用している開発手順が潜在的に抱えている問題点を事前に察知するためのシステムが用意されていれば、不具合の発生を回避することで作業の効率低下を回避することができるのではないだろうか。

そこで、開発手順に定義された工程ごとにスコア(得点)を計算し、手順へのフィードバックが必要な状況かどうかを判断する手法を提案する。同様の工程を採用する複数のプロジェクトについて、工程ごとのスコアの推移を列挙し、工程とスコアとの相関がないかをチェックする。



図【1-1】. 6 工程ごとにスコアを計算し、特異点を抽出する

多数のプロジェクトに共通して、特定の工程でスコアの低下が認められるならば、開発手順やツールの使用方法などに共通の問題点を抱えている可能性が考えられる。その工程について有効な対策を打ち出し、開発手順へのフィードバックを行う必要がある。特定のプロジェクトのみスコアが高い場合は、何らかの新規手法を導入した可能性が考えられる。その工程の担当者にヒアリングを行うなどして新規手法かどうかを判断し、同様に開発手順へのフィードバックを行う。

本研究事業で取り組む C 言語設計は従来手法とは全く異なる新規手法であり、多数の試行錯誤が必要となってくる。この中で成功した、あるいは重要と思われるエッセンスについて、早期に開発手順に取り込むという効果も期待できる。

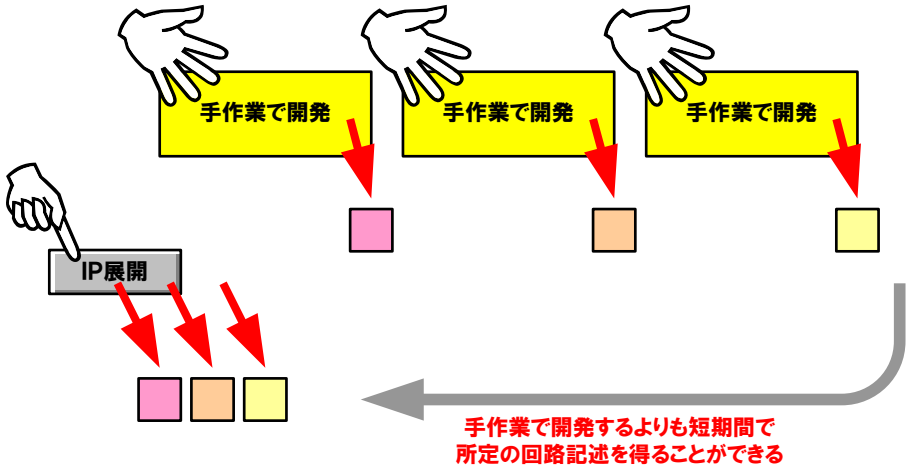
これらのサイクルを経て充分に細分化された開発手順は、その手順の一部または全てを支援ツールとして実装し、フレームワークから使用することでさらなる生産性向上とする。後述する統合置換ソフトウェアは、IP 利用に関する手順より派生した支援ツールと位置づけて開発を行う。

2-1-2-3　IP 利用に関する検討

本テーマで開発する統合置換ソフトウェアに関して、主な用途である IP について検討する。

2-1-2-3-1 問題点

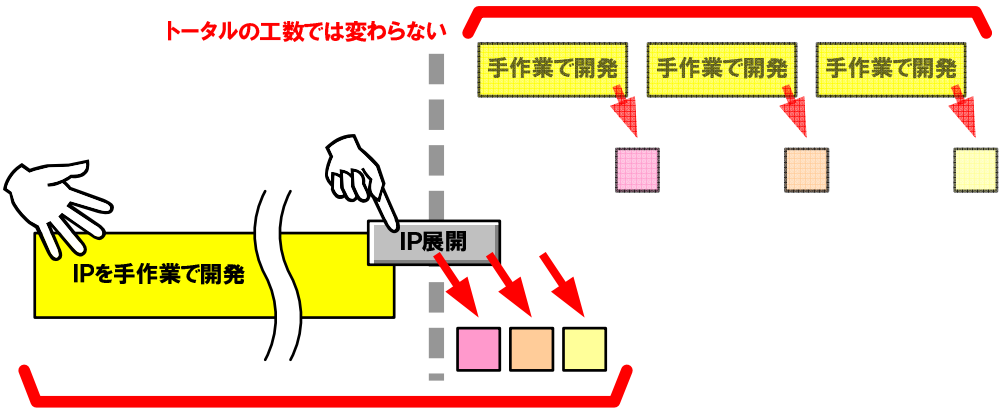
IP を利用することで、同等の機能を手作業で記述する場合と比べ、圧倒的な短期間で特定機能を実現するための回路記述を得ることができる。これを図に示すと、以下のようになる。



図【1-1】. 7 IP 利用の利点

上図に示すとおり、IP を利用することで効率的に開発を進めることができるように見える。しかし上図では「IP が既に用意されていること」という前提条件が考慮されていない。この場合、IP ベンダーなどの提供元から IP を入手した状態や、社内業務での派生物などが流用可能な形で用意されている状態を示す。

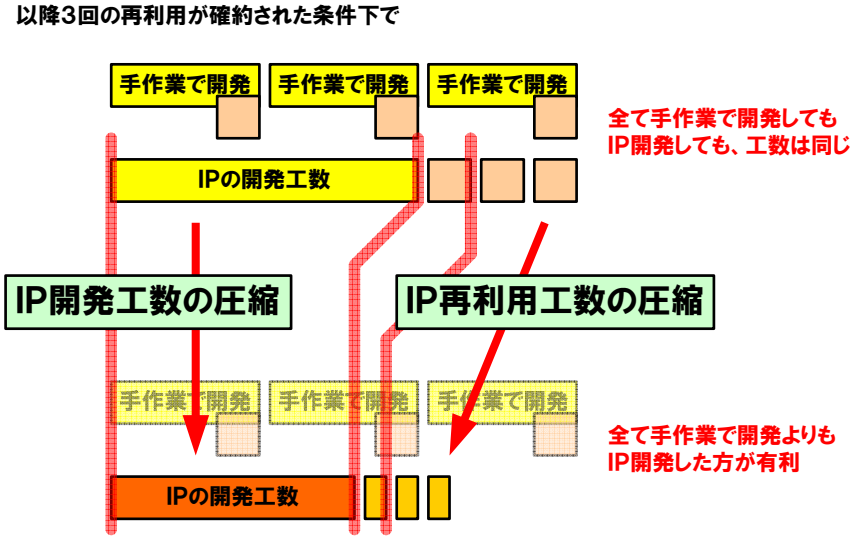
この前提条件を考慮し、IP が用意されていない状態での比較を図に示すと、以下のようになる。



図【1-1】. 8 IP 利用の盲点

2-1-2-3-2 解決の糸口

先述の IP 利用上の盲点については、IP の利点を減少させないよう IP 開発に関わる工数を圧縮することが直接的な問題解決となるはずである。そこで、本テーマでは、IP 開発に関わる工数の圧縮に主眼を置き、生産性向上の手法を開発する。



図【1-1】. 9 IP 開発および IP 再利用の工数圧縮イメージ

本テーマでは IP は統合置換ソフトウェアで使用するライブラリとして実装するため、汎用的かつ強力な言語仕様という点に注力して統合置換ソフトウェアの動作仕様を決定する。これにより、対応可能な IP の幅が広がり、いずれの用途においても開発のための期間を圧縮することが期待できる。

また副次的なメリットとして、IP 開発の工数を圧縮することで再利用しなければならない回数が抑えられるため、IP 開発に関わるリスクが低減される。つまり多様な IP を開発しやすい状況が発生し、それらの IP 再利用による、さらなる生産性向上をも狙うことが可能となる。

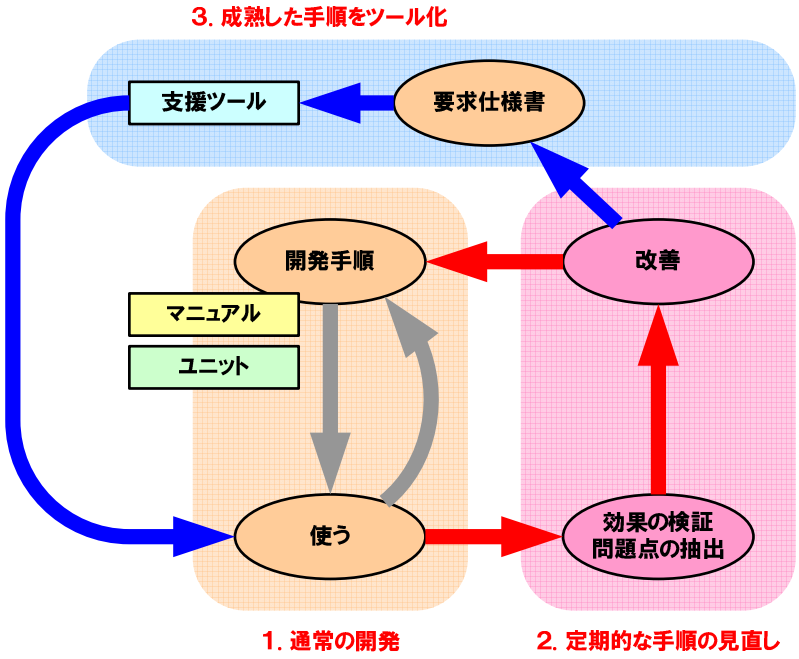
なお、再利用の回数が見込める IP の形態、すなわちどのような IP を開発すべきかについては、本テーマの趣旨からは外れるため、説明を割愛する。

2-1-2-4　生産性向上方法

上記 2 項目(開発手順，IP 利用)に関する検討を踏まえ、以下の要点に沿ってフレームワークおよび支援ツールの開発を行う。以下に開発にあたっての要点を挙げる。

2-1-2-4-1 フレームワーク

まず、最新の開発手順と支援ツールによる作業フローを以下図のように定義する。



図【1-1】. 10　フレームワークによる手順の改善フロー

これは、

1)　開発手順(マニュアル・手順書類、ユニット)および支援ツールを用いて開発業務を行う。

2)　複数の業務でスコアを計算し、開発手順の問題点や、有用な開発技術が発生していないかチェックする。チェック結果に従い、開発手順の見直し・改善を行う。

3)　開発手順の改善の結果　支援ツール化が可能な状態に成熟した手順は、手順を実現するための支援ツールを開発し、開発手順と置換する。

という流れを想定したものである。

フレームワークは、このフローに沿った業務遂行をサポートするための機能として、以下のものを有する。

表【1-1】. 1 本フレームワークが目標とする機能

項目	説明
使う	
任意のツールへの接続・使用	既存の業務ツールを使用することはもちろん、開発手順の見直しから派生した支援ツールも随時フレームワーク上から使用できる状態とする。
最新の開発手順の適用	他業務を起因とする開発手順の変更に追随し、進行中の開発業務にも最新の開発手順を適用できる。開発手順の変更は、後述の「改善」項で見直すことを想定。
情報の集約	開発に必要な情報をフレームワーク上のデータベースに集約し、各作業者で共有できる状態とする。また、情報の集約によって各作業者が個別に外部への調査をすることなく情報を利用できるようになる。
他業務とのリソース競合の回避	各業務・各作業者間で効率よく開発リソースを共有するための調停機能
開発	
プロジェクトの状態監視	開発手順と直結した進捗管理機能により、進捗の集計を自動で行う。従来は手作業で集計を行う必要があったが、本フレームワークが出力する進捗レポートをそのまま利用できる
開発業務の明確化	各作業者は、自身に割り当てられた業務にのみ着手することができる。 人的リソースの割り当て状況も進捗レポートから確認できる。異常な割り当てが行われている場合は、本フレームワークからリソースの再割り当てを行うことができる
改善	
問題点の抽出	スコアの計算を行い他業務との比較を行うことで、当業務および開発手順自体の問題点などを抽出するためのトリガとして利用できる。
反復による開発手順の強化	上図に示したサイクルを発生させ、見直しを反復させる。
開発手順更新の容易化	見直しが行われた開発手順を、容易にフレームワーク上のデータベースに取り込みできる。 また、開発手順だけでなく業務上有用と思われる外部の情報へのリンク(URL)も同様に取り込みできる。
支援ツールの派生	開発手順の改善に伴って派生した支援ツールは、上記「使う」項の通り、すぐにフレームワーク上から使用できる。

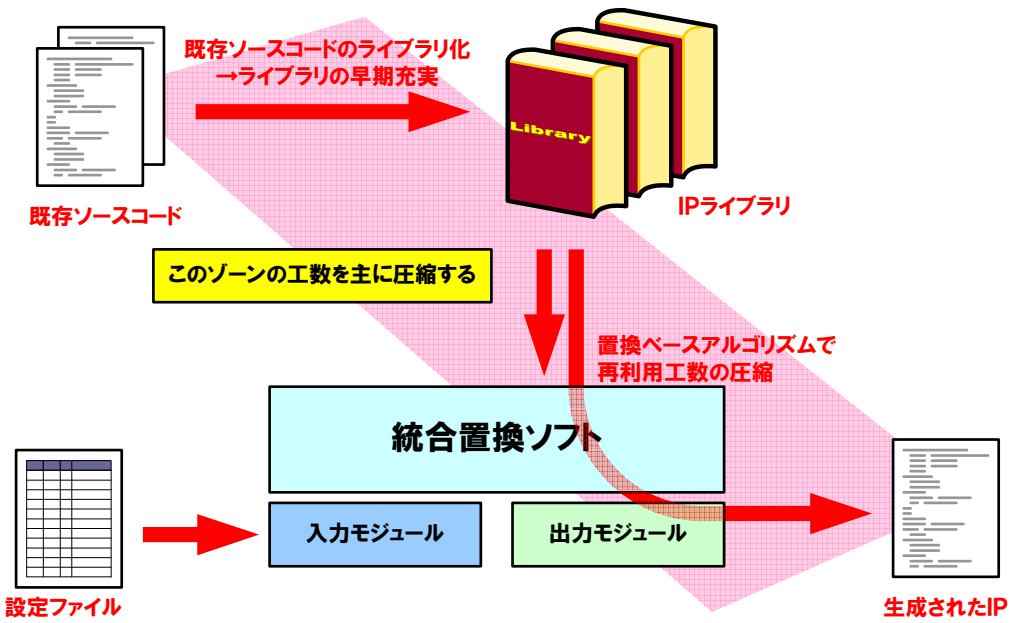
2-1-2-4-2 支援ツール

上述の通り、開発手順の改善・見直しによって種々の支援ツールが派生可能な状態となるが、本テーマで掲げた IP 開発に関わる問題点を解決するための支援ツール(統合置換ソフトウェア)を試験的に開発する。

統合置換ソフトウェアは、従来開発手順において作成したソースコード類をひな形として IP を生成するソフトウェアであり、以下の機能を有する。

表【1-1】. 2 統合置換ソフトウェアが目標とする機能

項目		説明
汎用性・柔軟性		
	ライブラリ	IP の目的ごとにライブラリとしてまとめ、それらを切り替えることで IP だけでなく、汎用的な記述や検証環境のひな形も生成可能とする。
開発工数の圧縮		
	既存ソースコードのライブラリ化	既存ソースコードからのライブラリ化をサポートし、ライブラリを早期に立ち上げ可能とする。
	置換ベースのアルゴリズム	大部分の IP は、既存の定型コードを元に、一部のパラメータを修正する程度で再利用可能な状態となる。また、従来はこの修正を手作業で行っていたために再利用工数が必要となっていたが、この作業を置換によって自動化することで再利用工数の圧縮を図る。
拡張性		
	任意言語での出力	設定の入力フォーマットを固定化せず、必要に応じて入力・出力モジュールを追加して多彩な生成を可能とする。 ターゲットとなる言語ごとに出力モジュールを用意し、将来の言語拡張・追加や合成ツールなど記述依存を持つツールへの対応を容易にする。



図【1-1】. 11 統合置換ソフトウェアの構成イメージ図

2-1-2-5　作成物 1.　C 言語設計フレームワーク「CDEF」

2-1-2-5-1 概要

CDEF は、本研究成果を元にした開発支援、プロジェクト管理、工程分析を主たる目的とし、短期的には開発支援面で、長期的にはフィードバックによる改善面で、生産性向上につなげていくことを目的とする。

形態としては、J2EE 準拠のアプリケーションサーバー上で稼動する WEB アプリケーションとした。

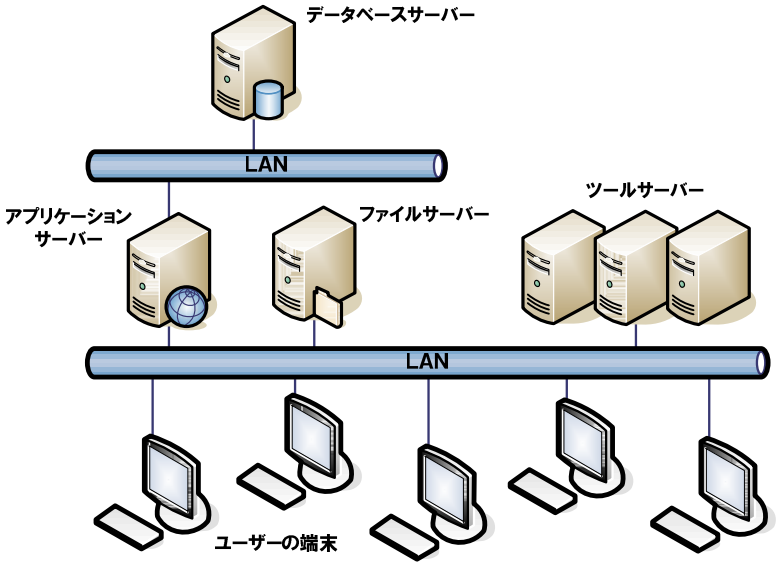
CDEF の名称は、C 言語設計フレームワーク (C-language DEsign Framework)の各頭文字に由来する。

2-1-2-5-2 システム構成

CDEF のシステムを構成する機器について、以下に示す。

表【1-1】. 3 システム構成要素一覧

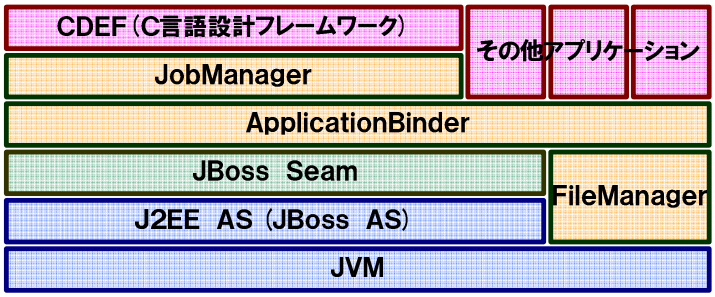
要素	機能
アプリケーションサーバー	CDEF が稼動するサーバー(下記例では WEB サーバーを兼ねる)
データベースサーバー	CDEF のデータベースが稼動するサーバー
ファイルサーバー	ユーザーが開発に使用するデータを格納するサーバー(各ツールサーバーにマウントする)
ツールサーバー	高位合成ツールなどを実行するためのサーバー
ユーザーの端末	ユーザーが作業を行う PC(WEB ブラウザで CDEF にアクセスし操作する)



図【1-1】. 12 システム構成例

2-1-2-5-3 アーキテクチャ構成

CDEF のアーキテクチャ構成を以下図に示す。



図【1-1】. 13 CDEF のアーキテクチャ構成図

表【1-1】. 4 CDEF の内製コンポーネントの機能一覧

要素	機能
CDEF	本体となる WEB アプリケーション。 プロジェクト管理、ドキュメント管理、開発支援、工程分析などの機能を持つ。
Application Binder	開発に直結する要素だけでは工程分析を十分に行うことは難しく、必要な情報を別途入力する必要が発生する。 そのため、外部システムとの連携や、新たなシステムの増築するための共通基盤として機能する箇所を、フレームワークとして分離し、拡張性を持たせている。
Job Manager	ジョブ毎の操作を統括し、ツールの自動実行、ジョブのロギングなどの機能を持つ。
File Manager	データベース化された文書を保管、管理するための機能を持つ。

表【1-1】. 5 CDEF の外部コンポーネント・ライブラリ等の名称・バージョン一覧

ツールおよびライブラリ	バージョン	機能
Java SE Development Kit	Version 6 Update 17	Java 言語開発キット
JBoss Application Server	Version 4.2.3	J2EE 準拠のアプリケーションサーバー
JBoss Seam	Version 2.1.2	J2EE アプリケーションフレームワーク
HSQLDB	Version 1.8.0	データベース

2-1-2-5-4 特徴

CDEF は、以下の 3 カテゴリについてそれぞれの特徴を持つ。

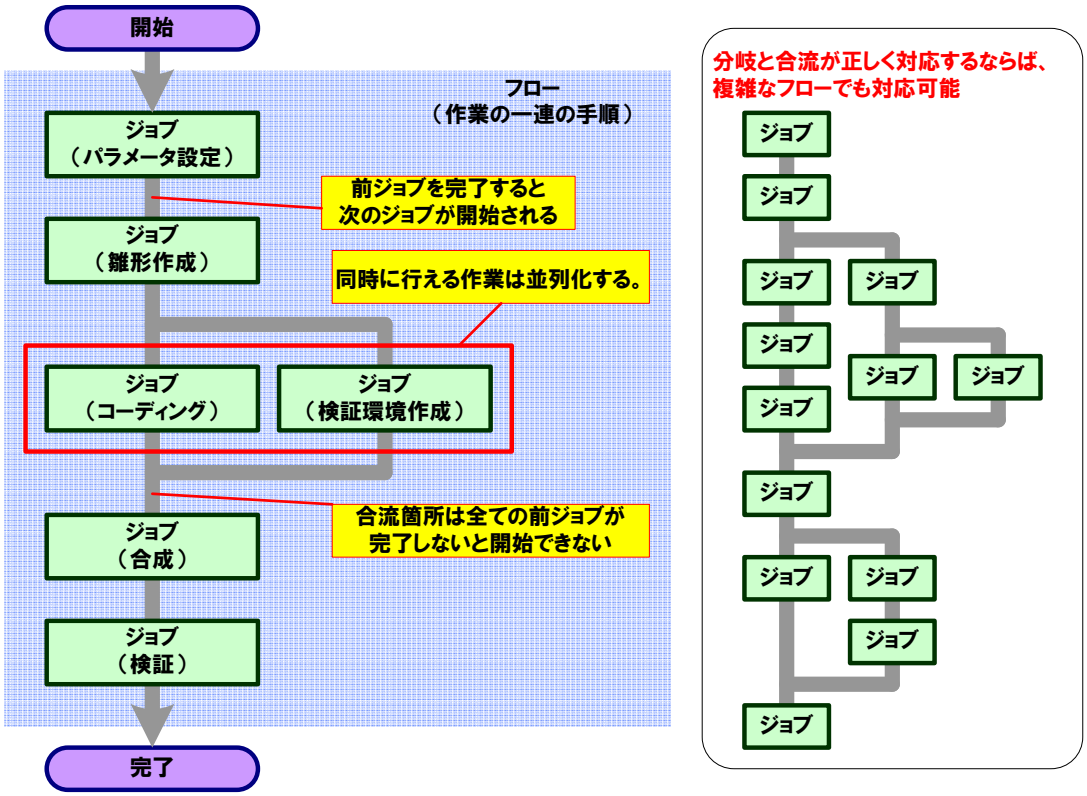
表【1-1】. 6 CDEF の特徴一覧

カテゴリ	特徴
開発支援	フローに沿った業務遂行のサポート
	ツールの自動実行
	マニュアル、規定、手順書および標準規格類のデータベース化
プロジェクト管理	進捗管理
工程分析	問題点抽出のための情報収集とその視覚化

以下、各特徴について説明する。

2-1-2-5-5 開発支援:フローに沿った業務遂行のサポート

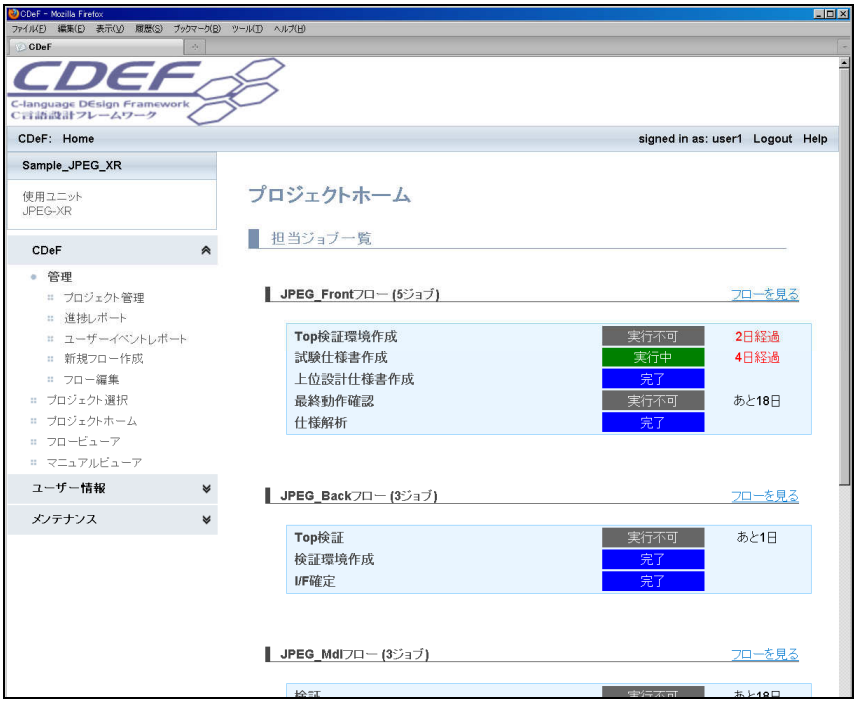
CDEF では、ソースコードの作成やツールの実行など、ユーザーが行う作業や自動実行される作業のひとつをジョブとして定義し、このジョブの連なりをフローとして管理することで手順と作業を関連づけることにした。



図【1-1】. 14 フローとジョブの関係

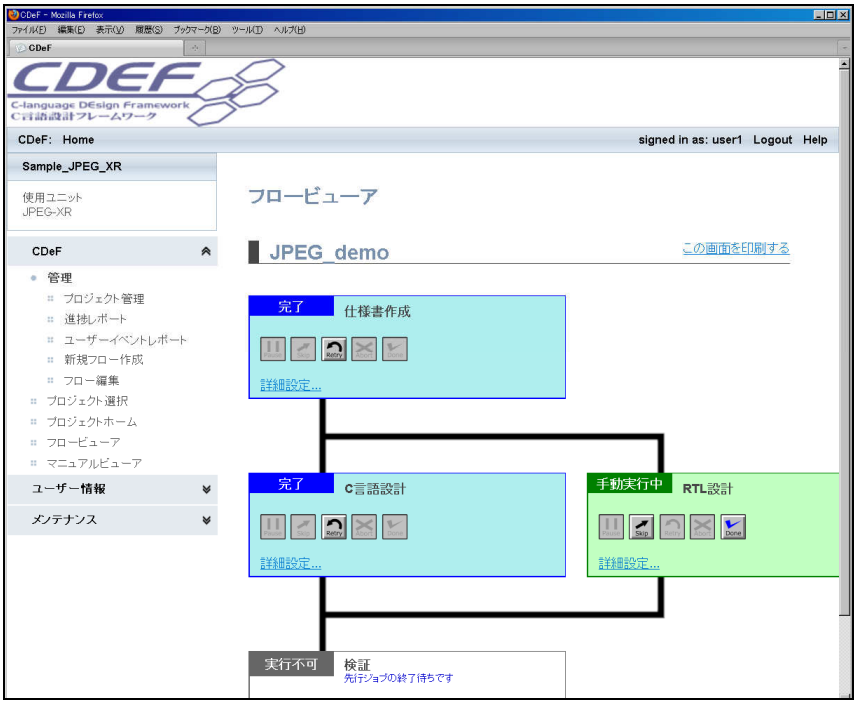
ユーザーはプロジェクトホーム画面にて、自身が担当しているジョブの状況、納期を確認できる。

そのジョブが着手可能かどうかも含めて表示されるため、ユーザーが次に取りかかるべき・取りかかることができるジョブを即時に把握することが可能となる。



図【1-1】. 15　CDEF:プロジェクトホーム画面

ユーザーはフロービューアにて、担当ジョブおよびその周辺の状況を確認することができる。これにより担当ジョブの納期だけなく、周辺の状況を踏まえた優先度判断が可能となる。

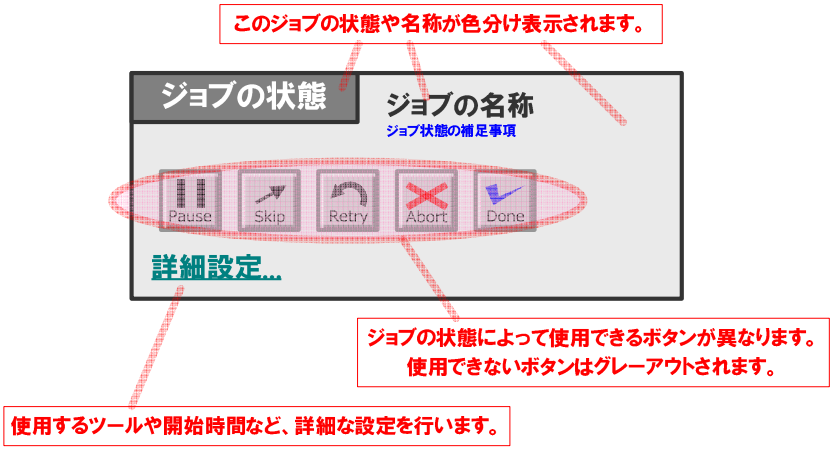


図【1-1】. 16　CDEF:フロービューア画面

表【1-1】. 7　ジョブの各状態

色	状態	意味	ユーザーに促す操作
	実行不可	前ジョブが完了していないため、このジョブを開始できない。	----
	スキップ	このジョブはフロー実行から除外されている。	スキップの解除
	一時停止	フロー実行がこのジョブに到達した時点で、フロー実行を一旦停止する。	一時停止の解除
	予約中	自動実行ジョブが、指定された開始時間まで待機している。	----
	手動実行中	このジョブは、手動ジョブとして開始しており、完了していない。	ジョブで指定された作業
	自動実行中	このジョブは、自動実行ジョブとして開始しており、完了していない。	----
	完了	このジョブは、エラーが発生せずに終了した。	----
	エラー	このジョブは、エラーが発生した。	原因分析と再実行

また、フロービューア上からジョブに対して直接操作し、フローの最新状態を確認しながら業務を遂行できる形とした。



図【1-1】. 17　ジョブ表示の見方

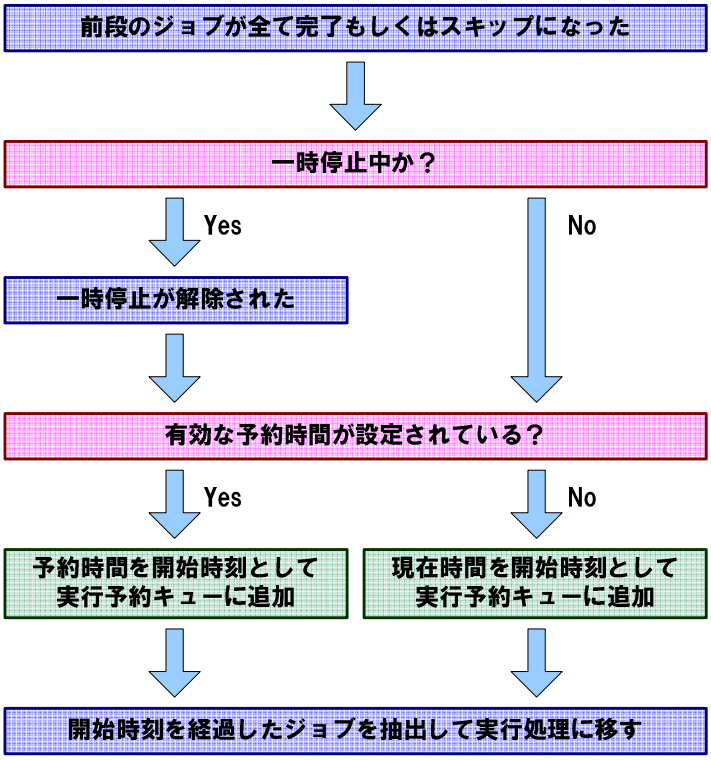
表【1-1】. 8　ジョブ表示内のボタンの機能

ボタン	ボタン押下によって行われる操作、機能
Pause 一時停止	ジョブの一時停止状態を設定もしくは解除する。一時停止が設定されているジョブは、前ジョブが完了したタイミングでは実行されず、一時停止が解除されるのを待つ。
Skip スキップ	ジョブのスキップ状態を設定もしくは解除する。スキップが設定されているジョブは実行さない。前ジョブが完了状態の場合、次のジョブが実行可能になる。
Retry リトライ	ジョブを再実行する。 この操作が行われたジョブの下流のジョブは、再実行のため実行不可の状態に戻る。
Abort 中止	ジョブの予約、実行を中止する。 予約中のジョブは一時停止扱いとなり、実行中のジョブはエラーとなる。
Done 完了	ジョブを完了状態にする。 ユーザーがそのジョブで行うべき作業が完了したタイミングで操作し、次のジョブが実行可能となる。

2-1-2-5-6 開発支援:ジョブの自動実行

ユーザー自身が行わなければならない作業に集中できるよう、各種ツールを利用して自動実行が可能な場合は、自動的に実行される方法を検討した。これによりユーザーが各種ツールを直接操作してツールを実行する工数を削減できる。

自動で実行されるジョブの実行タイミング、判定を以下図に示す。



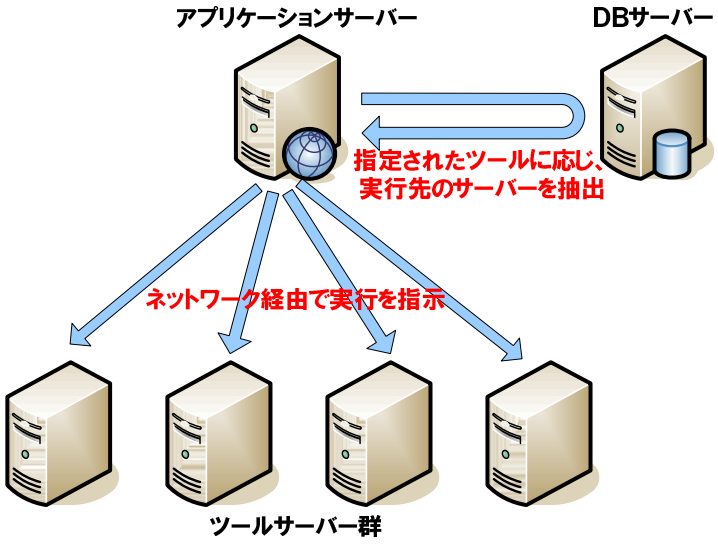
図【1-1】. 18 ジョブの実行タイミング判定フロー

■ツールサーバーの効率利用

ツールサーバーには、ハードウェア設計のためのツールが多数インストールされることを想定している。また、開発の条件としてツールのバージョンが指定されることも起こりうる。つまり、ツール種別だけでなくバージョン等も含めた使い分けが必要である。各担当者は開発条件に従って 指定ツールおよびバージョンが使用可能なツールサーバーを担当者自身が検索し、そのツールサーバーに対して個別に操作を行う必要があった。

ツールの管理情報を DB サーバー上に集約し、Job Manager によりユーザーが求める条件のサーバーに処理を投入する方式とした。これにより、ツールサーバー毎の差異を担当者が意識する必要が無くなり、使用頻度の低いツール、バージョンを用いる場合でも、既にインストール済みのツールサーバーが存在する場合、新たなインストールや設定なしで、容易に用いることができるようにした。

また、各ジョブから提供される占有時間の見積もり情報を元に、負荷予測を行い、ツールの実行時間の短縮につなげていく予定である。



図【1-1】. 19 指定ツール・バージョンが用意されたツールサーバーへ自動的に処理を投入

2-1-2-5-7 開発支援:マニュアル、規定、手順書および標準規格類のデータベース化

任意のプロジェクトにおいて必要なドキュメントが発生した場合に、そのドキュメントがユニット、フロー、IP、ツールに関連付けてシステム登録することができるようにすることで、同様のユニット等を用いるプロジェクトにて既存ノウハウとして活用することができると考えた。

プロジェクトやフローの作成時の情報を元に、設計に必要となる手順書、マニュアル等は、マニュアルビューア上に列挙されるため必要なときに参照でき、ユーザーがドキュメント類を検索する手間を省略することができる。

また、インターネット上の情報サイトについても、データベースにサイトアドレスを登録できるようにした。内製のドキュメントだけでなく外部の有用な情報も利用できるようにし、外部の新規ノウハウをドキュメント内容の補完に用いるといった効果が期待できる。

図【1-1】. 20 CDEF:マニュアルビューア画面

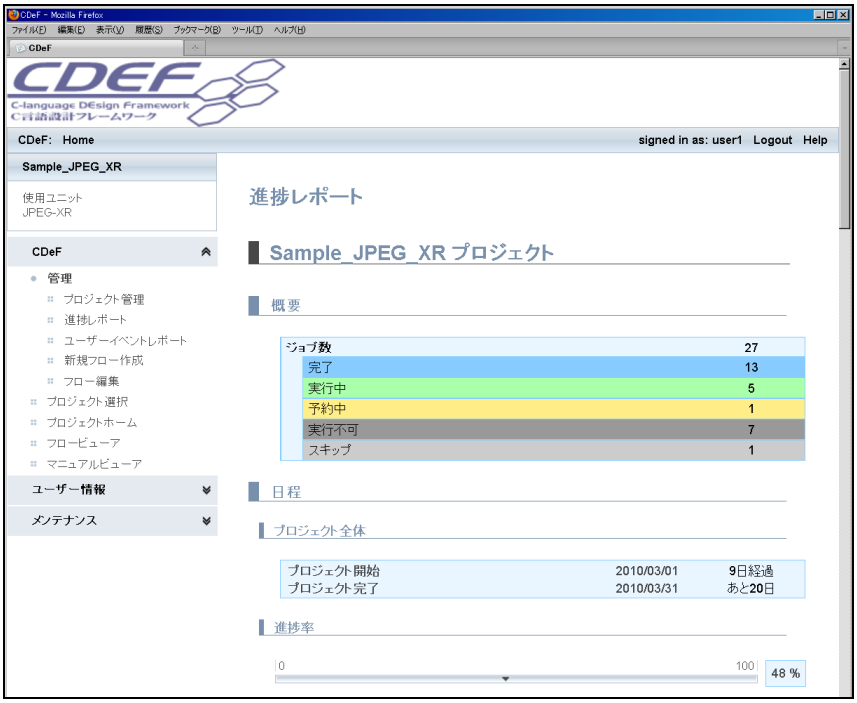
ドキュメント・情報の集約や蓄積を行うことにより、以前に他のユーザーが取得した情報を再度取得にいくといった無駄な作業を抑制し、発生しやすい問題の情報を事前に把握・共有することで不具合による手戻りを避けるといった効果が期待できる。

2-1-2-5-8 プロジェクト管理

開発では、ドキュメント作成、実装、検証など生産に直結した工程ではない作業も発生する。進捗確認、作業割り当てなど管理、進捗会議等でも少なからず工数が発生する。

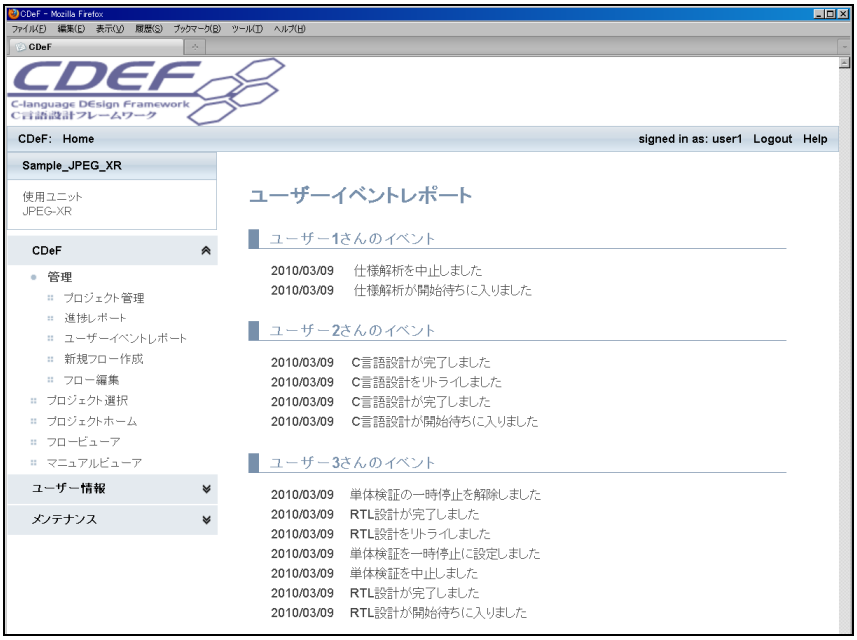
これらの管理工数がプロジェクト全体工数に占める割合を集計したところ、全体の約 10%であった。元データの判定基準が各担当者に依存しているため、この約 10%がすべて進捗管理等であるとは限らないものの、実際の開発工数の合間にも管理工数が潜伏しているという実情を踏まえるとやはり相応の工数が発生していることが推測される。この工数を削減し開発に専念できる環境を作るため、管理そのものの手間を抑えるリアルタイムな進捗管理機能を検討した。

各作業がジョブ単位で扱われ、フローによって工程の流れを制御できれば、リアルタイムな進捗が管理できると考えた。進捗管理の要素としては、各作業の進捗率、各作業の期日が上げられる。これらをジョブのエンティティに含めることによって、プロジェクトの各メンバーが進捗情報を入力することで部分的な進捗から全体的な進捗までを表示・確認可能とした。



図【1-1】. 21 CDEF:進捗レポート画面

また、プロジェクトの各メンバーがジョブに対して行った操作を記録しておき、それを一覧表示することで各メンバーの作業状況が確認可能となるよう検討した。



図【1-1】. 22　CDEF:ユーザーイベントレポート画面

ジョブごとの計画値に基づいて負荷の比率を考慮し、プロジェクト全体の進捗を把握できるようにする。この進捗は労務管理アプリケーション(※予定)と連動させることにより、開発業務だけでなく、全社的な業務改善とすることが可能であると考える。

2-1-2-5-9 工程分析

生産性向上は継続的に実施し続けなければならない。もし一過性のもので終わらせてしまうと、技術の成長速度についていけない事態になることは容易に想像される。継続的な生産性向上の重要なファクターとして、工程や手順の分析に着目する。

フローのボトルネックになっている工程や、非生産的な作業に陥っている工程を迅速、且つ的確に特定および抽出できなければ、対策を講じても十分な効果が期待できない。

CDEF 上では、工程を 1 つのジョブとして取り扱うことができるため、ジョブの情報を収集することで上記問題に対する対策が行えると判断し、以下 3 つのアプローチを検討した。

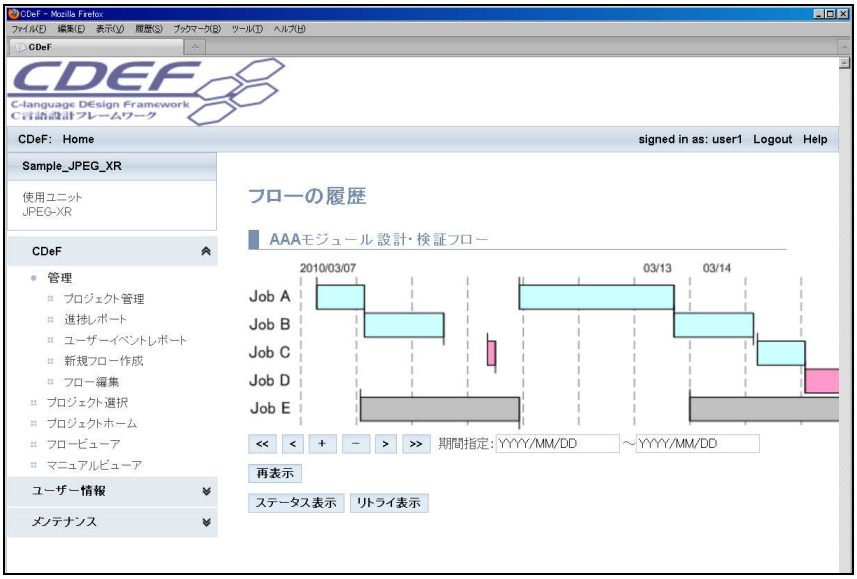
1) フローの実行状況を可視化
2) 特定プロセスの抽出
3) プロジェクトにおける傾向抽出

以下、これら 3 つのアプローチについて個別に述べる。

■フローの実行状況を可視化

フロー内の問題のあるジョブを明確にするため、フローの実行状況を可視化してユーザーに提供する。

縦軸ジョブ、横軸時間で実行開始から実行終了までが把握できるようグラフ化する。



図【1-1】. 23　CDEF:フロー履歴画面

ユーザーはこのグラフから、手戻りの発生状況、工程上の非常に時間のかかったジョブを視覚的に判断することができる。グラフ表示は、ジョブのステータス変更時に記録されたたログを元に自動生成する。各ジョブのステータスの遷移について以下表に示す。左端が変更前のステータス、上端が変更後のステータスを示す。

表【1-1】. 9 ジョブステータス遷移表

変更前＼変更後	実行不可	一時停止	予約中	手動実行中	自動実行中	完了	エラー	スキップ	設定変更のみ
実行不可		前ジョブ完了	前ジョブ完了	前ジョブ完了				スキップ設定	一時停止設定
一時停止	上流ジョブ再実行		一時停止解除	一時停止解除				スキップ設定	
予約中	上流ジョブ再実行	中止			自動ジョブ投入				
手動実行中	上流ジョブ再実行					完了		スキップ設定	
自動実行中	上流ジョブ再実行					自動ジョブ完了	中止 自動ジョブエラー		
完了	上流ジョブ再実行	自動ジョブリトライ		手動ジョブリトライ					
エラー	上流ジョブ再実行	自動ジョブリトライ		手動ジョブリトライ				スキップ設定	
スキップ	スキップ解除	スキップ解除		スキップ解除					

ステータス遷移表の各表記の意味は以下の通り。

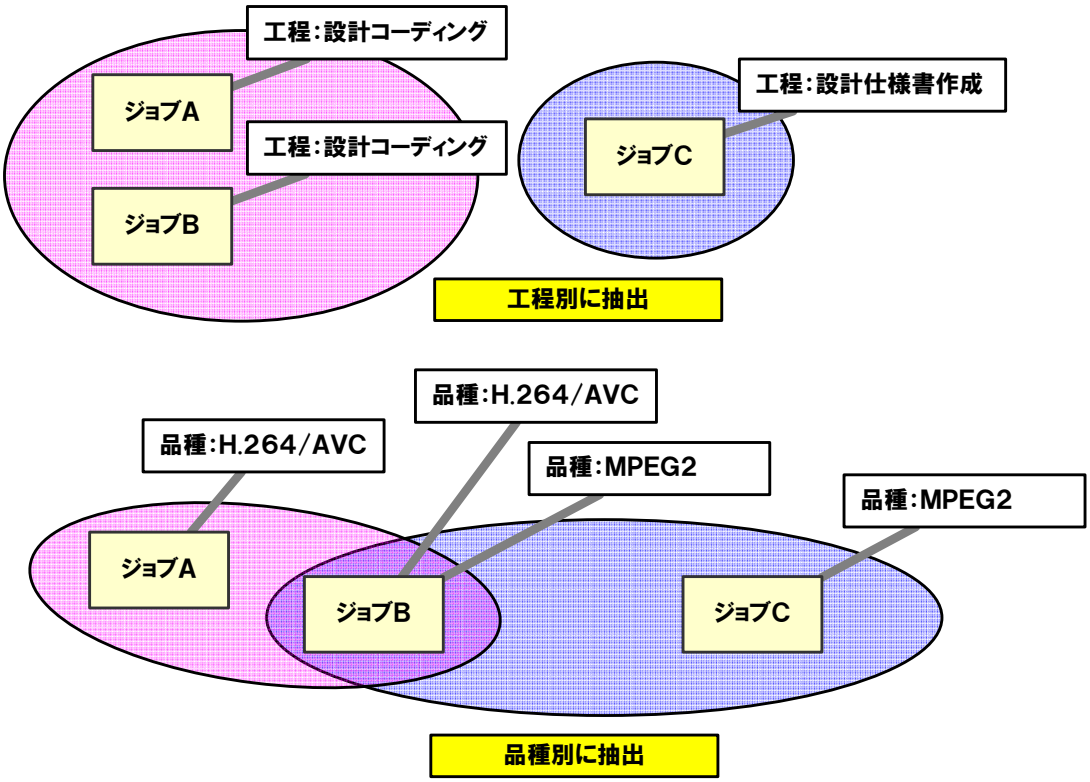
表【1-1】. 10 ジョブステータス遷移表の項目一覧

項目	意味
青字の項目	ユーザー都合による操作
赤字の項目	システム都合による操作
上流ジョブ再実行	自ジョブよりも上流にあるいずれかのジョブでリトライ、またはスキップ解除が発生した場合。
前ジョブ完了	自ジョブの直前にあるすべてのジョブが、完了またはスキップとなった場合。
自動ジョブ投入	自動実行対象のジョブが、指定された開始時刻となったために開始した場合。
自動ジョブ完了	自動実行対象のジョブが終了し、エラーが発生しなかった場合。
自動ジョブエラー	自動実行対象のジョブが終了し、何らかのエラーが発生した場合。
スキップ解除	フロー実行から除外(スキップ)されていた自ジョブを、フロー実行に復帰させる場合。
スキップ設定	自ジョブをフロー実行から除外する場合。
一時停止解除	自ジョブのところで停止しているフロー実行を再開させる場合。
一時停止設定	自ジョブのところでフロー実行を一旦停止させる場合。
自動ジョブリトライ	完了またはエラー状態にある自動実行対象のジョブを再実行させる場合。
手動ジョブリトライ	完了またはエラー状態にある非自動実行対象のジョブを再実行させる場合。
完了	非自動実行対象のジョブがユーザーによって完了と見なされた場合。
中止	自動実行中のジョブを強制停止させた場合。

このログからジョブごとに時系列の履歴を抽出し、ジョブの開始を起点として、次の履歴を終点に割り当てる。終点となった履歴から、ジョブがどのような状況で終了したか判定できるため、視覚的に確認しやすいよう色分けして表示する。

