修士論文

各種プロセッサアーキテクチャの設計に基づいた デザインパターンの検討

氏名	:	安倍 厚志
学籍番号	:	6162080004-9
指導教員	:	山崎 勝弘 教授
提出日	:	2011 年 2 月 14 日

立命館大学大学院　理工学研究科　創造理工学専攻

内容梗概

本論文では，コンピュータアーキテクチャを体系的に学習しながら，ソフトウェアとハードウェアのトレードオフをバランスよく理解できることを目的に，ハード/ソフト協調学習システムに対して，プロセッサ用デザインパターンの検討，オリジナルプロセッサの設計，及び評価を行っている．

本研究では，ハード/ソフト協調学習システムを用いた，各種プロセッサアーキテクチャの設計と過去の学習結果による評価について述べる．さらに，過去の設計データをプロセッサの仕様と構造の設計見本として活用することで，設計期間の短縮，他者の使用をイメージした設計の習得，各種プロセッサアーキテクチャ理解の促進，効率的な仕様の策定，及び設計データの再利用が可能なプロセッサ用デザインパターンについて，問題解決案として述べる．

本研究で提案するプロセッサ用デザインパターンは，初めてプロセッサを設計する人，より高度なアーキテクチャを搭載したプロセッサを設計する人それぞれに望まれる，活用しやすい過去の設計データの分割案を検討する．デザインパターンは名前，目的と用途，動機，アーキテクチャ，構成要素，データパス，アニメーションと Verilog-HDL ソースコードの 8 つの要素で構成される．学習者は求めるキーワードで，デザインパターンに検索をかけて，キーワードに類似したパターンがある場合に取り出せる．アーキテクチャ特有のハザードなどを学習しやすいように，個々の命令動作ごとにパターン化し，また，制御信号を含めた命令実行における機能ユニットの状態遷移を学習できるよう検討した．

デザインパターンを用いて，独自のプロセッサ設計を行った．その結果と学生による設計結果から考察を行っている．新たな機能の追加と改善による仕様の考察，それを元にした命令セットの定義に対しては，改善点や効率化のポイントを把握できるため，新たな設計案を考えるのに役立った．また，シングルサイクルアーキテクチャは，制御信号を含めた命令実行の観察ができるため学習効率を向上できた．さらに，パターンや単一機能ユニットの再利用によって，設計期間が短縮できることがわかった．

目次

図目次

表目次

1. はじめに

半導体の高集積化に伴って，LSI の軽量化，高速化，省電力化，高機能化，及び多機能化が可能になった．テレビ，デジタルカメラ，携帯電話，ゲーム機器，医療機器などの組み込み機器は，エンターテイメント，産業，医療分野など多岐に渡って人々の生活を支える重要な役割を担っている．これらの組み込み機器は，より一層の高機能性，多機能性，安全性，快適性，信頼性を維持しながらも，多様化したニーズに合わせた開発を行っていかなければならない．現場の開発者は消費者のニーズを満たすために，より小型化，低消費電力化に努める必要を迫られている．そのような環境の中で，回路規模の増加，用途の拡大による開発工程の複雑化と設計期間の短縮による開発労力の増大などが大きなボトルネックとなっている．開発要求の中でハードウェアとソフトウェアは共に密接な関係があり，両方の知識を持つ人材を育成するためには早期の教育が必要である．こうしたハードウェアとソフトウェア両方の知識を持つ人材の育成を実現するために，大学教育の中でも広島市立大学の City-1[11]、九州工業大学の KITE[12]，慶応義塾大学の PICO²[13]などの教育用マイクロプロセッサを用いたハードウェアとソフトウェアの関係を意識した教育システムが多くの大学で開発されている．