■特定プロセスの抽出

単一のジョブに対し、キーワードとなる項目は複数存在する。これらの項目が増減する可能性を考慮し、各ジョブに対して汎用的なキーワードをタグとして設定できる形とした。これにより、分析目的に応じてジョブを抽出可能となり、問題の多いジョブの特定や IP 化による生産性向上の見積を可能とする。



図【1-1】. 24 ジョブごとにタグの割り当て

■プロジェクトにおける傾向抽出

前述の「特定プロセスの抽出」と同様の手法で工程ごとにデータを抽出しスコアを計算し、工程を軸としてグラフにプロットさせることを考えた。まずスコア計算に含める要素として、以下事項を検討した。

スコアの算出は、該当工程に属するジョブを起因とする不具合数の総和を元に行う。

進捗の遅延

納期が設定されたジョブを納期通りに完了できているかをスコア化する。いずれのジョブもリトライが可能であるため、単一ジョブに対して複数回の完了が発生しうる。どの完了日時を以て計画との評価に使用するかは、そのときの設計データの品質や顧客要求の変遷などと照らし合わせなければならないため、CDEF 単体で特定することは困難である。

簡易化のため、単一ジョブで最初に発生した完了日時を評価対象と定義する。これは、計画当初の顧客要求およびチェック条件に合致したものであるため、計画との差異という意味で使用しやすいデータである。このデータを使用し、ジョブ納期との遅延方向の差分を元にスコアを計算する。また、単一ジョブで最後に発生した完了日時との差が非常に大きい場合は、顧客要求の変遷による仕様変更の発生や、周辺ジョブでの異常発生を示している可能性が高い。これも別スコアとして考慮する必要があると考える。

作業計画のズレ

当初計画された作業時間(計画時間)と実際にかかった作業時間(実績時間)の差から、効率の良し悪しが判定可能と考えるため、スコア化する。その工程に含まれるジョブの計画時間の総和と、実績工数の総和からスコアを計算する。

手戻りの発生頻度

リトライにより、実行回数が極めて大きな値となっているジョブについて、手戻りの発生が多い工程としてスコア化する。その工程に含まれるジョブの実行回数の最大値を元にスコアを計算する。

次に、これらの 3 つの要素をどう複合して表示させるかについて検討した。

それぞれの要素は係数値を変更することにより、どの要素を重視して表示するかを変更可能な式とする。その上で、3 つのスコアの和と積を取る方法が考えられるが、ここでの目的は特異点の視覚化にあるため、最大値、最小値がはっきりと出る積をとる方法が良いと考える。

以上を踏まえ、CDEF の工程分析画面を実装した。

図【1-1】. 25　CDEF:工程分析画面

2-1-2-6　作成物 2. 統合置換ソフトウェア「FGen」

2-1-2-6-1 概要

FGen は、先述の通り支援ツールの一つで、置換ベースのアルゴリズムを採用している。置換によって IP ライブラリからのファイル生成を行う。

FGen の名称は、ファイル(File)の生成器(GENerator)であることに由来する。

2-1-2-6-2 特徴

- 入力モジュール(論理)と出力モジュール(言語)の機能分担
- 種々の利用シーンを想定したファイル生成シーケンス
- 補助ソフトウェア TGen(Template GENerator)を用いて、ライブラリのひな形を生成可能

2-1-2-6-3 構成

FGen はスクリプト言語 Perl で記述されており、コマンドラインでの操作を前提としている。

設定ファイルやライブラリなど、XML フォーマットでのデータを利用するため、XML ライブラリ LibXML および Perl モジュール XML::LibXML が必要となる。

2-1-2-6-4 使用方法

■環境変数の設定

FGen では、環境変数 FGEN_HOME を必要とする。また、各 FGen ツールへのパス指定も必要となる。

表【1-1】. 11 FGen に必要な環境変数

環境変数	設定値	意味
FGEN_HOME	(FGen のインストールディレクトリ)	FGen のインストールディレクトリ
PATH	$FGEN_HOME/bin を追加	FGen 実行ファイルへのパス追加

```
> setenv FGEN_HOME /opt/eda/primegate/fgen
> setenv PATH  $FGEN_HOME/bin:$PATH
```

図【1-1】. 26 環境変数の設定

■初期設定ファイルの生成

使用するライブラリを指定し、FGen オプション「-create」を指定して初期設定ファイルを生成する。

生成される初期設定ファイルは現時点では「setting.xml」に固定される。

この初期設定ファイルに対して後述する編集作業を行い、IP への指示内容が記述された設定ファイルとなる。

```
> mkdir work
> cd work
> ls

> fgen.pl -create libregister.xml
> ls
setting.xml
>
```

図【1-1】. 27 初期設定ファイルの生成

上記例で使用した、FGen へのオプション指定は以下のようになっている。

表【1-1】. 12　図【1-1】. 27 で使用した FGen のオプション解説

入力コマンド	設定項目	動作
fgen.pl	FGen 本体	
-create	初期設定モード	初期設定ファイルを setting.xml として出力する
libregister.xml	初期設定モードで使用するライブラリ	出力される初期設定ファイルはレジスタ IP 専用となる

■設定ファイルの編集

設定ファイルの内容を編集し、再利用する IP の属性を指定する。

以下に挙げる各設定項目のうち、BASE_CONFIGURATION 項以外はライブラリによって構成が異なるため、ここではレジスタライブラリ(libregister.xml)から生成された初期設定ファイルを例に挙げる。

<category name="BASE_CONFIGURATION"/>

BASE_CONFIGURATION 項では、設定ファイル自身の設定が記述されている。ユーザーは特に修正する必要はない。

<category name="COMMON"/>

COMMON 項では、生成される IP の設定のうち、ユーザーに関する設定項目が記述されている。

ユーザーは、以下の赤文字の部分を修正する。

```
<category type="HASH" name="COMMON">
    <!-- 生成されるファイルに関する設定 -->

    <item name="PJNUM" remark="プロジェクト番号">
        <item name="VALUE">■プロジェクト番号を入力してください</item>
    </item>
    <item name="PJNAME" remark="プロジェクト名">
        <item name="VALUE">■プロジェクト名を入力してください</item>
    </item>
    <item name="DESCRIPTION" remark="タイトル">
        <item name="VALUE">■このコードのタイトルを入力してください</item>
    </item>
    <item name="AUTHOR" remark="作成者">
        <item name="VALUE">■このコードの作成者名を入力してください</item>
    </item>
    <item name="COPYRIGHT" remark="著作権">
        <item name="VALUE">(C) PrimeGate Ltd.</item>
    </item>
    <item name="REVISION" remark="バージョン">
        <item name="VALUE">0.01</item>
    </item>
    <item name="DATE" remark="作成日">
        <item name="VALUE">■このコードの作成日を入力してください</item>
    </item>
    <item name="REMARK" remark="">
        <item name="VALUE">new</item>
    </item>
    <item name="MODULENAME" remark="モジュール名">
        <item name="VALUE">■モジュール名を入力してください</item>
    </item>
</category>
```

図【1-1】. 28 基本情報の修正

<category name="HOST"/>

HOST 項では、生成される IP 設定のうち、ホスト側ポートに関する設定項目が記述されている。

ユーザーは、必要があればポート名およびデータバス幅、アドレスバス幅を修正する。

<category name="REGLIST" type="LIST"/>

REGLIST 項ではレジスタマップの指定を行う。

初期設定ファイルでは、例として 32 ビットリード/ライト型のレジスタを 2 つ(0x0000, 0x0004)定義してある。ユーザーは、生成するレジスタマップに従って REGLIST 項に追加していく。

```
<category name="REGLIST" type="LIST">
     <!-- レジスタマップに関する設定 -->

     <item>
          <item name="REGNAME">SCRATCH1</item>
          <item name="REMARK">汎用 1</item>
          <item name="PORT">o_scratch1</item>
          <item name="TYPE">WR</item>
          <item name="ADDR">0</item>
          <item name="MSB">31</item>
          <item name="LSB">0</item>
     </item>

     <item>
          <item name="REGNAME">SCRATCH2</item>
          <item name="REMARK">汎用 2</item>
          <item name="PORT">o_scratch2</item>
          <item name="TYPE">WR</item>
          <item name="ADDR">4</item>
          <item name="MSB">31</item>
          <item name="LSB">0</item>
     </item>

</category>
```

図【1-1】. 29 レジスタマップの指定例

■ファイルの生成

編集された設定ファイルを用いて、FGen でレジスタ RTL を生成する。

```
> ls
setting.xml
> fgen.pl setting.xml -lang vlog -o . -f
> ls
setting.xml       smp_reg.v
>
```

図【1-1】. 30 FGen を用いたレジスタ RTL の生成例

上記例で使用した、FGen へのオプション指定は以下のようになっている。

表【1-1】. 13　図【1-1】. 30 で使用した FGen のオプション解説

入力コマンド	設定項目	動作
fgen.pl	FGen 本体	
setting.xml	設定ファイルの指定	
-lang vlog	出力モジュールの指定	Verilog HDL 用出力モジュールを使用して生成する
-o .	出力先ディレクトリの指定	カレントディレクトリに出力する
-f	ファイルが既に存在する場合の強制上書き	

ここまでの作業によって、レジスタ IP を使用して新たなレジスタ RTL を生成することができる。
以下、生成されたレジスタ RTL の例(途中略)を示す。

```
// $Id$
// ********************************************************************************************
//  File Name   : smp_reg.v
//  Description : sample register
//  Author      : jan paul belmont
//  Copyright   : (C) PrimeGate Ltd.
//  Project     : xxxxxxxx sample project
//  Edit_history:
//  Rev.    yyyy.mm.dd  by           description
//  ------+------------+------------+-------------------------------------------------------
//  0.01    2009/13/32  jan paul belmont new
// ********************************************************************************************
//--------------------------------------------------------------------------
module smp_reg (
     clk,
     xrs,
     i_addr,
     i_cs,
     i_rxw,
     i_wdat,
     o_rdat,
     o_rdy,
     o_scratch1,
     o_scratch2
);
     input           clk;
     input           xrs;
     input[9:2] i_addr;

(略 ------------------------------------------------------------)
     always @( posedge clk or negedge xrs ) begin
          if (~xrs) begin
               r_rdat_sel[31:0] <= 32'h0;
          end
          else begin
               if (i_cs && i_rxw) begin
                    r_rdat_sel[31:0] <= 32'h0;
                    case ({i_addr[9:2], {2{1'b0}}})
                         10'h0 : begin
                              r_rdat_sel[31:0] <= r_SCRATCH1_[31:0];
                         end
                         10'h4 : begin
                              r_rdat_sel[31:0] <= r_SCRATCH2_[31:0];
                         end
                         default : begin
                              r_rdat_sel[31:0] <= 32'h0;
                         end
                    endcase
               end
               else begin
                    r_rdat_sel[31:0] <= 32'h0;
               end
          end
     end

(略 ------------------------------------------------------------)

endmodule
//--------------------------------------------------------------------------
```

図【1-1】.31 作成されたレジスタ RTL 例

■IP ライブラリの開発

新たに IP ライブラリを開発する場合は、TGen を使用してライブラリのひな形を取得することで、早期にライブラリを立ち上げることが可能となる。

```
> cd work
> ls
template/
> ls template                                          既存ソースコードの内容
bin/  sv/  svh/  tbench/

> tgen.pl -lang vlog -lib svsimenv -gh -gl ./template  TGen の実行
tgen.pl: generate template: libsvsimenv_tmp_vlog.xml...
tgen.pl:     | language: vlog
tgen.pl:     | library : libsvsimenv
tgen.pl: searching file to include....
tgen.pl:  +- found ./template/svh/
tgen.pl:  +- found ./template/svh/c_ahbmaster.svh
tgen.pl:  +- found ./template/svh/constval.svh
tgen.pl:  +- found ./template/svh/includeall.svh
tgen.pl:  +- found ./template/tbench/
tgen.pl:  +- found ./template/tbench/t_basic_all.sv
tgen.pl: generate template-holder: libsvsimenv_tmp.xml...
tgen.pl:     | library : libsvsimenv
tgen.pl:     | add-to  : libsvsimenv_tmp_vlog.xml for vlog
tgen.pl: generate library: libsvsimenv.xml...
tgen.pl:     | library : libsvsimenv

> ls -lgG
-rw-rw-rw- 1   4129 xx 月 xx xx:xx libsvsimenv.xml          ライブラリ本体
-rw-rw-rw- 1    173 xx 月 xx xx:xx libsvsimenv_tmp.xml      テンプレートホルダ
-rw-rw-rw- 1 148273 xx 月 xx xx:xx libsvsimenv_tmp_vlog.xml Verilog HDL 用テンプレート
drwxrwxrwx 6   4096 xx 月 xx xx:xx template/

>
```

図【1-1】. 32 TGen による、既存ソースコードからのライブラリひな形取得例

上記例で使用した、TGen へのオプション指定は以下のようになっている。

表【1-1】. 14　図【1-1】. 32 で使用した TGen のオプション解説

入力コマンド	設定項目	動作
tgen.pl	TGen 本体	
-lang vlog	ライブラリ言語の指定	Verilog HDL 用として生成する
-lib svsimenv	ライブラリ名の指定	ライブラリ名は libsvsimenv
-gh　-gl	テンプレートホルダ，ライブラリも作成する	
./template	既存ソースコードを納めたディレクトリ	

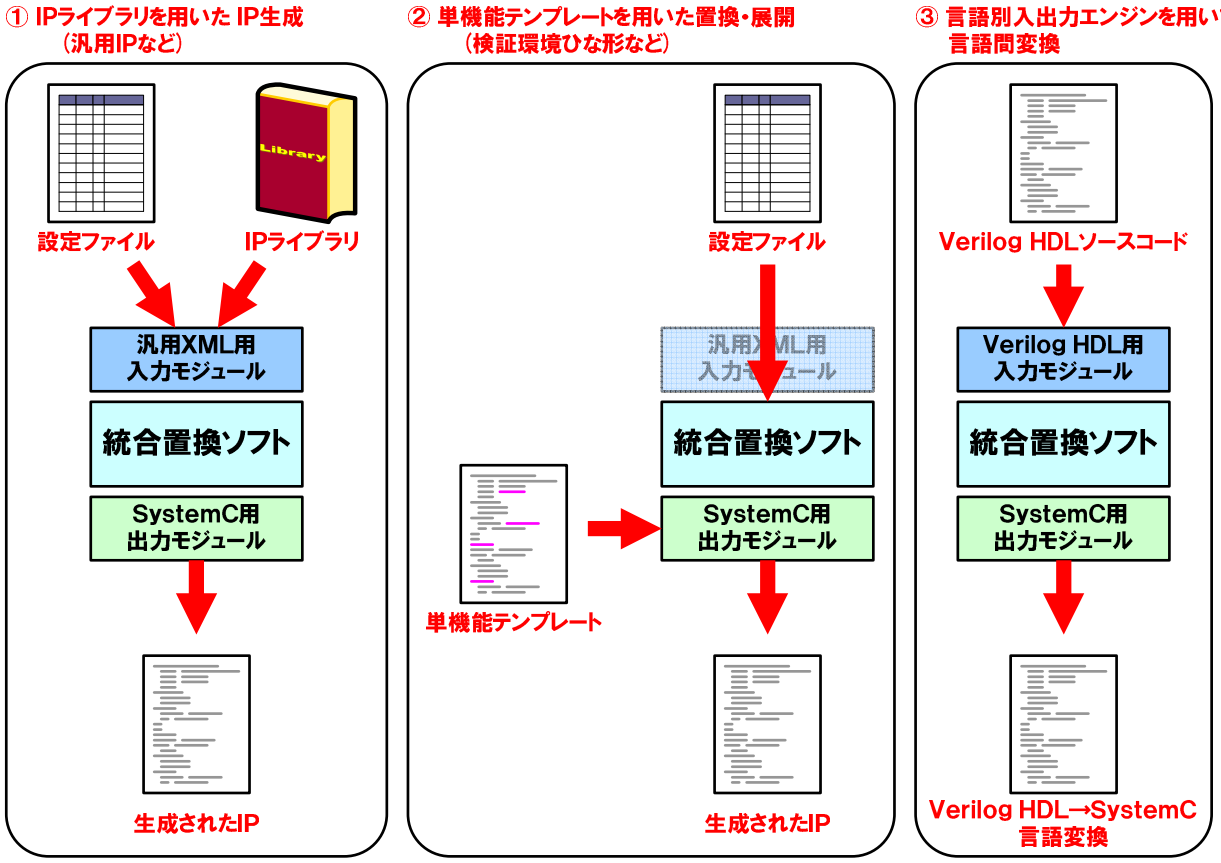
これによって、ライブラリ本体、テンプレートホルダ、Verilog HDL 言語用テンプレートの 3 ファイルが取得できる。ユーザーはこのファイルに対して修正を行い、IP ライブラリとして調整する。

調整が完了した IP ライブラリは、FGen インストールディレクトリ下にあるライブラリディレクトリに保存することで、以降の開発作業にて、先述のレジスタ IP を使用する場合と同様に使用することができる。

2-1-2-6-5 応用

本テーマで開発した統合置換ソフトウェアFGenは入出力モジュールが独立した構造となっているため、これらのモジュールを追加・増設することで、先に述べた IP 生成以外にも単純な記述ひな形や既存設計データの形式変換、言語間変換などのケースにも利用できる。

現時点の FGen は、入力設定ファイルおよびライブラリに沿った論理生成機能と、定型化された回路を生成するテンプレート機能のみを備えている。上述の論理エンジン部分は、この入力設定ファイルに特化した解析ルーチンで構成されているため、任意の変換元記述言語に対応した論理エンジンおよび変換先の記述言語に対応した言語エンジンを用意することで、言語間変換や合成ツール間の記述形式変換を行うことも可能となる。



図【1-1】. 33　FGen の応用例

2-1-2-7　生産性計測

本テーマの生成物によってもたらされる生産性について、以下の項目について数値の抽出を行った。

- CDEF の使用による、業務遂行効率
  - 先に述べたとおり、CDEF のホーム画面ではユーザーが担当するジョブの一覧が表示でき、どのジョブを優先して行うべきかが即時に把握できるようになっている。これは、業務遂行全般において工程間の空走・無駄を排除することに効果があると言える。
  - また、担当ジョブやフローに関連するドキュメントも逐次参照できることは、同様にドキュメントの検索に費やされる時間を排除することが可能と言える。
  - これらの最適化・集約による無駄の排除を生産性の向上として取り上げる。
  - ただし、CDEF 開発時点では他テーマで開発されているユニットの適用例がなく、また擬似的な状況でユニットを適用したとしても「無駄な工程」を再現することは難しいと考え、本項目は予測データとして抽出を行う。

- FGen の使用による IP 生成に関する工数の圧縮
  - CDEF の使用によって、反復的なフィードバックを行い、開発手順およびユニットを改善するというサイクルが自律的に発生することになる。この改善の結果、十分に詳細化された開発手順を元に自動化ツールを派生させることも CDEF の役割の一つである。
  - 本研究テーマにおいて、自動化ツールが派生したという想定で FGen を開発したが、この FGen による工数の圧縮を生産性の向上として取り上げる。
  - LSI 設計においてほぼすべての回路に搭載されるレジスタ機能について、今回開発したレジスタ IP を使用して以下のケースを比較する。
    - FGen を用いた IP 開発の工数（＝初回利用に係る工数）
    - FGen を用いた IP 展開の工数（＝再利用に係る工数）
    - 同規模のレジスタ機能を、既存設計データから複製・変更した場合の工数
      (既存設計データを疑似 IP とみなし、FGen を用いずに再利用するという想定)
    - 同規模のレジスタ機能を、手作業で設計した場合の工数

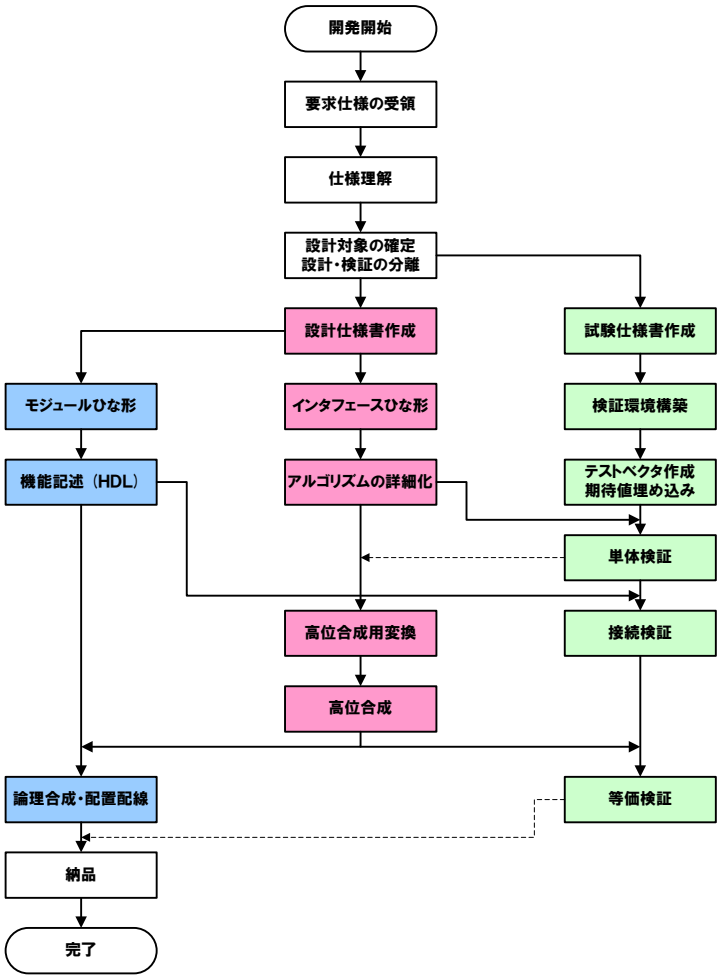
- CDEF＋マニュアル・手順・ユニット・支援ツール等を連結した生産性向上値
  - 概要項にて挙げた生産性向上目標値：10 倍については、CDEF で使用する開発手順・ユニットそのものが他テーマの生成物となっているため、詳細は各テーマを参照いただきたい。
  - 本項では、上記「業務遂行効率」と同様の条件を設定し、連結時の生産性の予測を行った。

2-1-2-7-1 業務遂行効率

業務遂行効率は、従来手順にて発生していた工程間の空走期間について、CDEF を使用することで排除できた度合いについて検討する。

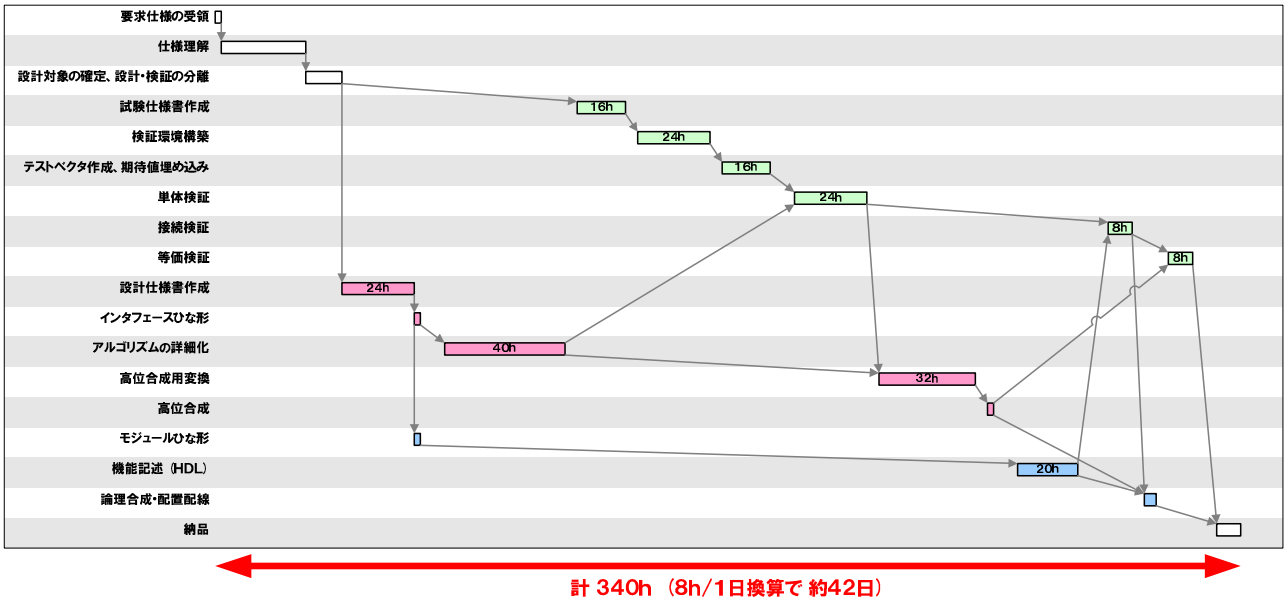
まず、予測のための前提として、開発条件を以下のように設定する。

表【1-1】. 15　設定した開発条件

項目	条件設定値
ドキュメントの作成	必要
開発範囲	要求仕様の受領～設計、検証～FPGA データの納品
対象	JPEG-XR 方式コーデック(一部機能のみ抜粋)
要求仕様形態	C で記述された動作アルゴリズムおよび期待値データ
想定される作業フロー	
作業人数	1 名
支援ツール	なし

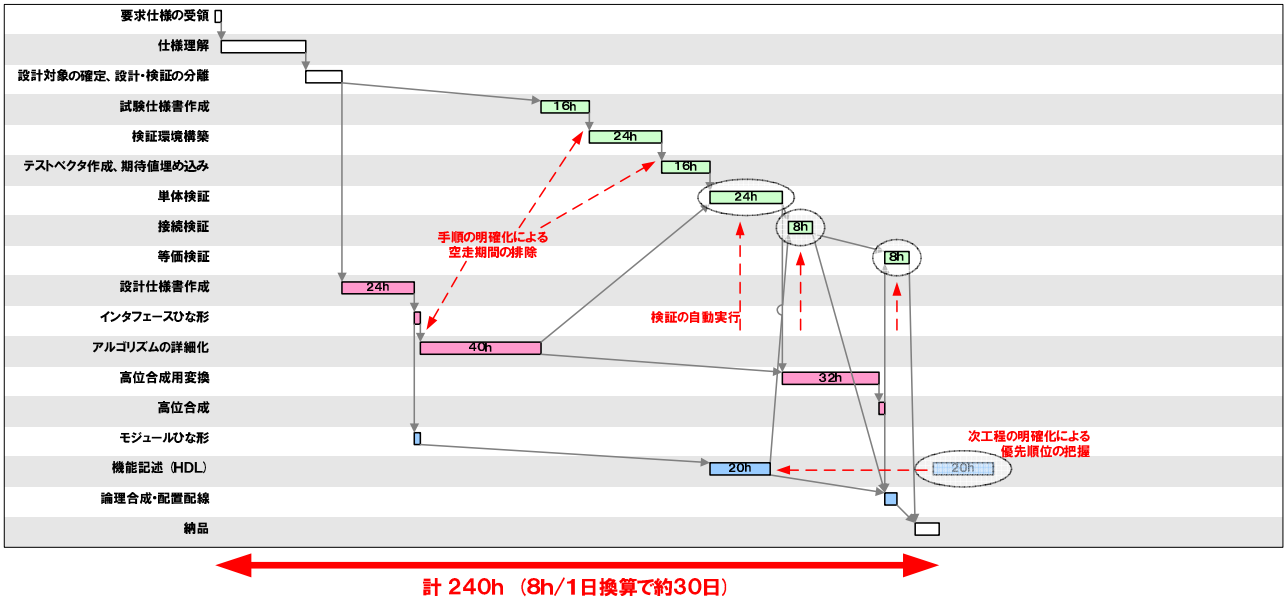
まず、上記条件で従来手順に沿った開発を行った場合の工程進捗を以下に示す。

各工程間では 試行錯誤や資料・データ準備等の空走期間が半日～1 日程度発生するとして、約 2 ヶ月(20日/月で換算)を要する。



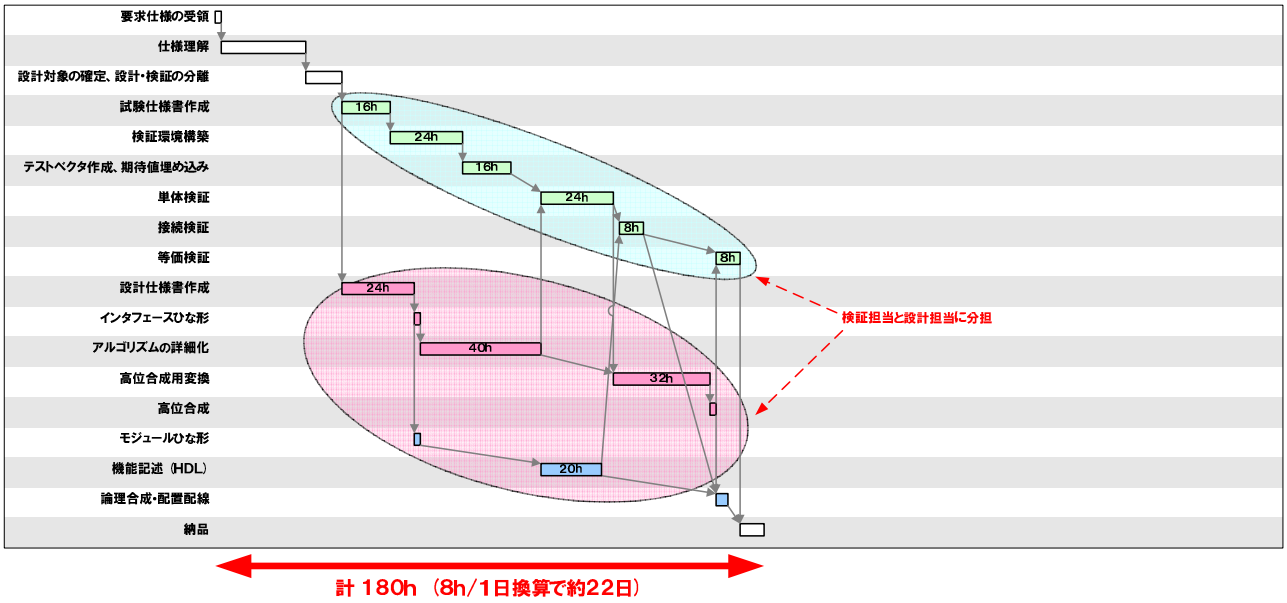
図【1-1】. 34　CDEF による支援なし、一人で開発を行った場合

この進捗を元に、CDEF を使用して次工程の明確化、ドキュメント等の蓄積、バックグラウンド実行を行った場合の工程進捗を以下に示す。



図【1-1】. 35　CDEF による支援あり、一人で開発を行った場合

また、CDEF を使用して二人に作業を割り振った場合の工程進捗を以下に示す。

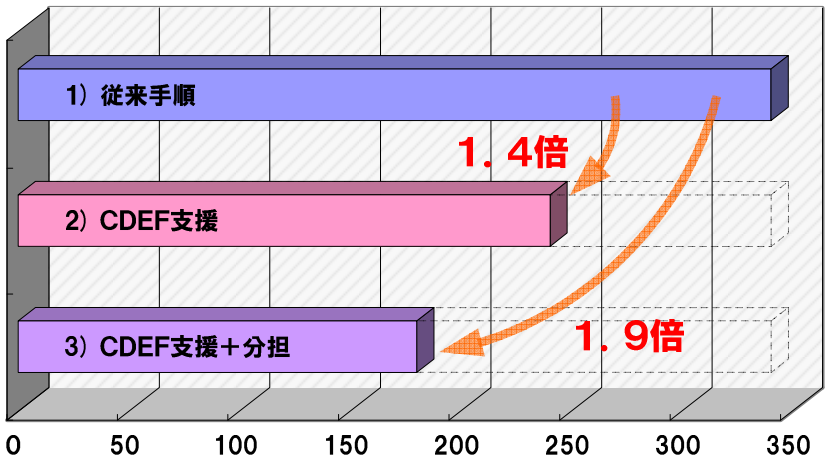


図【1-1】. 36　CDEF による支援あり、二人に分担を割り振った場合

それぞれの予想工数を比較すると、以下のようになる。
二人で分散した場合では工期を 47%削減することが可能となる。

表【1-1】. 16　設定条件下での予想工数の比較

ケース	予想工数	削減率	ケース 1 を基準とした倍率
1) 従来手順に沿った開発	340h	----	----
2) CDEF による支援	240h	29.4%	x1.4
3) CDEF による支援＋分担	180h	47.0%	x1.9



図【1-1】. 37　比較グラフ

2-1-2-7-2 IP 生成工数の削減率

IP 作成工数の削減率は、以下の各ケースで計測・抽出したデータを用いて算出する。

1) 手作業による設計工数、ソースコード有効行数
2) 既存ソースコードを流用して手作業による改変、同工数および有効行数
3) FGen を使用した IP 展開、同工数および有効行数
4) FGen を使用した IP 開発、同工数および有効行数

なお、ケース 4)は「IP 利用に関する検討・問題点」項で述べたとおり、ケース 3)を行うための前準備として必要な作業を示す。このため、ケース 4)単体では比較は行えない。IP の初回利用時はケース 3)＋ケース 4)との比較、IP の 2 回目以降利用時はケース 3)のみとの比較、という扱いになる。

表【1-1】. 17　各ケースでの設計工数データ

ケース	項目	データ	備考
ケース 1) 手作業による設計	ソースコード総行数	10,353 行	
	ソースコード有効行数	8,706 行	総行数－コメントおよび空行
	作業者による記述行数	9,908 行	総行数－空行
	ファイルサイズ	527,483 Byte	
	所要時間	10 日	
ケース 2) 既存 RTL の流用	※ 比較に使用できる同規模の類似事例が存在しないため、本データは設計見積にて代用した。		
	ソースコード総行数	----	
	ソースコード有効行数	5,000 行	
	作業者による記述行数	2,000 行	修正対象となる行数
	ファイルサイズ	----	
	所要時間	2 日	
ケース 3) FGen による IP 展開	ソースコード総行数	6,909 行	FGen によって展開されたコードの行数
	ソースコード有効行数	5,501 行	総行数－コメントおよび空行
	作業者による記述行数	1,768 行	FGen 設定ファイルの記述行数
	ファイルサイズ	315,981 Byte	
	所要時間	2 時間	設定ファイルの作成工数＋展開工数
ケース 4) FGen による IP 開発	ソースコード総行数	----	(記述対象がソースコードではないため除外)
	ソースコード有効行数	----	(記述対象がソースコードではないため除外)
	作業者による記述行数	1,395 行	FGen ライブラリの記述行数
	ファイルサイズ	62,098 Byte	
	所要時間	1 日	

上記のデータについて、ケース 3 の 5,501 行を基準に所要時間を補正したデータを以下に示す。本来ソースコードには数 10～数 100 行程度の固定的な記述が発生するが、5000 行を超える規模では十分に無視できるとして一次関数的な加工にて補正する。

表【1-1】. 18　各ケースの比較

ケース	総行数(行)	有効行数(行)	記述行数(行)	所要時間(h)
1) 手作業	6,540※	5,500※	6,260※	151※
2) 流用＋修正	----	5,500※	2,200※	53※
3) FGen IP 展開	6,909	5,501	1,768	2
4) FGen IP 開発	----	----	1,395	24

※補正したデータ

各ケースの所要時間に着目して比較したデータを以下に示す。手作業で開発した場合と比較し、IP の開発を含む工数は 5.8 倍の生産性向上となるが、IP の利用のみであれば 75 倍の生産性向上となる。

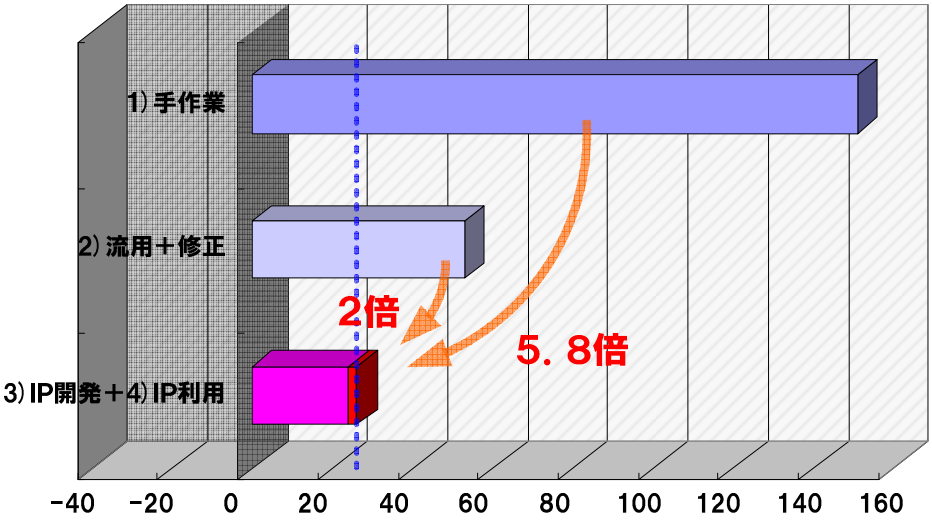
表【1-1】. 19　各ケースの所要時間の比較

ケース	所要時間(h)	対 IP 開発(ケース 3+4)		対 IP 利用(ケース 3 のみ)	
		削減率	倍率	削減率	倍率
1) 手作業	151※	82.7%	x5.8	98.6%	x75
2) 流用＋修正	53※	50.9%	x2.0	91.3%	x26
3) FGen IP 展開	2	----	----	----	----
4) FGen IP 開発	24	----	----	----	----

※補正したデータ

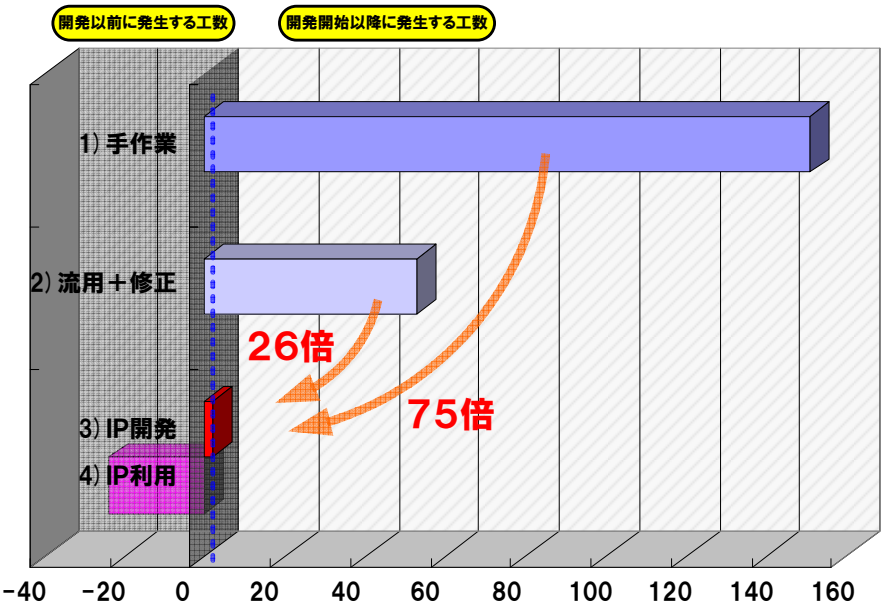
比較結果のグラフを以下に示す。工数軸は開発に必要な時間を示し、その 0 位置は「開発開始位置」を示すものとする。

まず、IP の開発を含む、新規の状態からの工数(ケース 3＋ケース 4)と従来手順による開発との比較を示す。
従来手順における流用を行った場合と比較しても、2 倍の生産性向上となっている。



図【1-1】. 38　ケース 3＋ケース 4(IP 開発＋展開)に対する従来手順による開発との比較

次に、IP が既に開発された状態での工数(ケース 3 のみ)と従来手順による開発との比較を示す。ケース 4)は開発の前準備として行うべき作業であるため負数側にプロットし、従来手順との比較には含めない。
この場合、IP 展開工数のみとの比較となり、従来手順の流用と比較しても 26 倍、手作業と比較した場合では 75 倍の生産性向上となった。



図【1-1】. 39　ケース 3(IP 展開のみ)に対する従来手順による開発との比較

2-1-2-7-3 全体の生産性向上率

各テーマに由来する生産性向上は、以下のように分類される。この分類と併せて、各テーマにて評価・導出された削減工数を以下に示す。

表【1-1】. 20　各テーマでの生産性向上値の分類

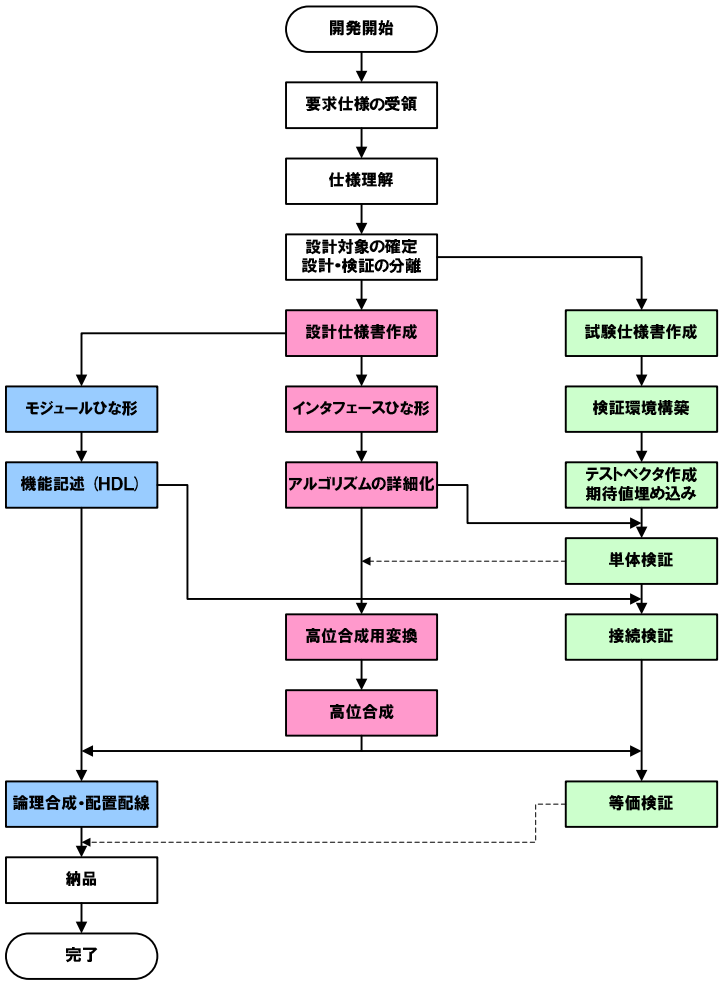
No.	項目	1-1	2-1	2-2	2-3	2-4	3-1	3-2	4-1
①	無駄工数の削減	○							
②	開発手順の支援ツール化による工数の圧縮	○	○						
③	ノウハウの蓄積による工数の圧縮		○	○	○	○	○	○	○
④	IP の開発、流用による工数の圧縮		○			○	○	○	○
⑤	新規手順の導入による工数の圧縮				○		○	○	○

表【1-1】. 21　各テーマの生産性向上値一覧

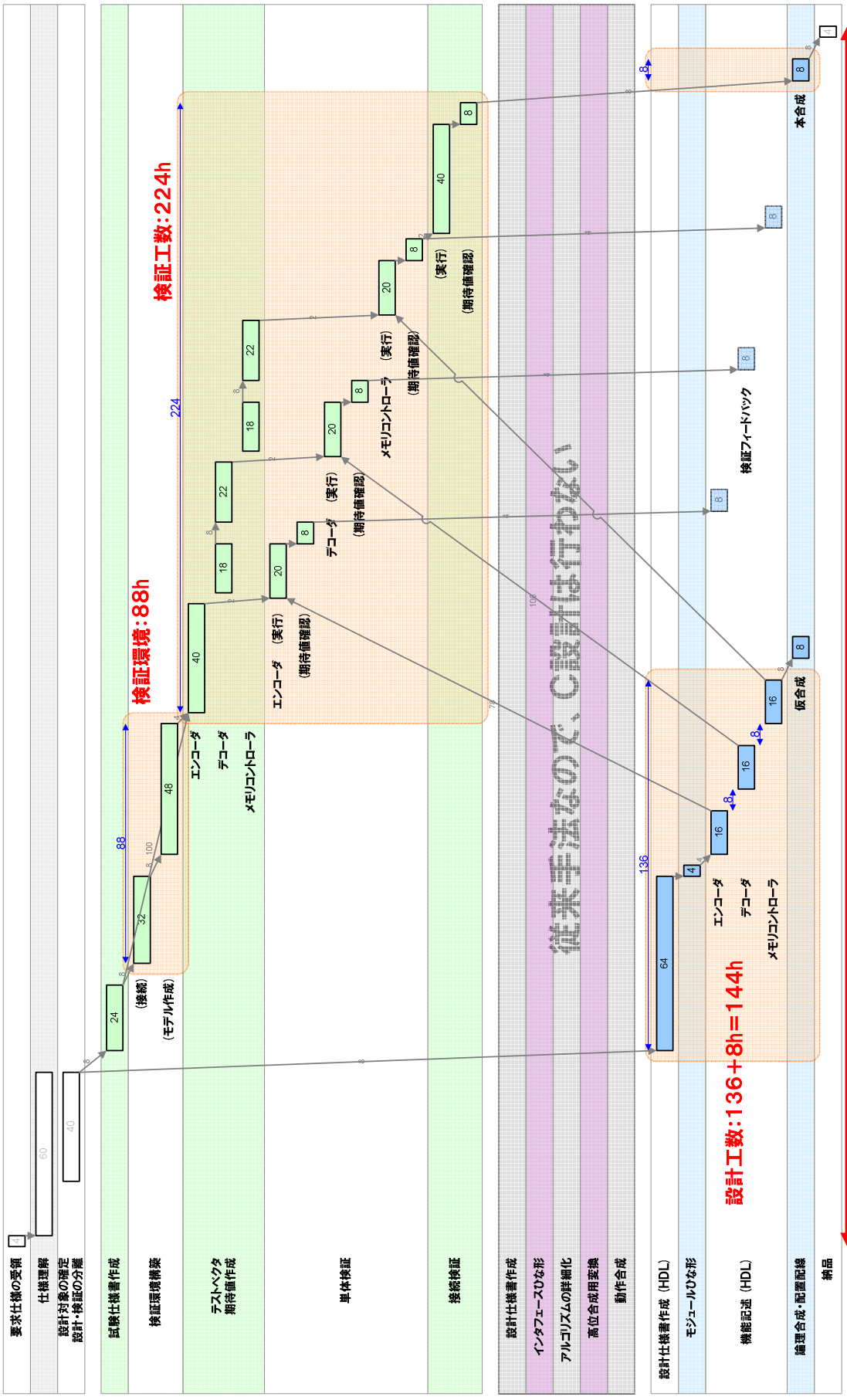
テーマ	項目	分類	元工数(h)	改善工数(h)	削減率
1-1	無駄工数の削減	①	340.00	240.00	x1.42
	支援ツール:統合置換ソフトウェア	②	151.00	26.00	x75.00
2-1	SystemC 非同期 FIFO	④	8.00	0.25	x32.00
	画像ビューアを用いた期待値比較手法	②	6.00	3.00	x2.00
2-2	高位合成ツール A を用いた JPEG-XR 方式コーデック開発	③	252.00	84.25	x2.99
	高位合成ツール B を用いた JPEG-XR 方式コーデック開発	③	211.75	87.25	x2.42
	高位合成ツール C を用いた JPEG-XR 方式コーデック開発	③	251.25	111.50	x2.25
2-3	従来手法による JPEG-XR 開発の状況分析	③,⑤	----	----	----
2-4	検証用ソフト IP:SDRAM コントローラ	④	196.00	151.00	x1.30
	SoC 検証環境	③,④	50.00	20.00	x2.50
3-1	ハード IP:SDRAM コントローラ	④	41.00	22.50	x1.82
	SystemVerilog 検証環境	③,⑤	90.00	60.00	x1.50
3-2	相互変換用ソフト IP 試作:SDRAM コントローラ	④,⑤	45.00	48.00	x0.93
	相互変換用ハード IP 試作:SDRAM コントローラ	④,⑤	45.00	3.00	x15.00
	SoC 検証環境	④,⑤	115.00	0.00	----
4-1	設計・検証分離手順、C 言語記述ひな形	③,④	190.25	88.50	x2.15
	SystemC 検証環境ひな形	③,④	61.00	23.00	x2.65