これらの背景から，本研究室でもハード/ソフト協調学習システム（Hardware/Software Co-learning System：HSCS）の開発を行い，学生が学びやすい環境を整えている[14]-[20]. HSCS の特徴は学習者が独自のマイクロプロセッサの仕様の策定と設計をしていく中で，段階的に各種プロセッサアーキテクチャの学習，独自の命令セットの定義，アプリケーションの設計，及びプロセッサ設計技術など，コンピュータアーキテクチャを体系的に学習しながら，ハードウェアとソフトウェアをバランスよく学習できることである．HSCS ではプロセッサ設計をサポートするために，プロセッサ設計支援ツールの開発も行っている．HSCS のソフトウェア側の学習要素はアセンブリプログラミングと命令セットの考案である．また，ハードウェア側の学習要素は Verilog-HDL によるプロセッサ設計，各種プロセッサアーキテクチャの学習と FPGA を用いた動作検証である．

本研究室ではこれまで 4 人の学生が現在のシステムを利用して，シングル，マルチ，パイプラインアーキテクチャの学習，プロセッサを設計，及び FPGA 上での検証を行い，システムの評価を行った[21]-[24]．その評価を元に現状のシステムの問題点は，プロセッサ設計において，各種プロセッサアーキテクチャの理解，データパスの構成，及びそれらの制御信号の生成が最も困難であり，学習に長い時間を要することである．独自のプロセッサを設計するときに，プロセッサアーキテクチャなどいかに重要な部分に時間を確保することができ，他の要素が短時間で効率よく学習，設計できる必要がある．

従来，ハードウェアの設計データから切り出したものをIP(Intellectual Property)と呼び，ソフトウェアではミドルウェアやライブラリと呼ぶことが多い[7]．デザインパターンとは仕様を再利用するために，構造のノウハウを設計見本の形で表したものである．過去の設計

事例を類型化し，見本化することで設計対象が持つ性質とデザインパターンとの関係性を見つけることができ，関連するデザインパターンを参考にできる．この考えは特別なものではなく，システム構造やコードの再利用は開発者が意識的に行っている．しかし，有用なデータがあっても一元化されておらず，各所に点在していては利用効率を上げることはできずにかえって探し出す時間を要してしまう．

本研究ではこの問題に対して，ハードウェア開発で用いられるデザインパターン[7]とソフトウェア開発に用いられるデザインパターン[8]の概念を，プロセッサ設計に応用することで柔軟なプロセッサ設計能力が効率よく習得できることを目的としている．プロセッサ設計におけるデザインパターンは，過去の設計データを集めて分類を行い，プロセッサアーキテクチャの理解，命令セットの分析と新たな設計，パターン化されたIPを用いた設計期間の短縮，及び過去のプロセッサとの性能比較・評価を助けることで，学習者のプロセッサ設計における負担を軽減することが可能である．

本研究では，HSCSを使用した過去のプロセッサ設計データから，デザインパターンとしてプロセッサアーキテクチャ毎に分類を行い，シングル，マルチ，パイプラインアーキテクチャのデザインパターンを作成した．それらのデザインパターンを用いて，4回生にオリジナルプロセッサを設計してもらい，ハードウェアとソフトウェア学習時間についてHSCSと比較して，全体の評価と考察を行う．

本論文では，第2章において，これまで設計したシングル、マルチ、パイプラインプロセッサについて設計思想，命令形式とデータパスを示す．第3章ではプロセッサ用デザインパターンの説明，システムの構成とデザインパターンの例を示す．第4章ではHSCSを用いて過去に設計された各種アーキテクチャに基づくプロセッサの説明とデザインパターンへの分類を行う．第5章ではデザインパターンを用いた新規プロセッサの設計と4回生による活用結果を示し，デザインパターンの考察を行う．

2. 各種アーキテクチャによるプロセッサ設計

2.1 HSCS の概要

HSCS の学習フローを図 1 に示す．HSCS とは，学習者が独自のマイクロプロセッサの設計をしていく中で，コンピュータアーキテクチャを体系的に学習しながらハードウェアとソフトウェア両面のトレードオフを理解していくことを目的とした教育システムである．本システムは，プロセッサアーキテクチャの学習，命令セットの定義とアプリケーションの設計を行うソフトウェア学習，プロセッサ設計，設計したプロセッサを FPGA ボード上で検証するハードウェア学習，及びそれらの学習をサポートするプロセッサ設計支援ツールから構成される．HSCS で使用する命令セット MONI の命令形式を図 2 に示す．MONI は MIPS のサブセットであり，以下の特徴をもつ．