各テーマの成果を使用し、本研究事業全体で連結した生産性を評価するにあたり、CDEF 単体での生産性評価と同様に開発条件を設定する。

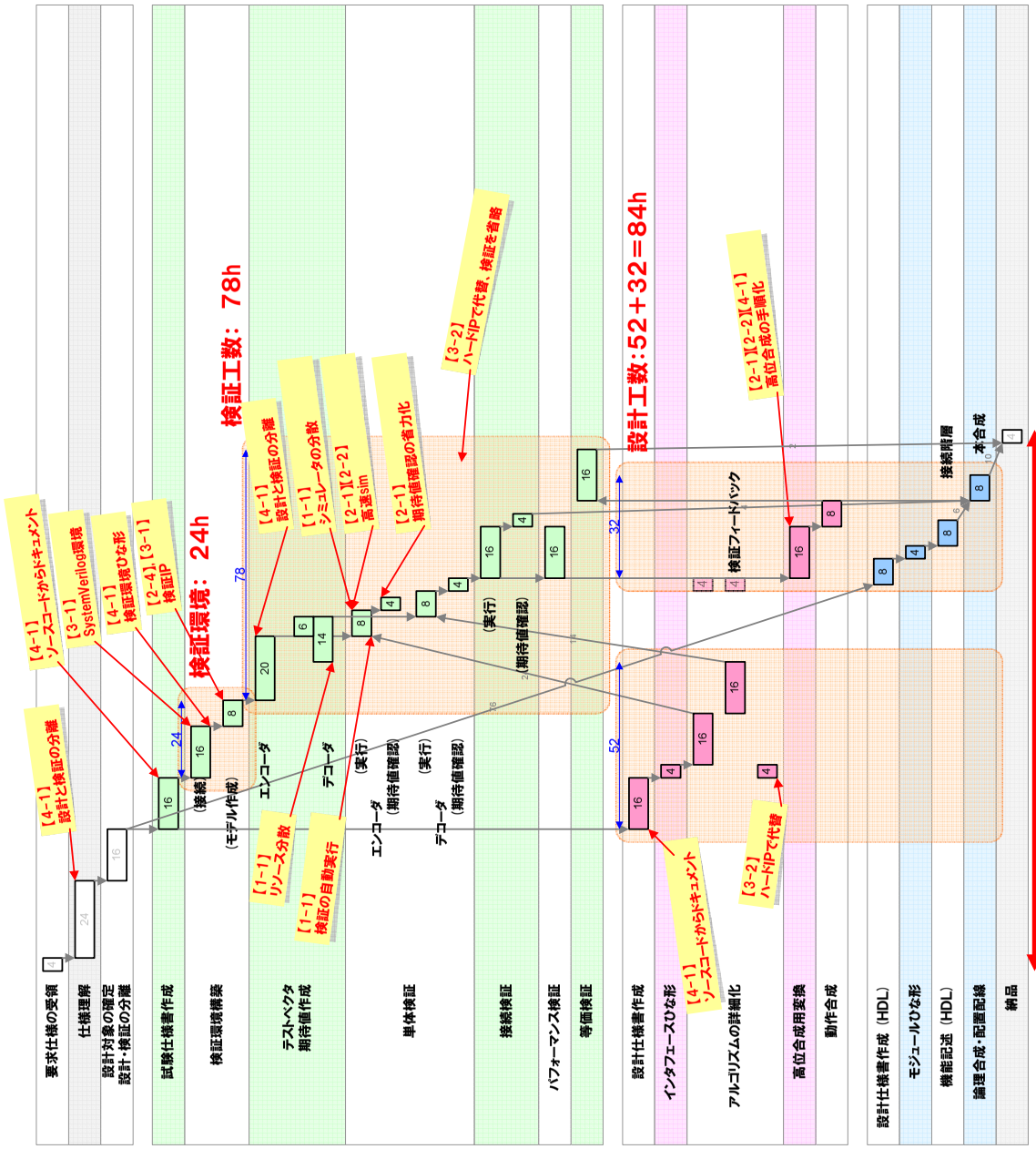
表【1-1】. 22　設定した開発条件 #2

項目	条件設定値
ドキュメントの作成	必要
開発範囲	要求仕様の受領～設計、検証～FPGA データの納品
対象	JPEG-XR 方式コーデックおよび SDRAM コントローラ
要求仕様形態	C で記述された動作アルゴリズムおよび期待値データ
想定される作業フロー	「業務遂行効率」の項にて示したフローと同様 
作業人数	2 名
支援ツール	統合置換ソフトウェア　(IP，モジュール/検証環境ひな形)

この条件に従って従来開発手法および C 言語ベースの開発手法でそれぞれの工程進捗の予測を行った。以下にその進捗図を示す。なお各テーマで導出された値とは異なる条件で予測を行っているため、各テーマの改善工数とは異なる数値が使用されている。



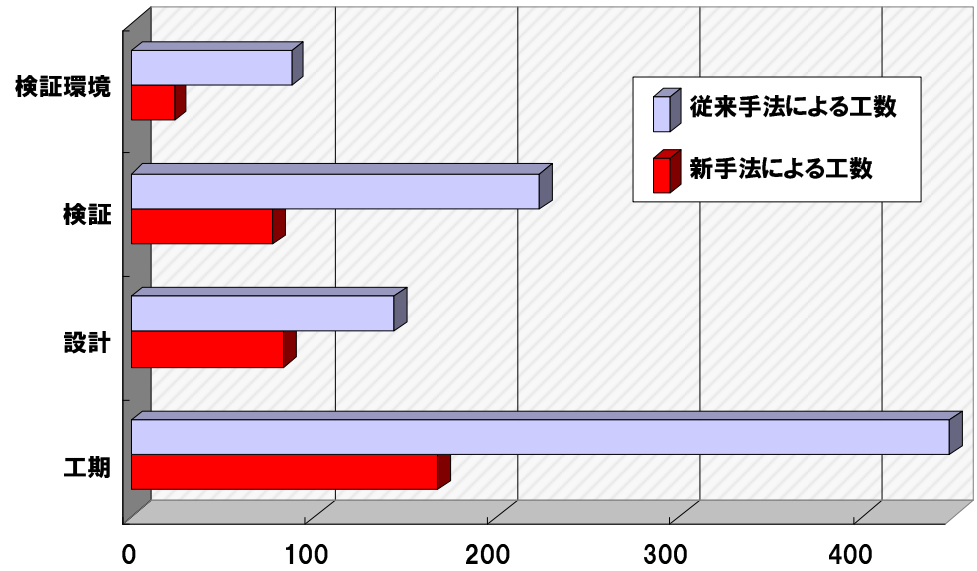
図【1-1】. 40　従来開発手法での進捗図



図【1-1】. 41　C 言語ベース開発手法を適用した進捗図

表【1-1】. 23　区分ごとの工数比較

区分	従来手法（h）	新手法（h）	削減工数（h）	削減率（倍率）
検証環境	88	24	64	73%(x3.6)
検証	224	78	146	65%(x2.9)
設計	144	84	60	42%(x1.7)
工期	448	168	280	63%(x2.7)



図【1-1】. 42　比較グラフ

各テーマを連結した成果は、この予測に使用した条件では、工数が 63%削減され 2.7 倍の生産性向上となった。

2-1-3　まとめ

2-1-3-1　成果

本テーマでは、フレームワーク CDEF と、統合置換ソフトウェア FGen の検討および開発を行った。

CDEF を使用した条件を想定し、予測工数の見積を行った。その結果、従来手法に含まれていた無駄工数を排除し 1.4 倍(30%削減)の生産性向上が見込まれることを予測できた。さらに、人的リソースの支援がある場合は 1.9 倍(48%削減)の生産性向上となることも予測できた。

FGen およびレジスタ IP ライブラリを使用するという条件を想定し、従来開発手法との工数比較を行った。IP 開発も含む状態では完全な手作業と比較して工数が 83％削減され 5.8 倍の生産性向上、IP 再利用のみの状態では 75 倍の生産性向上となることが比較された。

CDEF、FGen および他テーマの成果を連結した状態で、従来開発手法と C 言語ベース開発手法との予測工数の比較を行った。その結果、工期が 63%削減され 2.7 倍の生産性向上となることが予測できた。本研究事業全体での目標とした生産性向上 10 倍(90%削減)には達しなかったが、新手法を導入することで従来開発手法と比較して 2 倍以上の効果となることは確認された。

2-1-3-2　結論

フレームワーク CDEF および統合置換ソフトウェア FGen を使用することで、生産性が向上することが予測により確認できたが 2.7 倍の生産性向上にとどまり、当初目標である 10 倍には到達しなかった。

「図【1-1】. 42　比較グラフ」に示したとおり、総工期に対して検証環境・検証・設計の工数計が近い値となっており、各工程の並列度が高くない状態で作業が進められていることを示している。これは予測に使用した工程フローが設計と検証との強い依存関係を持つため、依存元の工程が完了するまで依存先の工程を開始できない状態が発生していると考えられる。生産性向上が 10 倍に到達しなかった原因は、この依存関係による空白期間が発生したためと思われる。CDEF に組み込まれる工程分析において、このような工程の役割に着目した分析は想定していなかったが、可視化することで新たな視点の掘り起こしが期待できることがわかった。今後の改善によって可視化機能と工程分析の充実を図っていきたい。また、設計と検証の依存関係を抑えた作業フローへ改善することで生産性向上度を 10 倍以上に高めていきたい。

CDEF および FGen の開発にあたり、実用上 必須となる機能の実装を優先したため、以下の機能および課題が残ってしまった。今後、実装方法・アルゴリズムなどを検討していきたい。

- CDEF
  - システム管理画面　(内部データの登録、管理など)、フロー編集画面
  - ユーザーごとのアクセス制御
  - 労務管理など、他アプリケーションとの連携
  - 開発手順改善の正当性評価
  - 工程分析の評価式
  - 複数サーバーの管理手法
- FGen
  - 他言語用の入出力モジュール
  - 業界標準フォーマットに準拠させ、さらなる汎用化

第3章 本論・画像・動画技術開発

3-1 【2-1】画像・動画圧縮用ユニットの開発

3-1-1 概要・目的

画像・動画圧縮用に特化したユニットとし、画像・動画処理のコア技術、周辺部品としてメモリに対して【1-1】で開発するフレームワークに合った規定書類、ソフトウェアおよびライブラリやIPデータベースを作成する。

3-1-2 内容

3-1-2-1 内容説明

回路設計において、全ての回路がプロジェクト固有の物であることはほとんどなく、多くの場合はプロジェクトに依存しない回路部品が存在する。しかし各プロジェクトで扱うデータの数やビットの幅等が異なり、そのまま再利用できないため、新規に設計もしくは既存回路の流用・修正を行う必要がある。

また、回路検証時に使用するデータ生成プログラム等も同様に、分野(通信系/画像系等)が同じであれば似たような内容の物である場合が多い。

そこで本項目では C 言語を用いた動作合成設計を対象に、プロジェクトをまたいだ繰り返し設計作業の削減を目的として、C++/SystemC 言語を用いた汎用的な設計/検証用ライブラリを作成した。

3-1-2-2　開発の成果物

回路設計業務において最も工数の削減が見込まれるのは、頻繁に使用される回路部品の設計や検証環境の構築といった作業である。本項目ではこれらの作業の工数削減を目的として以下のような設計/検証用の IP/ライブラリを作成した。作成物とその概要を次に示す。

■[設計用]

- SystemC サイクルベース非同期制御 FIFO

  外部 RAM を接続し非同期 FIFO として使用するモジュールで、以下の特徴がある。

  - データ先読み式
  - データビット幅可変
  - ワード数可変
  - Almost Full 出力ワード数可変

これは Verilog HDL で作成したモデルをサイクルベース SystemC に変換した物である。したがって SystemC/Verilog HDL で等価な検証を行うことが可能となる。

上記のパラメータは可変であるため、フレームワークを使って設計対象に応じた設定のモジュールを生成することが可能であり、生産性の向上が図れる。

■[検証用]

- 画像生成 C++ライブラリ

  検証時によく使用される画像パターンを生成するプログラム。以下の機能をまとめた関数群となっている。

  ・ファイル入出力

  ・パターン画像生成(現状 2 種)

  このライブラリは新しい画像パターンを容易に追加できる構成とした。既存の構成に合わせて画像生成関数を作成することでライブラリに組み込めるようになっている。(手順については説明文書「生成画像パターン追加手順書」を作成)また、ファイル入出力についても PPM 形式等に対応した関数を用意してあるため、基本的な検証であればユーザーは用意された設定/生成実行関数を記述するだけで画像データを生成/読込/書出することができる。また、使用方法をまとめた「画像生成ライブラリマニュアル」やプログラムの詳細説明資料も併せて作成した。

  C++言語で記述しているため、アルゴリズム C コードと動作合成 C コードとの等価性検証や SystemC 検証環境にそのまま組み込んで使用可能である。

- 画像ビューア

  C++で記述された画像閲覧プログラム。単体で使用できる他、上記画像生成ライブラリや SystemC 検証環境との連携使用が可能となっている。

3-1-2-3　設計/検証用の IP/ライブラリを用いることによる生産性に対する効果

本項目にて作成した上記の IP/ライブラリを使用することにより設計/検証作業において以下のような効果が見込まれる。

a) 設計作業における効果

[SystemC サイクルベース非同期制御 FIFO]

設計作業時に IP を使用せず、非同期制御 FIFO を新規で設計もしくは既存の物を流用・修正した場合の作業工数はそれぞれ下記の通りである。

表【2-1】. 1 新規設計および既存回路流用・修正作業工数

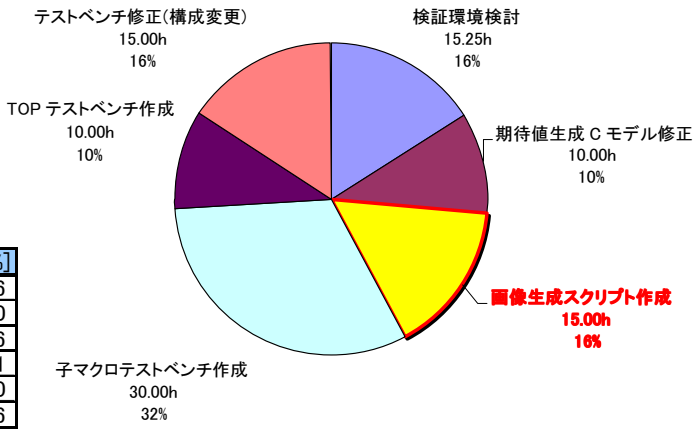
設計方法	必要設計工数(h)	必要デバッグ工数(h)	合計工数(h)
新規設計	4	4	8
流用・修正	2	3	5
IP 使用	～0.25	－(IP のため不要)	～0.25

この工数は設計作業全体における割合から考えると決して大きいとは言えないが、作業時間としては無視できない時間である。これをライブラリとして用意しておくことで、この工数を削減することができ、設計時間が短縮される。

b) 検証作業における効果

[画像生成 C++ライブラリ]

ある画像圧縮プロジェクトにおける検証環境構築の作業ごとの工数割合を下図に示す。表示項目は上から項目名・作業工数・割合となっている。(詳細については「3-3【2-3】画像圧縮 JPEG-XR 方式開発実績の分析」参照)



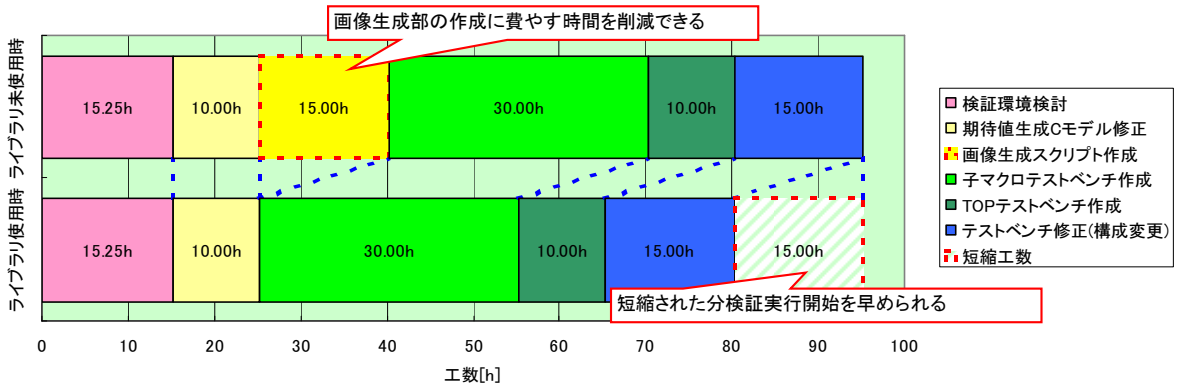
作業項目	工数[h]	割合[%]
検証環境検討	15.25	16
期待値生成Cモデル修正	10.00	10
画像生成スクリプト作成	15.00	16
子マクロテストベンチ作成	30.00	31
TOPテストベンチ作成	10.00	10
テストベンチ修正(構成変更)	15.00	16

図【2-1】. 1 ある画像圧縮プロジェクトにおける検証環境構築作業の各項目の割合

上図より、入力画像データ生成のプログラム作成に検証環境構築全体の 16%、時間にすると 15h の時間がかかっている。しかしこの画像データ生成部分についてはプロジェクト固有の部分はほとんどない。つまり他の画像系回路の検証においてもほぼ同等の物を作成することになり、プロジェクト単位での繰り返し作業が発生してしまう。そのため、画像データ生成部を汎用化しライブラリとすることで、この繰り返し作業をなくすことができ、15

～20%の工数削減により 1.1～1.2 倍の生産性の向上に繋がると考えられる。

本項目で作成した画像生成ライブラリは画像データファイルの読込/保存や画像データの生成といった基本的な機能を有しており、画像生成設定値をパラメータとして持たせることにより、プロジェクトに依存することなく汎用的に使用できるようになっている。このため入力データ(画像データ)生成部分の作成に費やす工数を削減することができ、検証実行作業の開始を早めることができる。また、単体デバッグ作業も同様に早期開始が期待されるため、全体検証開始前により多くのデバッグを行うことが可能となる。



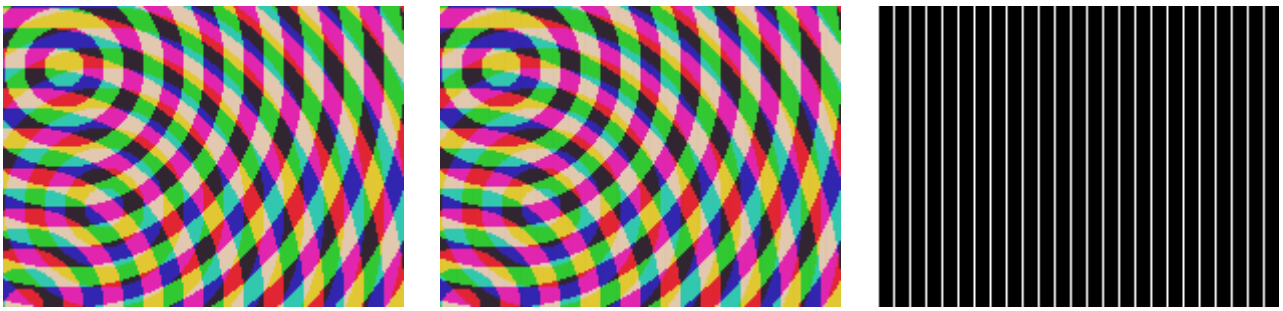
図【2-1】. 2 画像生成ライブラリ使用による検証環境構築時間短縮効果(見込)

さらに新規で作成した場合、作成者によってプログラムの構成や使用方法に違いがあり、検証を始める前に仕様を理解するための時間が発生してしまう。しかし、共通化されたライブラリと併せて作成したマニュアルや詳細説明資料により、使用方法理解等の工数短縮も見込まれる。加えて、継続利用・メンテナンスによりライブラリ自体の改良・拡張やマニュアルの内容改善にも期待でき、更なる生産性の向上ができるものと考えられる。

[画像ビューア]

画像系の回路検証作業においては、初期のデバッグ作業はシミュレーション結果と生成した期待値との数値の比較ではなく、画像として視覚的に比較を行う方が有効である場合が多い。しかし、一般的な画像形式ではファイルの内容が不完全な状態で閲覧することができず、シミュレーション途中に出力データを画像として確認することができない。

当社では画像系の回路検証に、構造が簡単な独自の形式で画像データを扱っている。この画像ビューアは前述の独自画像形式にも対応したもので、画像ファイルが不完全であってもファイルを画像として表示することが可能である。そのため、シミュレーションの途中であっても出力画像データをすぐに確認することができる。これにより待ち時間が短縮され、デバッグ作業をスムーズに進めることが期待される。さらに、機能拡張により期待値とシミュレーション出力の差分表示、値の一致/不一致箇所のモノクロ表示(図【2-1】. 3 差分表示例参照)といった使い方もできるため、視覚的にエラー発生箇所を特定しやすくなり、バグの原因特定までの時間短縮にも効果が見込まれる。これらの効果により、シミュレーション結果確認時間全体の 3～5%の工数が削減できると推測される。



(a)期待値画像　　(b)処理結果画像　　(c)差分表示(モノクロ)

図【2-1】. 3 差分表示例

3-1-3　まとめ

3-1-3-1　成果

本項目では生産性向上のために設計/検証作業の工数短縮を目的として、各種 IP/ライブラリを作成し、それを適用した場合に予測される削減工数を見積もった。その結果、検証環境構築工数が 95h から 80h へと 16%削減され、1.2 倍の生産性向上を見込める。

3-1-3-2　結論

設計作業における IP/ライブラリ使用の利点は、新規設計回路が減ることによる設計工数削減だけでなく、デバッグ済み回路を用いることで人為的なバグの混入を防ぐことができるという点である。そのため全体検証時にバグが見つかった場合、IP 部分は調査対象から除く(もしくは調査優先度を下げる)ことができ、バグの発生箇所特定に有利となる。

検証ライブラリを使用することによる一番大きな利点は、検証環境構築の時間短縮である。実際の検証作業では時間的なリソースが限られており、検証環境構築に時間を費やしてしまうと、その分検証実行の時間を圧迫してしまうことになる。ライブラリの使用によって短縮された時間は検証実行作業に充てることができ、より多くの検証を行えるため、品質の向上にも繋がる。

また、画像ビューアのような補助ツールの導入によって、検証作業を円滑に進められるだけでなく、有償の波形ビューア等の使用時間も減らすことができるため、設備的なリソースの節約という点においても生産性の向上が見込まれる。

以上のことからライブラリの開発によって、作業工数削減だけでなく品質向上や設備資源の節約といった面でも生産性が向上すると考えられる。

今回は画像圧縮をターゲットに設計/検証ライブラリの開発を行い、設計・検証環境構築作業における工数短縮の効果を見積った。今後の展開としては通信系の回路設計をターゲットとして、フレームワークに組み込むことを意識したパケットデータ生成ライブラリやパケットデータの可視化ツールのような補助ツールの開発を行うことにより、通信系回路設計においても同様の生産性の向上が見込まれる。

今回の研究で作成した画像生成ライブラリは動画を意識した機能を有していないが、画像のシフトや回転、明暗変化のような画像加工等の機能拡張を行うことで、動画検証時にも柔軟に対応でき、更なる生産性の向上を期待できる。またファイル入出力に関しても現在は未対応形式が多く、自然画などを扱う場合は画像フォーマット変換を行う必要がある。しかし、将来的に対応画像形式を増やすことで様々な画像フォーマット変換を行わず直接利用できるようになり、利便性向上によって効率化に繋がると考えられる。

3-2　【2-2】ソフトウェアレベルで画像圧縮技術とメモリコントローラの実装手法の開発

3-2-1　概要・目的

テーマ【1-1】のフレームワークに画像圧縮技術用ユニットを搭載して生産性向上の検証を行う。

ユニットでは画像圧縮技術を実現する為の組込みソフト、画像圧縮用 LSI および周辺メモリを必要とする。この時、画像圧縮 LSI には周辺メモリを制御するメモリコントローラを含む。

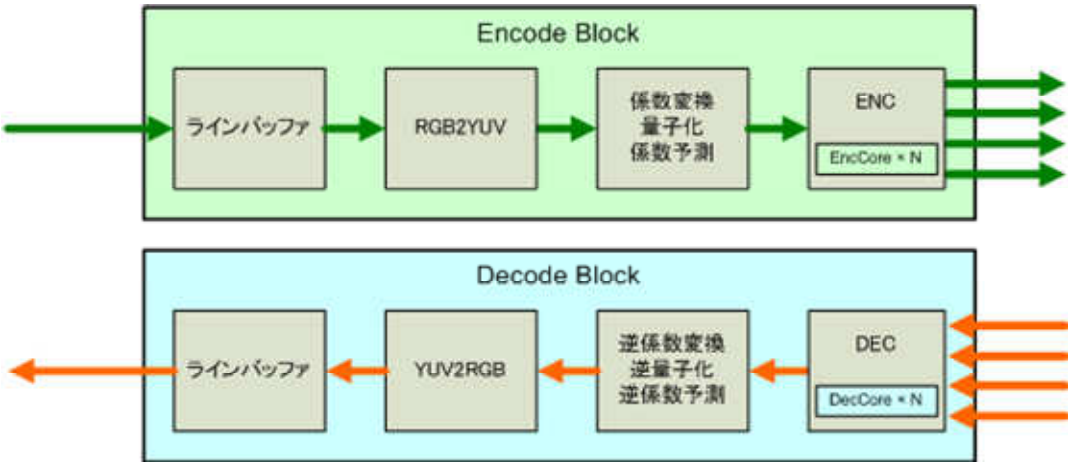
また、フレームワークを使用する初期段階では実装すべきシステム全体を全てソフトウェア化することで、アルゴリズム検討、アーキテクチャ検討、回路検討および回路検証を高い生産性で実現可能であり、その効率を測定することで生産性向上率を検証する。手設計に対する高位合成ツールを用いた設計の生産性向上率についてはテーマ【2-3】に示す。

3-2-2　内容

3-2-2-1　高位合成ツールを使用した JPEG-XR 開発

画像圧縮 JPEG-XR のアルゴリズム C ソースを元に、エンコードおよびデコード処理を行う機能を有する回路の開発を 3 社の高位合成ツールを使用して行い、その生産性について検討を行った。ここで 3 社の高位合成ツールを高位合成ツール A/B/C と定義する。

エンコードおよびデコード処理の概要は下記図に示す内容となっている。



図【2-2】. 1 エンコードおよびデコード処理の概要

エンコード処理を行うアルゴリズム C ソースの一覧と処理内容は下記表に示す通りである。

表【2-2】. 1 エンコード処理用アルゴリズム C ソース一覧

コード名	処理内容	処理単位
enc_main.c	RGB2YUV から符号化までの接続	1 フレーム(1920x1080 画素)77
rgb2yuv.c	RGB 2x2 画素を YUV420 に変換	1 フレーム(1920x1080 画素) 子関数に 2x2 画素単位の処理有り
pct.c	係数変換	1 マクロブロック(8x2 画素)
quantize.c	量子化	1 マクロブロック(8x2 画素)
predict.c	係数予測	1 マクロブロック(8x2 画素)
enc_core.c	可変長符号による符号化	1 フレーム(1920x1080 画素) 子関数に 1 マクロブロック単位の処理有り

デコード処理を行うアルゴリズム C ソースの一覧と処理内容は下記表に示す通りである。

表【2-2】. 2 デコード処理用アルゴリズム C ソース一覧

コード名	処理内容	処理単位
dec_main.c	復号から YUV2RGB までの接続	1 フレーム(1920x1080 画素)
dec_core.c	可変長符号の復号	1 フレーム(1920x1080 画素) 子関数に 1 マクロブロック単位の処理有り
ipredict.c	逆係数予測	1 マクロブロック(8x2 画素)
iquantize.c	逆量子化	1 マクロブロック(8x2 画素)
ipct.c	逆係数変換	1 マクロブロック(8x2 画素)
yuv2rgb.c	YUV420 を RGB 2x2 画素に変換	1 フレーム(1920x1080 画素) 子関数に 2x2 画素単位の処理有り

なお、開発にあたり以下の要求仕様を定義した。

- 入出力画素信号:RGB 各 10bit
- 入出力レート:2pixel/75MHz
- 対象デバイス:Stratix II EP2S180F1508C5
- ラインバッファは 1 ライン分
- 圧縮符号データの I/F は外部 FIFO と接続され、FIFO 要因のストールをケアする必要はない
- enc_core および dec_core は 8 並列以下を目標とする(必須ではない)

3-2-2-1-1 高位合成ツール A を使用した開発

(開発内容説明)

「高位合成ツール A」を利用して、画像圧縮 JPEG-XR をモチーフとして開発を行った。

回路構成を検討した結果、Encode ブロックおよび Decode ブロックをそれぞれ次の様に構成することとした。

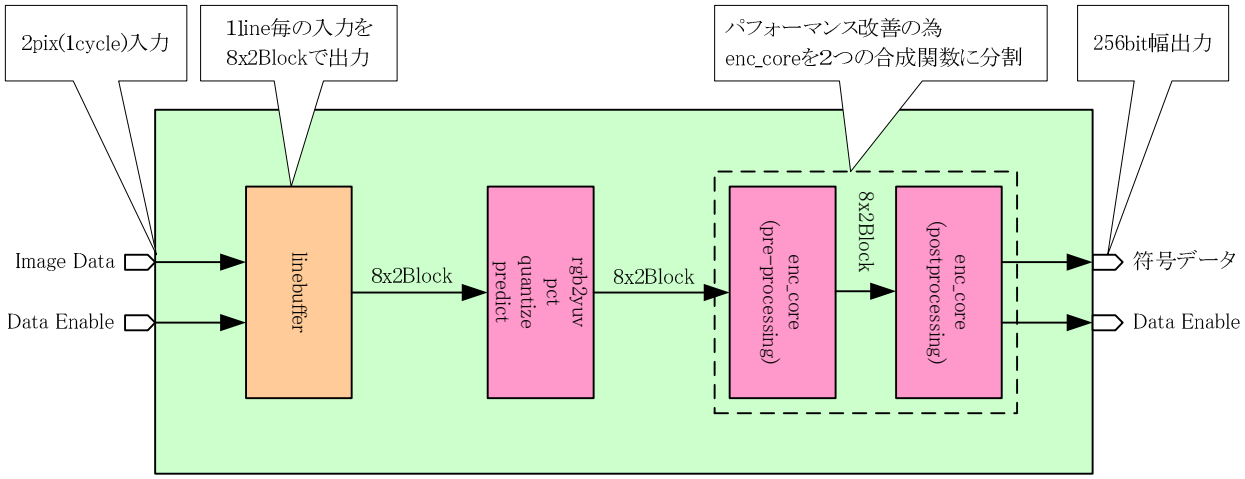
まず Encode ブロックの構成について説明する。

画像圧縮アルゴリズムとなるアルゴリズムコードは、全て 8x2 ブロック単位で処理を行うように変更を加え、高位合成対象として、「rgb2yuv から predict までの処理」「enc_core」の 2 モジュールに分割することにした。しかし「enc_core」を合成した結果、「enc_core のスループット」が「外部からの画像入力レート」を大きく下回った為、「enc_core(前処理)」「enc_core(後処理)」の 2 モジュールにして、スループットの改善を行った。

またアルゴリズムコード以外に実装が必要になる「linebuffer」は、画像アルゴリズムモジュールの処理単位に合わせ、8x2 ブロック単位のデータ出力と定めた。

以上の通り、「linebuffer」「rgb2yuv から predict までの処理」「enc_core(前処理)」「enc_core(後処理)」の 4 モジュールとして設計を行った。

なお符号データは、「外部からの画像入力レート」と「圧縮率の最悪ケース」から 256bit 幅で出力するものと定めた。



図【2-2】. 2 Encode ブロック構成(高位合成ツール A)

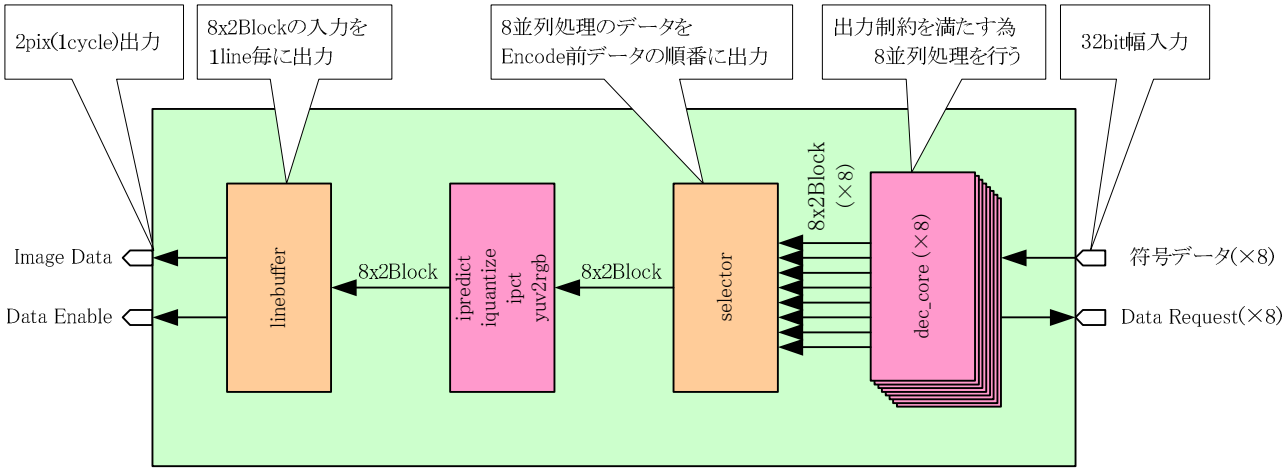
次に Decode ブロックの構成について説明する。

Encode ブロックと同様に画像圧縮アルゴリズムとなるアルゴリズムコードは、全て 8x2 ブロック単位で処理を行うように変更を加え、高位合成対象として、「dec_core」「ipredict から yuv2rgb までの処理」の 2 モジュールに分割することにした。「dec_core」を合成した結果、「dec_core のスループット」が「外部への画像出力レート」を大きく下回ったが、これ以上のスループット改善が難しいと判断した。「外部への画像出力レート」は 1 ブロック(dec_core の処理単位)4 サイクルであり、この時の dec_core のスループットが 32 サイクルであった。その為、「dec_core」を 8 並列に動作させることで、画像出力レートを満たすように構成した。

またアルゴリズムコード以外に実装が必要になる「linebuffer」は、画像アルゴリズムモジュールの処理単位に合わせ、8x2 ブロック単位のデータ入力と定めた。さらに「dec_core」を 8 並列動作させる為、「dec_core」と「ipredict から yuv2rgb までの処理」の間にデータを並び替える必要があり、「selector」モジュールを追加した。「selector」モジュールは「ipredict から yuv2rgb までの処理」モジュールに適切な「dec_core」の出力データを供給するよう設計した。

以上の通り、「dec_core」「selector」「ipredict から yuv2rgb までの処理」「linebuffer」の 4 モジュールとして設計を行った。

なお符号データは、「画像出力レート」「圧縮率の最悪ケース」「『dec_core』の並列数」の 3 点から 32bit 幅で入力することと定めた。



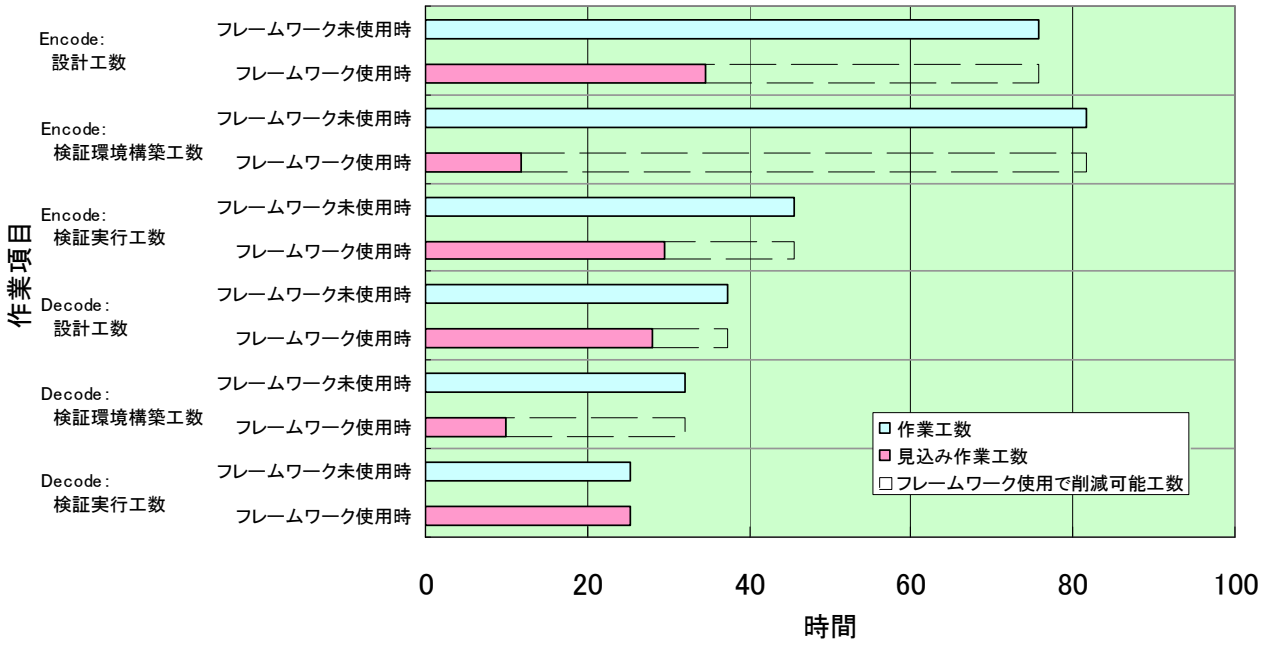
図【2-2】. 3 Decode ブロック構成(高位合成ツール A)

（設計総工数）

本ツールを用いた JPEG-XR 開発工数を下記グラフに示す。作業工数について全体的に encode より decode の工数が短いのは、内容の似ている部分やツール利用の慣れにより工数が短縮されたためである。

＜補足＞

- フレームワーク未使用時　　：フレームワークを利用しない場合の「作業工数」。
- フレームワーク使用時　　：フレームワークを利用した場合の「見込み作業工数」。



図【2-2】. 4 見込み作業工数(高位合成ツール A)

(設計記述量)

上記で説明したように回路を構成して元のアルゴリズムコードを高位合成用コード(注 1)に書き換えた。下記表は元のアルゴリズムコードと高位合成用コードのコード量差分を示したものである。

削除行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードから削除した行数。

追加行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードに追加した記述の行数。

行数　：　各コードの総行数と増減率［高位合成用コード行数/元のアルゴリズムコード行数］

流用率　：　元のアルゴリズムコードに手を加えず、流用した箇所を割合で表記。全ての記述を流用できれば 100%(最大)、全ての記述を書き換えた場合は 0%(最小)となる。

注 1:等価検証用記述を含むコードあり。

表【2-2】. 3 Encode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール A)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
rgb2yuv.c	127	275	291/143 (+103%)	80	検証コード(45 行)含む
pct.c	72	178	385/246 (+57%)	90	記述の並列展開有り 検証コード(61 行)含む
quantize.c	48	208	222/62 (+258%)	90	記述の並列展開有り
predict.c	22	175	239/38 (+546%)	90 (*1)	記述の並列展開有り 検証コード(48 行)含む
enc_core.c	446	742	748/452 (+65%)	5	検証コード(129 行)含む
linebuffer.c	-	-	180/- (-%)	-	新規設計

*1: 記述の並列展開(同じ記述の複数コピー)の為、行数に対して流用率が高くなっている。

表【2-2】. 4 Decode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール A)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
yuv2rgb.c	163	157	157/163 (-4%)	80	
ipct.c	57	84	253/243 (+4%)	90	記述の並列展開有り
iquantize.c	50	84	85/51 (+67%)	90	記述の並列展開有り
ipredict.c	37	111	112/38 (+195%)	85	記述の並列展開有り
dec_core.c	567	1585	1585/567 (+180%)	0	検証コード(37 行)含む
selector.c	-	-	130/- (-%)	-	新規設計 検証コード(68 行)含む
linebuffer.c	-	-	215/- (-%)	-	新規設計 検証コード(107 行)含む

等価性検証用の記述を追加したため、全体的に記述量が増加している。

enc_core.c と dec_core.c コードは合成後のパフォーマンス(動作速度)を満たすように記述を大幅に変更したため流用率が低くなっている。それ以外のコードは流用率が高い。

（検証作業）

高位合成ツールを利用して回路を設計する場合は検証作業に、アルゴリズムコードと高位合成用コードの等価検証が必要となる。（アルゴリズム以外のコードは期待した動作の確認用の検証となる。）等価検証は、アルゴリズムコードと高位合成コードに同じ入力値を与え、同じ出力値であることを確認するものとする。

等価検証を行うことにより各高位合成モジュールのバグの流入を防ぐことができる。その為全体検証で、各高位合成モジュールのデバッグ工数が存在しない。ただし従来のように、モジュール間でのデータの受け渡しが正常に行えているかを検証する接続検証とインターフェース検証は必要である。

（論理合成）

ALTERA 社の Stratix II (EP2S180F1508C5)で論理合成を行い、規模の確認を行った。下記表にトータル・論理回路・レジスタの利用率を示す。

＜補足＞

- 合成対象関数毎に論理合成を実行し、全結果を集計
- 「Logic utilization」レポート結果が 1%以下の場合、1%の使用率として集計

表【2-2】. 5 論理合成規模(高位合成ツール A)

論理合成対象	Logic Utilization(%)	Combinational ALUTs(%)	Dedicated logic Registers (%)
Encode	16	12.9	5.6
Decode	57	48.3	30.2

EncodeブロックとDecodeブロックを比較するとDecodeブロックがEncodeブロックの２倍程度の規模になった。これは Decode の core モジュールの並列数によるものであった。

なお論理合成結果のレポートを確認したところ、enc_core(後段)と dec_core でタイミングエラーが発生していた。タイミングエラーを修正する為に、スループットが変わらない範囲で高位合成制約に修正を加えた。結果は改善したが、タイミングエラーは発生していた。

3-2-2-1-2 高位合成ツール B を使用した開発

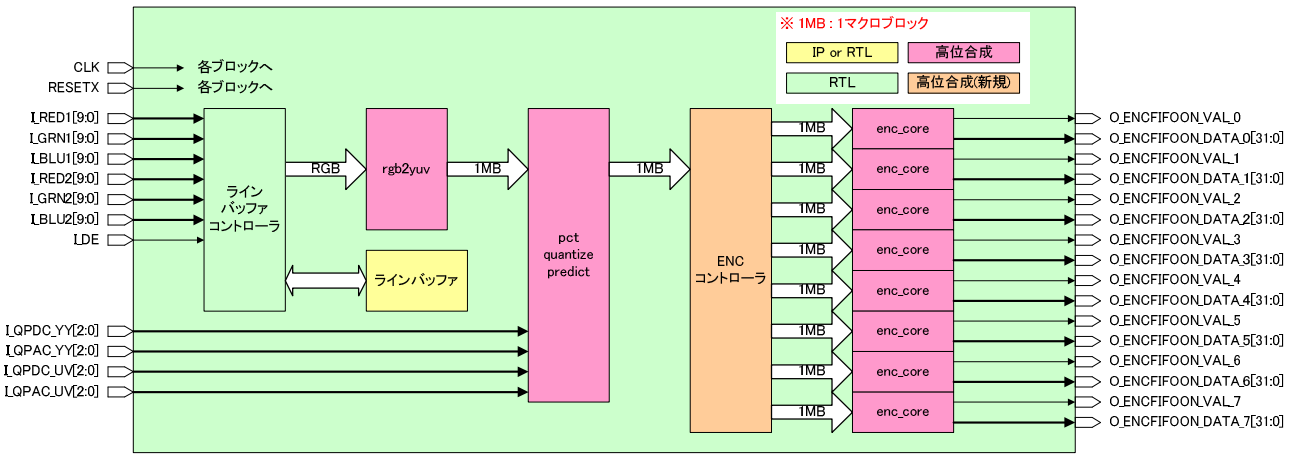
(開発内容説明)

■Encode ブロック

本開発のアルゴリズムコードは、8x2 画素単位(1 マクロブロック(1MB))で処理するアルゴリズムであり、スループットを満たすためには 4 サイクルで Encode 処理を行う必要がある。

- ラインバッファコントローラ (RTL)
  2pix/clock のデータをラインバッファに一旦格納し、次のラインが入力されてくるときにラインバッファから後段のブロックへデータを出力する。
- ラインバッファ (IP)
  1 ライン分のデータを格納する。
- rgb2yuv (高位合成)
  RGB データを 8x2 の YUV データ(1 マクロブロック(1MB))に変換する。
- pct、quantize、predict (高位合成)
  1MB 単位で係数変換、量子化、係数予測処理を行う。
- ENC コントローラ (高位合成新規設計)
  8 並列で構成された enc_core に対して、1MB 単位のデータを振り分ける。
  処理開始前に enc_core をストールさせておき、1MB データ処理時、4 サイクル置きに並列接続された enc_core のストールを順次解除する。
- enc_core (高位合成)
  1MB データを圧縮符号データに変換する。
  アルゴリズムコードを関数分解して、パイプライン合成した結果、スループットの最大値が 31 サイクルとなった。今回はスループットを 32 サイクルとし、4 サイクルの入出力レートを満たすために 8 並列とした。

以下に Encode ブロック構成図を示す。



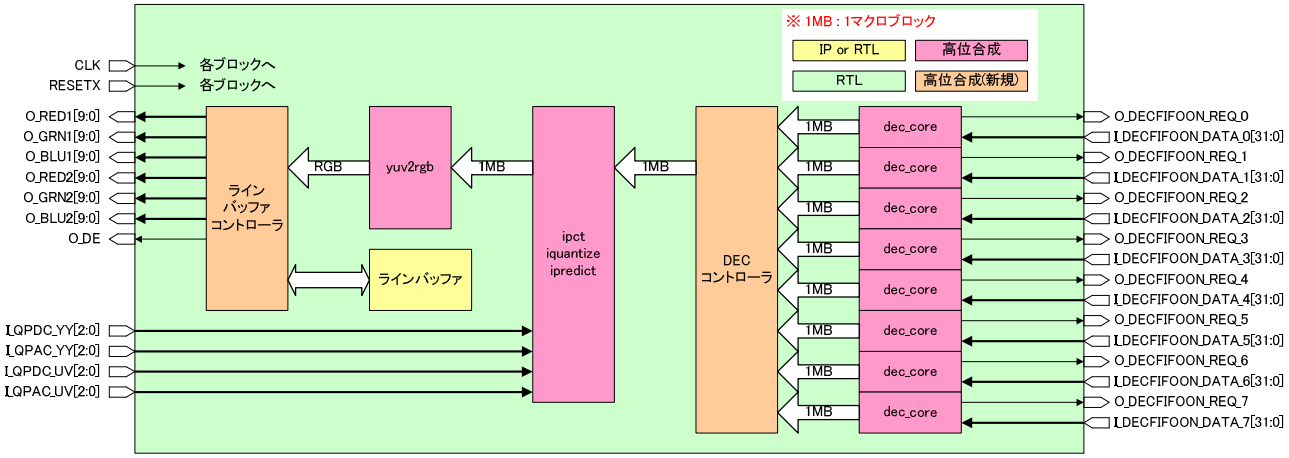
図【2-2】. 5 Encode ブロック構成(高位合成ツール B)

■Decode ブロック

本開発のアルゴリズムコードは、8x2 画素単位(1 マクロブロック(1MB))で処理するアルゴリズムであり、スループットを満たすためには 4 サイクルで Decode 処理を行う必要がある。

- dec_core (高位合成)
  圧縮符号データを 1MB データ単位で伸長する。
  アルゴリズムコードを関数分解して、パイプライン合成した結果、スループットの最大値が 30 サイクルとなった。今回は、スループットを 32 サイクルとし、4 サイクルの入出力レートを満たすために 8 並列とした。
- DEC コントローラ (高位合成新規設計)
  8 並列で構成された dec_core から出力される 1MB 単位のデータをセレクトする。
  処理開始前に dec_core をストールさせておき、処理開始すると、4 サイクル置きに dec_core のストールを解除する。
- ipct、iquantize、ipredict (高位合成)
  1MB 単位で逆係数変換、逆量子化、逆係数予測処理を行う。
- yuv2rgb (高位合成)
  1MB の YUV データを RGB データに変換する。
- ラインバッファ (IP)
  1 ライン分のデータを格納する。
- ラインバッファコントローラ (高位合成新規設計)
  YUV2RGB ブロックからの 2 ライン分の RGB データを、1 ラインは出力端子へ、もう 1 ラインはラインバッファへ格納する。
  1 ライン分出力を終えたら、ラインバッファに格納してあるもう 1 ラインのデータを出力する。

以下に Decode ブロック構成図を示す。



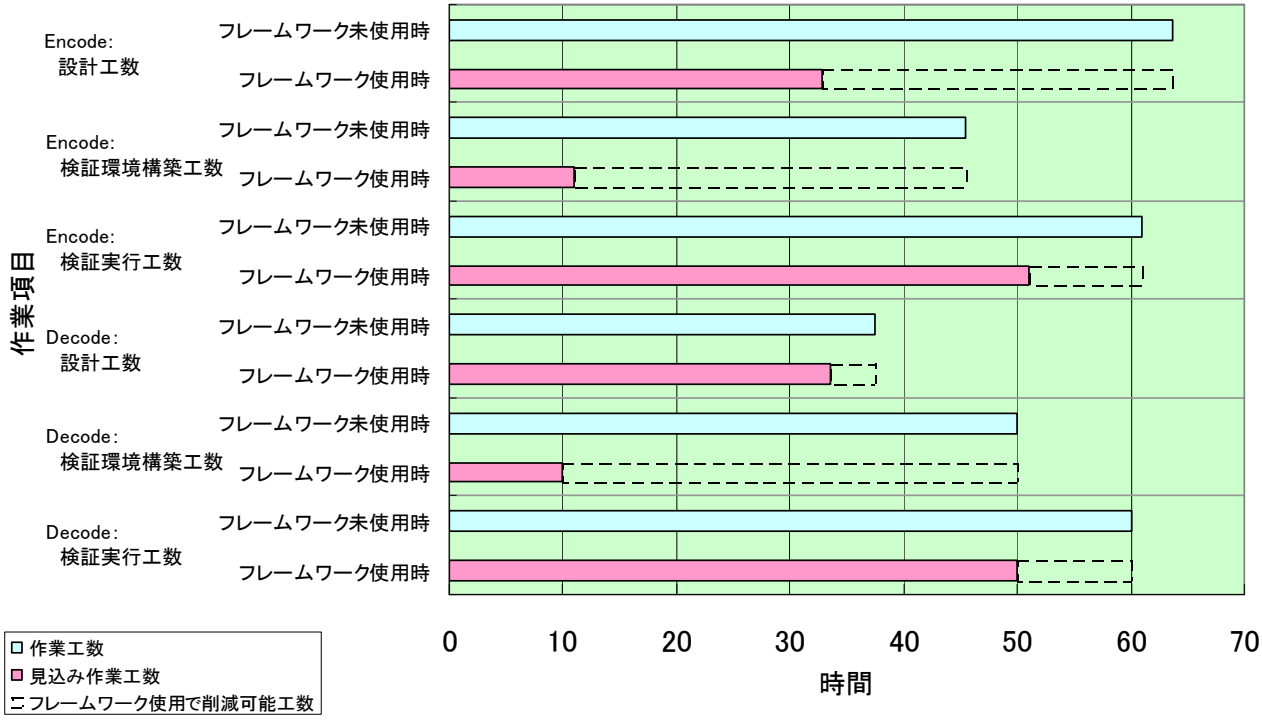
図【2-2】. 6 Decode ブロック構成(高位合成ツール B)

(設計総工数)

本ツールを用いた JPEG-XR 開発工数を下記グラフに示す。

＜補足＞

- フレームワーク未使用時　　：フレームワークを利用しない場合の「作業工数」。
- フレームワーク使用時　　：フレームワークを利用した場合の「見込み作業工数」。



図【2-2】. 7 見込み作業工数(高位合成ツール B)

（設計記述量）

上記で説明したように回路を構成して元のアルゴリズムコードを高位合成用コードに書き換えた。下記表に元のアルゴリズムコードと高位合成用コードのコード量比較結果を示す。

削除行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードから削除した行数。

追加行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードに追加した記述の行数。

行数　　：　各コードの総行数と増減率［高位合成用コード行数/元のアルゴリズムコード行数］

流用率　　：　元のアルゴリズムコードに手を加えず、流用した箇所を割合で表記。全ての記述を変更なく流用できれば 100%（最大）、全ての記述を書き換えた場合は 0%（最小）となる。

表【2-2】. 6 Encode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール B)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
enc_main.c	-	-	-	-	高位合成対象外
rgb2yuv.c	107	125	161/143 (+12%)	60	
pct.c	13	83	316/246 (+28%)	80	記述の並列展開有り
quantize.c	32	110	140/62 (+125%)	80	記述の並列展開有り
predict.c	14	104	128/38 (+236%)	75	記述の並列展開有り
enc_core.c	315	253	390/452 (-14%)	40	
ENCCNT.c	-	284	284/- (-%)	-	新規設計

表【2-2】. 7 Decode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール B)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
dec_main.c	-	-	-	-	高位合成対象外
yuv2rgb.c	42	144	267/165 (+61%)	70	
ipct.c	3	56	300/244 (+22%)	85	記述の並列展開有り
iquantize.c	10	98	141/53 (+171%)	85	記述の並列展開有り
ipredict.c	9	100	130/39 (+233%)	80	記述の並列展開有り
dec_core.c	100	474	941/567 (+65%)	80	
LBUFCNT.c	-	-	150/- (-%)	-	新規設計
DECCNT.c	-	-	160/- (-%)	-	新規設計

高位合成記述として入出力変数や高位合成オプションを追記しているため、殆どのコードにおいて行数が多くなっている。しかし、アルゴリズムが記述された部分にはあまり手を加えていないため流用率は高い。

ただし、「enc_core.c」は例外で、行数が減り流用率が低い。これは、高位合成用コードに書き換えた時に、合成対象関数内で使用していない不要な関数を削除したことにより行数が減り、制御系記述を追加した箇所のボリュームが大きいため、元のアルゴリズムコードの流用率が低くなっている。

（検証作業）

高位合成時に、サイクルベース SystemC モデルも同時に生成するので、SystemC にて接続を行い全体検証が可能となる。

今回の調査では作業者が SystemC 設計に不慣れであるため、作業工数が多くかかっているが、フレームワークを利用することで工数削減が見込まれる。

（論理合成）

ALTERA 社の Stratix II (EP2S180F1508C5)で論理合成を行い、規模の確認を行った。下記表にトータル・論理回路・レジスタの利用率を示す。

＜補足＞

- 合成対象関数毎に論理合成を実行し、全結果を集計
- 「Logic utilization」レポート結果が 1%以下の場合、1%の使用率として集計

表【2-2】. 8 論理合成規模(高位合成ツール B)

論理合成対象	Logic Utilization(%)	Combinational ALUTs(%)	Dedicated logic registers(%)
Encode	78	60.4	2.8
Decode	86	65.7	8.8

上記結果より、高位合成では組み合わせ論理が極端に多くなる傾向が見られる。

これは「高位合成ツール B」のアルゴリズムの特色であり、高位合成時に生成した DFF を極力使いまわすように合成されるため、セレクタ論理が多くなり、レジスタ数が少なくなる。

回路規模を削減させるためには、合成対象関数の細分化が有効と思われるが、これは高位合成ツールを使用する利点から逸脱していると考えられるため、今回は検討対象外とした。

3-2-2-1-3 高位合成ツール C を使用した開発

(開発内容説明)

■Encode ブロック

要求仕様のスループット(2pixel/サイクル)を満たす為、rgb2yuv 以降を 2 並列、enc_core を 13 並列で構成した。

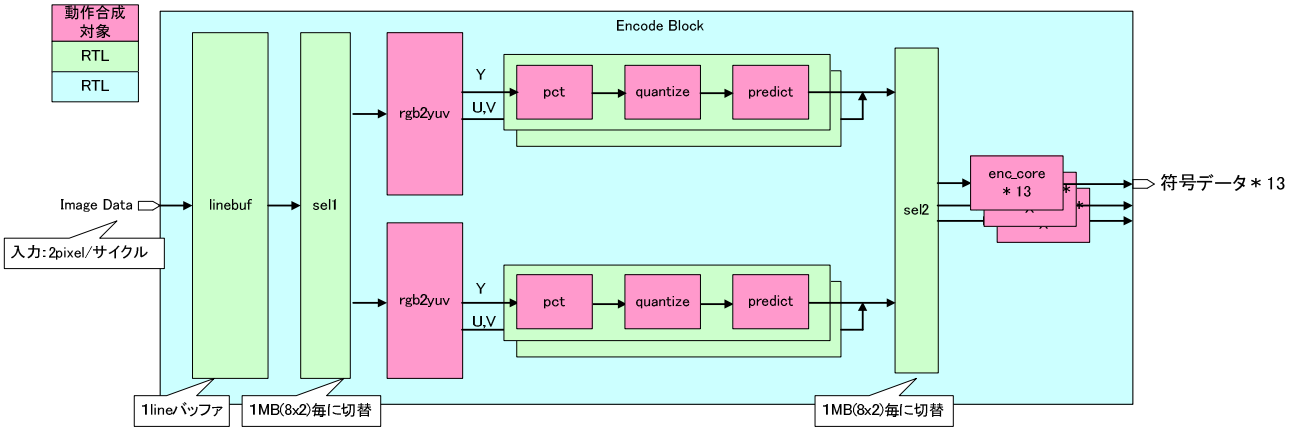
rgb2yuv のスループットは 2x2 pixel/サイクルであるが、初期検討用に高位合成した回路では 2x2 pixel の処理に 2 サイクル必要であった。高位合成ツール C ではデータ入力から出力までをパイプライン処理できないため、2x2pixel を 4 つ処理し 8x2pixel でまとめて出力するように更にコードを書き換えることで 5 サイクルでの処理が可能になった。これ以上の処理サイクル短縮が見込めないため、rgb2yuv 以降を 2 並列にすることで要求仕様のスループットを満たした。

enc_core は初期に高位合成した回路では出力までに 62 サイクルであったが、更にコードの修正を行い 49 サイクルまで短縮した。これ以上の処理サイクル短縮が見込めないため、13 並列にすることで要求仕様のスループットを満たした。

以下に回路構成ブロックを示す。

- linebuf (RTL)
  入力:2x1 pixel、出力:2x2 pixel
  2pix/clock のデータをラインバッファに一旦格納し、次のラインが入力されてくるときにラインバッファから後段のブロックへデータを出力する。
- sel1
  入力:2x2 pixel、出力:2x2 pixel
  8x2 (1MB)毎に出力経路を切替える。
- rgb2yuv (高位合成)
  入力:2x2 pixel、出力:8x2 pixel
  2x2 の RGB データを YUV に変換し、8x2 で出力する。
- pct、quantize、predict (高位合成)
  入力:8x2 pixel、出力:8x2 pixel
  8x2 単位で係数変換、量子化、係数予測処理を行う。
- sel2
  入力:8x2 pixel、出力:8x2 pixel
  8x2 pixel 毎に出力経路を切替える。
- enc_core (高位合成)
  入力:8x2 pixel、出力:圧縮符号データ
  8x2 単位で 圧縮符号データに変換する。

下記に Encode ブロック構成図を示す。



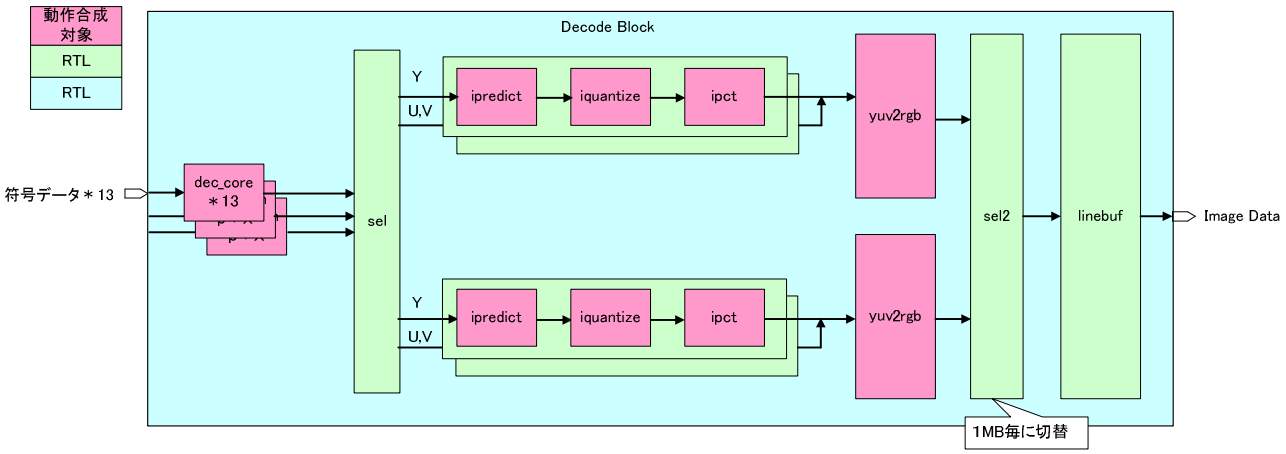
図【2-2】. 8 Encode ブロック構成(高位合成ツール C)

■Decode ブロック

要求仕様のスループット(2pixel/サイクル)を満たす為、dec_core を 13 並列・iPREDICT 以降を 2 並列、で構成した。ENCODE 回路と対称になる構成となっている。

- dec_core (高位合成)
  入力:圧縮符号データ、出力:8x2pixel
  圧縮符号データを 8x2(1MB)単位のデータに変換する。
- sel1
  入力:8x2 pixel、出力:8x2 pixel
  1MB 毎に出力経路を切替える。
- ipct、iquantize、ipredict (高位合成)
  入力:8x2 pixel、出力:8x2 pixel
  8x2 単位で逆係数変換、逆量子化、逆係数予測処理を行う。
- yuv2rgb (高位合成)
  入力:8x2 pixel、出力:2x2 pixel
  8x2 の YUV データを RGB に変換し、2x2 で出力する。
- sel2
  入力:2x2 pixel、出力:2x2 pixel
  8x2 pixel 毎に入力経路を切替える。
- linebuf (RTL)
  入力:2x2 pixel、出力:2x1 pixel
  1 ライン分のデータを格納する。

下記に Decode ブロック構成図を示す。



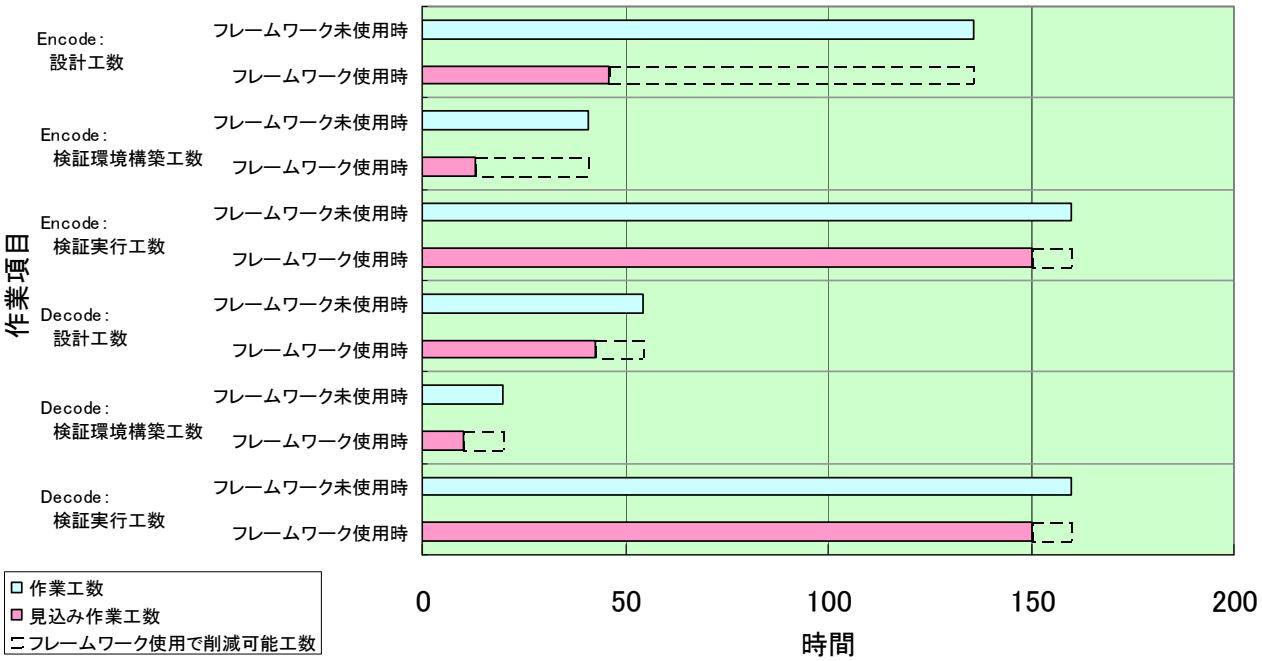
図【2-2】. 9 Decode ブロック構成(高位合成ツール C)

(設計総工数)

本ツールを用いた JPEG-XR 開発工数を下記グラフに示す。

＜補足＞

- フレームワーク未使用時　　：フレームワークを利用しない場合の「作業工数」。
- フレームワーク使用時　　：フレームワークを利用した場合の「見込み作業工数」。



図【2-2】. 10 見込み作業工数(高位合成ツール C)

（設計記述量）

上記で説明したように回路を構成して元のアルゴリズムコードを高位合成用コードに書き換えた。下記表に元のアルゴリズムコードと高位合成用コードのコード量比較結果を示す。

削除行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードから削除した行数。

追加行数　：　高位合成用コードに書き換え時、元のアルゴリズムコードに追加した記述の行数。

行数　　：　各コードの総行数と増減率［高位合成用コード行数/元のアルゴリズムコード行数］

流用率　　：　元のアルゴリズムコードに手を加えず、流用した箇所を割合で表記。全ての記述を流用できれば 100%（最大）、全ての記述を書き換えた場合は 0%（最小）となる。

表【2-2】. 9 Encode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール C)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
enc_main.c	-	-	-	-	高位合成対象外
rgb2yuv.c	75	208	268/143(+87%)	20	記述の並列展開有り
pct.c	87	221	380/246(+54%)	60	記述の並列展開有り
quantize.c	33	127	156/62(+152%)	60	記述の並列展開有り
predict.c	12	98	124/38(+226%)	50	記述の並列展開有り
enc_core.c	74	1028	1406/452(+211%)	5	記述の並列展開有り

表【2-2】. 10 Decode ブロック 高位合成用コード対元のアルゴリズムコード量比較(高位合成ツール C)

コード名	削除行数(行)	追加行数(行)	行数(行)	流用率(%)	備考
dec_main.c	-	-	-	-	高位合成対象外
yuv2rgb.c	68	221	318/165(+93%)	40	記述の並列展開有り
ipct.c	18	79	100/39(+156%)	60	記述の並列展開有り
iquantize.c	19	71	105/53(+98%)	60	記述の並列展開有り
ipredict.c	95	230	379/244(+136%)	50	記述の並列展開有り
dec_core.c	66	376	886/576	10	記述の並列展開有り

アルゴリズムコードに高位合成用記述を追加した為、全てのファイルでサイズが大きくなった。

文字数・行数も同じ傾向となった。

enc_core.c、dec_core.c は流用率が低くなったが、その理由はスループットを満たすためにデータ処理順序を修正したためである。

enc_core.c、dec_core.c 以外はポートとなる引数を書き換える等の軽微な修正であり、流用率は高くなった。

(検証作業)

高位合成結果の RTL を SystemC 言語にて RTL 検証を行う。

(論理合成)

ALTERA 社の Stratix II (EP2S180F1508C5)で論理合成を行い、規模の確認を行った。下記表にトータル・論理回路・レジスタの利用率を示す。

＜補足＞

「Logic utilization」レポート結果が 1%以下の場合、1%の使用率として集計

表【2-2】. 11 論理合成規模(高位合成ツール C)

論理合成対象	Logic Utilization(%)	Combinational ALUTs(%)	Dedicated logic Registers(%)	備考
Encode 高位合成 enc_core 以外	7	6	1	
Encode 高位合成 enc_core	60	50	1	単体での合成結果
Decode 高位合成	-	-	-	未実施

上記結果より、組み合わせ回路(Combinational ALUTs)が多くなっていることが分かる。これは高位合成ツール C を使用して高位合成した RTL では組み合わせ回路が大きくなるためである。また、enc_core は他の論理合成対象の RTL より演算処理が多いため、組み合わせ回路が他のものより大きくなったと考えられる。

また組み合わせ回路に比べ、DFF(Dedicated logic registers)が少なくなっている。これは DFF を使いまわすように合成されるためである。

3-2-3　まとめ

3-2-3-1　成果

3 社の高位合成ツールを利用した開発について、一部作業を除きほぼ完了した。

図【2-2】. 4、図【2-2】. 7、図【2-2】. 10 の各高位合成ツールによる作業工数を見て分かるように、Encode の高位合成の経験を生かすことで、Decode の高位合成ツールによる設計工数を削減することができる。同様に、検証の検証実行工数においても削減することができる。このことより、本調査で得た経験を手順書にフィードバックすることで、今後の同様の開発において設計工数および検証実行工数の生産性をそれぞれ 50%および 35%向上することができる。

また、検証環境構築工程の効率化を図るため、画像入出力機能を持つ共通の SystemC 検証環境のベースを作成した。このベースをフレームワークに組み込み再利用することで、今後の同様の開発において検証環境立ち上げ工程の生産性を 80%向上することができる。

3-2-3-2　結論

高位合成ツールを利用した開発の場合、以下の特徴により生産性を向上することが可能と見込む。

＜特徴＞

- 高位合成用コードはアルゴリズムコードの流用率が高く、RTL 記述よりコーディング時間が短い。
- 記述による回路の論理段数を考慮する必要がない。
- 出力の有効桁数が決定している場合は、途中演算の有効桁数を考慮する必要がない。
- 高位合成コードに対しての論理段数を容易に確認できる機能を持つツールもあり、パフォーマンス改善すべき箇所が容易に特定できる。
- インターフェース機能が容易に設定できる機能を持つツールもあり、制御系の記述の削減が行える。
- レイテンシーとスループットが容易に確認できる機能を持つツールもあり、記述に対する動作パフォーマンスの確認が容易である。

また、高位合成ツールを利用した開発において、より生産性を向上させるための課題を検討した。

■タイミング設計時のユーザビリティ向上

ハンドシェイクや、入出力タイミングに制約がある回路の場合、タイミングを意識したコーディングを行う必要があるため、実際に合成しデバッグしなければならない。

タイミング設計部分をツール GUI やオプションで、グラフィカルに設定（または、ファイルなどで設定）を行えると、ソースコードに手を加える機会が減り、効率化や不具合混入の減少に繋がると思われる。

■データ処理系と制御系記述の切り離し

今回のモチーフでは、一部データ処理系と制御系の記述が混在する関数を高位合成対象としていたため、設計制約を満たす合成結果を得られるまでに時間を要した。また、他の制御系、設計制約の場合、今回得られた設計方法がそのまま生かせる訳ではなく、別のアプローチが必要であると考えられる(オプションの使用方法、コーディング方法など)。

このため、データ処理系と制御系記述を分離させ、制御系は SystemC で記述するなど、体系的な設計方法の確立が必要と思われる。

データ処理系と制御系を分離させて高位合成する場合、それぞれの出力信号を DFF 出力で高位合成する必要がある。これは、それぞれの出力信号が組み合わせ出力であると、それぞれを接続し、

論理合成まで行わないと動作速度の保証が困難となるためである。また、この場合、出力信号に追加した DFF により、レイテンシーが大きくなることに留意しなければならない。

■リントツールの作成、利用

各社高位合成ツール向けのリントチェッカーを作成し利用することで、高位合成後の回路が意図しない動作となることを未然に防ぐことが可能であると考える。現時点で考えられるチェック内容としては以下のものが挙げられる。

1) 入力ポート信号の直接参照が 2 回以上発生している場合のチェック
高位合成ツールに依存するが、直接参照する度にポートアクセスの為のサイクルが発生する可能性がある為。

2) 配列変数の代入・参照箇所をレポート
制御内容によるが、RAM アクセスが発生するため、代入・参照箇所の把握とチェックは必要。

3) 異なる変数型への代入チェック
キャストしているか否かのチェック。特にシフト演算がある時は注意する。

4) 回路規模が大きくなる記述
定数の乗算などをチェック(乗算器で高位合成を行うツールがある)。

3-3 【2-3】画像圧縮 JPEG-XR 方式開発実績の分析

3-3-1 概要・目的

最新の開発事例である画像圧縮 JPEG-XR 方式の開発実績では、アルゴリズムレベルから実装レベルまでの手戻りが多かった内容を分析して、高い生産性を示す開発手法、開発支援システムおよび開発支援ライブラリを開発する。設計納期、設計工数および設計量(プログラム記述量、検証回数など)を比較・分析し、仕様を決定する。

また、設計生産性を定量化して、トータルでの生産性向上部分をフレームワーク部分およびユニット部分に分けて仕様を設計する。

3-3-2 内容

3-3-2-1 RTL 手設計開発工数の分析

従来の RTL 手設計による JPEG-XR の開発実績を工程毎に分析し、生産性向上のためのポイントを考察する。

RTL 手設計の工数は、プロジェクト進行時に利用していた工数データベースから工程毎の集計を行った。集計結果を次の表に示す。

表【2-3】. 1 RTL 手設計工程工数

	実工数(h)	工数割合(%)	主観工数(h)
PJ 管理	46.75	8.25	
要求理解	82.25	14.51	82.25
要求仕様書作成	4.00	0.71	
要求レビュー	9.00	1.59	
設計仕様書作成	15.25	2.69	
設計コーディング	92.75	16.36	110.50
設計レビュー	1.75	0.31	
試験仕様書作成	12.50	2.20	
検証環境構築	95.25	16.80	95.25
検証実行	177.50	31.31	216.50
不具合対応	2.25	0.40	
検証レビュー	4.50	0.79	
合成、フィティング	18.50	3.26	
合成レビュー	3.75	0.66	
(その他)	1.00	0.18	
合計	567.00		

※担当者の経験差による偏りを補正した工数を「主観工数」としている

「要求理解」「設計コーディング」「検証環境構築」「検証実行」の各項目で全体の 10%を超える値を示しており、時間がかかっていることが判る。

ここで 10%を超える項目に対して作業項目を分解して次の表に示す。

表【2-3】. 2 RTL 手設計工程工数（詳細分解）

作業項目	実工数(h)	主観工数(h)
要求理解（合計）	82.25	82.25
ENC/DEC 仕様理解：C ソース理解		22.25
ENC/DEC アルゴリズム RTL 化検討		20.00
全体構造検討：RAM 構成、全体タイミング		35.00
その他		5.00
設計コーディング（合計）	92.75	110.50
ENC/DEC クリティカルパス確認用、仮コーディング・合成		6.00
ENC 前処理/DEC 前処理		29.50
RGBYUV		5.50
ENC コーディング		23.75
DEC コーディング		6.00
		(17.75)
ラインメモリ制御、全体制御回路、ブロック TOP		22.00
検証環境構築（合計）	95.25	95.25
検証環境検討		15.25
期待値生成 C モデル修正		10.00
画像スクリプト作成		15.00
子マクロベンチ作成		30.00
TOP ベンチ作成		10.00
ベンチ修正(構成変更)		15.00
検証実行（合計）	177.50	216.50
シナリオ作成		20.00
実行スクリプト作成		5.00
ENC/DEC 前処理		43.50
RGBYUV		11.25
ENC 検証		41.50
DEC 検証		2.50
		(39.00)
ラインメモリ制御、全体制御回路		33.75
顧客担当ブロック		20.00

※DEC の担当者が可変長符号の経験者であり、ENC に比べると極端に工数が少ない。工数比較による正当な生産性向上率導出のため、DEC の設計・検証工数は ENC と同じ工数となるよう括弧書きで補正している。

3-3-2-1-1 設計工程の分析

設計に着目して工数分析を行う。「設計コーディング」では全体の工数の 16%を占めている。各モジュールの

工数の内訳は次の通りである。

- ENC／DEC 共通
  可変長符号設計の経験者がアルゴリズム C から、DEC／ENC 処理のスケジューリングを事前検討した。
- ENC
  ENC は上記スケジューリング検討に基づき、アルゴリズム C を HDL へ記述変更。
  C 言語ではタイミングを意識しなくて済むが、HDL の場合は判定フラグを DFF で持ち越す重たい処理は、参照されるタイミングの数クロック前から判定を開始する検討が必要となった。
- RGB2YUV、PCT 等
  画像処理未経験者が試行錯誤の時間も含めて設計した為、工数がかかっている。

3-3-2-1-2 検証工程の分析

次に検証に着目して工数分析を行う。ENC ではC 言語の期待値の不一致が発生し、調査の繰り返しが発生した。原因として次の理由があげられる。

- モジュール結合した状態で、フレーム単位のデータを流して検証していたため、期待値不一致が発生した時に C 言語の期待値と回路シミュレータ波形の問題パターンの発生箇所を一致させるのに時間がかかった。
- 出力期待値（コード化された出力）のエラー箇所と元の画像の位置の特定に手間（時間）がかかった。
- アルゴリズムCコードとRTL の対応箇所の中間ダンプをする為に、アルゴリズムCコードの中間ダンプを適宜入れるのに手間（時間）がかかった。
- RTL とFIFO に出力するバレルシフタ、前処理モジュール、ENC モジュール本体、とデバッグ対象が 3 箇所にあったので、問題の切り分けが必要となった。
- バレルシフタ等、ENC 処理以外の部分の単体デバッグを行っていなかったので、どこが正しいか間違っているのか、順番に確かめていく必要があった。
- 論理合成後の要求クロック周波数を満たせなかったので、コード修正・再検証も必要となった。（今回は全体の構成を大きく変えるほどの修正ではなかった）

ENC（エンコードモジュール）のバグ発生箇所は次の通りである。

- バレルシフタ（RTL コーディングミス）
- ENC 本体の処理（アルゴリズム C コードからの記述変更ミス／RTL コーディングミス）
- カウンタのカウント数間違え
- 制御パルス発生タイミングミス、等

DEC（デコードモジュール）のバグ発生箇所は次の通りである。

- アンダーフロー検出ミス（2 ワードの先読み制御の考慮漏れ）
- 10 進数／16 進数記述ミス（RTL コーディングミス）

3-3-2-2　高位合成ツールを使用した開発との比較

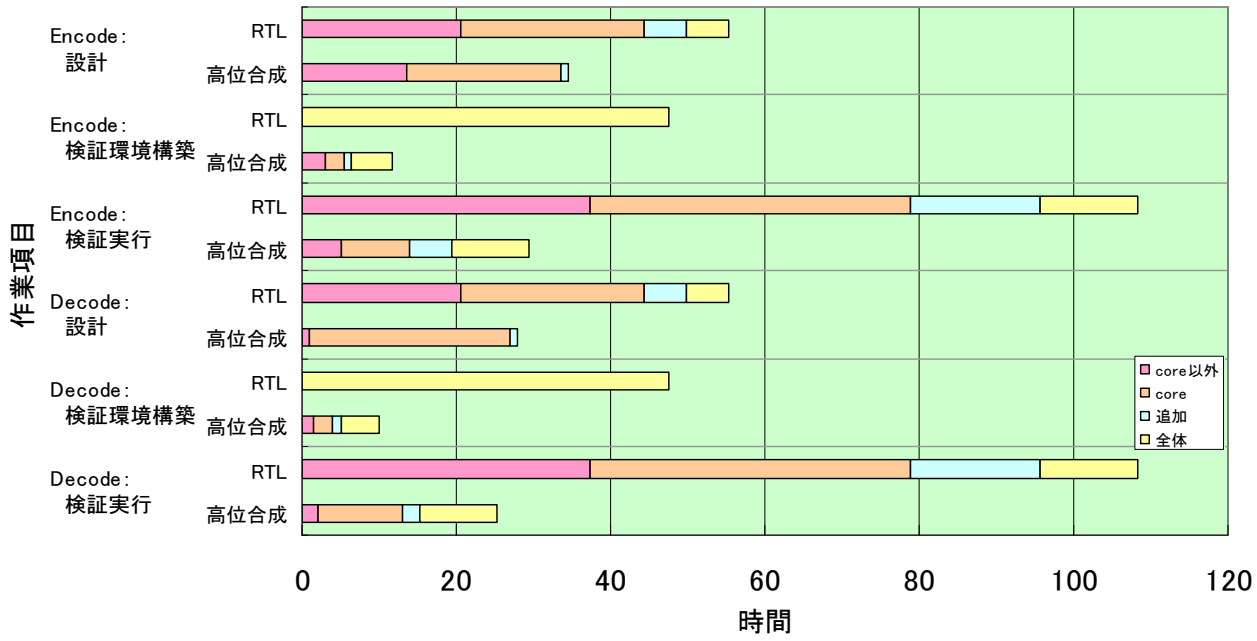
従来の RTL 手設計による JPEG-XR の開発工数と【2-2】で行った高位合成ツールを使用した開発の工数を比較し、生産性向上が見込める工程の考察を行う。

3-3-2-2-1 高位合成ツール A を使用した開発との比較

ここからは RTL 手設計と高位合成ツール A を使用した場合の開発比較を行う。

「高位合成ツール A を利用した開発」と「従来の手法である RTL 手設計」を比較する。

下記のグラフは、「高位合成ツール A を利用した開発」と「従来の手法である RTL 手設計」の設計総工数を比較したものになる。

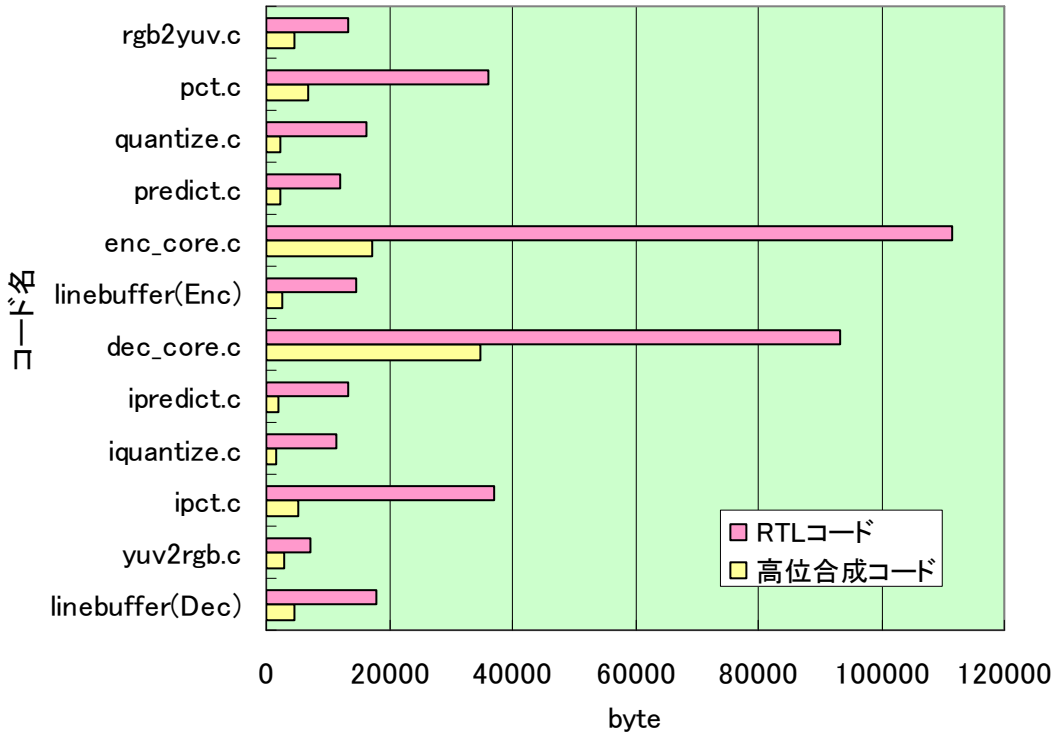


図【2-3】. 43 高位合成ツール使用設計対 RTL 手設計工数比較（高位合成ツール A）

EncodeブロックとDecodeブロックの見込み総工数は、上記のグラフのように「設計工数は40%程度」「検証環境構築工数は75%程度」「検証実行工数は70%程度」削減できると推測する。各項目で削減された理由は後に記載するが、全体で 65%程度削減が可能であろう。

■コード量の比較

下記のグラフは、手書き RTL と高位合成用コードとを比較した結果になる。



図【2-3】. 44 高位合成用コード対 RTL コードファイルサイズ比較(高位合成ツール A)

記述量を比べると、高位合成コードの方が RTL コードに比べどのコードでも少ない。高位合成コードはアルゴリズムコードから流用しているので、さらに差は大きくなる。その為設計工数が 40%程度削減できると推測する。

■検証の比較

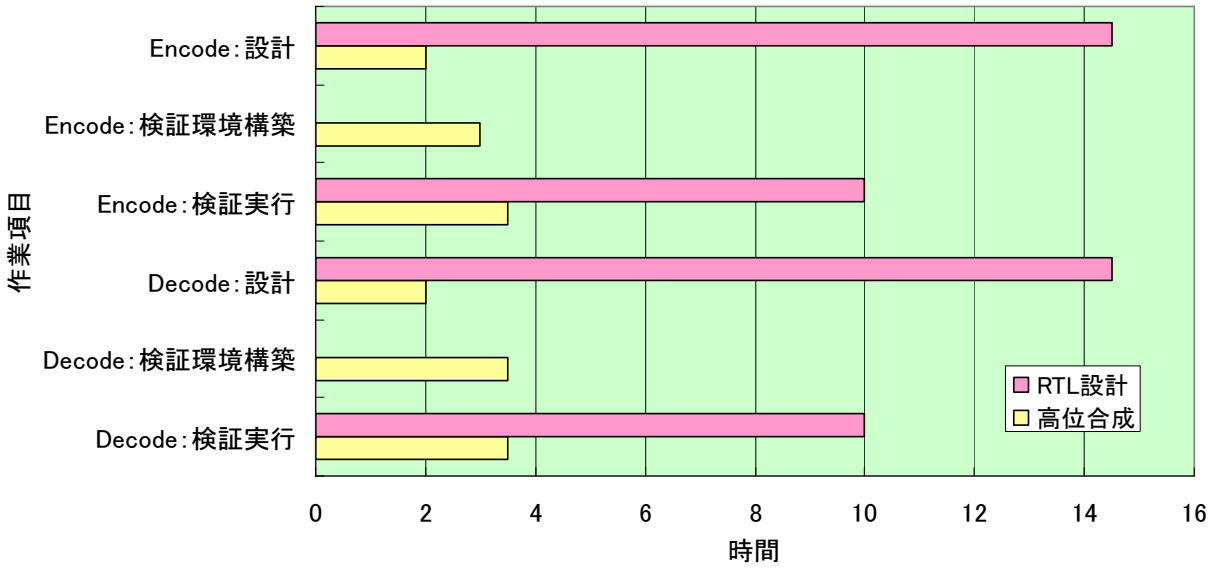
検証環境構築工数は、アルゴリズムコードと高位合成用コードの等価検証の工数が追加される。しかし SystemC/RTL 検証の環境構築を行う時間は、フレームワークを利用により従来の設計手法と比べ格段に少なくなる。その為検証環境構築工数が 75%程度削減できると推測される。

検証実行工数は、検証環境構築工数と同様にアルゴリズムコードと高位合成用コードの等価検証の工数が追加される。しかしこの等価検証により、アルゴリズムと高位合成モジュールの動作が一致していることを確認できる。その為検証シナリオを使った SystemC/RTL 検証では、接続検証とモジュール間のインターフェース検証が主な作業内容となり、従来の RTL 手設計で必要であったモジュール単体の検証時間が削減される。よって、全体の検証実行工数として 70%程度削減できると推測される。

■アルゴリズムを変更した場合

仕様変更が発生した場合における、高位合成による開発の優位性を考察する。

圧縮アルゴリズムの単位を 8x2 のブロック単位から 8x8 のブロック単位に変更した場合に追加される修正工数を見積もった結果、下記グラフの様になった。グラフには RTL 手設計と高位合成設計の工数が比較できるように並べて記載している。



図【2-3】. 45 アルゴリズム変更時 高位合成ツール使用設計対 RTL 手設計工数比較（高位合成ツール A）

修正工数は Encode ブロックで 9h、Decode ブロックで 9h、合計で 18h となった。（修正工数の内訳は下記表にて記載している。）これは、RTL 手設計での見積もり修正工数の約 35%になる。

開発設計と同様に修正工数も 65%程度削減されるのではないかと考える。

下記表に工数の内訳を記載した。なお表には RTL 手設計での修正工数も載せている。

表【2-3】. 3 アルゴリズム変更時作業項目(高位合成ツール A)

作業項目	設計方法	作業内容	追加工数(h)	合計(h)
Encode 設計	高位合成	enc_core 以外	1.0	2.0
		enc_core	0.5	
		linebuffer	0.5	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		enc_core 設計変更	8.0	
		enc_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Encode 検証環境構築	高位合成	enc_core 以外等価検証	0.5	3.0
		enc_core 等価検証 ※期待値モデル修正含む	1.0	
		linebuffer 等価検証	0.5	
		SystemC 接続変更	1.0	
	RTL 手設計	無し	0.0	0.0
Encode 検証実行	高位合成	enc_core 以外等価検証	0.5	3.5
		enc_core 等価検証	0.5	
		linebuffer 等価検証	0.5	
		接続検証	2.0	
	RTL 手設計	RTL 検証	10.0	10
Decode 設計	高位合成	dec_core 以外	1.0	2.0
		dec_core	0.5	
		linebuffer/selector	0.5	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		dec_core 設計変更	8.0	
		dec_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Decode 検証環境構築	高位合成	dec_core 以外等価検証	0.5	4.0
		dec_core 等価検証 ※期待値モデル修正含む	0.5	
		linebuffer/selector 等価検証	0.5	
		SystemC 接続変更	2.0	
		ベーステストベンチ変更	0.5	
	RTL 手設計	無し	0.0	0.0
Decode 検証実行	高位合成	dec_core 以外等価検証	0.5	3.5
		dec_core 等価検証	0.5	
		linebuffer/selector 等価検証	0.5	
		接続検証	2.0	
	RTL 手設計	RTL 検証	10.0	10.0

下記表に各コードの具体的な修正箇所を示す。

表【2-3】. 4 アルゴリズム変更によるコード修正箇所(高位合成ツール A)

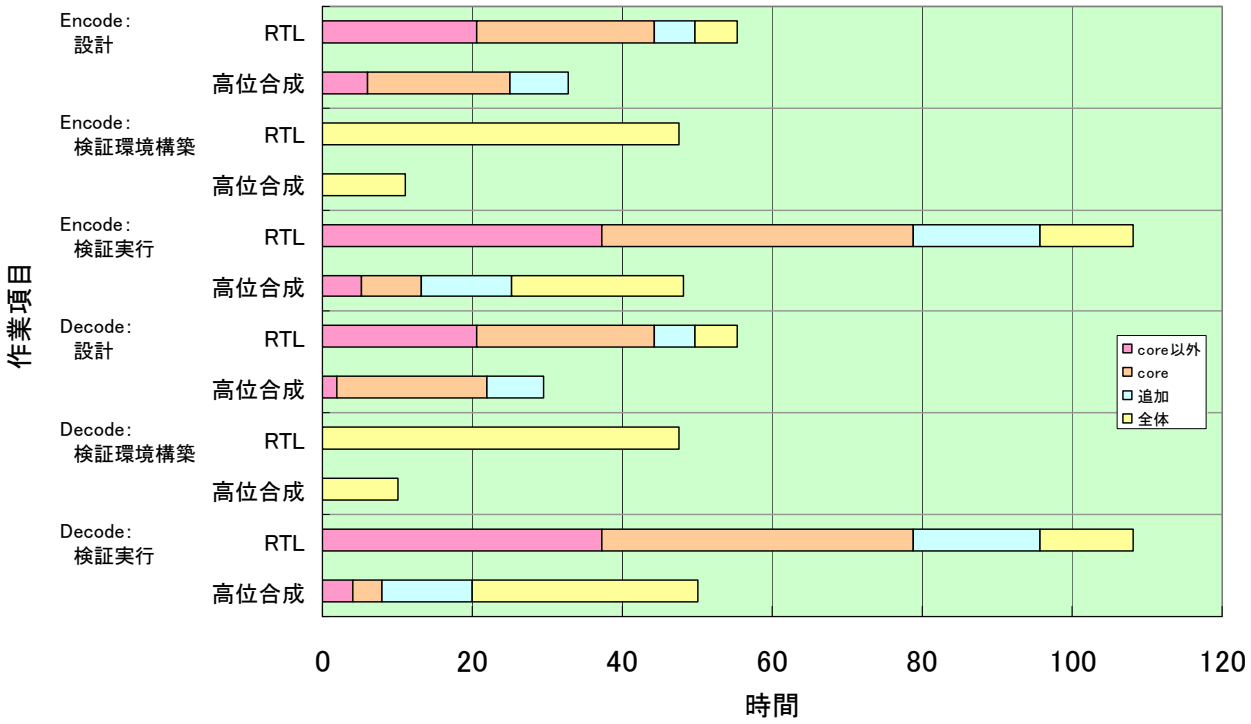
コード名	修正箇所	修正時間(h)
pct.c	8x2 ブロックの高位合成コードを破棄し、 8x8 ブロックのアルゴリズムコードから変更。	+0.5
enc_main.c quantize.c predict.c	「変数の宣言」と「ループ文の実行回数」を変更。	+0.5
enc_core.c	「変数の宣言」と「ループ文の実行回数」と「分岐条件」を変更。	+0.5
linebuffer.c (enc)	8x2 ブロックから 8x8 ブロックに変更。 ※「変数の宣言」と「ループ文の実行回数」と「分岐条件」を変更。	+0.5
ipct.c	8x2 ブロックの高位合成コードを破棄し、 8x8 ブロックのアルゴリズムコードから変更。	+0.5
enc_main.c ipredict.c iquantize.c	「変数の宣言」と「ループ文の実行回数」を変更。	+0.5
dec_core.c	「変数の宣言」と「ループ文の実行回数」と「分岐条件」を変更。	+0.5
linebuffer.c (dec)	8x2 ブロックから 8x8 ブロックに変更。 ※「変数の宣言」と「ループ文の実行回数」と「分岐条件」を変更。	+0.5
selector.c	32 並列動作に変更。	+0.5

3-3-2-2-2 高位合成ツール B を使用した開発との比較

ここからは RTL 手設計と高位合成ツール B を使用した場合の開発比較を行う。

(工数の比較)

図内[RTL]は RTL 手設計時の工数、「高位合成」は高位合成ツール B を利用した開発の工数である。

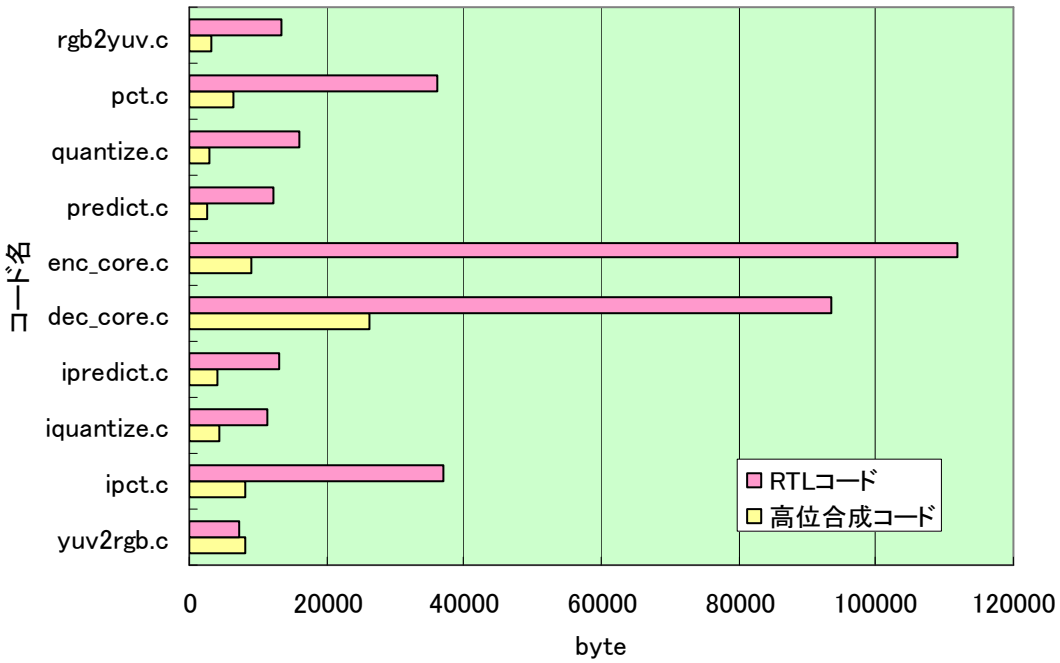


図【2-3】. 46 高位合成ツール使用設計対 RTL 手設計工数比較 (高位合成ツール B)

- Encode ブロックの設計は、RTL 手設計に対して約 40%の工数削減。
- Decode ブロックの設計は、RTL 手設計に対して約 45%の工数削減。
- 検証環境はフレームワークを使用することにより、Encode/Decode ブロックそれぞれ約 30%の工数削減。
- 検証実行時間は SystemC 環境のため実行速度が向上し、Encode/Decode ブロックそれぞれ約 55%の工数削減。

(コード量の比較)

下記のグラフは、手書き RTL と高位合成用コードのファイルサイズを比較したものである。



図【2-3】. 47 高位合成用コード対 RTL コードファイルサイズ比較(高位合成ツール B)

- RTL ソースに比べて、高位合成用コードの記述量が少なくなるという結果が得られた。
- 高位合成用コードは、元のアルゴリズムコードを流用している部分が多いため、さらに記述する行数は少なくなる。
- 単語数、行数を比較した場合も上記のグラフと同じ傾向が見られる。

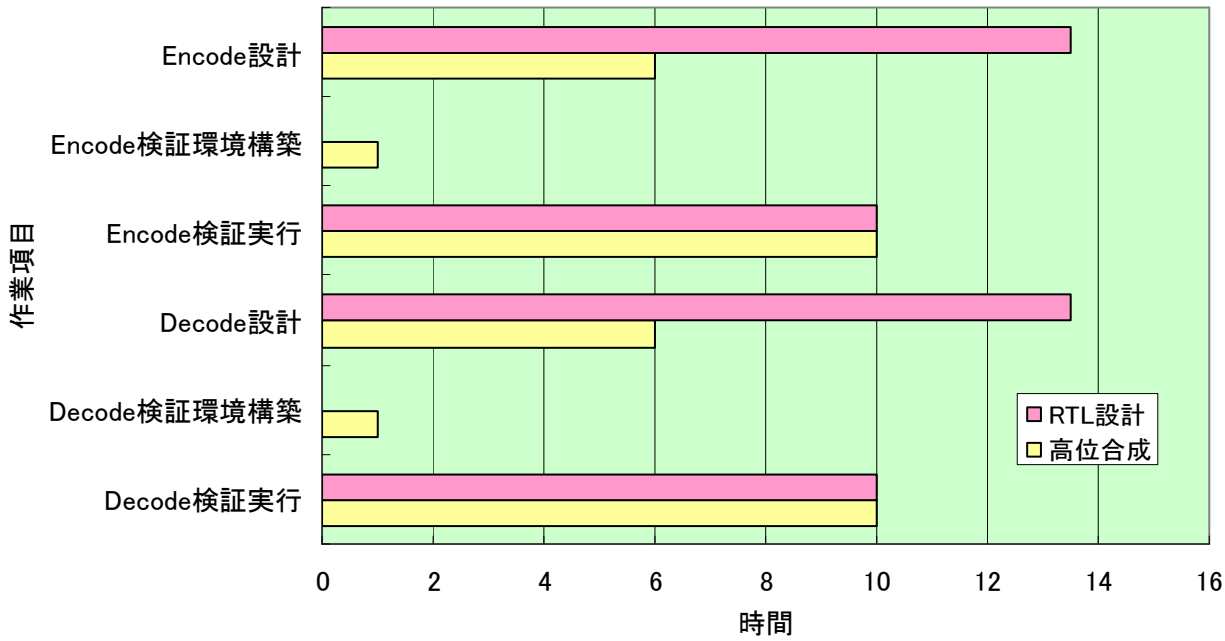
(検証の比較)

高位合成ツールがサイクルレベル SystemC モデルを生成するので、フレームワークを利用し SystemC モデルの接続や、テストベンチひな形生成などを自動化することで、検証環境構築の際、RTL 手設計時と比べ、大幅な効率化が見込まれる。

Encode/Decode それぞれのシミュレーション実行時間が、RTL 手設計時と比べ約 1/2 となった。検証環境が SystemC 環境となり、これまでの Verilog HDL ベースの環境と比べシミュレーション実行速度が向上したことにより、検証作業の効率化も見込める。

（アルゴリズムを変更した場合の考察）

今回のモチーフでは圧縮アルゴリズムの処理単位を 8x2 の 1 マクロブロックで行ったが、8x8 に変更した場合の予想工数を下記に示す。



図【2-3】. 48 アルゴリズム変更時 高位合成ツール使用設計対 RTL 手設計工数比較(高位合成ツール B)

表【2-3】. 5 アルゴリズム変更時作業項目(高位合成ツール B)

作業項目	設計方法	作業内容	追加工数(h)	合計(h)
Encode 設計	高位合成	各合成対象関数の入出力変数変更	1.0	6.0
		マクロブロック処理のループ文変更	1.0	
		enc_core 制御記述変更	2.0	
		enc_core 並列数変更	2.0	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		enc_core 設計変更	8.0	
		enc_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Encode 検証環境構築	高位合成	SystemC 接続変更	1.0	1.0
	RTL 手設計	無し	0.0	0.0
Encode 検証実行	高位合成	SystemC/RTL 検証	10.0	10.0
	RTL 手設計	RTL 検証	10.0	10.0
Decode 設計	高位合成	各合成対象関数の入出力変数変更	1.0	6.0
		マクロブロック処理のループ文変更	1.0	
		dec_core 制御記述変更	2.0	
		dec_core 並列数変更	2.0	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		dec_core 設計変更	8.0	
		dec_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Decode 検証環境構築	高位合成	SystemC 接続変更	1.0	1.0
	RTL 手設計	無し	0.0	0.0
Decode 検証実行	高位合成	SystemC/RTL 検証	10.0	10.0
	RTL 手設計	RTL 検証	10.0	10.0