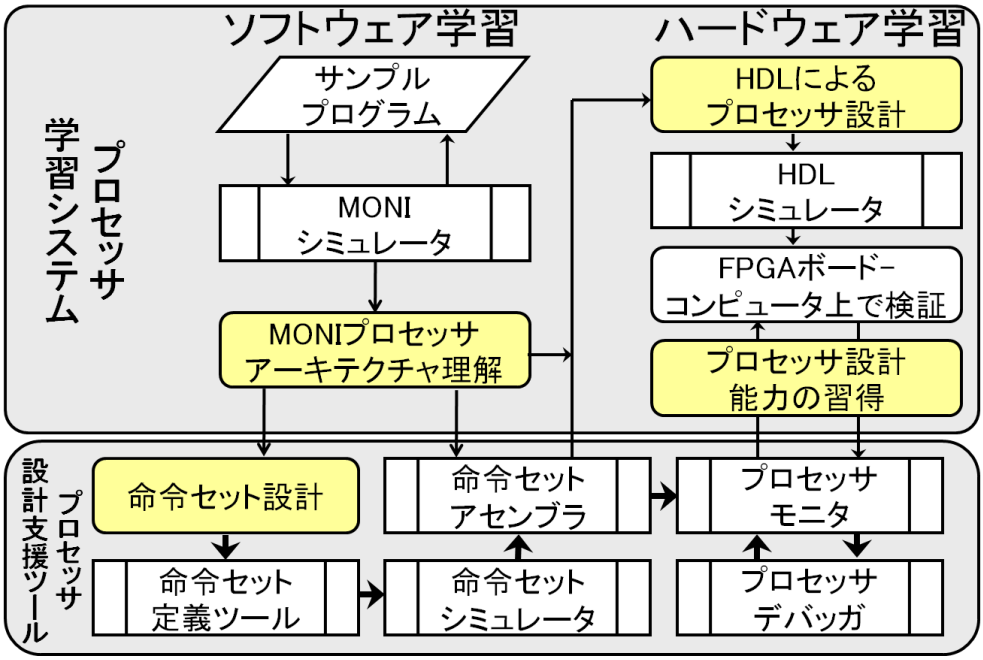

図 1：ハード/ソフト協調学習システム

(1) 命令語長は 16bit 固定
(2) 3 オペランド命令方式
(3) 全 39 命令
(4) 4 命令形式

ビット長 / 命令形式	5	3	3	3	2
Register	Opecode	Rs	Rt	Rd	Function
Immediate5	Opecode	Rs	Rt	Immediate	
Immediate8	Opecode	Rs	Immediate/Address		
Jump	Opecode	Target absolute address			

図 2：MONI 命令形式

(1) ソフトウェア学習

本研究室で定義した MONI を用いて学習を進める．MONI 命令セットを元にアセンブリ言語でアプリケーションの設計を行い，MONI シミュレータ上で各種プロセッサアーキテクチャにおける動作を観察する．MONI シミュレータでは，プログラムカウンタ，レジスタファイル，データメモリ，命令メモリ，ハザード処理などの回路機能にテストデータを与えてクロック，命令ごとにデータの遷移を細かく把握できる[15]．以上からプロセッサアーキテクチャごとの命令実行効率などの長所と短所，アーキテクチャ発展の経緯を理解する

(2) ハードウェア学習

MONI シミュレータでデータ構造を理解して，実際に MONI シングルの Verilog-HDL によるハードウェア設計を行う．また，プロセッサデバッガを用いて，ソフトウェア学習で設計したアプリケーションをプロセッサの動作と協調させて ModelSim 上でデバックする．最後に設計したプロセッサをプロセッサデバッガと接続し，プロセッサモニタを用いてデータを送受信することで FPGA ボードに実装したプロセッサの実機検証，評価と完成を目指す[20].

以上からハードウェアとソフトウェアの理解を深めることが HSCS の目的である．

(3) システムの評価

これまで 4 回生の卒業研究として 4 人の学生が HSCS を利用した[21-24]．それぞれの学生は，一連の学習の後に独自の命令セットを定義してオリジナルプロセッサの設計を行っている．表 1 に HSCS を利用した 4 人の学習時間を示す.

学習時間を見てわかるようにハードウェア学習とソフトウェア学習ではハードウェア学習が大きなウェイトを占めている．ここで設計したプロセッサアーキテクチャに注目するとシングル，マルチが 2 人とシングルのみが 2 人である．現在の主流がパイプラインやスーパースカラアーキテクチャだと考えると十分な教育段階まで達しているとは言えない．また，各々の命令セットを観察すると JINT を除きほとんどの命令セットが MONI の縮小命令セットである．このことは，実際はソフトウェア学習における理解度が不十分である

ため，命令をより効率よく動作できる高度なアーキテクチャや命令セットを採用できなかったと思われる．最終的にプロセッサの性能を上げるためにはハードウェアとソフトウェア両方の観点から設計進める必要があるため，設計と学習のより充実したサポートが求められる．

表 1：HSCS を利用した学習時間　　単位：時間

プロセッサ	MONI		SARIS		PSCSF	JINT
アーキテクチャ	シングル	マルチ	シングル	マルチ	シングル	シングル
ソフトウェア学習	25		18		28	15
ハードウェア学習	183		230		122	110

2.2　シングルサイクルプロセッサ

(1)　設計思想

本研究では HSCS を用いたプロセッサ設計において，設計のどの段階に問題が存在するかオリジナルプロセッサのシングル，マルチ，パイプラインを設計することで確かめる．設計したプロセッサは MONI アーキテクチャを参考にして，浮動小数点演算器，浮動小数点演算命令の追加と MONI アーキテクチャの煩雑な部分を削り，SAIX と名付けた．浮動小数点演算を搭載した理由は，画像処理に興味があり，整数と実数の演算が行える汎用プロセッサの設計を目的としたからである．SAIX プロセッサには以下のような特徴がある．

・命令語長は 16 ビット固定
・3 オペランド命令方式
・全 27 命令
・5 命令形式
・浮動小数点演算命令に対応

図 3 に示すように，SAIX プロセッサには，Register 形式，Floating point 形式，Immediate5 形式，Immediate8 形式，Jump 形式の 5 つの命令形式を用意した．R 形式ではレジスタ間の整数演算を行う命令を定義している. F 形式ではレジスタ間の浮動小数点演算を行う命令を定義している．I5 形式ではレジスタ値と即値 5 ビット間演算を行う命令とメモリ・レジスタへのデータ転送命令を定義している．I8 形式では条件分岐命令を定義している．J 形式では無条件分岐命令の JUMP，無実行命令の NOP，プログラム終了命令となる HALT 命令を定義している．R 形式の下位 2 ビット Fn は Opecode と合わせて命令の種類分けができるため，1 命令で 4 種類の命令を定義できる．また，MONI とは異なり，格納先レジスタ Rd フィールドが固定のため，各フィールドの動作が理解しやすい．

(2)　IEEE754 浮動小数点数規格

実数を用いるには小数点の扱える演算器が必要である．浮動小数点数は固定小数点数では誤差として消えてしまうような大きな数や小さな数の計算に向いている．最も多く使われている規格であり，単精度，倍精度で浮動小数点を表現する形式が挙げられる．
IEEE754 における単精度浮動小数点の形式を図 4 に示す.