高位合成では、マクロブロック処理のループ文や入出力変数の書き換え、検証環境の接続変更のみで、圧縮アルゴリズムの変更を行うことが出来る。制御系や入出力信号タイミング設計における工数の比重が大きいため、アルゴリズムの変更による追加工数量は微少と考えられる。

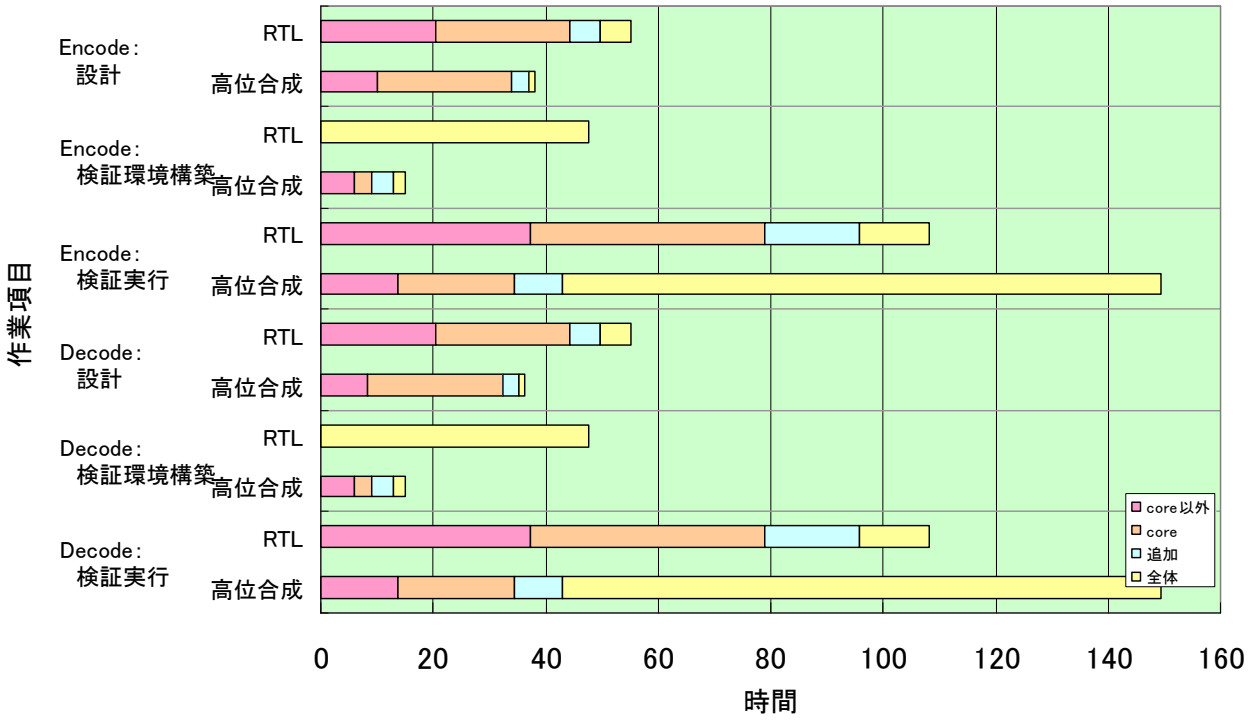
一方 RTL 手設計では処理単位の変更により、ラインバッファや enc_core および dec_core の全体制御記述などに変更を加える必要があり、高位合成に比べ設計段階における追加工数が大きい。

3-3-2-2-3 高位合成ツール C を使用した開発との比較

ここからは RTL 手設計と高位合成ツール C を使用した場合の開発比較を行う。

（工数の比較）

RTL 手設計での設計工数と高位合成ツール C を使用した場合の見込み設計工数を以下に示す。

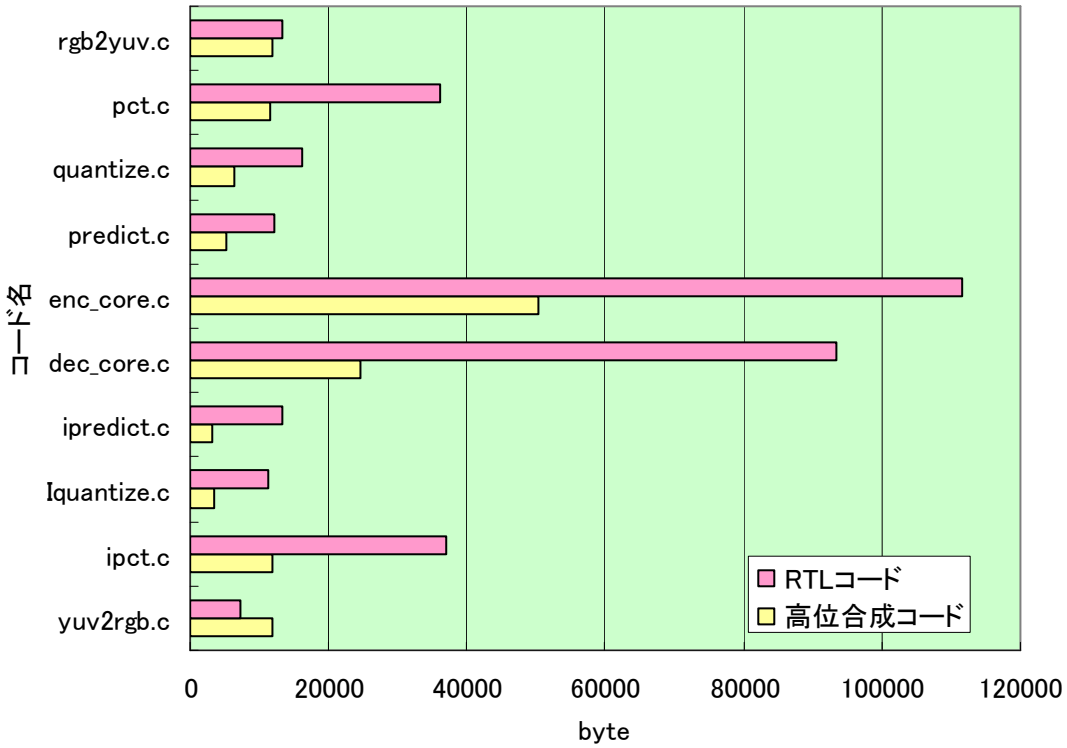


図【2-3】. 49 高位合成ツール使用設計対 RTL 手設計工数比較（高位合成ツール C）

- 設計は高位合成手順書を参照することにより、約 30%工数削減する見込みとなった。
- 検証環境はフレームワークを使用することにより、約 75％工数削減する見込みとなった。
- 検証実行はシミュレーション実行時間の増加により約 50%工数追加する見込みとなった。

(コード量の比較)

下記のグラフは、手書き RTL と高位合成用コードのファイルサイズを比較したものである。



図【2-3】. 50 高位合成用コード対 RTL コードファイルサイズ比較 (高位合成ツール C)

RTL ソースに比べて、高位合成用コードのコード量が少なくなるという結果が得られた。高位合成用コードは、アルゴリズムコードを流用して記述しているため、さらに記述量は少なくなる。

(検証の比較)

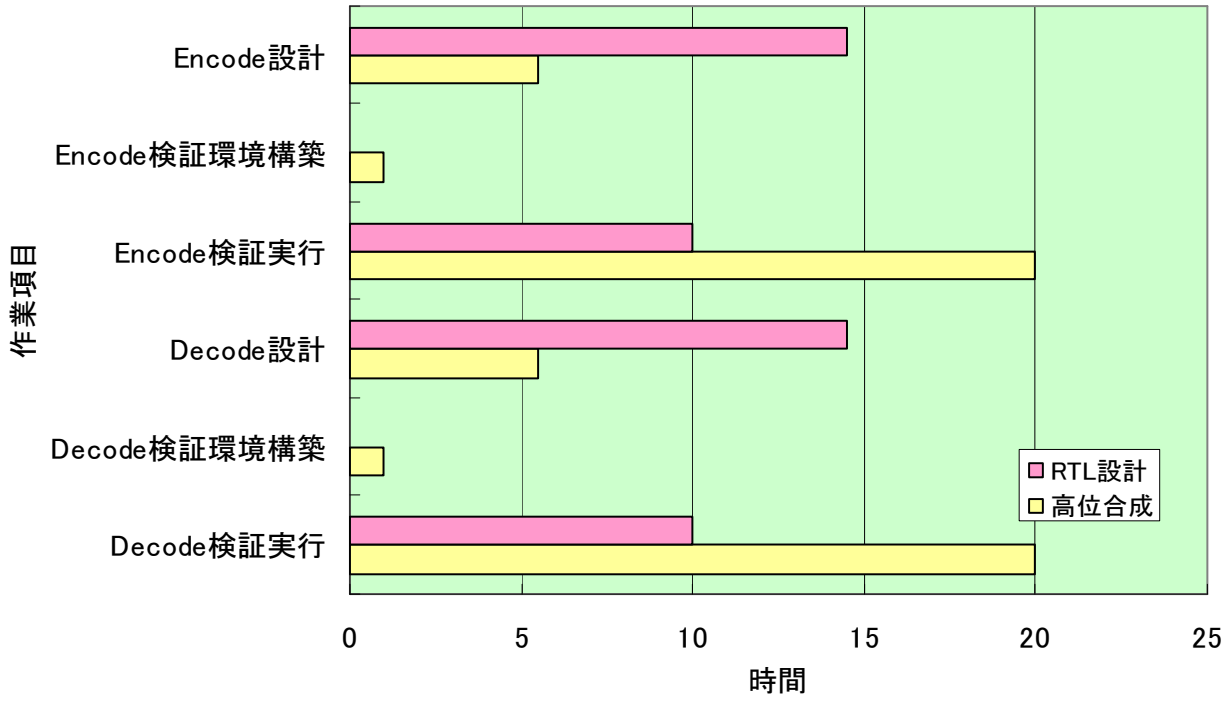
検証環境構築については、フレームワークを利用しテストベンチひな形生成などを自動化することで、検証環境構築の際、RTL 手設計時と比べ、工数削減が見込まれる。

検証実行について、高位合成の工数は等価検証と接続検証を合わせた時間になる。RTL 検証での確認項目を RTL 検証よりも高速である等価検証で行うことで RTL 検証での確認項目を削減でき、RTL 検証では接続検証のみを行うことで工数削減を期待した。しかし、高位合成結果 RTL のシミュレーション実行時間が RTL 手設計より増加したため、RTL 手設計より高位合成のほうが、検証実行の工数が増加する見込みとなった。シミュレーション時間の増加は高位合成結果 RTL の回路規模が RTL 手設計より大きいことが原因として考えられる。

(アルゴリズムを変更した場合の考察)

今回のモチーフでは圧縮アルゴリズムを 8x2 のマクロブロック単位で行った。

ここでは開発完了後に仕様変更があった場合を想定し、8x2 から 8x8 のマクロブロック単位に変更した場合の高位合成ツールを利用した場合と RTL 手設計の場合の予想追加工数を下記グラフに示す。



図【2-3】. 51 アルゴリズム変更時 高位合成ツール使用設計対 RTL 手設計工数比較(高位合成ツール C)

- 設計は、高位合成では主に入出力データ数や演算の繰り返し処理を追加するだけでよく、追加工数は少ない。
- 検証環境構築は、検証環境の信号修正のため、高位合成のほうは追加工数が多い。
- 検証実行は前述した理由により、高位合成のほうは追加工数が多い。

表【2-3】. 6 アルゴリズム変更時作業項目(高位合成ツール C)

作業項目	設計方法	作業内容	追加工数(h)	合計(h)
Encode 設計	高位合成	enc_core 設計変更	1.0	5.5
		enc_core 以外の設計変更	4.0	
		接続階層変更	0.5	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		enc_core 設計変更	8.0	
		enc_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Encode 検証環境構築	高位合成	無し	1.0	1.0
	RTL 手設計	無し	0.0	0.0
Encode 検証実行	高位合成	RTL 検証	20.0	20.0
	RTL 手設計	RTL 検証	10.0	10.0
Decode 設計	高位合成	dec_core 設計変更	1.0	5.5
		dec_core 以外の設計変更	4.0	
		接続階層変更	0.5	
	RTL 手設計	入出力バッファ設計	2.0	14.5
		dec_core 設計変更	8.0	
		dec_core 以外の設計変更	2.5	
		接続階層変更	2.0	
Decode 検証環境構築	高位合成	無し	1.0	1.0
	RTL 手設計	無し	0.0	0.0
Decode 検証実行	高位合成	RTL 検証	210	210
	RTL 手設計	RTL 検証	110	110

3-3-2-3　高位合成ツールを使用した開発との比較

今回は画像圧縮アルゴリズムに対して、RTL 手設計と高位合成ツールを利用した場合で比較を行った。メリットとデメリットを以下に示す。

1) RTL 手設計のメリット
   - 細かいタイミング設計が可能であるため、符号化回路などの IF 文の分岐やパターン判定を組み合わせる回路の場合は、自由なレイテンシー（出力までのクロックサイクル数）設計ができる。
2) RTL 手設計のデメリット
   - 要求速度を満たせない場合に処理レイテンシーを変更する必要があり、全体の構成および回路規模が変わってくる。（初期見積時の SRAM 総容量にも影響する）
   - 期待値不一致発生時に、長いデータ列中の一箇所の不一致からエラーが発生したマクロブロックおよびマクロブロック内のどの係数の符号が間違っていたかを特定するために時間がかかる。
     これは、処理中のフレーム番号および入力マクロブロック番号を示す変数と共に、各係数の符号単位にファイルダンプする仕組みを HDL と C アルゴリズムの両方に埋め込み、期待値不一致発生時に、パターンやタイミングをすぐに特定できるようにする工夫が必要である。
   - 全体の組み上げ後に問題が判明した場合、手戻りが大きい。（高位合成ツールを使えば、早期に処理レイテンシーを見積もることが出来る）
   - 可変長符号のため、圧縮／伸長部分の回路記述にバグがあった場合、問題箇所の特定に時間がかかる。
3) 高位合成ツールを利用した場合(C 言語設計)のメリット
   - ENC/DEC 処理等の合成対象関数の内部タイミングのスケジューリングをユーザーが考える必要がない。（DFF 自動挿入など）
   - 高位合成ツールを使えば、オリジナルのアルゴリズム C コードと高位合成用 C コードの間で等価検証を行うことで、トレース結果の比較が容易に行える
4) 高位合成ツールを利用した場合(C 言語設計)のデメリット
   - ツール依存だが、基本的に START／DONE 型の処理モジュールしか生成できない。（常に毎サイクルデータを取り込む動作は合成が難しい）
   - パイプライン回路（常に毎サイクルでデータを取り込む）は、制御無しの演算モジュールには適用できるが、それ以外の制御を含むモジュールへの適用はデータ制御や外部ポートアクセスタイミングの制約が多く、パイプライン合成が出来るようにするのはかなり難しい。
   - 高位合成結果のレイテンシーが要求を満たせない場合に、動作結果改善のために、高位合成ツールにあわせてソースコードを書き換える工数が発生する。
   - 高位合成結果 RTL の内部信号はネットリストに近く、内部信号を確認しながらの波形デバッグが困難である。設計時に合成対象関数内のサブ関数をモジュールとして合成しサブモジュールの出力ポートを利用してデバッグするなどの問題点を切り分けしやすい工夫が必要である。

3-3-3　まとめ

3-3-3-1　成果

高位合成ツールを利用した設計では、本来アルゴリズムでは必要の無いハードウェア特有の部分（バスインターフェースや制御回路）を表現するには RTL と同じかそれ以上の記述変更や追加を行わなければならなかった。しかし、純粋な演算アルゴリズムの部分ではユーザーが内部タイミングを考える必要がなく、さらにソースコードの変更も工数をかけずに高位合成が行える。動作周波数が満たせない場合もツールの設定を変更するだけで DFF 等を挿入し、周波数制約を満たすことが簡単に行うことができた。

RTL 手設計では細かいタイミングやブロック間インターフェースを動作仕様としてまとめた後に設計を行うため、詳細なタイミング設計を行える。しかし、回路規模、仕様変更などの問題が発生した場合、手戻りの影響範囲が大きい。このため、設計初期に動作スピードと規模を見積もるための RTL を作成することが必要である。しかし、高位合成ツールを使用した場合ではこの作業が簡単に行うことができる。

今回の画像圧縮をモチーフで RTL 手設計と高位合成設計の比較を行った。一番良い結果が得られたツールでは設計および検証での工程で 420h から 140h へと 66%削減され、3 倍の生産性を達成した。圧縮アルゴリズムを変更した場合についても比較を行ったが、一部ツールでは検証工程で RTL 手設計と同程度の工数となったが、設計変更工程も含めると全体的に効率化できる結果が得られた。

3-3-3-2　結論

高位合成ツールを利用する本来の目的としては、アルゴリズムコードから自動的に回路が生成できることによる効率化向上である。しかし、各高位合成ツールによって記述量は異なるが、アルゴリズムコードからの書き換えは必須であり、さらに制御が複雑な部分においては RTL と同等の記述が求められる場合もある。この部分を他のテーマで研究している高位合成用コード記述手順整備、および高位合成ツール毎の得意不得意部分を認識した上でのノウハウ整備や検証環境自動構築、さらに設計・検証 IP 群を整備によって圧縮することが可能となる。

高位合成ツールで生成した回路は RTL 手設計のものに比べ全般的にゲート数が大きくなる傾向にある。これは高位合成ツールが DFF を極力使いまわし、その分組み合わせ回路や配線が多くなるためである。ゲート数の増加は検証工程や論理合成工程の生産性向上を阻害する要因であり、FPGA の場合ゲートサイズの大きいものを選ぶ必要があり、コストアップに繋がってしまう。そのため、高位合成ツールで生成した回路が最適なゲート数となるように今後の進化を期待する。

3-4 【2-4】画像圧縮技術と周辺部品のユニット開発

3-4-1 概要・目的

画像圧縮技術の開発実績としては JPEG 以外に MPEG や H.246 など多数の方式が存在する。これらと JPEG_XR 方式の基本開発実績と本開発内容による生産性向上の手法に問題が無いことを検証するとともに、本項目ではメモリ全般に着目、ユニットとしての汎用性を考慮して仕様を設計し、ライブラリまたは IP 化手法を開発する。

今回開発するユニットは、画像圧縮 JPEG_XR 方式に特化したユニットとし、画像・動画処理のコア技術と周辺部品としてメモリに対してのライブラリ・IP を開発する。この時、メモリは複数種類存在するので、そのメモリ群をメモリ用ライブラリとして開発し、個別のメモリ用 IP として詳細情報を開発する。

メモリ用ライブラリはアルゴリズムレベルでメモリの詳細が決定しない状態で利用可能であり、設計の最終段階でメモリの変更が発生しても容易に変更可能なソフトウェアソース用ライブラリにする。メモリ用 IP は高位合成可能な水準の回路および詳細検証実施が可能な個別のメモリ詳細情報のソフトウェアソースにする。

本テーマでは、検証に使用する汎用的なメモリ IP および周辺 IP を開発すると共に検証に使用するライブラリ・IP（以下検証用ソフト IP）の開発フローを立案し、手順化する。また、大規模なシステムなどでは複数のモジュールを接続してシステム全体のつながりやパフォーマンスなどの検証を行う場合がある。ここでは、ソフトウェアレベルでのシステム検証（以下 SoC 検証）の検証環境構築手法を立案し、手順化する。

なお、ここで言うソフト IP とは C/C++/SystemC で設計したソフトウェアソースのライブラリや IP のことを指す。

3-4-2 内容

3-4-2-1 内容説明

3-4-2-1-1 検証に使用する汎用的なメモリ IP および周辺 IP の作成

本テーマで挙げた「検証に使用するライブラリ・IP（以下検証用ソフト IP）」とは、高位合成の対象となるコアユニットの検証を行うことに特化したソフトウェアレベルのモデルのことを指している。現状、確立された検証用ソフト IP の開発フローが存在しないため、まずはベースとなる開発フローを立案し、開発を行う。なお、検証用ソフト IP 自体は高位合成の対象としては考えない。

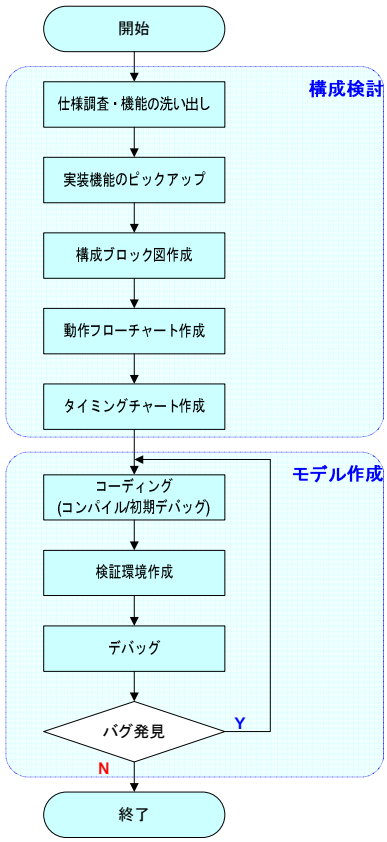
検証用ソフト IP の作成にあたり、まず仕様を調査し、必要な機能の洗い出しを行う。洗い出した機能をもとに検証モデルとして実装すべき機能のピックアップを行う。また検証用ソフト IP は、検証対象となるコアユニットの言語レベル（アルゴリズム、トランザクションレベル、サイクルベースなど）や検証フェーズ（パフォーマンス検証、サイクルベース検証など）に柔軟に対応するため、検証用ソフト IP 自身も「レベルやフェーズにかかわらず変更しないコア部分」と「レベルやフェーズにあわせて変更する周辺部分」に分離しておく必要がある。

以下に、メモリ IP を対象として実装すべき機能とその分離の一例を示す。

- レベルやフェーズにかかわらず変更しないコア部分
  - DRAM アクセスに対するアービトレーションの部分
  - DRAM へのアドレス変換の部分
- レベルやフェーズにあわせて変更する周辺部分
  - トランザクションレベルやサイクルベースで構成される I/F 部分
  - メモリのタイプ（DDR、DDR2、DDR3 など）に応じてタイミング情報が変わる I/O 部分

実装すべき機能のピックアップおよび分離を行った後、構成ブロック図、動作フローチャート図、概略タイミング図を作成し、モデルのコーディングに入る。

上記のことを踏まえて、以下に示すベースとなる開発フローを立案した。



図【2-4】. 1 ベースとなる検証用ソフト IP 開発フロー

上記ベースとなる開発フローに沿ってメモリ IP や周辺 IP の開発を行う。

3-4-2-1-2 検証用ソフト IP の開発フローの立案および手順化

前項「検証に使用する汎用的なメモリ IP および周辺 IP の作成」の工程での実際の開発結果をもとに、ベースとなる開発フローの評価を行った。その結果、開発フローに対して多くの問題点、改善点が見つかった。

フローとしての問題点としては、以下(1)->(2)->(3)の順番を踏まなかったことが挙げられる。

1) アルゴリズムモデル作成

2) UTF(Untimed Function)モデル作成

3) TF(Timed Function)モデル作成

(1)->(2)->(3)の順番を踏まなかったことで多くの問題が様々な工程から発生したため、まずは開発フロー自体の見直しを行う。

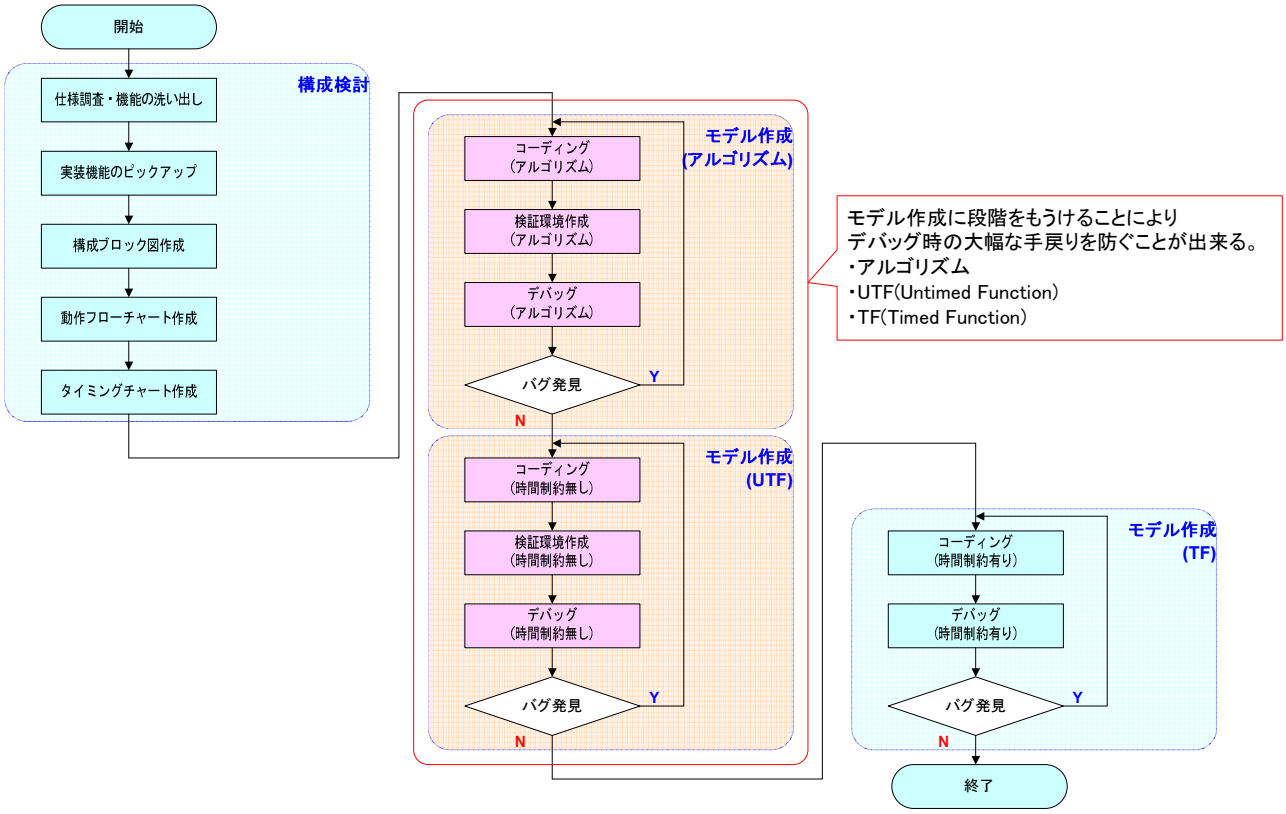
また、工程毎の問題点としては、コーディングとデバッグに多くの時間がかかったことが挙げられる。工程毎の問題点を分析した結果、以下の点を改善することで開発にかかる工数を減らすことができ、生産性向上が見込める。

1) SystemC や TLM2.0 の習得に要する工数を抑制する

2) デバッグ方法の確立による効率化

上記分析結果をもとに改善案を検討する。

改善案を盛り込んだ新たな開発フローは以下のようなフローとなる。



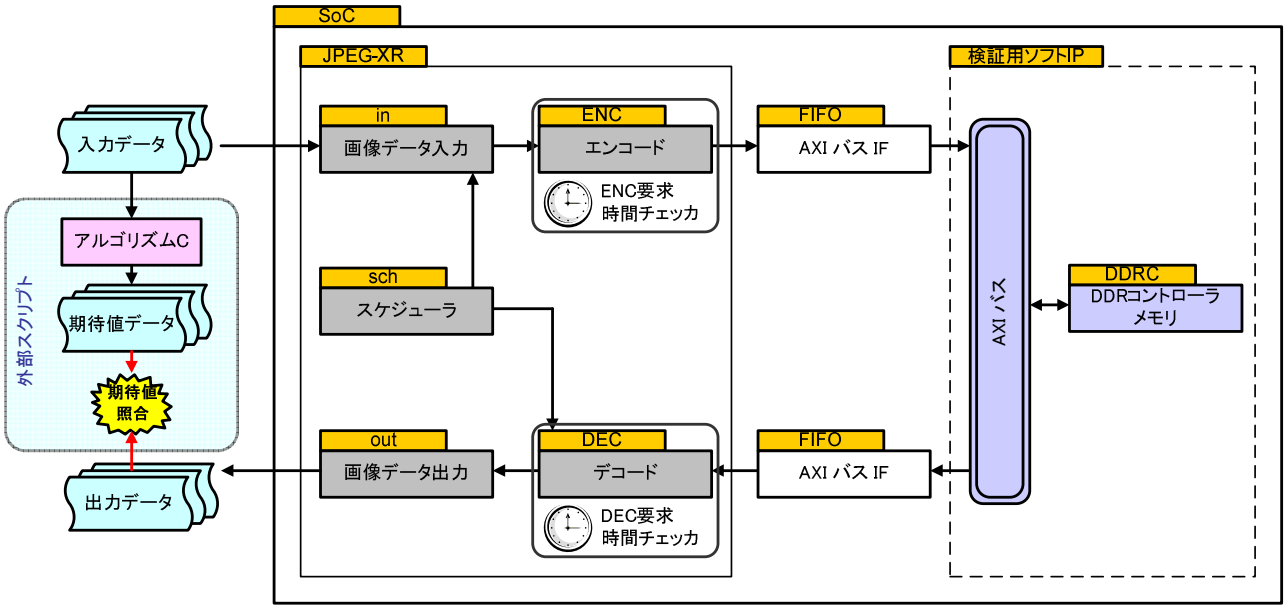
図【2-4】. 2 改善案を盛り込んだ検証用ソフト IP 開発フロー

上図の改善案を盛り込んだ検証用ソフト IP 開発フローに従って手順化する。

3-4-2-1-3 ソフトウェアレベルでの SoC 検証環境構築手法の立案および手順化

本テーマで作成したメモリ IP や周辺 IP などの検証用ソフト IP と、テーマ【2-1】が作成した JPEG-XR の「アルゴリズム C ソース＋TLM I/F」の C ソースを組み合わせて、ソフトウェアレベルでの SoC 検証を実施する。

ソフトウェアレベルでの検証環境の全体構成を以下の図に示す。



図【2-4】. 3 ソフトウェアレベルでの SoC 検証環境構成図

検証用ソフト IP 同様、ソフトウェアレベルでの SoC 検証環境構築に関しても確立された開発フローが存在しないため、まず実際に SoC 検証環境の構築を行い、その上で検証環境構築に当たっての問題点の洗い出しを行う。洗い出した問題点に対して改善点を検討し、改善点を盛り込んだ形で SoC 検証環境構築フローの立案および手順化を行う。

3-4-2-2　生産性評価

3-4-2-2-1 検証に使用する汎用的なメモリ IP および周辺 IP の作成

検証用ソフト IP として、以下の IP を作成する。

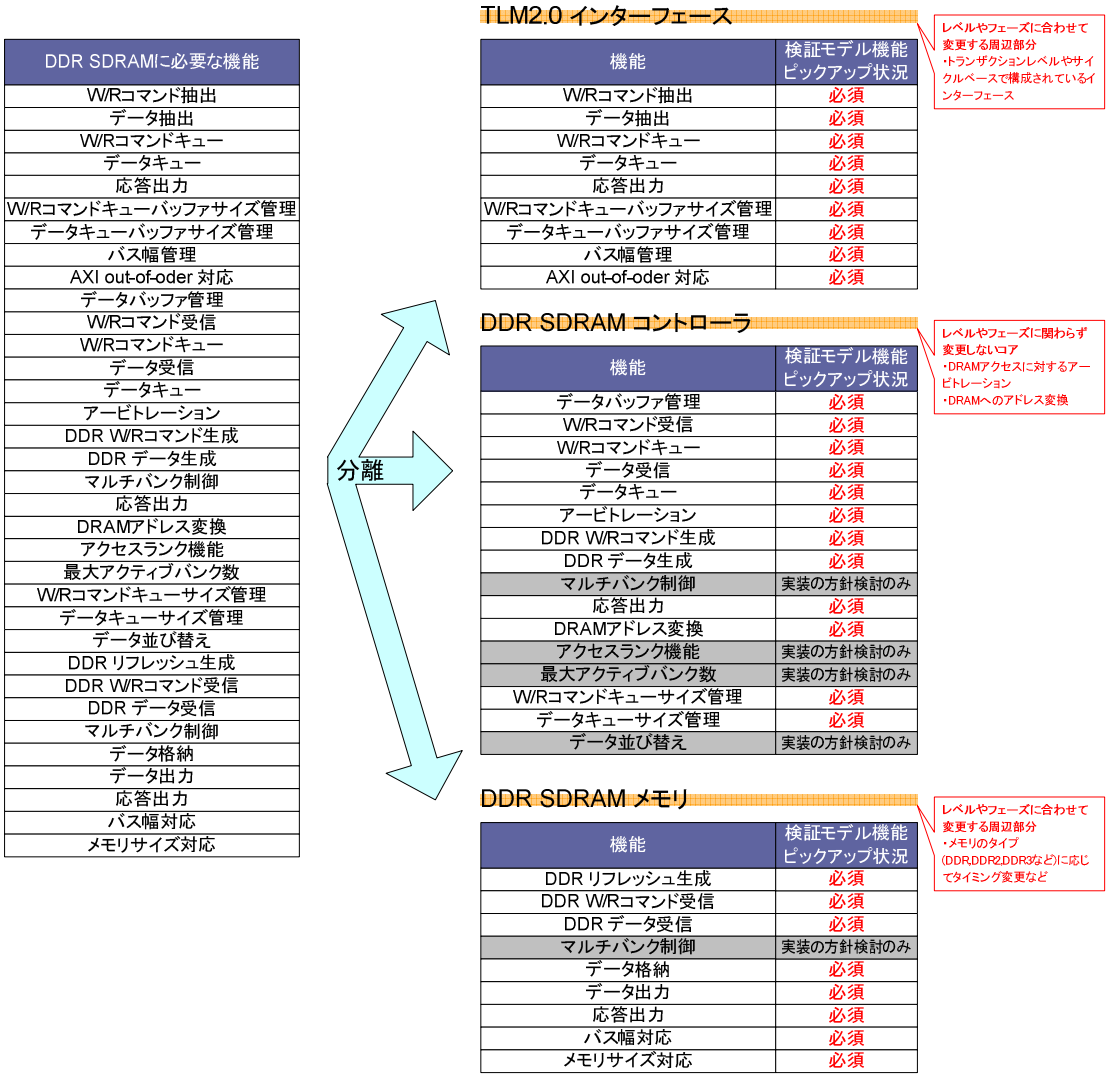
1)　汎用的メモリ IP ： DDR　SDRAM コントローラモデル

2)　周辺 IP ： AXI バスモデル

なお、上記 2 つの IP を作成するにあたった理由は、高位合成の対象となるコアユニットとの単体検証およびシステム検証（SoC 検証）を行うことを前提に必要な IP を検討したためである。

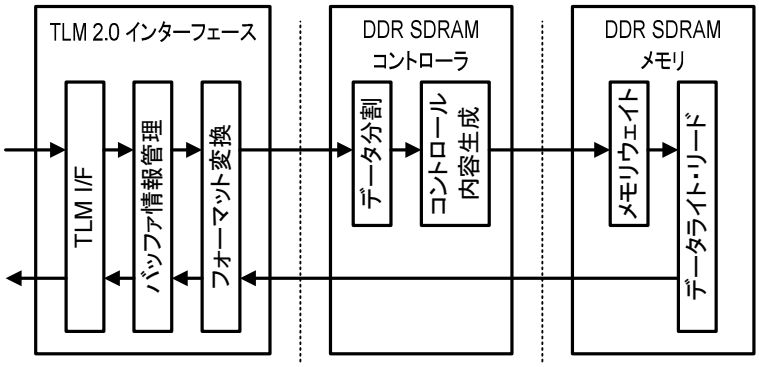
1)　汎用的メモリ IP ： DDR　SDRAM コントローラモデル

DDR　SDRAM コントローラモデルを作成するにあたって、まず仕様調査を行い、機能の洗い出しとピックアップ、そして分離を行った。その過程を以下の図に記す。図の左側が機能の洗い出しで、右側がピックアップと分離を行った結果である。分離を行った結果については、「レベルやフェーズに関わらず変更しないコア部分」と「レベルやフェーズに合わせて変更する周辺部分」に分離するという思想をもとにしている。



図【2-4】. 4 DDR SDRAM コントローラモデル　機能洗い出し～ピックアップ・分離

また、前述の機能ピックアップ・分離後には、構成ブロック図、動作フローチャート図、概略タイミング図を作成しモデルのコーディングを実施した。一例として、構成ブロック図を以下に示す。



図【2-4】. 5 DDR SDRAM コントローラモデル 構成ブロック概要図

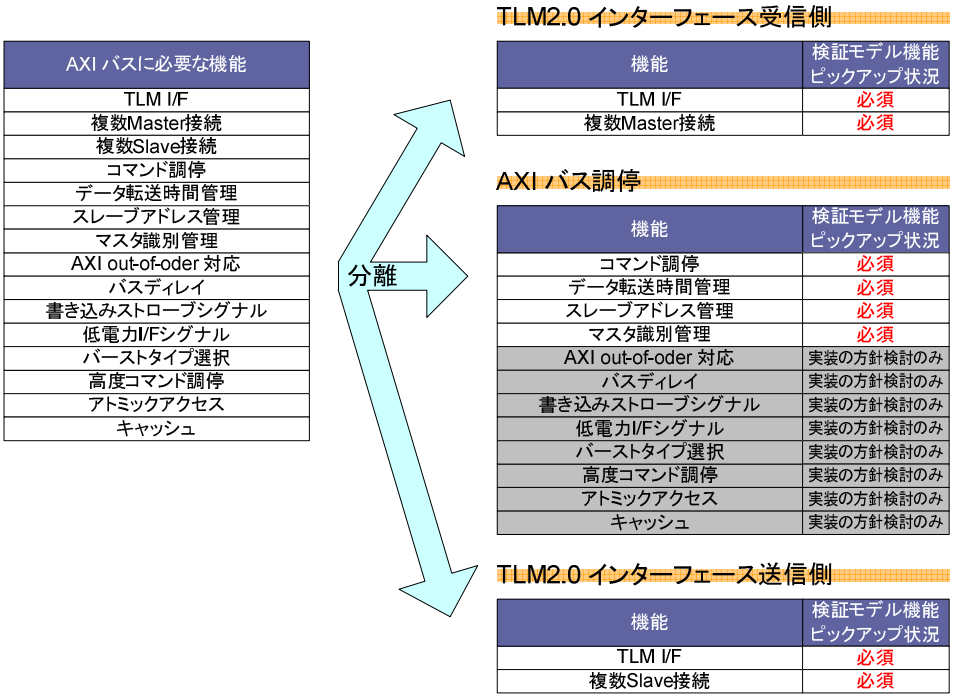
上記構成でモデリングおよびデバッグを行った結果、以下の工数を要した。

表【2-4】. 1 DDR SDRAM コントローラモデル作業工数

項目		作業時間（単位:時間）
仕様		84.0
	仕様調査・機能の洗い出し	16.0
	洗い出した機能から必要な機能のピックアップ	8.0
	構成ブロック図作成(モジュール構成検討)	16.0
	動作フローチャート作成(各モジュール動作詳細検討)	32.0
	タイミングチャート作成(詳細タイミング図)	12.0
設計		48.0
	コーディング(コンパイル/初期デバッグ)	48.0
検証		64.0
	検証環境作成	4.0
	デバッグ	60.0
合計		196.0

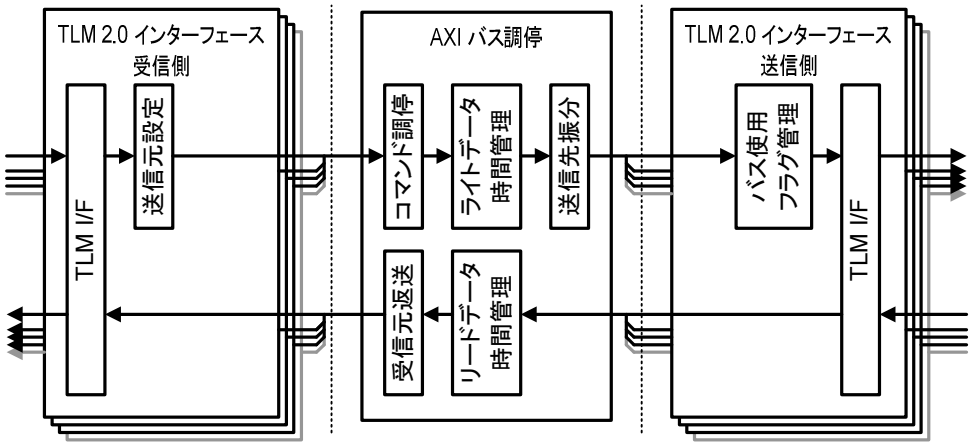
2) 周辺 IP ： AXI バスモデル

AXI バスモデルを作成するにあたって、まず仕様調査を行い、機能の洗い出しとピックアップ、そして分離を行った。その過程を以下の図に示す。図の左側が機能の洗い出しで、右側がピックアップと分離を行った結果である。



図【2-4】. 6 AXI バスモデルの機能洗い出し～ピックアップ・分離

また、上記の機能ピックアップ・分離後には、構成ブロック図、動作フローチャート図、概略タイミング図を作成しモデルのコーディングを実施した。一例として、構成ブロック図を以下に示す。



図【2-4】. 7 AXI バスモデル 構成ブロック概要図

上記構成でモデリングおよびデバッグを行った結果、以下の工数を要した。

表【2-4】. 2 AXI バスモデル作業工数

項目		作業時間(単位:時間)
仕様		24.0
	仕様調査・機能の洗い出し	8.0
	洗い出した機能から必要な機能のピックアップ	4.0
	構成ブロック図作成(モジュール構成検討)	8.0
	動作フローチャート作成(各モジュール動作詳細検討)	4.0
	タイミングチャート作成(詳細タイミング図)	0.0
設計		24.0
	コーディング(コンパイル/初期デバッグ)	24.0
検証		12.25
	検証環境作成	0.25
	デバッグ	12.0
合計		60.0

上記 2 つの検証用ソフト IP を作成した結果、以下の問題点が挙げることができた。

1) SystemC や TLM2.0 に対する基礎知識不足
   - 作業項目のコーディングにおいて、SystemC 特有である Thread 記述で、Thread どうしを接続する際、コンパイルエラーやセグメントエラーが多発し、エラー除去に多大な時間を要した。
2) デバッグ方法の未確立
   - 作業項目のデバッグにおいて、データの入力方法・出力値確認方法などのデバッグ方法が確立されていない為、時間を費やしてしまった。
3) 開発フローの問題点
   - デバッグを行う際、アルゴリズムと UTF(Untimed Function)、TF(Timed Function)を同時に実施してしまった。このため、エラーが発生した際の原因解析が非常に困難であった。

3-4-2-2-2 検証用ソフト IP の開発フローの立案および手順化

前項「検証に使用する汎用的なメモリ IP および周辺 IP の作成」の工程での検証用ソフト IP の作成工数とその問題点を踏まえて、改善提案および開発フローの立案を行い、さらに予測される生産性向上率を測定する。

なお、ここで評価する対象は、DDR SDRAM コントローラモデルとする。

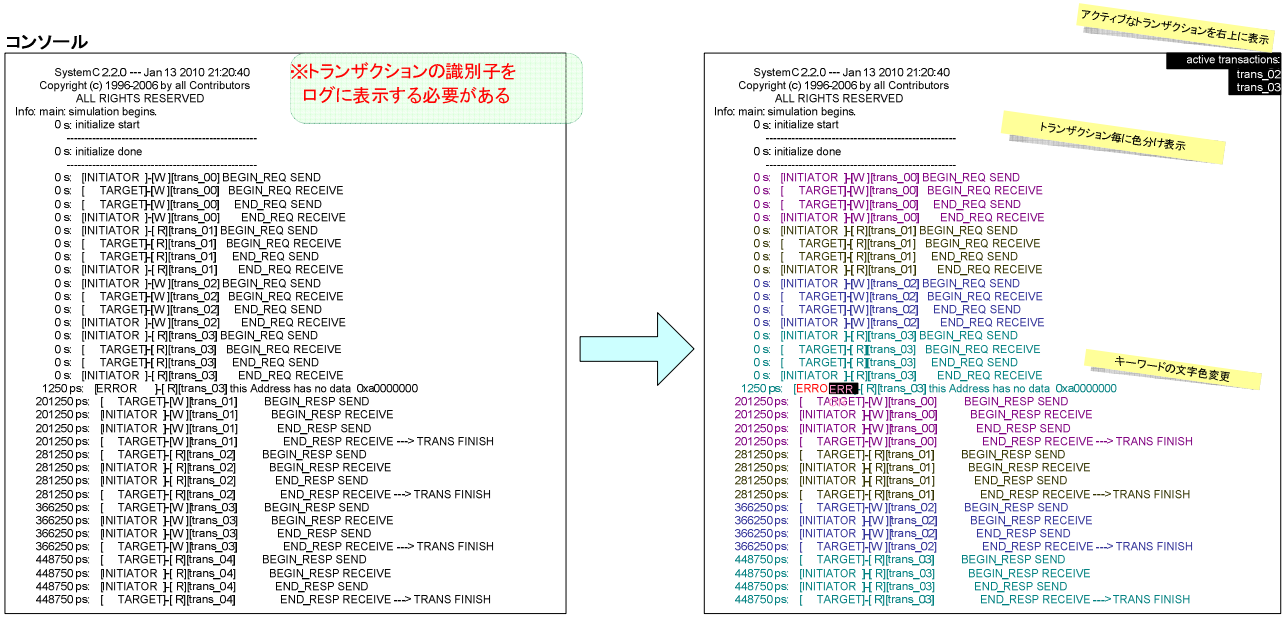
問題点(前項より)

1) SystemC や TLM2.0 に対する基礎知識不足
2) デバッグ方法の未確立
3) 開発フローの問題点

改善点

1) SystemC や TLM2.0 の基本部分をひな形化
   - どのようなソフト IP を作成する場合でも、SystemC や TLM2.0 の基本部分は同様であるため、基本部分をひな形化し、さらに Thread 間の接続等を自動的に行えるようなツールを整備する。これにより基本部分や接続部分のコーディングにかかる工数を削減する。さらにユーザーは作成対象のアルゴリズム部分だけに注力でき、アルゴリズム部分以外の問題・エラー解析工数を大幅に削減する。
   - SystemC のデザインルールチェッカーを用いることで、デバッグ中の予期しないエラーを回避し、エラー解析工数を削減する。

2) デバッグ方法の確立およびデバッグ支援ツールの整備
   - 検証環境に関しても、基本部分のひな形化が可能である。そこで、テーマ【4-1】によって確立された検証環境を検証用ソフト IP 作成における検証環境のベースとする。これにより、検証環境構築の工数を削減する。
   - SystemC を使用する上で、デバッグツールは欠かせないものとなる。そこでデバッグツールの「使用方法」、「便利なコマンド」などを明記した手順書を作成し、デバッグを効率化させる。
   - デバッグ中に出力されるログの可視化を行い、デバッグを効率化させる。方法としてコンソール上に表示されるログを色分けするツールや出力されたログをデバッグ情報ごとに仕分けするスクリプトなどの簡単なツールを整備する。ログの色分けツールやログ仕分けスクリプトのイメージ図を以下に示す。

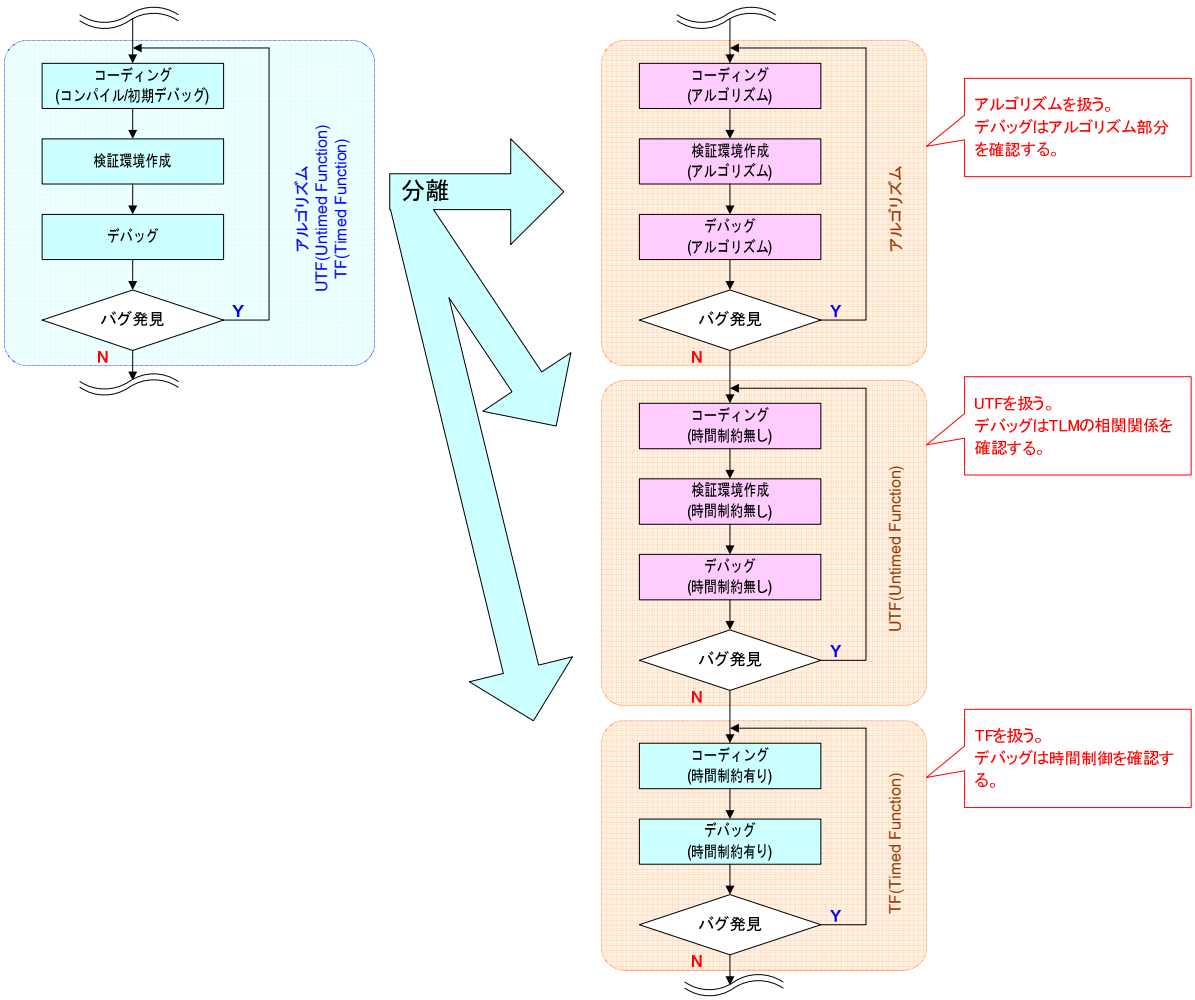


図【2-4】. 8 ログの色分けツールのイメージ図

図【2-4】. 9 ログ仕分けスクリプトのイメージ図

3) 開発フロー改善による効率化

- ベースとなる開発フローでは、アルゴリズム・UTF(Untimed Function)・TF(Timed Function)を明確に分離せずにコーディング・デバッグを行った。そのため、デバッグ時にバグが上記 3 種のどれに該当するのか切り分けから行わなくてはならず、効率的なデバッグが行えなかった。このことを踏まえて開発フローの改善を行う。内容としては、開発フローの「コーディング～検証環境作成～デバッグ」までを、アルゴリズム・UTF(Untimed Function)・TF(Timed Function)の 3 種に分ける。1 種ずつ注力することにより、主にデバッグを効率化する。開発フロー改善に伴うフロー変更を以下図に示す。図の左側がベースフローの「コーディング～検証環境作成～デバッグ」部分で、右側が改善を行った結果である。