浮動小数点 float32 ビットにおいて, float[31]は符号部, float[30:23]が指数部, float[22:0]が仮数部となる．条件として指数部においては 0 と 255 は予約語として例外処理に割り当てられている．取りうる指数の範囲は -126 <= 指数<=127 となる．実際にはゲタばき表現がとられるため 1<=指数<=254 までの値として指数は扱われる．また，丸めこみの表現として 0 捨 1 入の考えが用いられている. 仮数部の最下位ビットを upl(unit in the last place)と名付けており，演算の途中で仮数部 10 ビット以上の数が存在する時は upl と upl より下位のビットを参照することで丸めこみを行う．1/2upl を G ビット(guard bit)，1/4upl を R ビット(round bit)，R ビット以下を S ビット(sticky bit)と呼ぶ．S ビットは R ビット以下のビットで OR 計算を行い，丸めこみの判定に用いる．

(3)　SAIX における浮動小数点数の扱い

SAIX は 16 ビットプロセッサであるので，IEEE754 の形式を 16 ビットのデータ表記へ修正する必要がある．SAIX では符号部 1 ビット，指数部 5 ビット，仮数部 10 ビットとした．この形は，指数部が-14<=指数<=15 の範囲，仮数部が 10 ビットなので 2^{-23}<=精度<=2^{15} まで精度を保ったまま演算が可能である.

SAIX の各命令フィールドの意味を表 2 に，MONI シングル，SAIX シングルのデータパスをそれぞれ図 5，図 6 に示す.

MONI と比較するとマルチプレクサを 1 つ減らせ，また，命令形式を見るときに書き込みレジスタを判断しやすいためプロセッサのデータ，設計構造が把握しやすい.

ビット長 / 命令形式	5	3	3	3	2
R	Opecode	Rd	Rs	Rt	Fn
F	Opecode	Rd	Rs	Rt	Fn
I5	Opecode	Rd	Rs	Immediate	
I8	Opecode	Rd	Immediate/Address		
J	Opecode	Target absolute address			

図 3：SAIX 命令形式

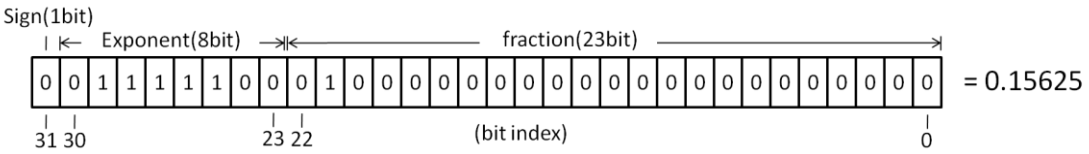

図 4：単精度浮動小数点演算形式

sign:符号部、exponent:指数部、fraction:仮数部

表 2：各命令フィールドの意味

フィールド	意味	bit 幅	用途
Opecode	Operation Code	5	操作コード
Rd	Destination Register	3	格納先レジスタ
Rs	Source Register	3	ソースレジスタ 1
Rt	Source Register	3	ソースレジスタ 2
Imm	Immediate	5-11	即値
Fn	Function	2	機能コード．演算を分類