図【2-4】. 10 検証用ソフト IP 開発フローの改善

次に、新たな開発フローで検証用ソフト IP を作成した場合の生産性向上率を測定する。

ただし、現時点では実際に新たな開発フローでの検証用ソフト IP の作成まで行えていないため、コーディングとデバッグの視点から改善効果を考察し、生産性向上率の予測値を算出する。

コーディング(コンパイル/初期デバッグ)

- 「SystemC や TLM2.0 の基本部分をひな形化」で挙げた自動化ツールの使用により、SystemC の接続部分の記述を削減でき、さらに接続ミスによるエラー解析工数を削減出来る。また、SystemC のデザインルールチェッカーを用いることで、初期デバッグの精度が向上する。これらにより、コーディング(コンパイル/初期デバッグ)フローは約 27%の工数が削減され、1.3 倍の生産性向上が見込まれる。

デバッグ

- 「デバッグ方法の確立およびデバッグ支援ツールの整備」で挙げた検証環境のひな形化により、検証環境構築の時間を削減出来る。また、デバッグツールの手順書とログの可視化により、デバッグ中のエラー解析工数を削減出来る。さらに「開発フロー改善による効率化」の開発フロー変更により、アルゴリズム・UTF(Untimed Function)・TF(Timed Function)の、それぞれの視点よりデバッグを行える為、開発の後戻りを削減出来る。これらにより、デバッグフローは約 55%の工数が削減され、2.2 倍の生産性向上が見込まれる。

以下に生産性向上率を示した表を示す。

表【2-4】. 3 検証用ソフト IP 作成 生産性向上率

項目		作業時間(単位:時間)		生産性向上率
		実測値	改善後の予測値	
仕様		84.0	82.0	----
	仕様調査・機能の洗い出し	16.0	16.0	----
	洗い出した機能から必要な機能のピックアップ	8.0	8.0	----
	構成ブロック図作成(モジュール構成検討)	16.0	16.0	----
	動作フローチャート作成(各モジュール動作詳細検討)	32.0	30.0	----
	タイミングチャート作成(詳細タイミング図)	12.0	12.0	----
設計		48.0	35.0	27% (1.3 倍)
	コーディング(コンパイル/初期デバッグ)	48.0	35.0	27% (1.3 倍)
検証		64.0	34.0	47% (1.8 倍)
	検証環境作成(アルゴリズム)	----	3.0	----
	検証環境作成(SystemC)	4.0	4.0	----
	デバッグ	60.0	27.0	55% (2.2 倍)
合計		196.0	151.0	23% (1.3 倍)

3-4-2-2-3 ソフトウェアレベルでの SoC 検証環境構築手法の立案および手順化

ソフトウェアレベルでの SoC 検証環境を構築し、構築した上で挙げられた問題点・改善点を踏まえて SoC 検証環境構築フローの立案を行い、さらに予測される生産性向上率を測定する。

ここで作成する SoC 検証環境は、「テーマ【2-1】が作成した JPEG-XR「アルゴリズム C ソース＋TLM I/F」の C ソースと、本テーマで作成した検証用ソフト IP(DDR SDRAM コントローラモデル、AXI バスモデル)を組み合わせ、システム的な検証を行う」ための検証環境である。

以下の流れで SoC 検証環境を構築する。

1) テーマ【2-1】が作成した JPEG-XR の単体検証環境に、本テーマで作成した検証用ソフト IP(DDR SDRAM コントローラモデル、AXI バスモデル)を組み込む。検証シナリオは単体検証のものをそのまま流用する。
2) テーマ【4-1】が作成した検証環境のひな形をベースに、テーマ【2-1】が作成した JPEG-XR と本テーマで作成した検証用ソフト IP を組み込む。検証シナリオは単体検証のものをそのまま流用する。

現在 SoC 検証環境を構築中であるが、現時点までに挙げられた以下 2 点の問題点をもとに、改善点を検討する。また、SoC 検証環境構築に関する開発フローの立案を行う。

問題点

1) 検証環境構成の移行問題

- 今回テーマ【2-1】が使用した検証環境は、テーマ【4-1】が確立した検証環境に適合していない為、テーマ【4-1】の検証環境に移行する際に大幅な時間を要することが予測される。(実行スクリプト修正、シナリオ構成修正、接続修正、環境修正等)

2) デバッグ方法の未確立

- SoC 検証の場合、大規模になればなるだけ扱われるデータ量が多くなる。その結果、出力されるログの量も大量になり、バグの発見を遅らせる可能性がある。
- デバッグを行う際、パフォーマンスが規定速度まで達しない場合がある。このとき、使用率が高いモジュールの割り出しや、機能の修正が必要であり、時間増が予測される。

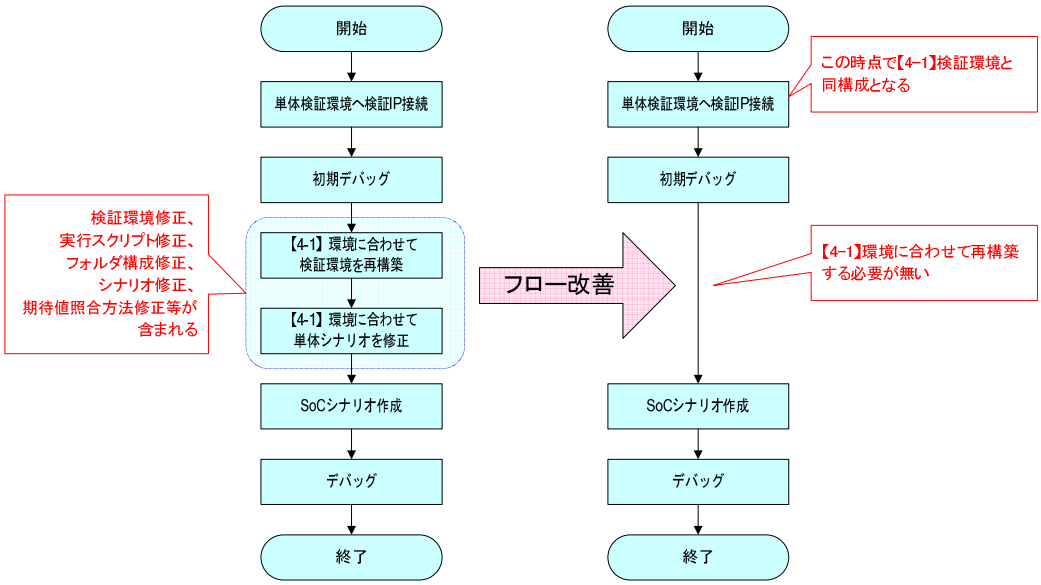
改善点

1) 検証環境の統一

- SoC 環境と、単体検証環境(今回ではテーマ【2-1】の JPEG-XR 単体環境)の構成が異なる場合、必然的に環境間の移行工数が発生する。そこで、検証環境を統一することにより、検証環境の移行工数(実行スクリプト修正、シナリオ構成修正、接続修正、環境修正等)を削減する。なお、統一された検証環境には、テーマ【4-1】により確立された検証環境を使用する。

2) デバッグ方法の確立およびデバッグ支援ツールの整備

- SoC 検証では扱われるデータ量が多くなるため、出力されるログの量も大量になりデバッグ効率の低下が予想される。そこで、あらかじめ検証環境にデバッグ状況にあわせて出力されるログの精度の切り替える機能を組み込む。これにより、デバッグ状況に応じたログ内容で効率的なデバッグが行える。
- SoC 検証でも、前項「検証用ソフト IP の開発フローの立案および手順化」で提案したログの色分けツールやログ仕分けスクリプトなどのデバッグ支援ツールの整備を行う。



図【2-4】. 11 SoC 検証環境開発フローの改善

次に、新たな開発フローで SoC 検証環境を作成した場合の生産性向上率を測定する。

ただし、現時点では実際に新たな開発フローでの SoC 検証環境の作成まで行えていないため、検証環境構築とデバッグの視点から改善効果を考察し、生産性向上率の予測値を算出する。

検証環境構築

- 「検証環境の統一」により、検証環境の移行が削減され、さらに、単体検証で作成した検証シナリオがそのまま SoC 環境にも流用できる。これにより、検証環境構築は約 63%の工数が削減され、2.6 倍の生産性向上が見込まれる。

デバッグ

- 「デバッグ方法の確立およびデバッグ支援ツールの整備」のデバッグ支援ツールにより、パフォーマンス未達成の原因であるモジュールを早期に発見できる。また、検証規模の変動によるログ出力精度の変更機能により、デバッグ効率の低下を抑制する。これらにより、デバッグは約 50%の工数が削減され、2 倍の生産性向上が見込まれる。

以下に生産性向上率を示した表を示す。

表【2-4】. 4 SoC 検証環境構築 生産性向上率

項目		作業時間(単位:時間)		生産性向上率
		実測値	改善後の予測値	
検証環境構築		42.5	16.0	63% (2.6 倍)
	単体検証環境へ検証 IP 接続	0.5	0.5	----
	初期デバッグ	28.0	11.5	59% (2.4 倍)
	【4-1】環境に合わせて検証環境を再構築	8.0	----	----
	【4-1】環境に合わせて単体シナリオを修正	2.0	----	----
	SoC シナリオ作成	4.0	4.0	----
デバッグ		8.0	4.0	50% (2.0 倍)
	デバッグ	8.0	4.0	50% (2.0 倍)
合計		50.5	20.0	61% (2.5 倍)

3-4-2-3　作成物

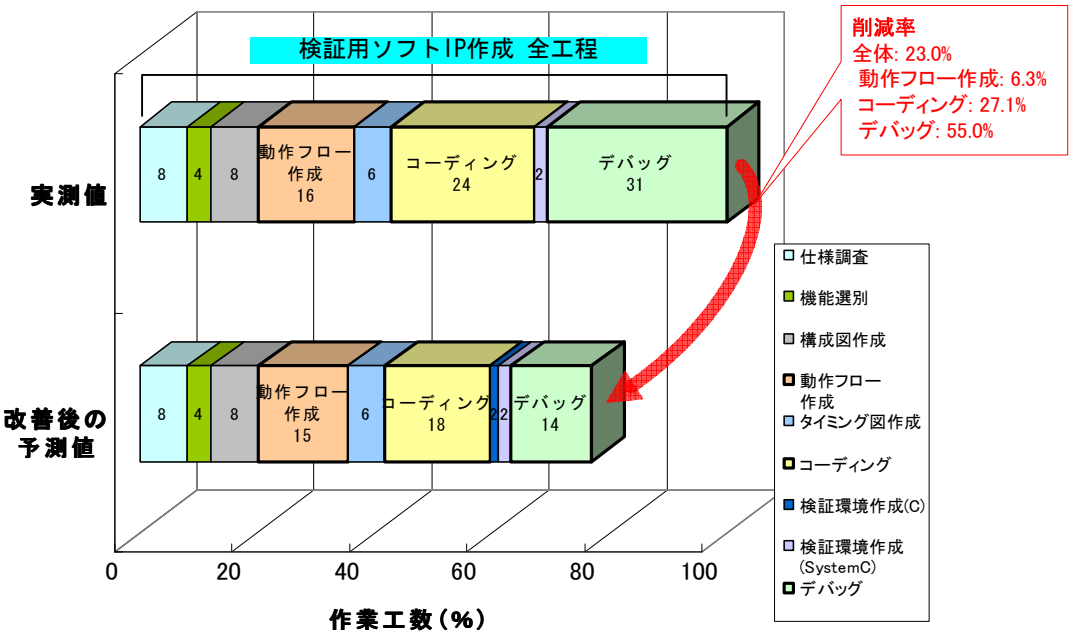
本テーマでは、以下に挙げる手順書および検証用ソフト IP を開発した。

- 検証に使用する汎用的なメモリ IP および周辺 IP
- 検証用ソフト IP 作成手順書
- ソフトウェアレベルでの SoC 検証環境
- ソフトウェアレベルでの SoC 検証環境作成手順書

3-4-3　まとめ

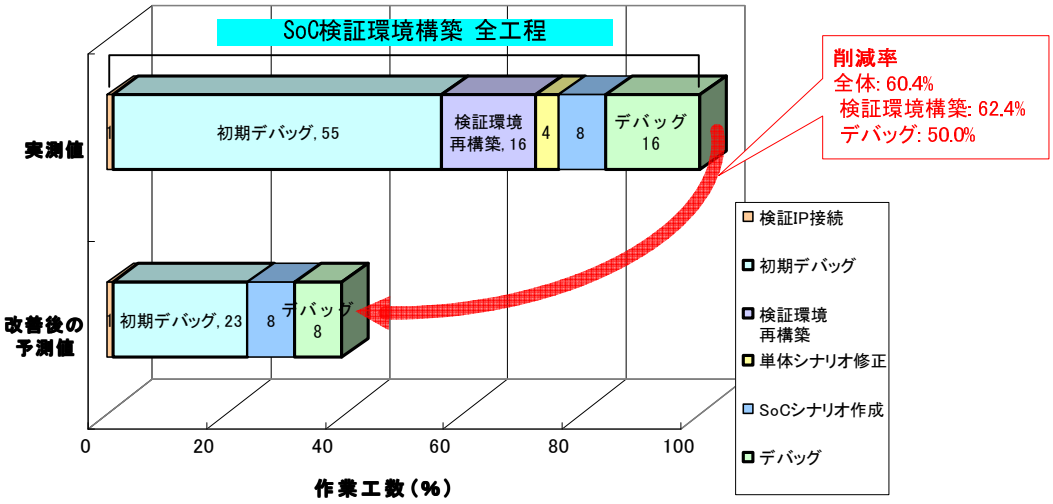
3-4-3-1　成果

本テーマでは、検証用ソフト IP の作成手順の検討を行った。検討した手順に準拠して検証用ソフト IP の作成を行うことで、手順に準拠する前は工数 196h であったものが、手順に準拠することで工数 151h となることが予測される。その結果、工数が約 25%短縮され、生産性は約 1.3 倍の向上が見込まれる。



図【2-4】. 12 検証用ソフト IP 作成 生産性向上率

また、ソフトウェアレベルでの SoC 検証環境の作成手順も検討を行った。検討した手順に準拠して SoC 検証環境の作成を行うことで、手順に準拠する前は工数 50h であったものが、手順に準拠することで工数 20h となることが予測される。その結果、工数が約 60%短縮され、生産性は約 2.5 倍の向上が見込まれる。



図【2-4】. 13 SoC 検証環境構築生産性向上率

3-4-3-2　結論

本テーマで検討した検証用ソフトIPの作成手順とソフトウェアレベルでのSoC検証環境の作成手順を利用することで、一定の生産性向上が見込まれる。

ただし、得られた生産性向上率はあくまでも予測値であるため、実際に手順に沿って開発を行って生産性向上の手法として問題がないかを検証し、手順としての精度を高めることが急務である。また、以下の課題を解決していくことも必要である。

1) ひな形の活用

ソフトIPの作成および検証において生産性を向上させるためには、テーマ【4-1】が検討しているひな形を活用する必要がある。特に検証に関しては、検証の形態にかかわらず、同じひな形をベースに検証環境を作成することが必須である。（例えば単体検証からSoC検証への移行や拡張の際、ベースとなっている環境が異なると整合をとるための工数が余分に発生する。単体検証もSoC検証も同じひな形をベースすることで、調整などの余分な工数が削減され、さらに検証シナリオの流用がスムーズに行えるため、大幅な生産性向上が見込まれる。）

2) デバッグの効率化

ソフトIPのデバッグでは、ログの解析が中心となってくる。どういったログを出力することでデバッグを効率的に行えるかは今後検討していく必要がある。また、波形ビューアのような大がかりなツールではなく、例えばコンソール上に表示されるログを色分けするツールや出力されたログをデバッグ情報ごとに仕分けするスクリプトなどの簡単なデバッグ補助ツールで可視化するだけでも、検証効率を向上させることができると思われる。出力されるログの内容の検討とあわせて、ログの可視化のための補助ツールの整備を行う必要がある。

3) 検証ソフトIPの整備

現状検証用ソフトIPが必要な場合、流用可能なソフトIPがほとんどないため新規で作成する必要がある。標準規格に準じたバスなどの汎用的に利用可能な検証用ソフトIPや手順書をあらかじめ数多く準備することにより、検証用ソフトIPを作成すること自体の削減になり、開発全体での大幅な生産性向上が見込まれる。

第4章 本論・高位合成ツール向けライブラリや IP の開発

4-1 【3-1】ソフト用、ハード用の個別ライブラリや IP の開発

4-1-1 概要・目的

現存するメモリや画像圧縮等のライブラリや IP は、ソフトウェアレベルまたはハードウェアレベルのどちらか一方でのみ動作するものが主流である。その傾向と実体を分析し、ソフトとハードの両レベルでデータ化が可能なライブラリおよび IP について抽出する。

その中で画像圧縮、メモリおよびメモリコントローラの IP 販売実態および市場動向情報を分析し、メモリサイズ・ビット幅・スピード等の詳細技術仕様として提案、フォーマットの違いに対応した共通ライブラリや高位合成ツール別の個別 IP を開発する。

具体的には、複数の高位合成ツールに着目、画像圧縮用メモリやメモリコントローラ等の画像圧縮に特化した複数種類のメモリ群をライブラリとして、個別のメモリや画像圧縮用の特殊なメモリコントローラの IP を開発する。

本テーマではハード IP の開発に着目し、まず従来の HDL 作成手順を生産性向上の観点から見直しを行い、新たなハード IP 作成手順を提案する。さらに、ハード IP の検証に関する生産性向上の手段として SystemVerilog による検証を導入し、そのリファレンスマニュアルの作成を行う。

なお、IP 販売実体および市場動向情報の調査/分析はテーマ【3-2】にて行う。

本テーマでは、ハードウェア用ライブラリや IP を総称してハード IP と定義する。

4-1-2　内容

4-1-2-1　内容説明

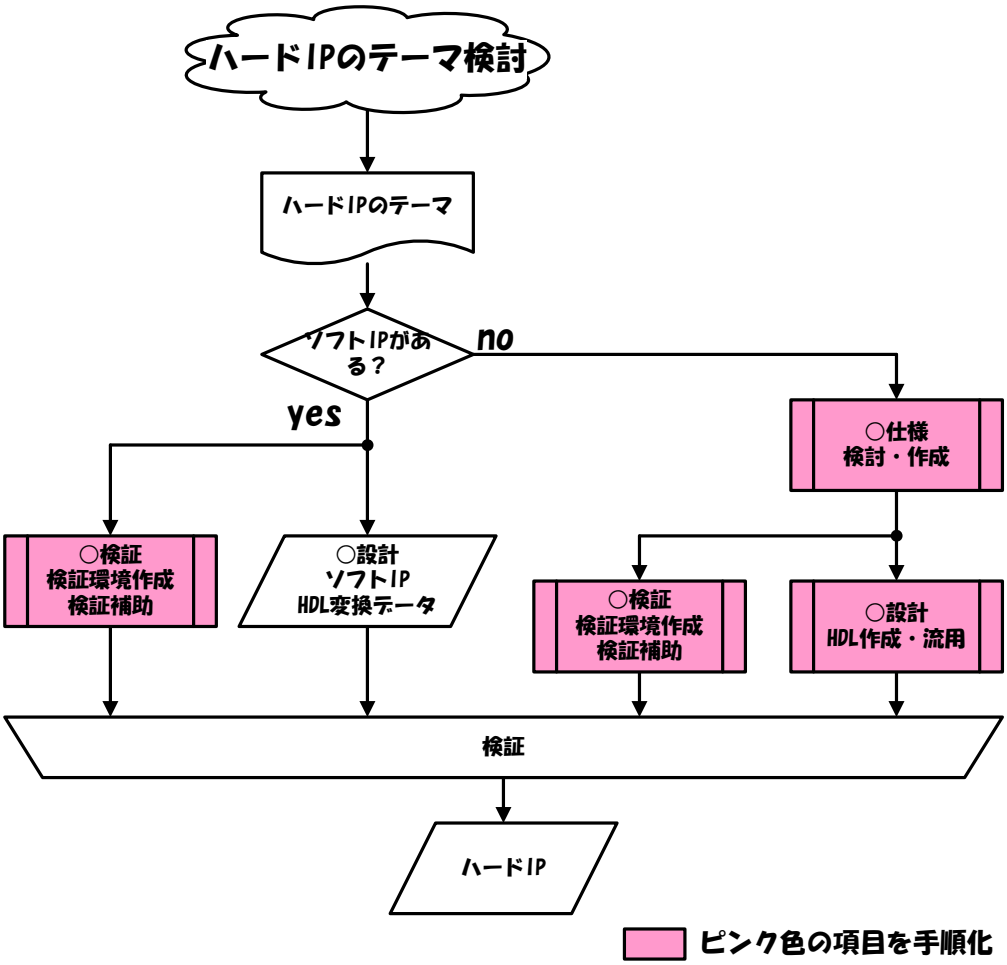
4-1-2-1-1 ハード IP 作成手順の提案

ハード IP 作成手順の提案にあたり従来の HDL 作成手順に対して、どういった方法を用いることで生産性が向上できるか検討した。生産性を向上させる方法として以下のポイントが挙げられる。

- 設計資産の流用（期待値ツールや汎用マクロなどの過去資産）
- 自動化（ひな形生成など）
- ノウハウの利用（開発データベースの利用）

上記のポイントを考慮し、開発フローの「仕様」、「設計」、「検証」の各項目で生産性向上が見込める手段を検討した。

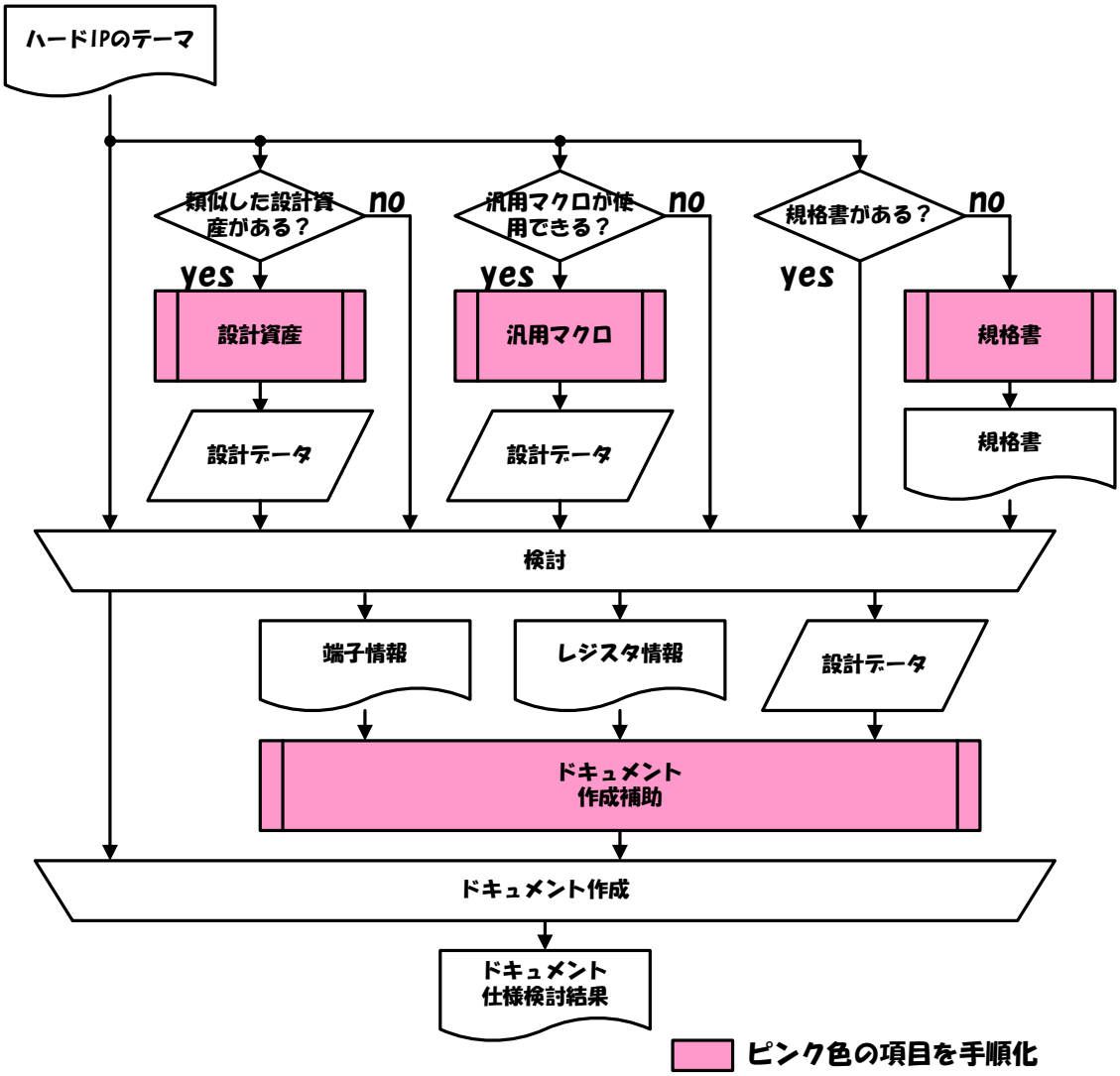
まず、新たに提案するハード IP 作成　全体フローを以下に示す。



図【3-1】. 1 ハード IP 作成　全体フロー

次に、ハード IP 作成 仕様フローを以下に示す。

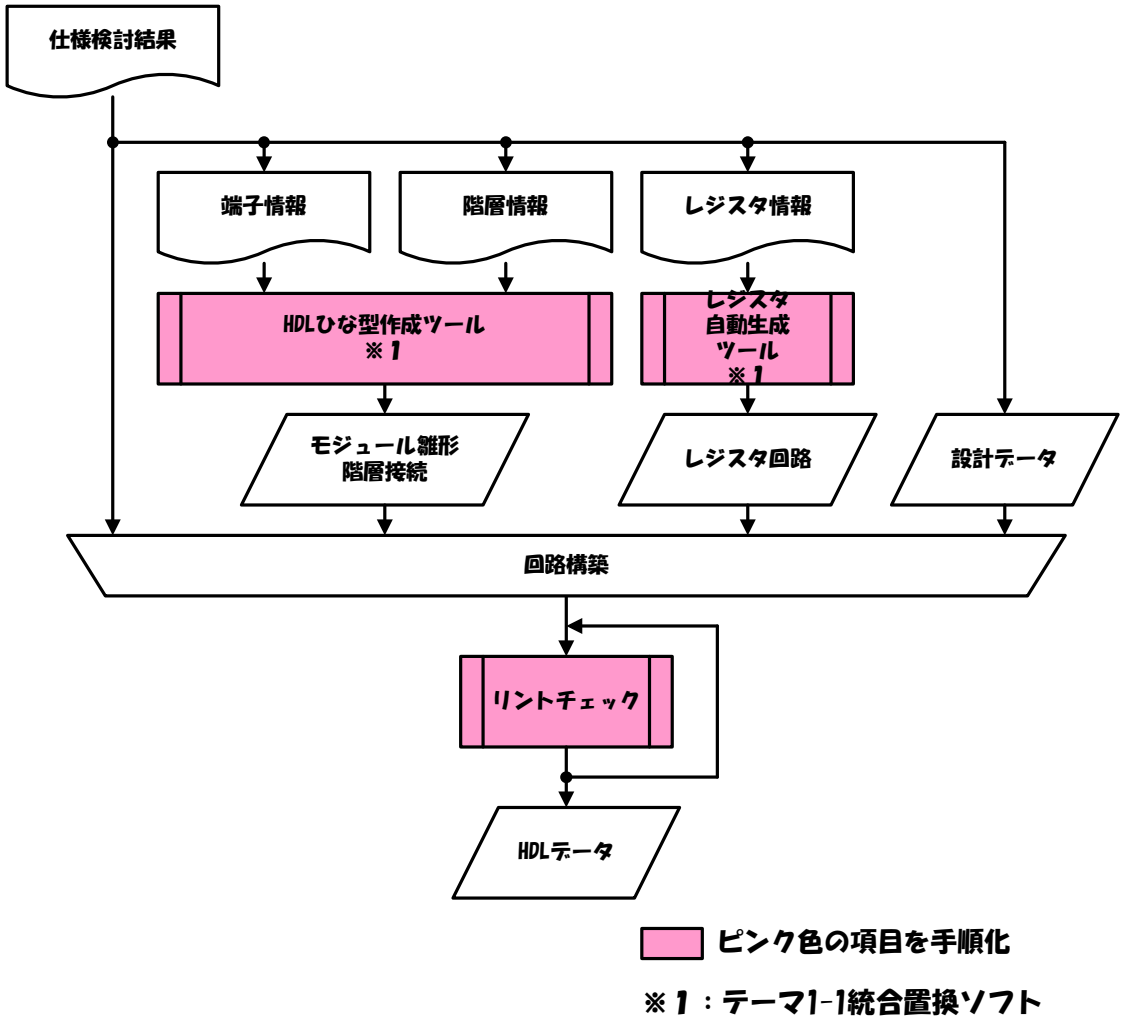
「仕様」では、設計資産の流用の観点から“設計資産”、“汎用マクロ”、“規格書”として、流用できる資産や情報の収集と整備行いデータベース化し、これらを流用することでの生産性向上を見込んだ。また、自動化の観点から“ドキュメント作成補助”として、仕様書作成での生産性向上を行うためあらゆる技法を取り入れた。



図【3-1】. 2 ハード IP 作成 仕様フロー

次に、ハード IP 作成 設計フローを以下に示す。

「設計」では、自動化の観点から“HDL ひな形作成ツール”を整備することにより、HDL のひな形だけでなくモジュール接続を自動で行う機能を持たせて作業時間を圧縮できるようにした。ノウハウの利用の観点から“リントチェック”ではログの成型ツールを整備し、視認性を良くすることでログ確認を容易なものとした。また、自動化の観点から“レジスタ自動生成ツール”を整備することにより、設計毎に作成するレジスタ回路を自動作成し、設計全体の生産性向上を見込んだ。

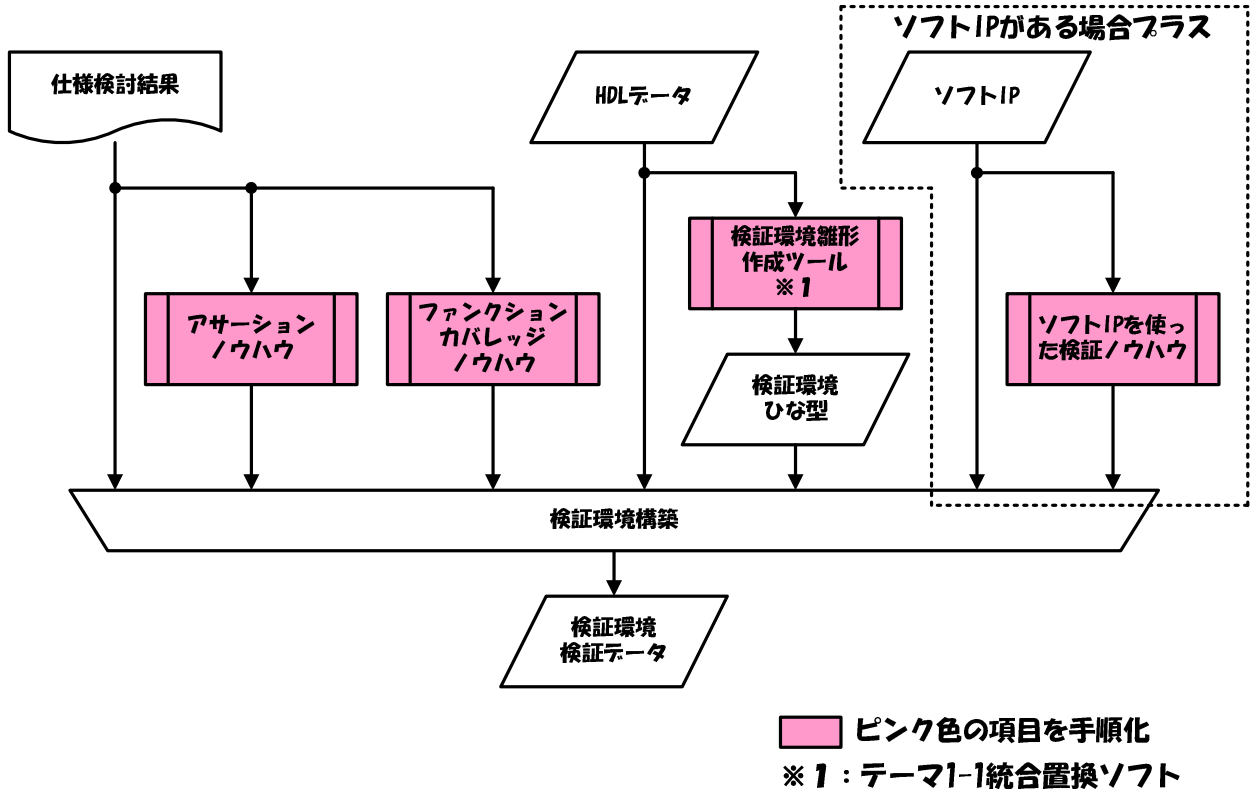


図【3-1】. 3 ハード IP 作成 設計フロー

最後に、ハード IP 作成 検証フローを以下に示す。

「検証」では、自動化の観点から“検証環境ひな形作成ツール”を整備することにより、検証環境作成を容易なものとした。また、ソフト IP からハード IP を作成する場合を想定し、ノウハウの利用の観点から“ソフト IP を使った検証ノウハウ”としてまとめた。

さらに「検証」では、Verilog HDL だけでなく SystemVerilog を利用することで、記述量や検証実行時間を削減し生産性向上を見込んだ。また、ノウハウの利用の観点から SystemVerilog による“アサーションノウハウ”や“ファンクションカバレッジノウハウ”を整理し、検証精度の向上による検証工数の削減を見込んだ。



図【3-1】. 4 ハード IP 作成 検証フロー

ここでは、上記で提案したハード IP 作成の「仕様」、「設計」、「検証」フローが生産性向上の手法に問題ないことを検証するとともに、上記フローに沿ったハード IP 作成手順を作成する。

4-1-2-1-2 ハード IP 開発の検証に対する SystemVerilog の導入

新たなハード IP 作成手順の提案にて開発フロー全体の見直しを行ったが、ここでは「ハード IP の開発および従来の HDL 開発における検証」の部分に着目した。

現在、従来の HDL 開発における検証にて主に利用されているのは Verilog HDL 言語であるが、その拡張言語であり尚かつ検証機能が強化された言語である SystemVerilog が新たに標準規格として認定され、この SystemVerilog を利用することで検証の高性能化と生産性向上が可能であると言われている。

Verilog HDL と SystemVerilog の違いについて以下に簡単に説明する。

- SystemVerilog は Verilog HDL の拡張言語
  - Verilog HDL はそのまま利用できる（Verilog HDL 資産の再利用）
- ランダム生成やテストベンチ専用記述が強化
  - 制約付きや重み付きのランダム宣言が可能
  - アサーション(SVA)やファンクションカバレッジに適した記述が追加
- SystemVerilog はオブジェクト指向の言語
  - 記述の抽象度を上げる記法の導入（シミュレーションの高速化）
  - C 言語で記述された機能モデルとの接続が容易（DPI 接続の用意）

そこで、SystemVerilog を導入することが「ハード IP の開発および従来の HDL 開発における検証」の部分の生産性向上の手法に問題ないことを検証するとともに、SystemVerilog を利用した検証に関するリファレンスマニュアルを作成する。

なお、SystemVerilog の評価は、Verilog HDL と SystemVerilog との比較で行い、以下のポイントに着目し評価する。

1) Verilog HDL と比較した際の SystemVerilog の優位性の検証（記述量、検証実行時間）
2) C 言語との親和性の検証（C 言語との接続の容易さ、記述量、検証実行時間）
   - 「Verilog HDL +ファイル渡し」と「Verilog HDL +PLI」と「SystemVerilog+DPI」の比較

4-1-2-2　生産性向上の評価

4-1-2-2-1 ハード IP 作成手順の提案

従来の HDL 作成手順を生産性向上の観点から見直しを行い、新たなハード IP 作成手順を提案した。このハード IP 作成手順を使用して作業した場合の生産性向上率を測定し、ハード IP 作成手順の評価を行う。

ハード IP 作成手順評価のポイント

- 「手順書あり」と「手順書なし」で同じ仕様の回路を作成し測定
- 設計対象者は設計経歴 4 年

ハード IP 作成手順の生産性向上率の測定方法を以下に説明する。

- 比較対象者
  - 手順あり:設計経歴 4 年
  - 手順なし:設計経歴 4 年
- 作成回路
  - 汎用バス スレーブ I/F を有するレジスタ回路
- 作業時間の測定
  - 設計仕様書作成～カバレッジ測定までの作業時間を測定
  - 作業に要した時間のみを測定し、仕様理解や環境理解は作業時間に含めない

汎用バス スレーブ I/F レジスタ作成の作業時間について以下に示す。

表【3-1】1 作業時間比較

作業項目		作業時間(h)		生産性向上率
		手順書なし	手順書あり	
仕様(設計仕様書)		12.75(h)	9.5(h)	25%(1.4 倍)
	階層表/端子表	1(h)	0.5(h)	50%(2 倍)
	ブロック図	4.5(h)	3.0(h)	33%(1.5 倍)
	説明	6.75(h)	6.0(h)	12%(1.1 倍)
設計		6.5(h)	5.5(h)	16%(1.2 倍)
	コーディング	5.75(h)	5.0(h)	14%(1.2 倍)
	リントチェック	0.75(h)	0.5(h)	33%(1.5 倍)
検証		22.0(h)	7.5(h)	66%(3 倍)
	環境作成	15.5(h)	3.0(h)	80%(5.2 倍)
	検証実施	6.5(h)	4.0(h)	38%(1.6 倍)
合計		41.25(h)	22.5(h)	45%(1.8 倍)

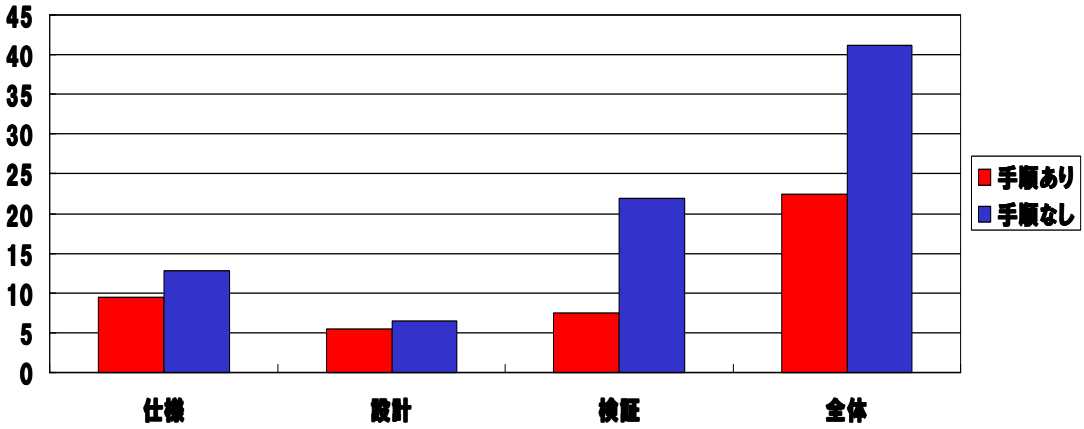
ハード IP 作成手順評価について以下のように考察する。

- 仕様(設計仕様書)
  - 手順化(ドキュメント作成補助)により、生産性が約 1.4 倍向上した。
- 設計
  - 手順化(ひな形生成、汎用マクロ流用、リントチェックログ成型)により、生産性が約 1.2 倍向上した。
- 検証
  - 手順化(ひな形生成、アサーション、ファンクションカバレッジ、SystemVerilog)により、生産性が 3 倍向上した。

- まとめ
  - ハード IP 作成を手順化することにより今回の評価から全体で 1.8 倍の生産性向上が測定された。
  - 一番生産性向上率が高かったのが検証の環境作成(5.2 倍)だった。これはアサーション等のサンプルを活用することにより生産性が向上した。
    ただし、検証で利用できるモデルやチェッカーのサンプル数が少ないため、標準規格に準じたモデルやチェッカーを数多く準備することで高い生産性が維持できると思われる。
  - 一番生産性向上率が低かったのが仕様(設計仕様書)の説明(1.1 倍)だった。これは、作りこみの部分が多かったため生産性の向上があまり無かった。
    上記から、作りこみの部分を減らすことで生産性が向上する。今後の課題は、作りこみの部分を減らすために汎用的に使用できるマクロなどを充実させる必要がある。

ハード IP 作成手順の評価の結果、以下の生産性向上が見込まれる。

- 「ハード IP の検証」または「従来の HDL 設計の検証」に関しては、ハード IP 作成手順を利用することにより下表のように全体で約 56%作業時間を圧縮でき、約 1.8 倍の生産性向上が見込まれる。



図【3-1】. 5 作業時間の比較

4-1-2-2-2 ハード IP 開発の検証に対する SystemVerilog の導入

1) Verilog HDL と比較した際の SystemVerilog の優位性

Verilog HDL と比較した際の SystemVerilog の優位性については以下の方法で検証する。

「実際にプロジェクトで使用された Verilog HDL で作成された汎用バスの検証モデル(アサーション&ファンクションカバレッジ)を SystemVerilog に移植し、記述量および検証実行時間を比較する。」

- 記述量比較
  - 以下 2 つの検証モデルを Verilog HDL から SystemVerilog へ移植

表【3-1】2 記述量比較

検証モデル	Verilog HDL	SystemVerilog	削減量
アサーション	2225 行	1301 行	約 42%削減
ファンクションカバレッジ	5866 行	3518 行	約 40%削減

- 検証実行時間比較
  - 検証モデルのみ変更しテストシナリオ 146 本を連続実行(マシンなどの環境は同じ)

表【3-1】3 検証実行時間比較

Verilog HDL	SystemVerilog	削減量
約 5 時間 10 分	約 4 時間 30 分	約 15%削減

  - 一部のみの移植で検証実行時間が短縮できたので、検証環境全体を SystemVerilog で構築すればさらなる時間短縮が見込まれる(約 30%の短縮が予想される)

2) C 言語との親和性

C 言語と HDL 検証環境の接続方法には以下の 3 種類が挙げられる。
1) ファイル渡しでの C 言語との接続(HDL:Verilog HDL)
2) PLI を使用した C 言語との接続(HDL:Verilog HDL)
3) DPI を使用した C 言語との接続(HDL:SystemVerilog)

C 言語との親和性については以下の方法で検証する。
「C モデルとのインターフェースに関して上記 3 種類の方法で環境を構築し、接続の容易性および検証実行時間を比較する。」

「C 言語との接続の容易性」について
接続の種類と方法について以下に示す。

表【3-1】4 C 言語との接続の容易性

接続種類	接続方法
ファイル渡し	C 側/Verilog HDL 側双方にあらかじめファイル渡しの事前準備が必要
	ファイルとのダンプ/ロードに時間がかかる(検証時間に影響する)
	ファイルのサイズによっては実行マシンのディスクやメモリを圧迫する
PLI+Verilog HDL	C 側/Verilog HDL 側双方にコードの修正が必要 とくに C 側へ PLI 用の記述を加える必要があり、また接続方法も煩雑
	Verilog HDL 側から C 側のタスク、ファンクションを利用するのみ
	参考資料が少ない(Web 上にも)
DPI+SystemVerilog	C 側は修正の必要がない SystemVerilog 側の修正も数行程度(タスク、ファンクションの宣言のみ)
	C 側から SystemVerilog 側のタスク、ファンクションも参照可能(その逆も可)
	参考資料が多い(Web 上にも) C 言語と HDL の接続に関しては DPI が主流になりつつある

接続の容易性の観点では、DPI を利用することが、最も接続が容易であると思われる。

「検証実行時間」について
C モデル :配列を使用した演算モデル(入力データ＋期待値を生成)
HDL :入力データを元に演算

データ数 1000000 個作成時の検証実行時間

表【3-1】5 C 言語との接続による検証実行時間

接続種類	検証実行時間
ファイル渡し	3.020seconds
PLI 接続	2.570seconds
DPI 接続	2.310seconds

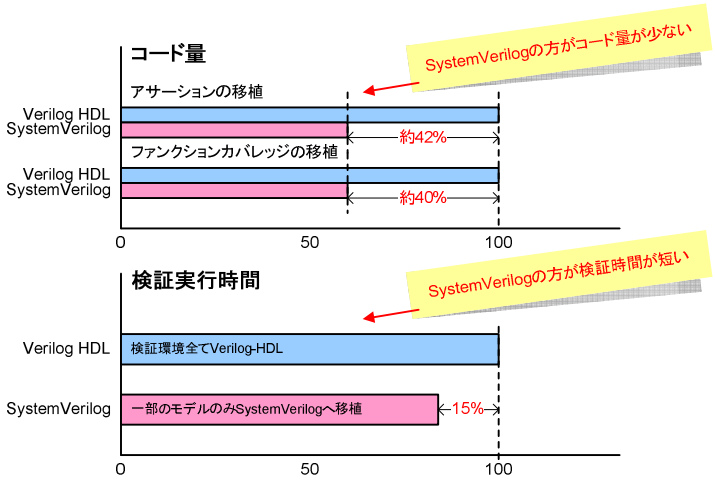
「ファイル入力 vs PLI」では PLI の方が約 14%短縮できる
「PLI vs DPI」では、DPI の方が約 11%短縮できる

検証実行時間の観点では、DPI を利用することで検証時間が短縮でき、生産性向上が見込まれる。

以上の SystemVerilog の評価の結果、以下の点で生産性向上が見込まれる。

1) Verilog HDL と比較した際の SystemVerilog の優位性

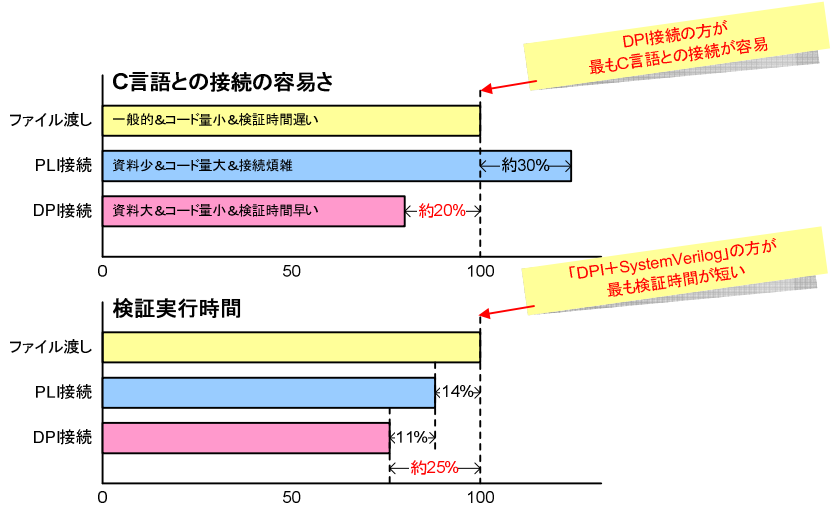
- 「ハード IP の検証」または「従来の HDL 開発の検証」に関しては、SystemVerilog を利用することにより下図のような生産性向上が見込まれる。
  - 記述量、検証実行時間ともに SystemVerilog の方が優位である。



図【3-1】. 6 Verilog HDL と比較した際の SystemVerilog の優位性の評価結果

2) C 言語との親和性

- 「ハード IP の検証」または「従来の HDL 開発の検証」の C 言語との接続に関しては、DPI 接続を利用することにより下図のような生産性向上が見込まれる。
  - 総合的に見て、「DPI 接続」が他の接続方法と比べて優位である。
  - また、DPI 接続は SystemVerilog で使用できるため、SystemVerilog での検証環境構築による生産性向上との相乗効果が期待できる。



図【3-1】. 7 C 言語との親和性の評価結果

4-1-2-3　作成物

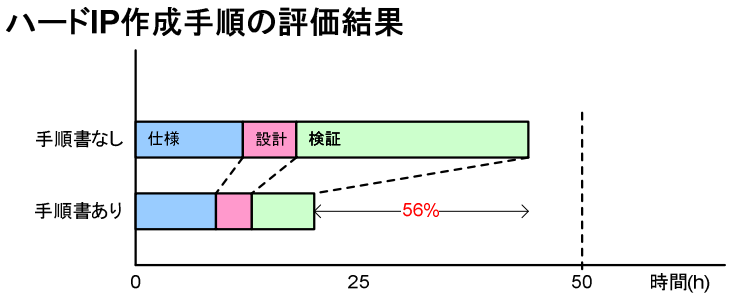
本テーマでは、以下に挙げる手順書およびハード IP を開発した。

- ハード IP 作成手順書
- SystemVerilog 手順書（SystemVerilog のリファレンスマニュアル）

4-1-3　まとめ

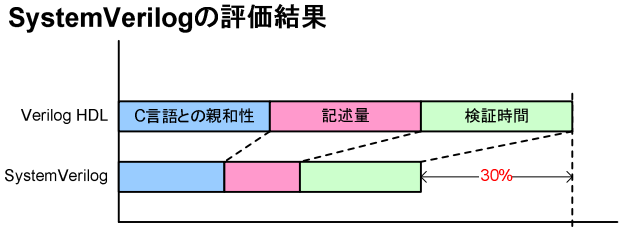
4-1-3-1　成果

本テーマでは、従来の HDL 作成手順をもとに新たなハード IP 作成手順を作成し、その手順が生産性向上の手法として問題がないか検証を行った。その結果、従来の HDL 作成手順では開発全体の工数が約 41h であったものが、新たに作成したハード IP 作成手順を用いることで工数が約 22.5h となり、工数が約 50%短縮され生産性は約 2 倍の向上が見込まれる。



図【3-1】. 8 ハード IP 作成手順の評価結果

また、Verilog HDL と SystemVerilog の比較を行い、SystemVerilog を導入することが生産性向上の手法として問題がないか検証を行った。「C 言語との親和性」、「記述量」、「検証実行時間」を総合的に判断した結果、SystemVerilog を導入することで、生産性は約 1.5 倍の向上が見込まれる。



図【3-1】. 9 SystemVerilog の評価結果

4-1-3-2　結論

本テーマで作成したハード IP 作成手順を利用および「ハード IP の検証」に対する SystemVerilog の導入により、一定の生産性向上が見込まれる。

今後さらに生産性の向上を行うためには、以下の課題を解決していくことが必要である。

1) 「仕様」段階で利用できる汎用マクロや汎用モジュールなどの IP 類の整備
   現状は、FIFO や非同期マクロなどの機能レベルの IP やマクロしか用意されていないため、種類およびバリエーションを増やす必要がある。また、メモリコントローラや標準規格に準じたバスブリッジなどの汎用的に利用可能なモジュールレベルの IP や手順書を数多く準備することにより、ハード IP を作成すること自体の削減になり、開発全体での大幅な生産性向上が見込まれる。
2) 「検証」段階で利用できる期待値生成やアサーションなどの検証ライブラリの整備

今回、検証環境の自動生成とあわせて、既に作成されていたアサーションやファンクションカバレッジ計測なのどのチェッカーを利用することで大幅な生産性向上が図れた。この生産性向上率を維持するためには、標準規格に準じたチェッカーなど利用機会が多くかつ汎用的に利用可能な検証ライブラリをあらかじめ整備しておく必要がある。また、画像系や通信系など規格化されたものに関しても、データ生成や期待値チェックの部分などは汎用的に利用可能であり、利用頻度も多いことが予想されるため、積極的に検証ライブラリとして整備する必要がある。

3) 「検証」に対して SystemVerilog の積極的な導入

今回、アサーションやファンクションカバレッジ計測などの一部の検証ライブラリを Verilog HDL から SystemVerilog へ移植するだけである一定の生産性向上が図れた。検証全体を SystemVerilog で行うことによりさらなる生産性向上が見込まれる。

また、今回は「ハード IP の開発」に関してのみ SystemVerilog の導入を行ったが、他のテーマで行われた C ベース設計に関しても、検証部分への SystemVerilog の導入で生産性向上が図れる可能性がある。今後は、「C ベース設計の検証」に関する SystemVerilog の導入について検討および評価が必要である。

4-2　【3-2】ソフト・ハード共用ライブラリや IP の開発

4-2-1　概要・目的

C/C++の高位合成による設計はアルゴリズムレベル(上位設計)では高い生産性が期待出来るが、メモリなどデバイスのインターフェースなど信号制御が必要な回路設計では優位性が低いとされている。そこでこのような回路をライブラリや IP として準備して C/C++設計した回路と接続する方法で生産性を上げる事を検討した。IP を準備しておけば既存の HDL による設計でも生産性を上げる事が出来る。ここでは、IP やライブラリを C/C++設計することで生産性を向上させる方法を調査し作成の手順や利点と問題点の洗い出しを行った。

また、将来的にライブラリや IP の作成手順や規格やフォーマットを国際的に提案することや、ライブラリや IP の販売をおこなう事も視野に入れて開発を行わなければ優位性をアピールしていく事が出来ない。そのため、ライブラリや IP に関連する、組織や考慮すべきインターフェース規格やフォーマットの調査と、また世界の IP 市場はどのようなシーズがあるのかの調査も行い今後の展開も考慮することとした。

ここでライブラリや IP を、C/C++/SystemC などソフトウェア用の言語で設計したライブラリや IP と、Verilog HDL や VHDL など設計したライブラリや IP とに分類して、前者をソフト IP、後者をハード IP と記述する。

4-2-2　内容

4-2-2-1 ライブラリや IP の市場動向の分析

IP の提供を行っているベンダーの調査を行った。国内の主要ベンダーと、標準規格への参画組織などを中心に合計で 100 以上のベンダーの調査を行った。調査方法は、各ベンダーのホームページからプロダクトページ等を目視で確認した。調査項目として、IP の有無、IP の提供されるソースコードの記述言語、機能別のラインナップなどを対象とした。提供している IP が、ソフト IP なのかハード IP なのかを集計した。以下の表の様な結果になった。

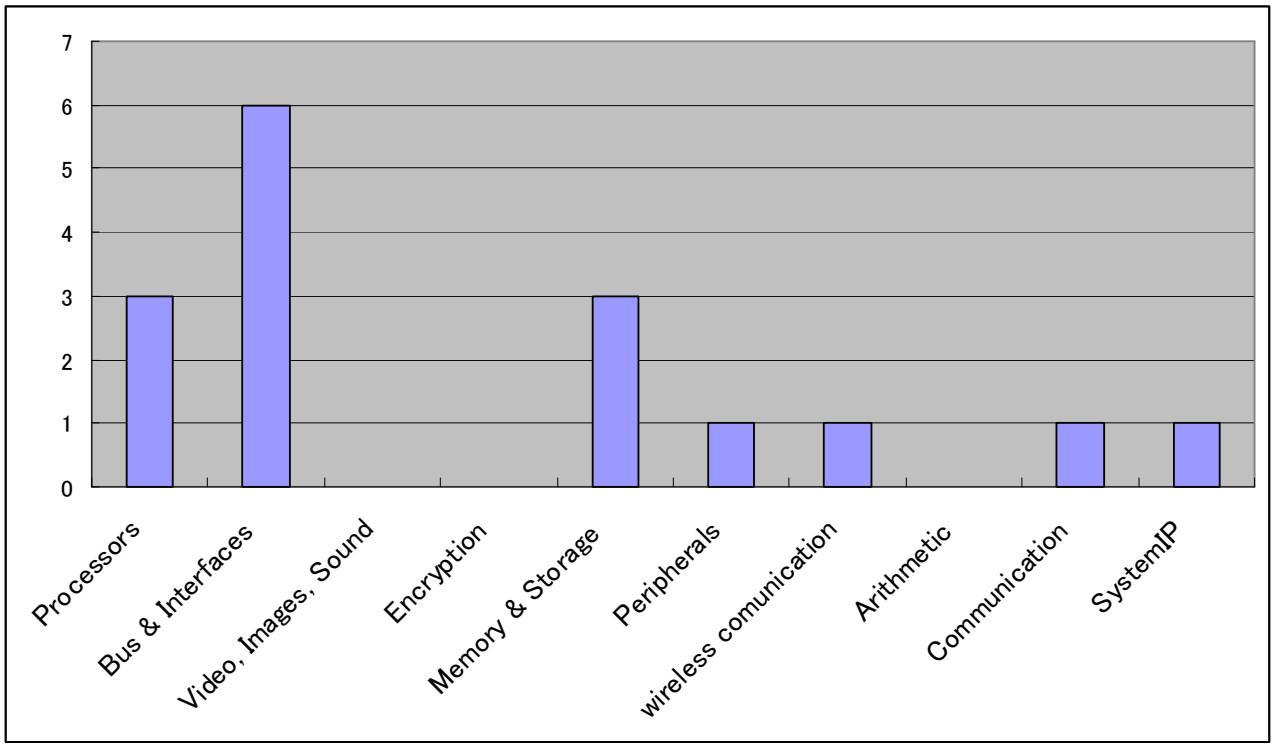
表【3-2】.1　IP コアベンダー数

	全ベンダー	ソフト IP ベンダー	ハード IP ベンダー
個数	108	13	108
割合(%)	100	12.0	100

ソフト IP を扱っているベンダーは、全体の 13%で 14 社見つかったが、その全てでハード IP の提供も行っていた。しかし、そのソフト IP と同じ機能のハード IP を持っているという事ではない。今回の調査は WEB のみで行ったが、そのほとんどが検証用と論理合成用を分けて表示してあり、同じ機能であるか判断出来なかったが、プロセッサなどいくつかの IP は、共用されていると思われる。

ソフト IP のベンダーとしては、大手 EDA ベンダーか独自のプロセッサやバスを持っているベンダーが大半を占める。これは、そのベンダーの戦略的な方針によるものと思われる。それ以外として、オープンソースの IP を提供するコンソーシアムが特別な例として挙げられる。利用する面では非常に有効であるが、販売としては競合することが考えられる。

ソフト IP の品種の傾向を以下のグラフに示す。



図【3-2】.1 ソフト IP の品種の傾向

ソフト IP は、プロセッサ、バス・インターフェース、メモリ関連の IP がほとんどを占めている。これは、ソフト IP は SOC などのシステム設計で使用される事が多いためと思われる。TLM でのパフォーマンス検証や、組み込みソフトとの協調検証での利用のためと思われる。

4-2-2-1　共用ライブラリや IP のフォーマットの検討

4-2-2-1-1 世界標準フォーマットや規格と策定団体の調査

IP 製品は、規格に準拠した機能を持つ製品が多くを占めている。規格にも、画像、通信、バスなど多くの種類が存在しているが、ここでは、機能の規格ではなく、ライブラリや IP そのものを作成、接続するために制定された標準規格を調査の対象としている。

ライブラリや IP を作成し販売を行う場合には、世界的な標準規格やデファクトスタンダードに準拠した製品である事が、他製品に対して差別化の重要なファクターとなる。また、それらの規格の技術や思想を取り入れる事で、設計の効率化や、そのための支援ツールの作成にも貴重な情報になる。そして、将来的には、自社開発の規格を制定して広めていく事も考えられる。

調査は、WEB の検索サイトを利用して行った。フォーマットや規格として、SystemC は既に IEEE で標準化されていることは周知だが、注目すべき規格として、IEEE の標準化に向けて動いている規格で、メタデータを XML データベースとして持つ事で、ツール間の接続や、ドキュメントへの変換がスムーズにできる事を目指している規格が存在した。当社としても IP 作成だけでなく、開発支援ツールの設計構成に大変参考になると思われる。その他で、IEEE の規格化への動きはなく、デファクトスタンダードか独自規格としてバスや検証環境を規定して製品として提供されている規格がほとんどであった。

4-2-2-2　共用ライブラリの作成手順の検討

ソフトIPは、主に検証用として作成されて、高位合成や論理合成を行う事ができないが、C/C++やSystemCで記述されているため、検証でシミュレーション時間が短縮できるという特徴を持つ。ハード IP は、論理合成が可能であり、詳細な回路記述を行うことに向いているが、抽象度の高い記述が困難で検証のシミュレーション時間もソフトIPに比べて時間がかかるという特徴を持っている。

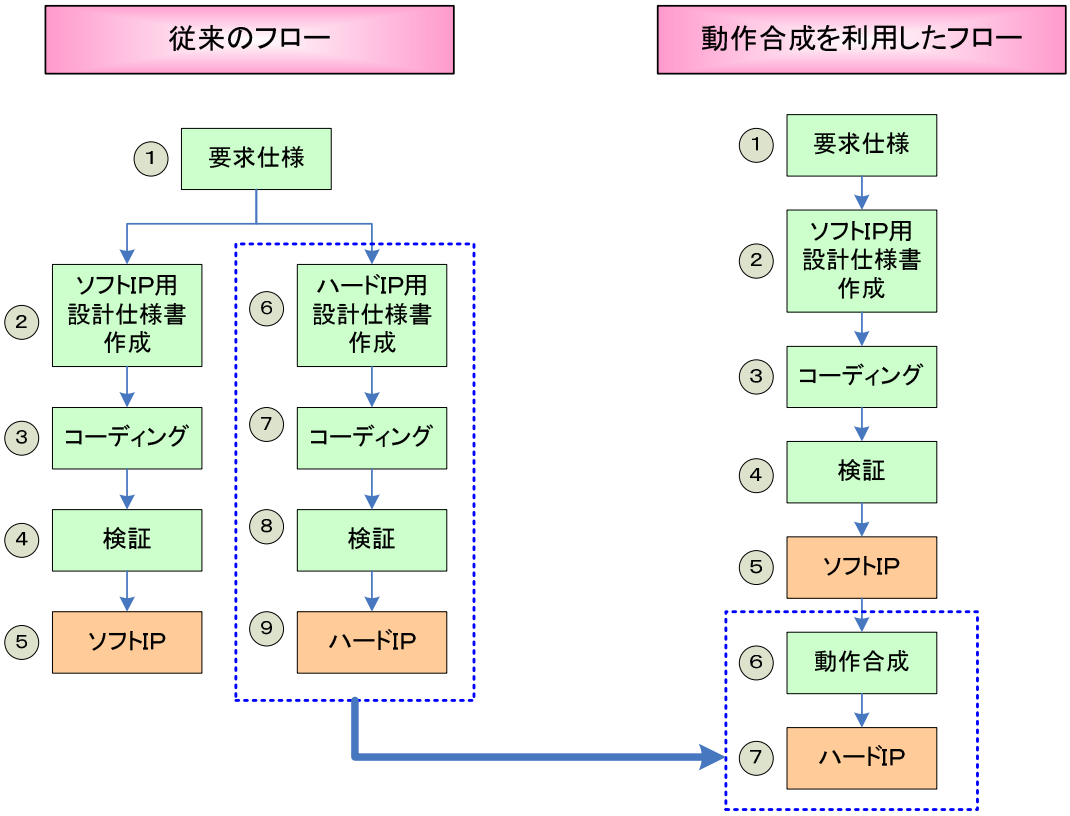
■共用ライブラリ

共用ライブラリとは、同じ機能を持つソフトIPとハードIPの両方を準備する事で検証の効率化と論理合成までの効率化を行う事を目的としたライブラリである。本研究では、共用ライブラリの作成が高位合成ツールを使用して効率的に行えるのか検証を行った。

高位合成ツールは、本来、アルゴリズムレベルで書かれた C/C++のモデルを高位合成することで飛躍的な設計効率の改善を期待出来るツールである。しかしメモリコントローラのような複雑な信号制御が必要な回路の場合には HDL 言語と同様の詳細な記述が必要となるが、高位合成用の C/C++記述では、詳細な記述は不得手であり、開発効率が低くなってしまう。そこで、各ツールでは、外部メモリやバスなどの規格用にライブラリを準備して高位合成時に適切なライブラリを選択する方法で対応しようとしている。だが、現状は全てのインターフェースへ対応するには不十分である。そこで、独自に共用ライブラリとシステム設計で必要な環境を準備して開発の効率を向上させる手法が必要となる。

■共用ライブラリ作成の効率化

共用ライブラリを作成する従来手法と高位合成ツールを使用した場合の新手法の比較を以下の図に示す。



図【3-2】.2 共用ライブラリ作成のフロー

従来手法は、既存の設計資産がない場合を前提条件としている。つまり新規設計の場合のフローを想定している。従来手法のフローは、ソフト IP とハード IP をそれぞれ同様の工程で作成する。その時、ソフト IP を作成して、それをリファレンスにしてハード IP を作成することでデバッグなどの効率が上がることは考慮にいれることとする。これを新手法の高位合成を利用したフローでは、C/C++検証と高位合成が可能なソフト IP を作成する事で、ハード IP の作成を高位合成ツールで行う事で、工数を半分程度に圧縮することが期待出来る。

■C/C++検証と動作合成が可能なソフト IP と高位合成用記述言語

新規手法を実現するためには、作成するソフト IP には、C/C++検証と高位合成の両方が可能である事が必要である。これを C/C++検証と高位合成が可能なソフト IP と呼ぶこととする。この C/C++検証と高位合成が可能なソフト IP の機能としては、サイクルベースでのタイミング制御などを含んだ回路の設計も出来る必要がある。つまり RTL モデルの記述が可能でなければならない。RTL モデルの記述としては、HDL の方が向いており、高位合成には向いていないとされているが、実際の実用性はどうなのかを検証する。

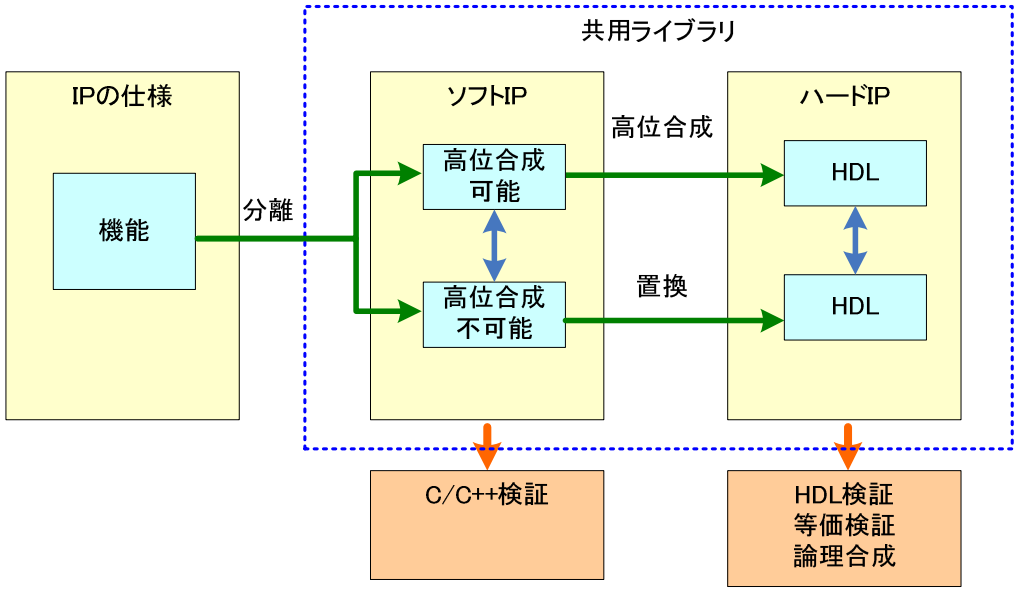
現在、高位合成のための C/C++記述の標準規格は存在していない。そのため高位合成ツール毎に記述方法が異なっていて、各社の記述言語は、独自の言語として存在している。今回は、3 社のツールを使い分けてソフト IP を作成したが、高位合成ツールによって RTL モデル記述の容易さが異なる。ツールによっては、コンパイル設定とディレクティブにより、RTL モデルでの記述が容易に行え HDL に近い記述を行う事が可能なものもあった。また、高位合成後に出力される HDL ソースコードにも違いが見られた。手書きで作成した HDL 記述に近くて視認性のよいソースコードを出力するツールや、基本構成で複数のモジュールに分割して記述されるツールスケジューリングのためのステートマシンが生成されるツールなど各機能を反映した出力結果となっていた。

■高位合成用記述言語と検証

高位合成用に記述したソースコードで C/C++検証を実施する場合、C/C++コンパイル用のライブラリが提供されていれば高位合成用のソースコードを修正なしで C/C++検証を行う事が出来る。しかし、ライブラリが無い場合は、高位合成用言語で記述したソースコードのままでは C/C++検証を行う事ができない。そのため、プリプロセッサにより検証用の記述と高位合成用の記述の切替えが必要となる。切替え記述を行った場合、その追加する記述量は多く効率的ではない。また記述が 2 重化されると C/C++検証では不具合が無いが高位合成を行うと不具合が起きてしまうという品質的な問題も発生する。今後は、2 重化される記述をより少なくする方法と等価性の評価により、効率と品質の両面での改善を検討する必要がある。ベンダーから C/C++検証用のライブラリが提供されるようになればこの問題は無くなるが、ベンダーの対応が無い限りはプリプロセッサで記述を切り替える方法でソースコードを作成するしかない。

■高位合成用記述言語による RTL モデル記述のサンプル回路検証

ツール毎の記述の違いや高位合成の可能性を確認するために、RTL モデル記述でよく使用する回路をいくつか作成して実現性の検証を行った。結果としては予想以上に各ツールとも RTL モデルの記述に対応出来た。ただ一部のツールでは論理の最適化により RTL モデルでの記述が困難であったり、記述出来ない回路もあったが、実用上の問題は少ないと思われる。たとえば 1 モジュールに複数クロック入力の記述が出来ないような場合がある。このような回路が必要な場合は、下図のフローのように、高位合成出来ない部分を切り出して、その部分は、高位合成不可能な検証専用のソフト IP と、手書きで作成したハード IP をペアにした共用ライブラリを準備し使用する事で回避できる。



図【3-2】.3 高位合成が不可能な場合の対応

■各高位合成ツールの入出力ファイルと検証の実施

検証と動作合成可能なソフト IP 作成時の検証で使用するソースファイルのレベルを検討する。ツールによって、入力ファイルの記述言語と出力できる記述言語が異なっている。入力可能な言語は、各社の独自言語が標準であるが、ツールによって SystemC 言語による入力も可能である。今回は SystemC 入力での検証は行っていない。出力可能な言語も各社で異なっている。等価検証のために、サイクルベースの C/C++や SystemC を出力するツールや、Verilog HDL をシミュレーション用と論理合成用の 2 種類出力出来るツールもある。今回は、サイクルベース SystemC と Verilog HDL を使用して検証を行っている。下表にソフト IP 記述レベルと検証の実施可能性についてまとめた。

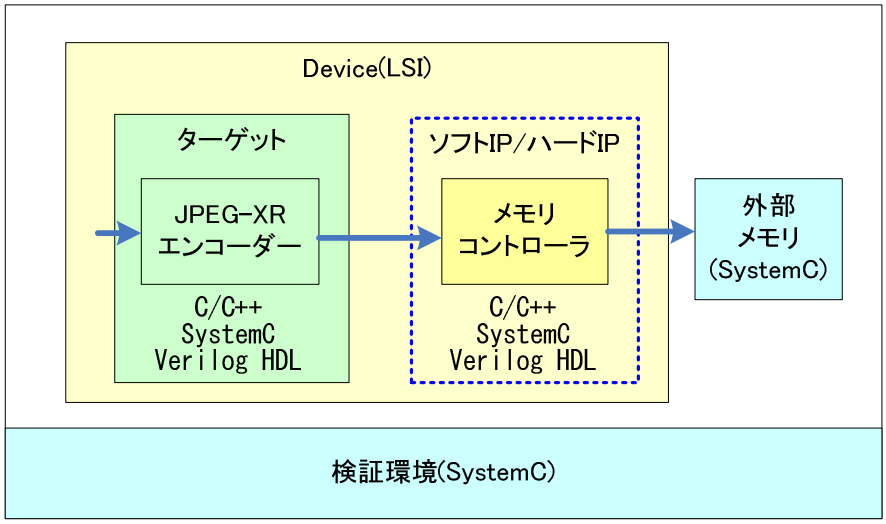
表【3-2】.2 ソフト IP 記述レベルと検証の実施可能性

ソフト IP の記述レベル				検証の可能性	備考
検証方法	エントリー	C/C++		△	C/C++ライブラリが提供されていないツールでは修正が必要。今回検証実施。
		SystemC		△	入力可能なツールもある。今回対象外。
	高位合成後	サイクルベース C/C++		△	出力可能なツールもある。今回対象外。
		サイクルベース SystemC		△	出力できないツールもある。今回検証実施。
		HDL	シミュレーション用	△	出力可能なツールもある。今回対象外。
			論理合成用	○	Verilog HDL と VHDL 両方に対応しているものもある。今回 Verilog HDL で検証実施。

4-2-2-3　共用ライブラリの検証

■FPGA モチーフでの実証

ここまで、各高位合成ツールでの RTL モデル記述や検証方法について調査してきたが、ここではその成果に基づいて実際にメモリコントローラを作成して生産性の検証を行って行く。高位合成可能なソフト IP を FPGA モチーフで実行させた場合の検証速度やコード量などによりどのように効率化が出来るのか検証する。FPGA モチーフとは、FPGA に入る比較的小規模な回路で複数のモジュールがバスで接続されている様な構成となる。FPGA モチーフの構成を下図に示す。



図【3-2】.4 IP の検証環境

JPEG-XR エンコーダーは、【2-1】で設計した C のソースコードと高位合成した SystemC と Verilog HDL を使用して検証を行う。JPEG-XR のエンコーダーから、1 マクロライン分を圧縮したデータが、ストリームとして出力され、メモリコントローラは、64x64Bit 単位で、外部メモリへ転送する。外部メモリは、64bit 幅の同期メモリを想定している。生産性比較のために、各高位合成ツールの記述以外に、RTL モデル記述の SystemC でモデルの作成も行う。

■ソースコード量

高位合成ツール毎のソースコードのライン数の比較を下表に示す。

表【3-2】.3　メモリコントローラソースコードのライン性

記述レベル		全ソース		論理記述部分のみ	
		ライン数	倍率	ライン数	倍率
RTL モデルの SystemC		157	1.3	80	1.0
高位合成前	高位合成が可能な C	120	1.0(基準)	80	1.0(基準)
	検証と高位合成可能な C (プリプロセッサによる切り分け)	165	1.4	124	1.6
高位合成後	サイクルベース SystemC	1720	14.3	—	—
	Verilog HDL	1729	17.3	—	—
手書きの Verilog HDL		170	1.4	81	1.0

高位合成が可能なCのソースコードと、RTLモデルのSystemCや手書きのVerilog HDLと比べると、全ソースコードの差は1.4倍程度あるが、これは、Verilog HDLでは信号別にalways文で囲うため冗長になっているためである。冗長な部分を除いた場合が論理記述部分のみの項目にあるが、どのソースコードもほぼ同じになっている事がわかる。このことから、RTL モデル記述では、言語による違いが、ほとんど無いことがわかる。検証と高位合成が可能な C のソースコードは、プリプロセッサにより検証用記述と高位合成用の記述の切り分けを行っている記述になるが、全ソースのライン数では、元ソースの 1.4 倍になっている。これはプリプロセッサにより専用ディレクティブを切り替えている箇所がほとんどで元のソースコードの再利用がある程度出来ていると言える。記述の手法やコーディングルール等で再利用率を上げる事は可能だと思われる。

■シミュレーション環境

シミュレーションは、SystemC でテストベンチを記述して、その上に言語の異なるモジュールを実装した。基本的にシミュレーターは、HDL シミュレーターを使用した。同じシミュレーターを使用した場合のモジュール記述レベルによる違いの測定を行うためである。また、SystemC シミュレーターを使用した場合とシミュレーション時間を比較するために、全てのモジュールが SystemC の場合に、SystemC シミュレーターでシミュレーションも行っている。

■シミュレーション時間

JPEG-XRモジュールの記述レベルと、合成可能なソフトIPの記述レベルを組み合わせてシミュレーション時間を測定した。

（検証条件）

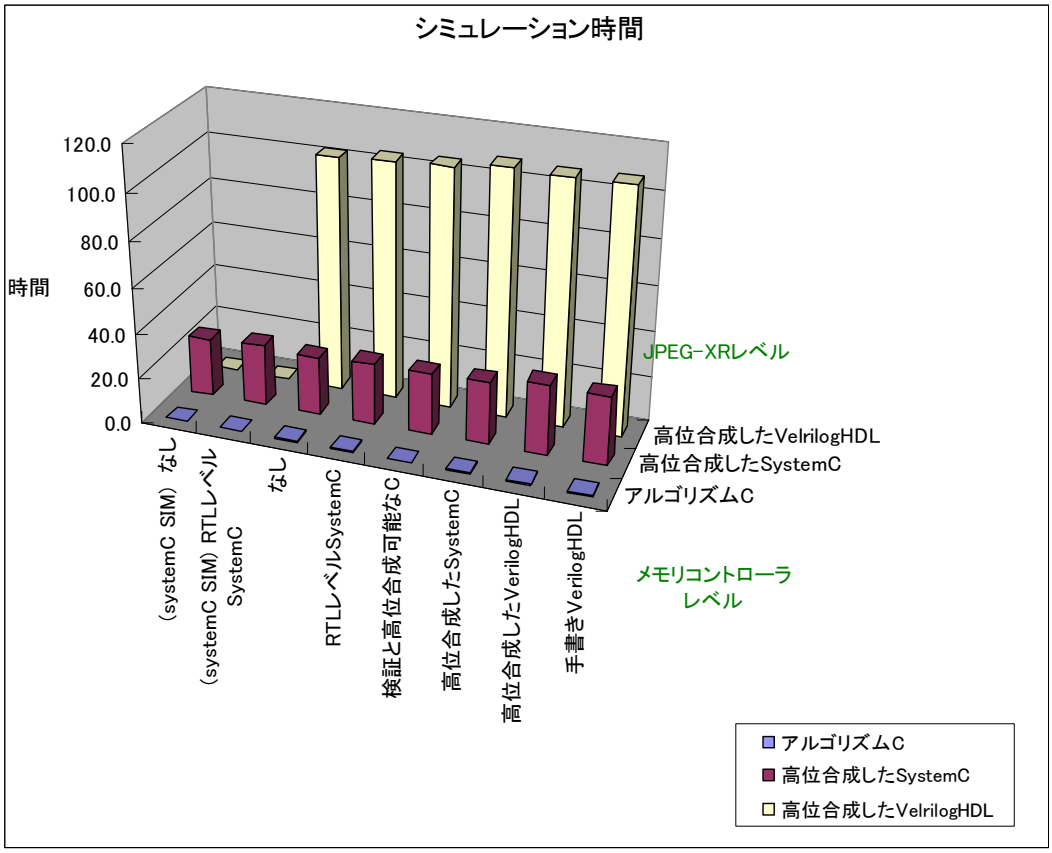
- SystemC シミュレーターは、OSCI のライブラリと標準 C コンパイラを使用した。SystemC と記載。
- HDL シミュレーターは、市販の HDL シミュレーターを使用した。HDL と記載。
- テストベンチは、SystemC で記述し、C/C++と SystemC と Verilog HDL の混在環境とした。
- JPEG-XR の高位合成した SystemC と高位合成した Verilog HDL は使用ツールと最適化方法が異なる。

結果を下表に示す。

表【3-2】.4　検証レベルとシミュレーション時間

No	記述レベル		シミュレーター	シミュレーション時間(秒)	比率(倍)
	JPEG-XR	メモリコントローラー			
1	アルゴリズム C	なし	SystemC	0.2	0.51
2		RTL モデルの SystemC	SystemC	0.3	0.75
3		なし	HDL	0.2	0.49
4		RTL モデルの SystemC	HDL	0.4	1.00
5		検証と高位合成可能な C	HDL	0.4	1.04
6		高位合成した SystemC	HDL	0.5	1.53
7		高位合成した Verilog HDL	HDL	0.8	2.29
8		手書き Verilog HDL	HDL	0.6	1.78
9	高位合成した SystemC	なし	SystemC	25.6	0.95
10		RTL モデルの SystemC	SystemC	27.1	1.00
11		なし	HDL	25.6	0.95

12		RTL モデルの SystemC	HDL	27.0	1.00
13		検証と高位合成可能な C	HDL	27.0	1.00
14		高位合成した SystemC	HDL	27.7	1.02
15		高位合成した Verilog HDL	HDL	31.1	1.15
16		手書き Verilog HDL	HDL	30.1	1.12
17	高位合成した Verilog HDL	なし	HDL	102.9	1.00
18		RTL モデルの SystemC	HDL	104.4	1.02
19		検証と高位合成可能な C	HDL	105.4	1.03
20		高位合成した SystemC	HDL	108.5	1.06
21		高位合成した Verilog HDL	HDL	107.4	1.04
22		手書き Verilog HDL	HDL	107.9	1.05



図【3-2】.5 検証レベルとシミュレーション時間の比較

今回作成したメモリコントローラーの IP は、JPEG-XR に比較して回路規模が小さくシミュレーション時間のほとんどが JPEG-XR に費やされていることがわかる。全体としては小さい差であるが、ソフト IP とハード IP のシミュレーションの負荷の違いが確認出来る。例えば、JPEG-XR が高位合成した Verilog HDL の場合であれば、メモリコントローラーが、無しと、RTL モデルの SystemC との差は 1.5 秒で、同じく、メモリコントローラーが、無しと手書き Verilog HDL との差は 5 秒あり、差は 3.3 倍にもなる。このことからも検証にソフトIPを使用することでシミュレーションの効率が向上することがわかる。

4-2-2-4　生産性計測

検証用ソフトIPと論理合成可能なハードIPからなる共用ライブラリの作成のための生産性の計測を行った。対象は、前述したメモリコントローラーで実際にかかった工数を計測している。計測方法は時計による実測ではなく作業者からヒアリングして工数を求めている。既存手法は、ソフトIPとハードIPを個別に作成する方法である。新手法は、ソフトIPから高位合成ツールを使用してハードIPを作成する方法である。新手法では、ハードIPの作成の工数が大幅に削減出来るため生産性の向上が期待出来る。

条件

- メモリコントローラの設計にかかった工数である。
- 工数は作業者からヒアリングして求めている。
- テストベンチは、1本のみで期待値の自動チェックは含んでいない。
- 高位合成記述は、C/C++ライブラリの提供されていないツールの言語を使用。

下表に工数の比較表を示す。

表【3-2】.5 手法の違いによる工数の比較

工程		工数(Hour)		工数差分	補足
		(1)既存手法	(2)新手法	(Hour)	
ソフトIP	設計	10	10	0	同等
	コーディング	5	8	-3	B社の記述では、検証用の記述の追加が必要となる。
	検証	30	30	0	検証と高位合成で記述が分かれているため両方の検証が必要
ハードIP	設計	10	0	10	全削減可能
	コーディング	5	0	5	全削減可能
	高位合成	0	1	-1	(2)の追加項目
	検証	30	2	28	(2)ではソフトIPと同じ検証を実施可能
合計		90	51	39	既存手法に比べ43.3%工数減で1.8倍の効率化

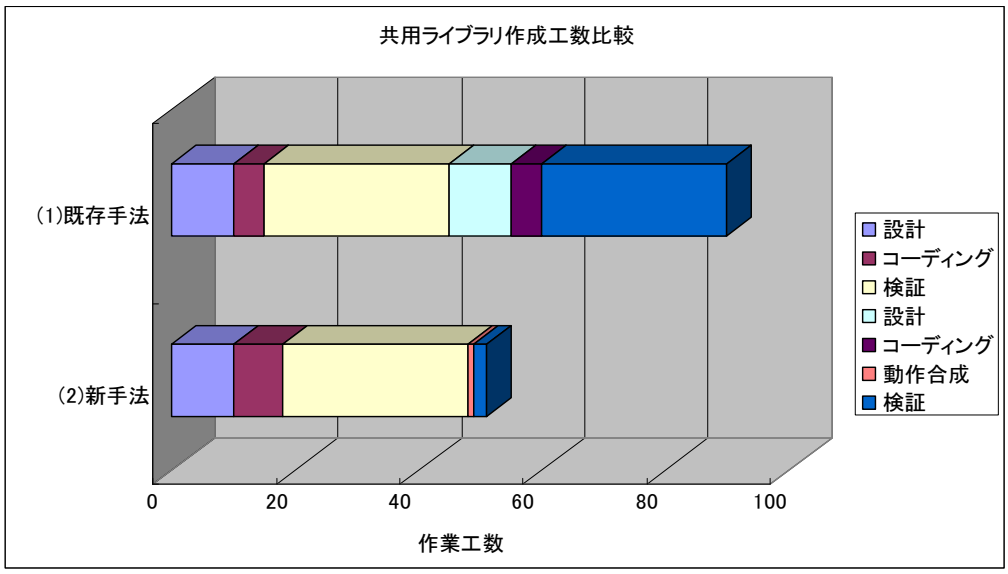
ソフトIPの検証環境が、ハードIPの検証でそのまま使用出来るようにしたため、ハードIPの検証は、対象ファイルを切り替えてシミュレーション実行するのみで確認ができたことが工数削減の大きなポイントになる。逆にソフトIPのコーディングの工数が多くなっているが、これは検証用の記述と動作合成用の記述を分けて記述したためである。C/C++ライブラリが提供されているツールを使用すればこの工数の削減が可能になる。

4-2-3　まとめ

4-2-3-1　成果

■生産性の向上

今回検討したソフトIP作成フローにより、ソフトIP作成の工数を43.3%ほど削減し、生産性を1.8倍向上する事が可能であると考えられる。



図【3-2】.6 共用ライブラリ作成工数比較

4-2-3-2　結論

■高位合成ツールを使用した共用ライブラリ作成の問題点

- 高位合成用の言語のライブラリが提供されていないツールは、そのままのソースコードで C/C++検証を行う事が出来ないため、ディレクティブでの切り分けが必要となる。
- 本来の高位合成ツールは、スケジューリングにより回路を合成するため、RTL モデルの設計で信号制御を行うには不向きである。各ツールの高位合成の仕様を考慮した記述手法の確立が必要である。
- 高位合成ツールが出力できるインターフェースの種類が少ないため、そのインターフェースに適応した記述が必須となり、必要な機能が制限される場合がある。

■共用ライブラリに求められる機能

- 共用ライブラリには、C 検証用のソフトIPと論理合成可能なハードIPの他に、TLM インターフェースのIPを含むとSOC モチーフにも対応でき生産性向上に有効である。
- 接続性として、バス規格への準拠が必要と思われる。世界標準にはなっていないが、広いシェアを持つプロセッサ用のバスや、物理層用のバス規格もあり、接続性による差別化が期待出来る。
- IP には独自機能による付加価値が求められる。同じバス準拠の製品であっても、高性能なアービターなど、他社に真似出来ない価値が必要である。

■ソフト IP の市場

IP を、ハード IP とソフト IP をセットにして提供しているベンダーは少なく、自社の LSI の検証用として提供している場合がほとんどである。ソフトIPとハードIPをセットにして販売することで新規需要を開拓出来ると思われる。

IP の種類としては、各ベンダーが規定した内部バス規格に準拠した IP の需要が多い。しかしFPGA分野に限ると、FPGA 各社は独自のバス規格を持つプロセッサとその周辺を自動に接続する機能を持つ IP を準備し無料や安価にて提供する事で、デバイスの販売促進に利用している。FPGA 分野を ASIC などで標準とされるバスに置き換える事は困難である。そこでFPGAとASIC の両方に対応するため、複数のバス規格に対応した複数のIPを準備しなくてはならない。そのためにも IP や共用ライブラリの生産性の向上が必要である。

IP の標準規格で XML スキーマと XML 生成用言語を提供している規格がある。このアプローチは、設計情報を共有してソースコードのひな形や設計ドキュメントの作成などを容易にする方法として優れている。この技術を自社の支援ツールに応用する事でより生産性を向上できると思われる。

■高位合成ツールの今後

高位合成ツールの現在の状況は、記述言語がベンダー毎に異なり、高位合成が可能なソースコードで直接検証が出来ないなど、広く使用されるには、克服しなくてはならない課題が残っている。そこで注目されているのが、SystemC である。国際標準規格であり、RTL モデル記述にも対応していて、検証環境も整備されつつある。高位合成記述サブセットも規定され、今後、高位合成の主流になることが期待される。SystemC エントリーができる高位合成ツールも増えており生産性の向上が期待出来る。

また、、高位合成ツールを使用して共用ライブラリを作成する方法でアプローチを行ってきたが、RTLモデルの記述で HDL を出力するのであれば、高位合成ツールではなく言語変換という方法もある。言語変換であれば独自に開発する事も充分可能であり、変換の精度を高くすることも可能である。回路情報を XML でもち各言語へ変換するというツール作成を検討したい。

第5章 本論・ソフトウェアを LSI 化する設計手法の生産性向上方法の開発

5-1 【4-1】ソフトウェア記述の標準化による生産性の向上

5-1-1 概要・目的

C 言語で記述する上での回路構造について記述フォーマットを提案することで明確化し、生産性を向上させる。

1) ソフトウェア記述の標準化
   - 変数宣言:LSI 回路の入力フォーマット、出力フォーマット、周辺回路としてのメモリ、LSI 内部メモリ、演算回路等の区別をソフトウェア上の変数として宣言する。
   - 機能表現:関数などの利用によって回路規模および回路特性に合わせた規則を決定する。
   - 記述規則:コメント文をプログラムで再利用する規則を決定する。
2) 生産性の測定
   上記項目 1)の効果を、作業時間と記述量で測定し、生産性の向上率を評価する。
3) 支援ツール:上記項目 1)および 2)をソフトウェア化する。
   本項目では、人為的作業の排除およびアルゴリズムの改善を両立できる環境を開発することで、ソフトウェア記述の標準化による生産性の向上を目指す。

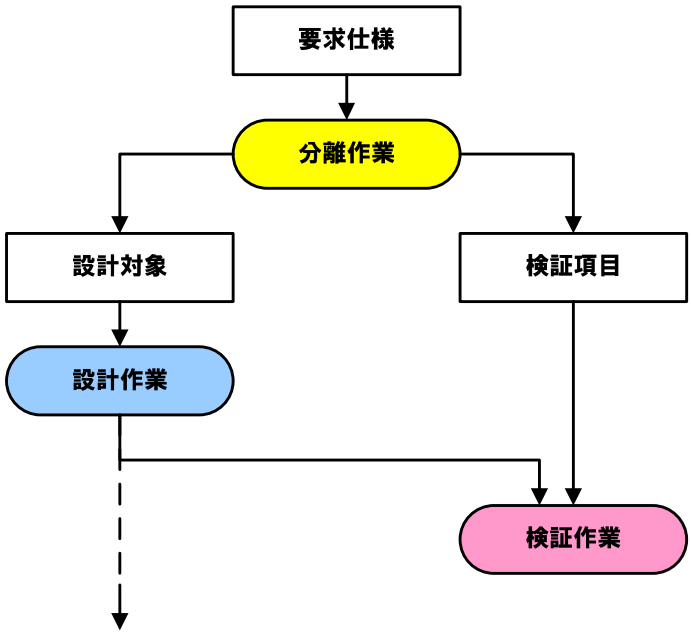
また、実施計画書のテーマ「【1-1】 支援ツールのフレームワーク開発」で定義されている「2) C 言語ベースの分離ソフトウェア」について、本テーマとの関連が深いため上記ソフトウェア記述の標準化と併せて検討・開発を進めていく。

なお、本テーマ単体では 30%程度の工数削減を目標とする。

5-1-2　内容

従来開発手法ではハードウェア記述言語(Hardware Description Language: HDL)で開発を行うのに対し、本研究事業では C 言語をベースとした新たな手法の開発を目的としている。C 言語ベースの開発手法では、C 言語で記述されたソフトウェアも要求仕様として取り扱うことを想定している。つまり、ソフトウェア形の要求仕様は設計と検証をそれぞれ含むものとして捉え、要求仕様から設計と検証に相当する項目をそれぞれ抽出する(分離作業)。抽出された設計、検証の各項目について開発を行うという手順を踏む。

各作業の関係図を以下に示す。



図【4-1】. 1 C ベース設計手法での分離・設計・検証の関係

設計作業では、分離作業によって抽出された設計対象を元に回路設計を行う。C 言語ベース開発手法では、高位合成ツールを用いることで抽象度の高い回路記述を行い、従来手法よりも生産性を高めることを目標とする。この目標を達成するためには、高位合成ツールにあわせた設計手順や 回路サンプルなどのノウハウを含む記述フォーマットが必要となるため、本テーマではこれらの手順・フォーマットを開発する。また先述の分離作業についても、高位合成ツールの特性・特徴を考慮した手順作りが必要となる。

検証作業は従来開発手法と同様に、検証項目ごとにテストベクタを作成し、設計した回路と接続して検証する。C 言語ベース開発手法では検証作業のベースとなる検証環境を共通化し、開発内の各工程で流用させることで従来開発手法よりも生産性を高めることを目指す。C 言語ベース開発手法では、開発規模や工程ごとに回路記述されている言語が異なることが想定される。そこで、どのような記述・言語・構成であっても接続できる検証環境があれば各工程での流用度を高められるのではないかと考え、相互接続性の高い検証環境を開発する。

上記 3 つの作業について、本テーマで開発する手順・フォーマット等の一覧を以下に示す。

表【4-1】. 1 本テーマでの開発目標一覧

作業分類	作成目標	内容・目標レベル
分離作業	分離手順	C 言語ソースコードまたはドキュメントで提示された要求仕様を、以下設計作業と検証作業で利用可能な要素に分解・整理できる 設計対象となる要素は、各高位合成ツールでの利用が容易となるように分離・抽出される
設計作業	設計ひな形	各高位合成ツールで適用可能な C 言語ベースの記述フォーマット
	設計手順	分離作業にて抽出された要素を利用して、高位合成可能なソースコードを作成できる
検証作業	検証ひな形	SystemC をベースとした検証環境 テストベクタを差し替えるだけで他の検証項目を検証実施できる SystemC だけでなく、ANSI-C、HDL(Verilog HDL)で記述されたモデルを接続可能 ポート単位の接続だけでなく、関数単位や TLM2.0 を用いたトランザクション単位での接続が可能
	検証手順	上記検証環境の取扱説明 SystemC 以外の言語で記述されたモデルの接続方法を網羅 ポート、関数、TLM2.0 などの接続への対応方法を網羅

これらの目標に対して、本研究事業の目的である生産性向上がどの程度行われるのか評価する必要がある。従来開発手法と C 言語ベース開発手法を用いて同一の要求仕様を元に分離・設計・検証(環境立ち上げ)の各作業を行い、その工数や記述量を多面的に判断し、本テーマの成果として評価する。

また、これらの手順をソフトウェア化する。テーマ【1-1】にて開発する統合置換ソフトウェアを用いて設計・検証の各ひな形を IP として開発し、必要に応じて速やかに展開・利用できるよう整備する。各手順についても同様に、テーマ【1-1】が提唱している開発手順の改善フローを利用することで、改善フロー内での支援ツール化を期待できる。

5-1-2-1　分離作業に関する検討

本テーマでは、C 言語ベースの設計手法について、その設計手法自体の開発を主たる目的とする。

先に述べたとおり、要求仕様としてアルゴリズム記述された C 言語ソースコードも想定しており、そのソースコードから後に述べる設計作業・検証作業で使用可能な仕様の要素に分解(分離)する手順について検討を行う。また、分離作業は後に述べる設計作業、特に高位合成ツールとの関連が強いため、分離作業だけでなく設計作業も考慮に入れた検討を行う。

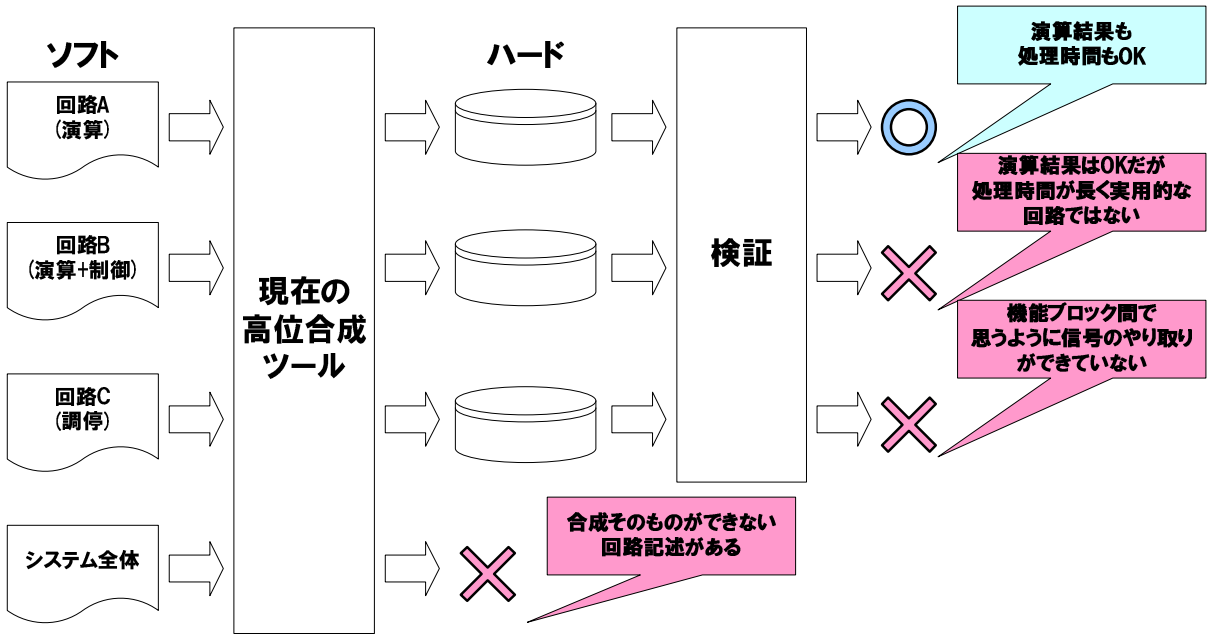
5-1-2-1-1 分離・設計作業で考慮すべき高位合成ツールの特徴

高位合成ツールとは、C 言語などの動作記述から順序回路や組み合わせ回路といったハードウェア固有の回路記述(RTL 記述)へと自動変換するソフトウェアである。従来手法では検討に時間がかかっていたタイミング設計を自動で行うこともできる。本テーマでは、高位合成ツールを使用した回路記述への変換に関わる手順を整備することで生産性を向上させる。このため、高位合成ツールに対する考慮が非常に重要な要素となってくる。

現在の高位合成ツールでは、システム全体、あるいは大規模な回路をまとめて合成することはツール性能上困難であり、合成できないか、あるいは所定の仕様を満たしていない回路が生成される可能性がある。このため、高位合成ツールで実用上変換可能な規模までシステム全体の記述を分離・分解していき、それぞれの単位で高位合成を行う必要がある。

つまり、分離作業および設計作業においては、上記のような高位合成ツールの特徴を把握して、作業を行うことが重要である。

※注意:高位合成ツールの種類毎に得意・不得意があり、合成可能な回路範囲には差がある。



図【4-1】. 2　高位合成ツールの問題点

5-1-2-1-2 要求仕様の分離

本テーマでは、開発の源泉となる要求仕様を、「システムの機能や動作、その制御方法など」を「文章と図面で示した要求仕様書」と、「C 言語でモデル化したアルゴリズムレベルのソースコード」の 2 通りで想定している。

要求仕様が文書として提示されている場合、従来手法による設計と同様、文書の内容を理解した上での構造検討を行う必要がある。あらかじめハードウェア化を意識した構造検討を行い、高位合成に適した回路構成、高位合成可能な記述方法とモジュール(関数)間のインターフェースにしておくことが重要である。これらの事項について考慮されたアルゴリズムレベルのソースコードを作成しておくことにより、以降の工程での高位合成に要する工数の削減を図り、生産性向上に繋げていく。

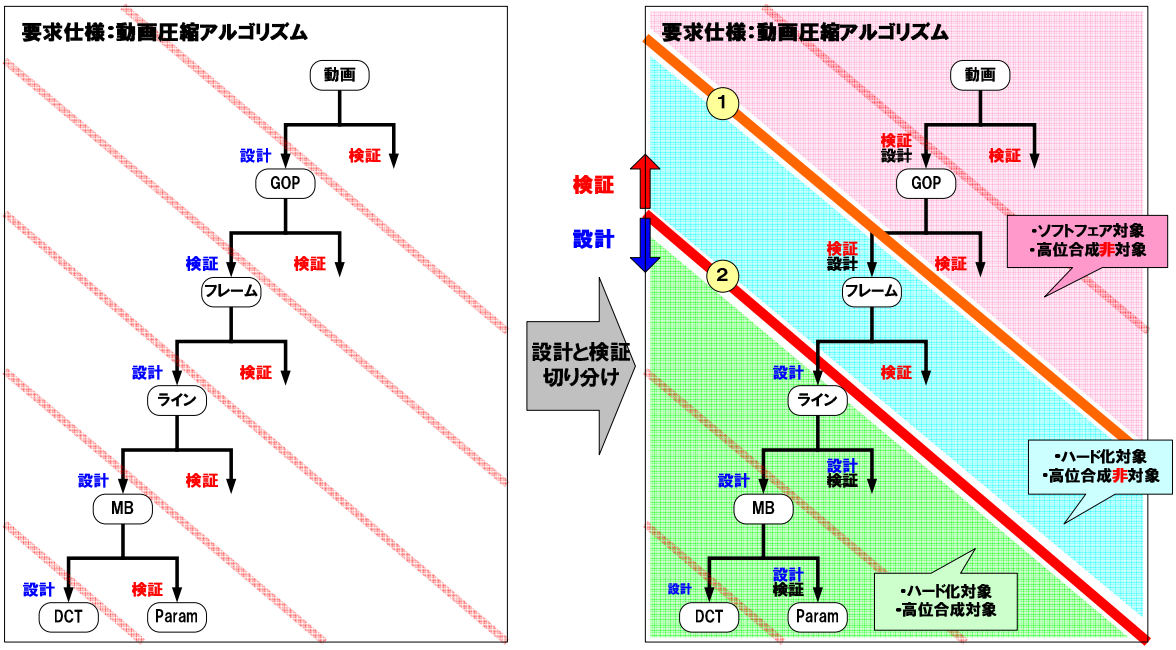
■ 設計と検証の界面

要求仕様が C 言語ソースコードで提示されている場合も同様にハードウェア化を意識した構造検討を行うが、このソースコードはハードウェア化する設計記述とテストベンチに相当する検証記述とが混在した状態であることに着目する。ここでの検証記述とは、製品上では組み込みソフトウェアとして実装される部分、あるいはハードウェア化対象(製品上の回路)の複数機能ブロックを制御する回路部分を示す。

実際に高位合成行う過程においては、まずチップ規模や要求定義によって、組み込みソフトウェアとして実装する部分とハードウェア(回路)として実装する部分との切り分けを行う必要がある。次に切り分け後のハードウェア化範囲において、高位合成対象ソースコードと高位合成非対象ソースコードとに分離するという手順が考えられる。

以下に動画圧縮アルゴリズムの分離イメージを示す。

- 界面①の上側:テストベンチ、上位ハードウェア、ファームウェア、人による操作等
- 界面②の下側:設計対象(高位合成対象)
- 界面①と②間:上位ハードウェア、テストベンチ もしくは ファームウェア

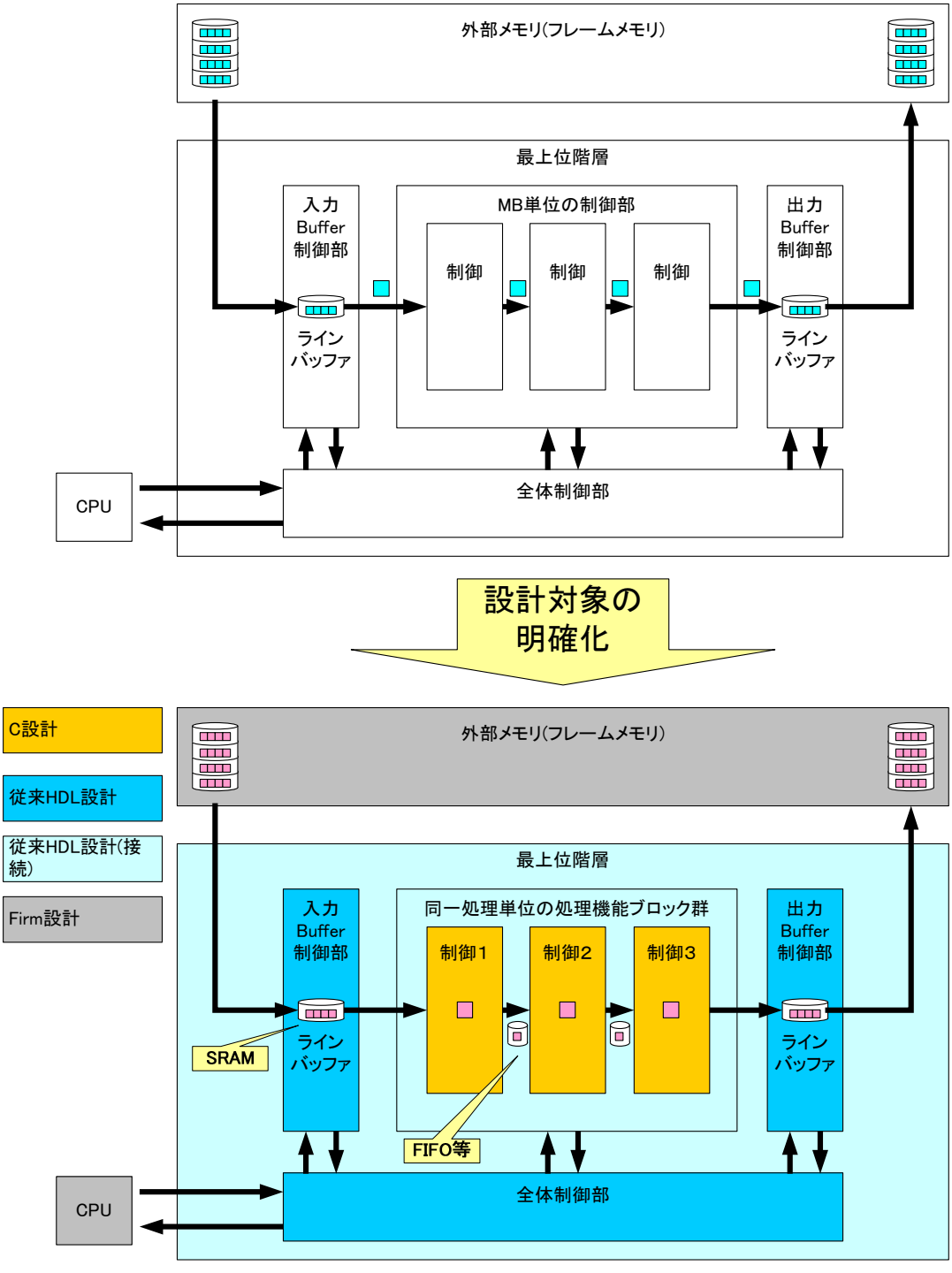


図【4-1】.3　動画圧縮アルゴリズムの分離イメージ概要

■各要素の取り扱い

切り分けられた高位合成非対象ソースコードに関しては、高位合成対象ソースコードを単体で検証する場合は検証記述となるが、システム全体として見たときは高位合成対象機能ブロックに対して制御する機能を有する回路となっている。これらは C 設計と並行して従来の HDL 設計で作成する。

上記により、元となるアルゴリズムは組み込みソフトウェア設計対象ソースコード／従来の HDL 設計対象ソースコード／C 設計(高位合成)対象ソースコードに切り分けられ、設計対象全体は以下の図のように内訳されると考えられる。



図【4-1】. 4　アルゴリズムの設計対象分担

5-1-2-2　設計作業に関する検討

先に述べた分離作業によって抽出された仕様要素を用いて、回路として組上げるのが設計作業である。

分離作業において、使用する高位合成ツールの特徴を踏まえて要求仕様を分解しているため、これを効率良く設計データとして再利用可能とするための記述フォーマットを作成する。

また、高位合成ツールの取り扱いに関するノウハウの開発・手順化も併せて行う。

5-1-2-2-1 設計作業における問題

現在の高位合成ツールが変換対象とするソースコードは、それぞれツール固有の記述スタイルとなっており、例えば入出力端子の宣言ではソースコード中に記述すべき位置やその記述方法が異なっている。信号やメモリの宣言・使用でも同様にツール固有の記述方法となっている。分離作業によって抽出された仕様要素や設計対象ソースコードを合成可能なソースコードにする際に、各高位合成ツールに適合する記述スタイルを取扱説明書やマニュアルなどで調べる必要があり、生産性向上の妨げとなっている。

5-1-2-2-2 記述フォーマットと手順化

上記問題を解決するために、合成可能なソースコード作成を支援するための高位合成ツール毎のフォーマットおよび記述方法の説明書を開発する。

フォーマットとなるソースコードについては、入出力ポートや分解された機能記述を挿入する場所をコメント等で明示する。また、各高位合成ツールで固定的に記述される定型文を予め記載する。これにより、合成可能なソースコードを作成する際の筋立てが明確になり、生産性が向上すると考えられる。

- 記述方法の説明書においては、上記フォーマットを利用した上で不明点のみを調べることが可能となり、フォーマットと併せて使用することにより、さらなる生産性向上が期待できる。また、この他にもノウハウとして高位合成ツール自体の使用方法や実行手順をまとめた手順書も作成する。

5-1-2-3　検証作業に関する検討

5-1-2-3-1 抽象度の問題

本研究事業では、C 言語ベースの新しい開発手法を導入し、主として C/C++、SystemC 言語で記述された検証モデル・設計対象を用いる。すなわちこれらのモデル等を用いて検証を行う必要がある。

たとえば SystemC では時間の概念を持たない UTF(Untimed Function Model)や、データ通信が抽象化された TLM(Transaction Level Modeling)などの多様な抽象度を取り扱わなければならない。C/C++においても UTF 相当の抽象度で記述された検証モデルを使用することが予想される。

5-1-2-3-2 言語の問題

上記に挙げた C/C++、SystemC の各言語だけでなく、従来開発手法で用いていた HDL(Hardware Description Language)で記述された既存資産を流用することで、検証モデルの省力化など、生産性を高める効果が期待できる。

上記 2 つの問題を鑑みて、異なる抽象度階層/言語間であっても接続可能な検証環境を構築する。また、HDL との記述の親和性、C/C++との流用などを鑑みて、検証環境は SystemC で構築する。

5-1-2-3-3 検証環境のビルドに関する問題

SystemC で構築された検証環境は、実行前にビルド作業を行う必要がある。検証対象、検証モデルを含む検証の規模が大きくなると、ビルド自体にかかる時間が無視できないほど大きいものとなる。検証環境自体は 1 度だけビルドを行い、検証項目に相当する部分だけ差し替える構成とする。この構成により、ビルド時間を抑えることができ、生産性の向上に繋げることができる。

5-1-3　成果物

他テーマを含む本研究事業全体での、本テーマに関連する成果物一覧を以下に示す。各成果物の内容については後述する。

表【4-1】. 2 本テーマ/本研究事業で開発した手順書一覧

手順書名	開発元	カテゴリ	概要
設計/検証 分離手順マニュアル	本テーマ	分離	アルゴリズムレベル C ソースコードの設計箇所と検証箇所の分離方法
高位合成ツール C 使用基準	本テーマ	設計	高位合成ツール C による C 言語ソースコードの動作合成の簡単な操作概要
高位合成ツール C 合成手順書	本テーマ	設計	高位合成ツール C による動作合成を行うための実行手順
高位合成ツール C ANSI C ソース作成基準	本テーマ	設計	C ソースコードから高位合成ツール C を使用して動作合成を行うための注意点・記述制限
SystemC 検証手順書	本テーマ	検証	SystemC の検証環境作成手順及び実行手順
高位合成ツール A 設計手順書	他テーマ	設計	高位合成ツール A で動作合成するための設計手順
高位合成ツール A 文法ガイド	他テーマ	設計	高位合成ツール A で動作合成するための文法ガイド
高位合成ツール B 文法ガイド	他テーマ	設計	高位合成ツール B 用の C 言語を記述するためのコーディングガイド
高位合成ツール B 使用手順書	他テーマ	設計	高位合成ツール B を使用して動作合成するための手順
高位合成ツール B 設計手順書	他テーマ	設計	高位合成ツール B を使用して、C 言語ベース設計を行う際の手順

表【4-1】. 3 本テーマで開発した記述フォーマット一覧

記述フォーマット	カテゴリ	概要
高位合成ツール A 動作合成用 C	分離/設計	高位合成ツール A 動作合成用 C の記述フォーマット
高位合成ツール B 動作合成用 C	分離/設計	高位合成ツール B 動作合成用 C の記述フォーマット
高位合成ツール C 動作合成用 C	分離/設計	高位合成ツール C 動作合成用 C の記述フォーマット
SystemC 検証環境	検証	SystemC 検証環境の記述フォーマット

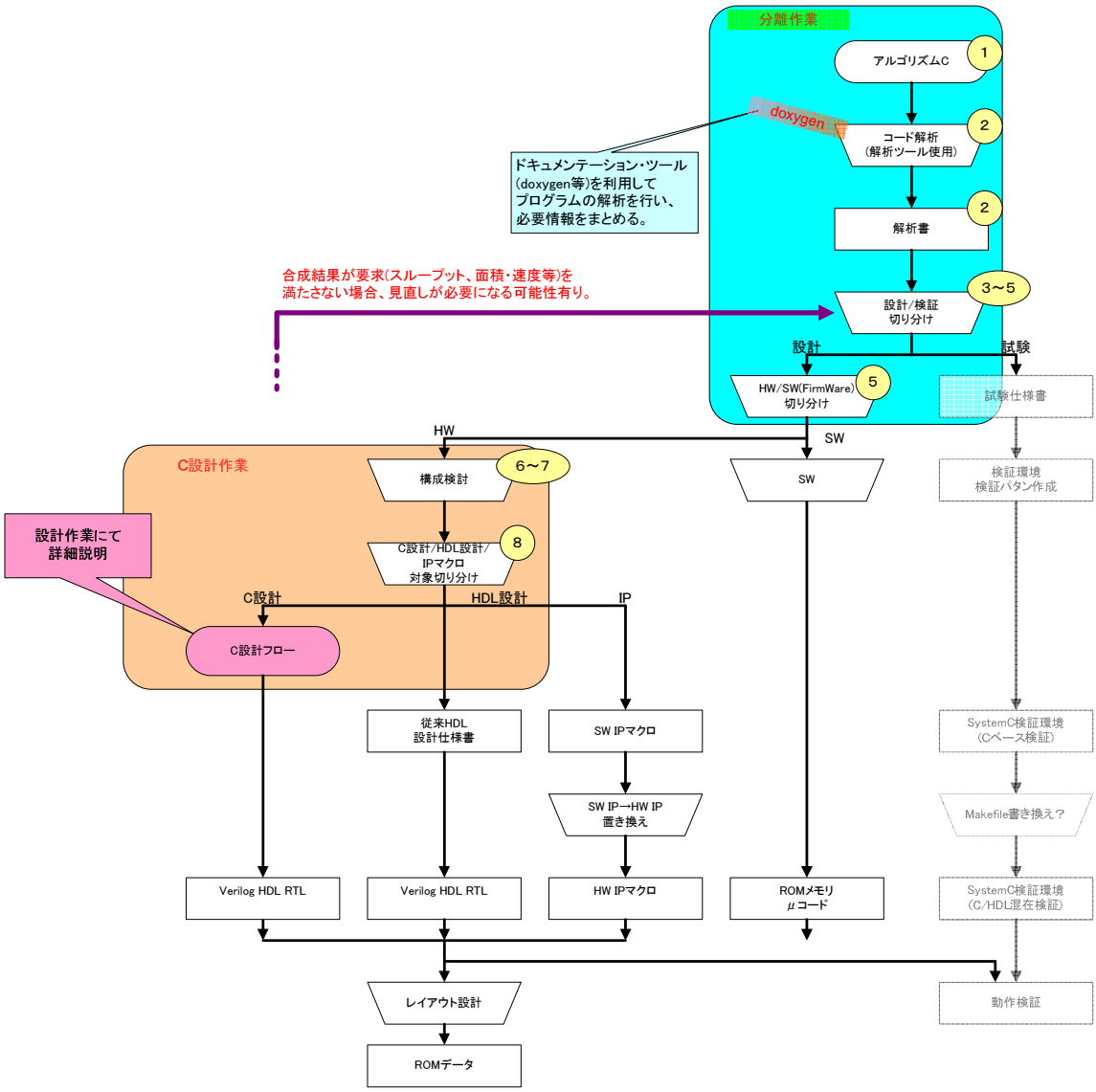
5-1-3-1　分離作業に関する成果物

5-1-3-1-1 分離作業の詳細

高位合成対象(設計記述)と高位合成非対象(検証記述)とを分離する際は、システム全体として所定の仕様・性能(スループット等)を満たす回路構成や、検証(デバッグ)容易性を考慮する必要がある。[5-1-2-1-1 分離・設計作業で考慮すべき高位合成ツールの特徴]でも述べたように、分離作業と設計作業は高位合成ツールの性能や特徴・特性による影響が大きい。特にパイプライン回路と関数間インターフェースについては、これらの特徴・特性をより考慮しなければならない。

本テーマでは、テーマ【2-2】(画像圧縮用途向けのソフトウェア開発手法として、アルゴリズムレベルから実装レベルのソフトウェアへの開発工程の詳細な検討)と共同して、上記を踏まえた高位合成フローおよびノウハウを研究した。テーマ【2-2】では、実際にこのフローとノウハウを使用して JPEG-XR 設計を行った。

下記に画像圧縮アルゴリズム C ソースから高位合成後 HDL ソースを得るまでの概略フローを示す。



図【4-1】. 5　高位合成概略フロー

C 設計対象部分の切り出しまでの過程としては、以下のようになる。(図中番号と対応)

1) アルゴリズム C ソースコード(要求仕様)を理解する。
2) アルゴリズム C ソースコードの解析を行い、クラス/配列/変数一覧等をまとめる。
3) 1 処理単位のサイズを把握する。(数画素,MB,ライン等)
4) データフローの解析および必要なメモリサイズを推定する。
5) ハードウェア化のためにアルゴリズムを組み替える。(メモリ量/回路面積の抑制)
6) バッファ(ラインメモリ)構成の検討。
7) 関数間インターフェース、全体構成の検討。
8) C 設計対象部分とマクロ IP 使用部分とに切り分ける。

[5-1-2-1 分離作業に関する検討]にて、分離作業および設計作業は相互の関連性が非常に強いと述べたが、これは双方が互いの作業を意識しつつの連続した作業であることを意味しており、この高位合成フローの 5)の時点で、検証記述と設計記述(ハードウェア用アルゴリズム C 記述)は必然的に切り分けされる。

この高位合成フローに沿って、C 言語で記述された 2 種類のアルゴリズムレベルのソースコード(JPEG-XR, RLE Bitmap)について分離・設計作業を行った。その結果、どちらでも正しく機能する高位合成回路を得ることができ、この高位合成フローの妥当性が確認できた。

分離作業としては 1)～5)までの工程として捉え、その詳細な手順・ノウハウを資料にまとめた。

以上の検討結果を踏まえ、以下の手順書を作成した。

設計検証分離手順マニュアル

- 要求仕様から設計と検証を分離する詳細を説明。
- 手順書に沿って 2 つの実例あり。

5-1-3-1-2 その他の特徴

合成ツール別 関数間インターフェース

- 回路構造や合成対象モジュール切り分けの検討をサポートし、開発工数を削減することを目的としたノウハウ資料を作成。
- 3 種類の高位合成ツールにて、合成可能かつ実設計に有効な関数間(モジュール間)のインターフェースの記述/使用方法を説明。

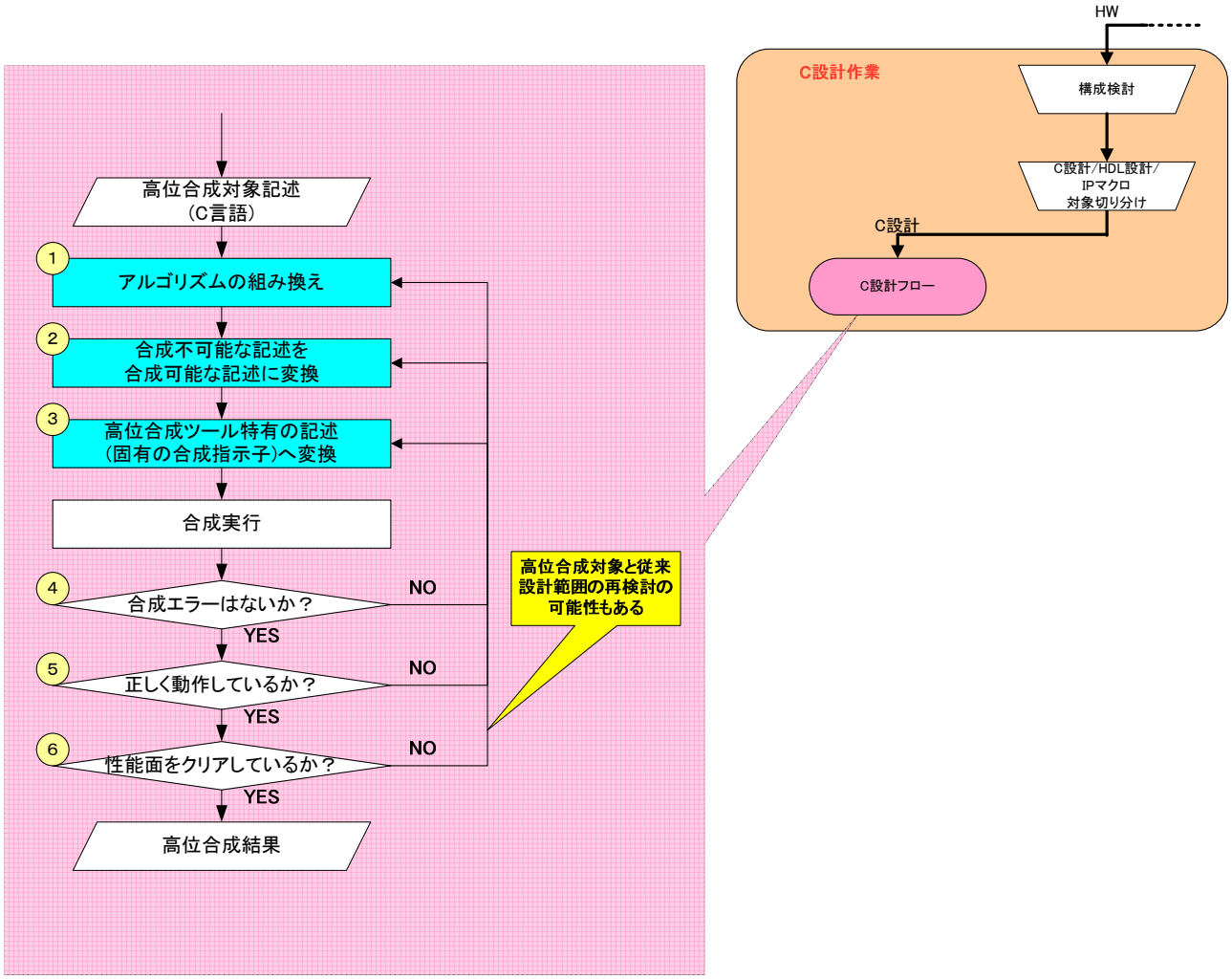
5-1-3-1-3 検証フローに関する特記事項

[5-1-3-1-1 分離作業の詳細]の高位合成概略フロー図では分離・設計作業に着目しているため、検証フローは簡略化したものになっている。本来検証とは、要求仕様を満たす回路になっているかを確認する作業であり、高位合成ツールを利用した回路設計においては、アルゴリズム C ソースコードと高位合成回路との等価検証にて同一機能であることを確認できれば良い。ただし、アルゴリズム C ソースコードから記述の分解や変更等が入る場合、都度その変更前後で等価検証を行うことが望ましい。これにより、ソースコード変更時のバグ混入を早期に発見し、全体検証にて高位合成対象モジュールのデバッグ工数を抑制することができ、生産性が向上する可能性がある。

5-1-3-2　設計作業に関する成果物

5-1-3-2-1 設計作業の詳細

[5-1-3-1-1 分離作業の詳細]で説明した高位合成概略フロー図にて、赤色で示した設計作業部分の C 設計フローの詳細を以下に示す。



図【4-1】. 6　C 設計詳細フロー

C 設計(高位合成)対象記述が切り出せた後、まずは主に以下 3 点に関して記述変換を行う必要がある。

1)　アルゴリズムの組み換え

2)　合成不可能な記述の書き換え

3)　高位合成ツール用の合成指示子挿入(高位合成ツール毎に書き方が異なる)

1)については、前項で説明した分離作業工程と重複する項目にもなる。ハードウェア化を行う前提であればバッファメモリ量の削減は検討必須事項であり、何度もフレームメモリなどに出し入れしながら処理をしているようであれば、必要最低限のデータをフレームメモリよりバッファ(途中格納)し、各関数が結果を引き渡していくように変更していく必要がある。

2)については、各高位合成ツールの制約に依存する。ポインタやループ記述など、よりソフトウェア的な記述

は合成できない場合があるため、これらを合成可能な構文のみを用いて等価な記述に置き換える必要がある。

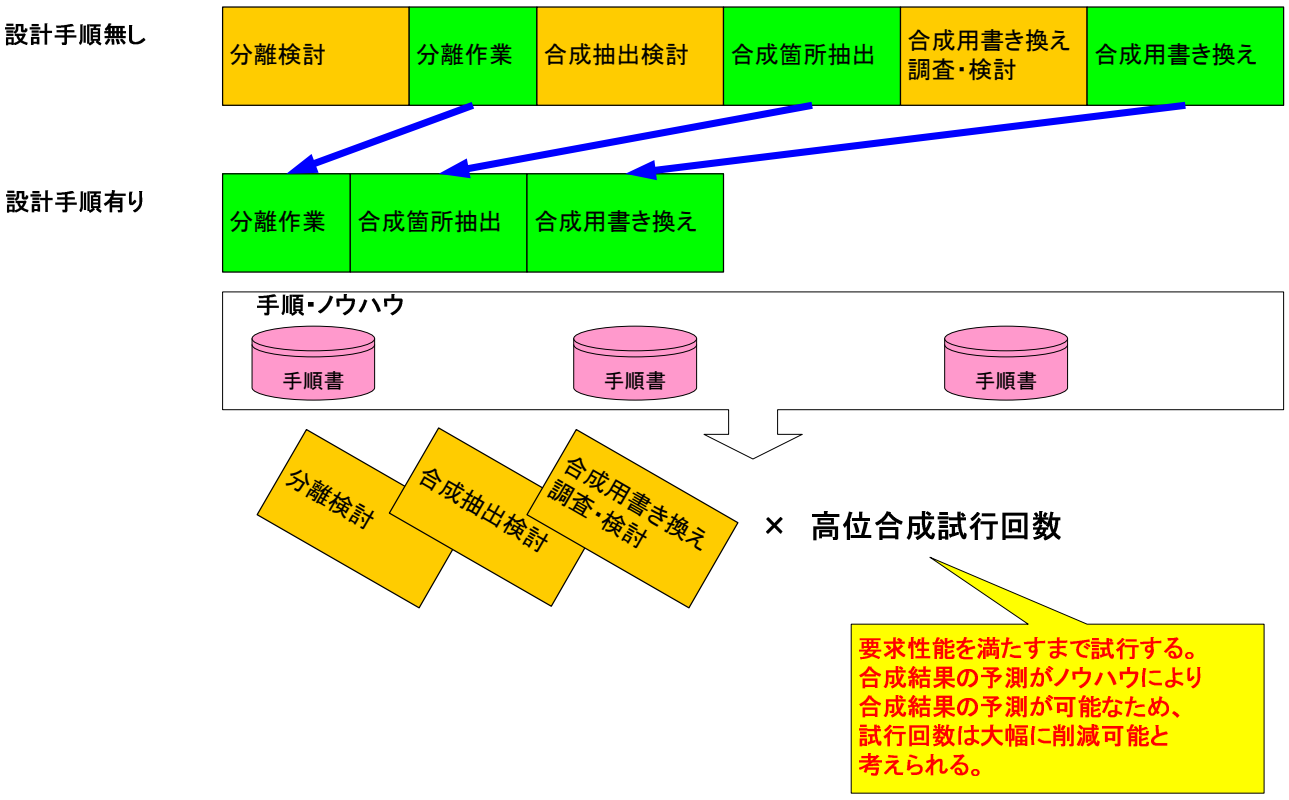
3)については、2)と同様に各ツール固有の機能であり、ディレクティブや指示子と呼ばれる記述を用いて高位合成ツールに対して補助的な情報を明示する。例えば、入出力ポート記述であることをツールに明示するために指示子を挿入する。

上記 1)から 3)の記述変更が完了後に高位合成を行い、正常に合成が完了すれば検証作業へと移行する。

ただし、この記述変更には高位合成ツールの特性を把握して、各ツールにあわせた設計手順や記述方法、ノウハウが必要であり、最も工数がかかる工程であると考えられる。実際、JPEG-XR 設計においては、ツール調査工数は設計時間において大きな比率となっている。

また、実際の作業においては、図内 4)～6)の判定箇所にもあるように、合成作業自体でのエラー発生や、動作検証においても所望の動作を得られないケースが数多く見受けられる。特に高位合成結果の回路性能(スループット等)に関しては、要求性能を満たすことができなければ、その要求性能に達するまで何度も「回路検討→高位合成→動作検査」を繰り返す必要がある。

そこで本テーマでは、各高位合成ツールにあわせた設計手順、記述方法や記述フォーマットを研究して、それをノウハウ集(手順書)としてまとめた。これにより、上記記述方法の調査工数は大幅に削減され、合成後の回路動作(関数間インターフェース等)が予測可能となるため合成試行回数も減少し、生産性向上が可能になると考えられる。



図【4-1】. 7　生産性向上イメージ

以上の検討結果を踏まえ、以下の手順書および C 言語記述ひな形を作成した。

高位合成ツール合成手順書

- 高位合成ツールによる動作合成を行うための実行手順を説明するために作成。
- 動作合成フローチャート、アルゴリズム C ソースコードの変更手順、高位合成手順について説明。

高位合成ツール C ソース作成基準

- 高位合成ツールの合成可能な記述を説明するために作成。
- 合成用記述の作成方法、合成指示子/予約関数の種類について説明。

高位合成ツール使用基準

- 高位合成ツールの操作方法を説明するために作成。
- デザイン作成フロー、基本/応用操作方法、その他インストール方法等について説明。

C 言語記述ひな形

- コメント等で合成可能なソースコードを作成する際の道筋を示しており、フォーマットに記載されたそれらの指示に従って記述を進めることによって理想的に回路実装することが可能となる。また、最低限必要なツール固有の定型文を予め記載している。

5-1-3-2-2 その他の特徴

ドキュメント作成の省力化

- ドキュメンテーション・ツール doxygen を利用し、設計データから設計ドキュメントを自動生成する。これによってドキュメント作成工数を圧縮する効果が期待できる。
- doxygen はコメント記述の内容を元にドキュメントを出力するため、C 言語ソースコードにコメント記述として実装内容と説明コメントを挿入しておけば、それを整理してドキュメントとして出力することができる。これを設計仕様書として流用する。
- また、設計と検証の分離作業時の解析サポートにも使用可能。

5-1-3-3　検証作業に関する成果物

5-1-3-3-1 検証環境ひな形の作成

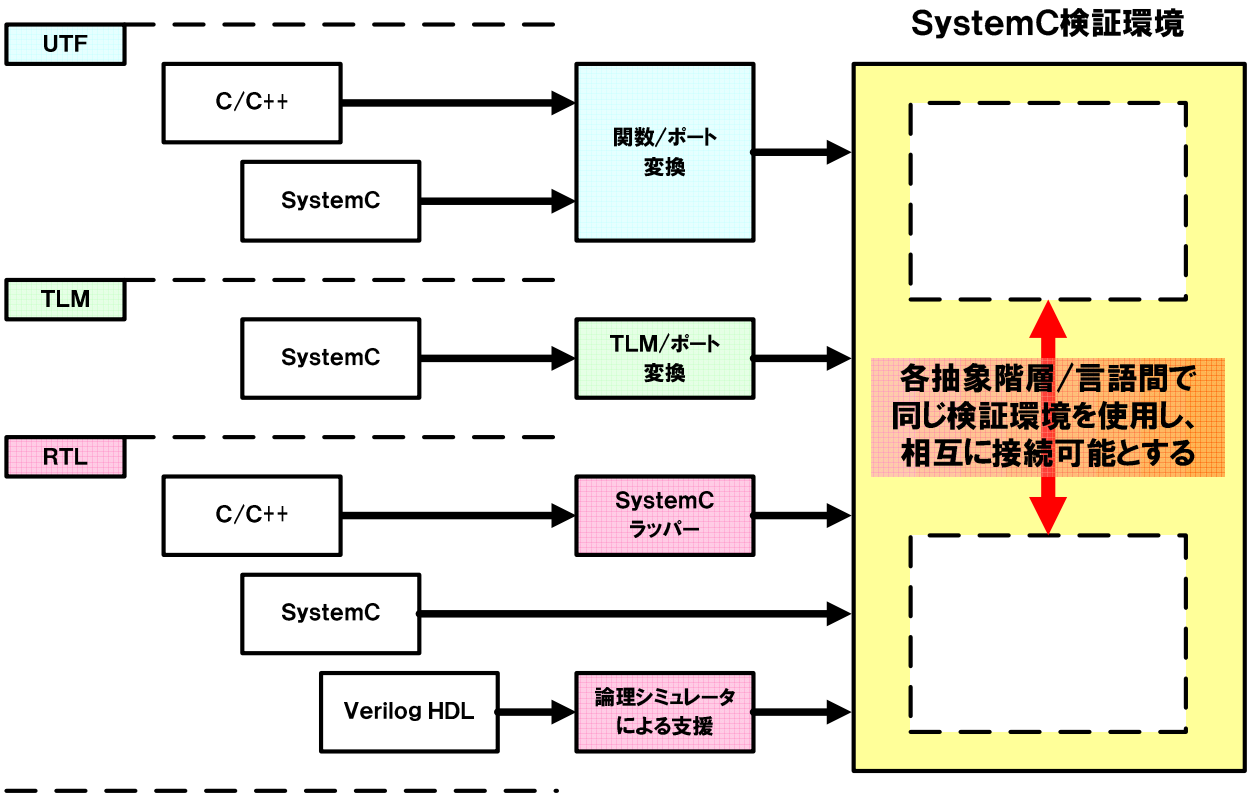
先に挙げた問題点に対して、下記に挙げる項目を含む例として JPEG-XR コーデックの一部機能について検証環境を作成し、この検証環境をベースとして検証環境のひな形を作成した。ひな形はテーマ【1-1】で生成した統合置換ソフトウェアのライブラリの一つとして機能し、必要に応じて展開・利用することができるようにした。

5-1-3-3-2 異なる抽象度階層・言語間の接続方法の確立

異なる抽象度で記述されたモデルを取り扱うために、それぞれの抽象度を検証環境に接続するためのラッパーモジュールを作成した。

現在用意されている SystemC 単体では、C/C++、SystemC 以外の言語で記述されたモデルを取り扱うことができない。このため、従来手法での検証に用いている論理シミュレータの機能を使用し、SystemC 検証環境に Verilog HDL で記述されたモデルを接続する方式を採用した。検証環境の作成と併せて、SystemC 検証環境に HDL 記述を組み込むために必要となる手法を手順化した。

これらを組み合わせることで、異なる抽象度階層・言語間で記述されたモデルであっても同一の検証環境に接続することが可能となる。



図【4-1】. 8　異なる言語/抽象階層接続概念図

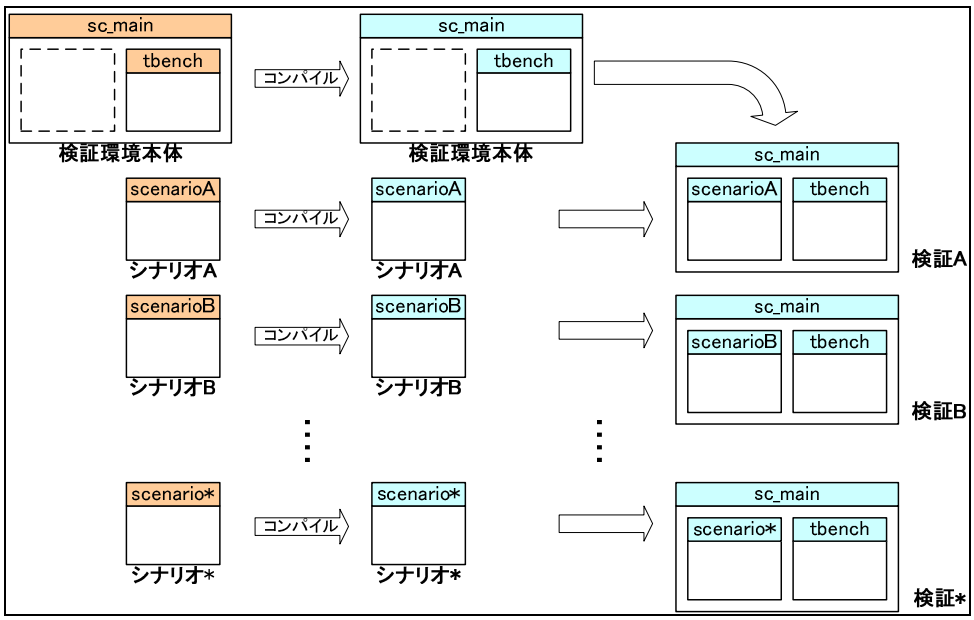
5-1-3-3-3 検証シナリオの分離

検証環境のビルド時間短縮のため、検証環境を環境本体と検証シナリオとに分離した。検証環境本体は実行ファイルとしてビルドし、検証の実行はこの実行ファイルを通じて行われる。検証シナリオは共有ライブラリとしてビルドし、検証環境の実行時に動的にロードして使用される。

これにより、検証環境本体のビルドこそ時間がかかるが、環境構築後に一度だけビルドしておけばよくなり、以降は検証シナリオのビルドだけで検証を実施することが可能となる。

表【4-1】. 4 検証環境の構成

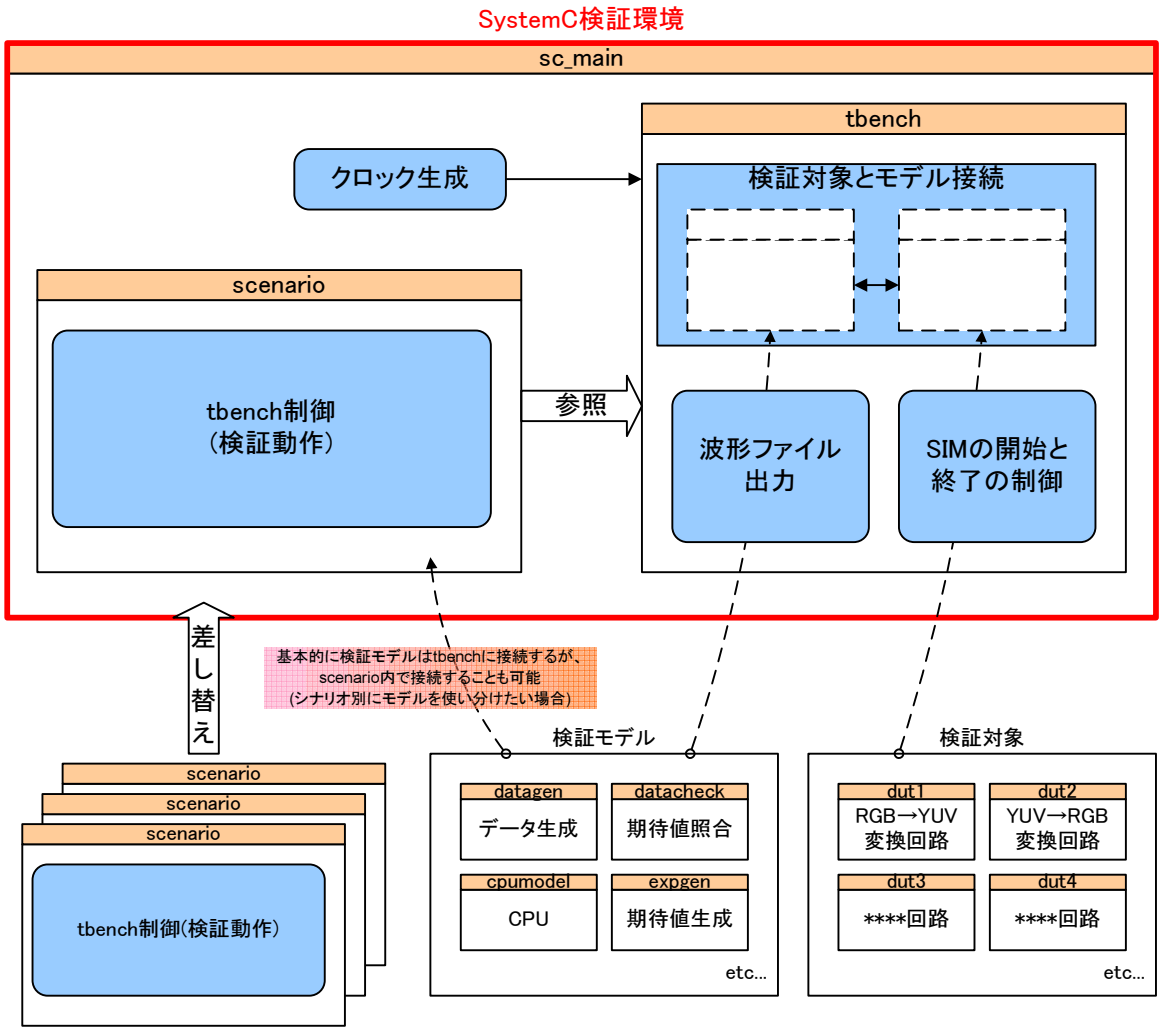
項目	形式, ファイル名	構成要素	機能
検証環境本体	実行ファイル simv	検証トップ	SystemC シミュレータのエントリポイント （sc_main）
		検証接続	検証対象、検証モデル間の接続
		検証モデル	検証対象に入力するデータの生成 検証対象が出力したデータのチェック
		検証対象	----
検証シナリオ	共有ライブラリ libscenario.so	検証項目	検証モデルの制御 検証終了条件の明示



図【4-1】. 9 検証環境構成

5-1-3-3-4 検証環境の構成

以上の点を踏まえ、本研究事業のテーマである JPEG-XR コーデックの一部機能を抜粋した検証環境を以下のように作成した。



図【4-1】. 10　検証環境構成図

5-1-3-3-5 手順書、ひな形の作成

以上の検討結果を踏まえ、以下の手順書とひな形を作成した。

SystemC 検証手順書.doc

- SystemC の検証環境作成手順及び実行手順を説明するために作成。
- 検証環境の全体構成、実行方法、他言語の組み込み方法等の応用例等について説明。

SystemC 検証環境ひな形

- SystemC 検証環境を作成する際の道筋を示すものであり、手順書の指示に従って記述を進めることによって検証環境の構築が可能である。

5-1-3-4　生産性計測

ソフトウェアから LSI への設計の作業工数について、画像・動画圧縮用 JPEG-XR コーデックの開発での実測値から、本テーマで開発した手順書及びひな形を用いることによる生産性向上への効果を考察した。

使用した高位合成ツール別に、設計と検証環境構築の 2 点について作業工数の実測値と予測値を比較した結果を下記に示す。予測値は以下に挙げる条件で算出した。

- 作成手順書及びひな形があらかじめ用意されていることによる、作成工数の削減。
- 設計ノウハウを加えた手順化による、手戻り工数の削減。

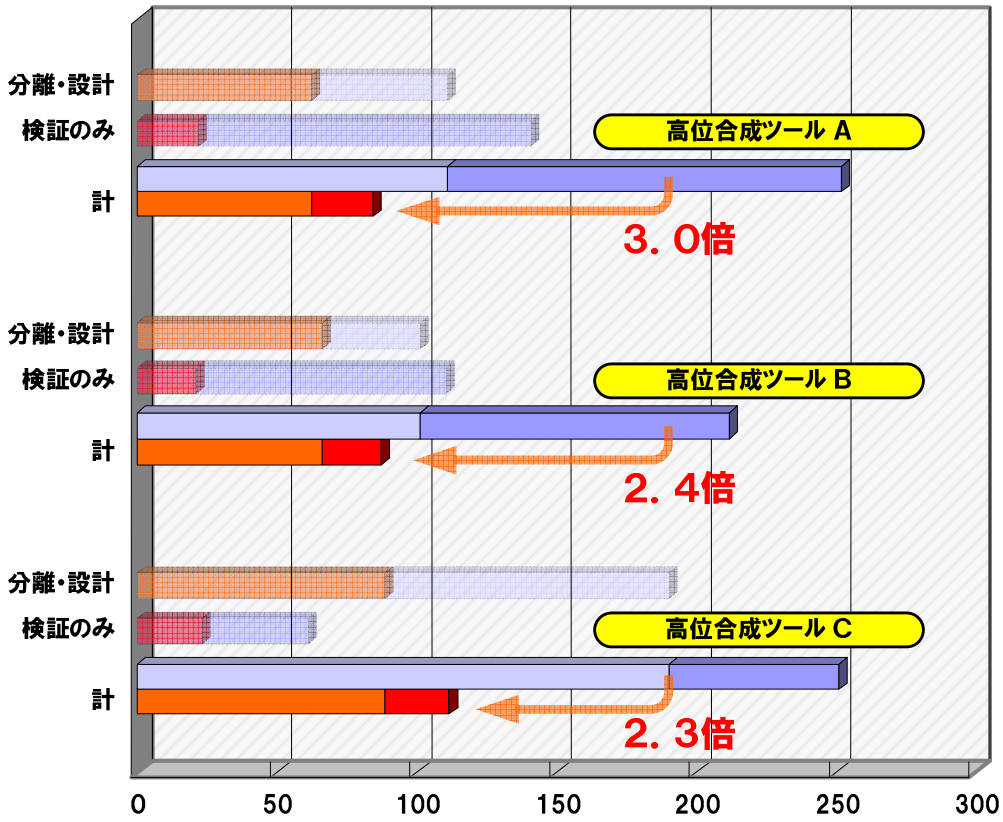
また、記述フォーマット利用による記述量比較に関しては、記述フォーマットが以下のような特徴を持つため、作業工数のみでの比較とした。

- 記述フォーマット作成の主たる目的が記述方法を誘導することであり、ノウハウ的な要素が強い
- 記述フォーマット自体の記述量が全体記述量に対して少なく、生産性への影響が少ない

表【4-1】. 5　作業工数比較

ツール	区分	手順化なし(h)	手順化あり (h)	削減工数(h)	削減率(向上度)
高位合成ツール A					
	設計(分離と設計)	111.00	62.50	48.50	44% (x1.8)
	検証環境構築	141.00	21.75	119.25	85% (x6.5)
	合計	252.00	84.25	167.75	67% (x3.0)
高位合成ツール B					
	設計(分離と設計)	101.25	66.25	35.00	35% (x1.5)
	検証環境構築	110.50	21.00	89.50	81% (x5.3)
	合計	211.75	87.25	124.5	59% (x2.4)
高位合成ツール C					
	設計(分離と設計)	190.25	88.50	101.75	53% (x2.1)
	検証環境構築	61.00	23.00	38.00	62% (x2.7)
	合計	251.25	111.50	139.75	56% (x2.3)

下記に比較結果のグラフを示す。



図【4-1】. 11　作業工数比較グラフ

5-1-4　まとめ

5-1-4-1　成果

本テーマでは、生産性向上を目的として分離・設計・検証の各作業を手順化し、この手順に沿った記述フォーマットおよび検証環境ひな形を作成した。作成した手順・記述フォーマットを実際に使用してアルゴリズムレベルの C 言語ソースコードを高位合成し、手順の妥当性を確認した。

また、手順の効果として各手順の有無による作業時間の予測を算出し最大で 3.0 倍(67%削減)の生産性向上という予測結果を得た。

5-1-4-2　結論

本テーマで作成した分離・設計・検証の手順を用いることで、生産性向上に効果があることを確認できた。実際の開発業務を用いて効果の計測を行ったため、他の開発業務にも同様の効果があることが期待できる。

なお今回は手順・記述フォーマット等の作成に注力したため、分離手順のソフトウェア化までは踏み込めなかった。今後の課題として、分離・設計・検証の各手順をフレームワークに搭載した上で手順の改善を進めていき、支援ツールとして派生させることでソフトウェア化を実現させたい。

第6章 全体総括

本研究の成果および課題を生かして骨組み部分をフレームワークとして汎用化を行い、ユニット部分を本研究の画像圧縮以外に情報家電、医療機器、産業系、通信系や自動分野系などとして付け替えまたは複合化可能な状態を最終形態として目指すための礎を築くことができた。

今後は本研究を元にC言語ベースIPによって200億円規模の売り上げを目指し、本IP群を使用したLSIチップ開発によって 3000 億円規模の売り上げに応じた地域社会への社会福祉貢献、納税および雇用創出を行う。

第7章 付録 添付資料

- 研究発表・講演、文献、特許等
- 参考文献リスト など

無し