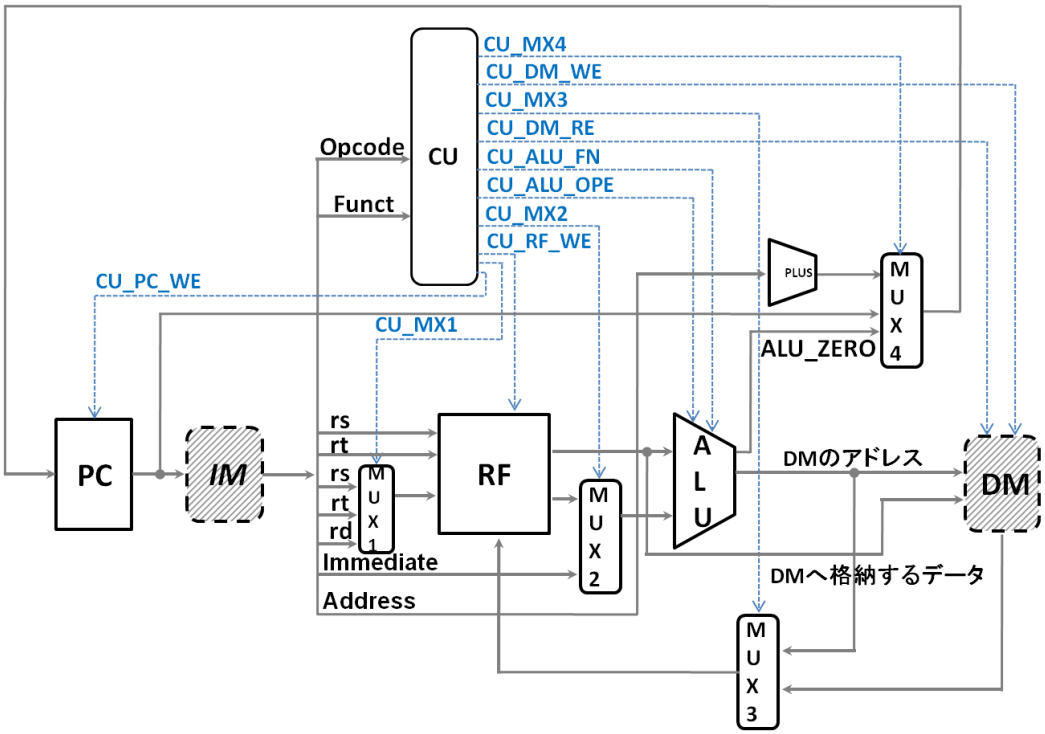

図 5：MONI シングルのデータパス

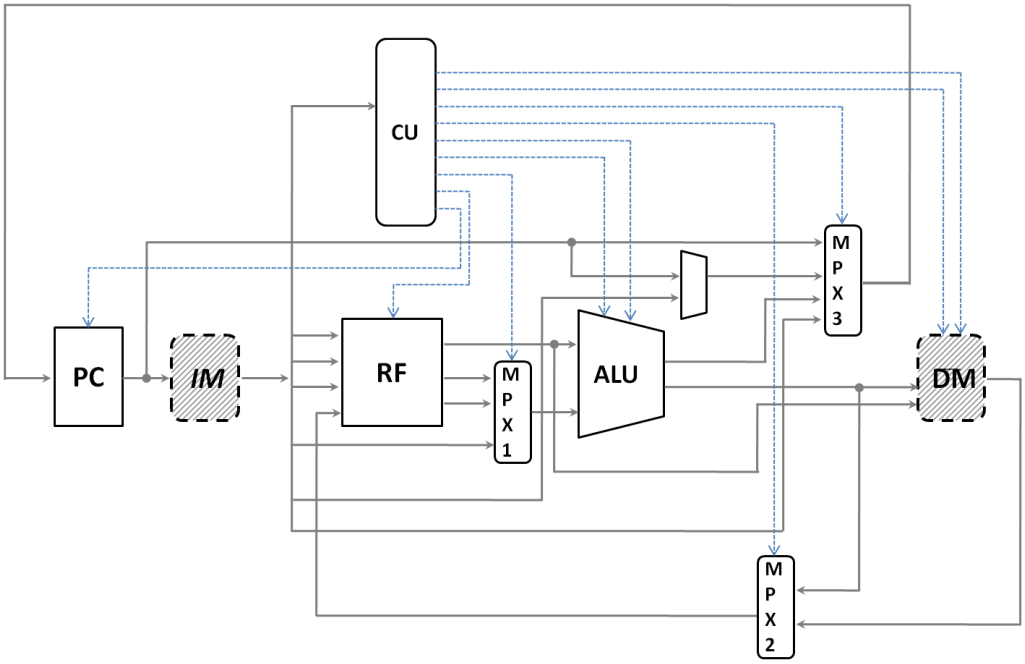

図 6：SAIX シングルのデータパス

2.3 マルチサイクルプロセッサ

2.2 の SAIX アーキテクチャを元に設計した SAIX マルチのデータパスを図 7 に示す．このデータパスは，ヘネパタに掲載されている MIPS の動作ステップを参考にした[1].

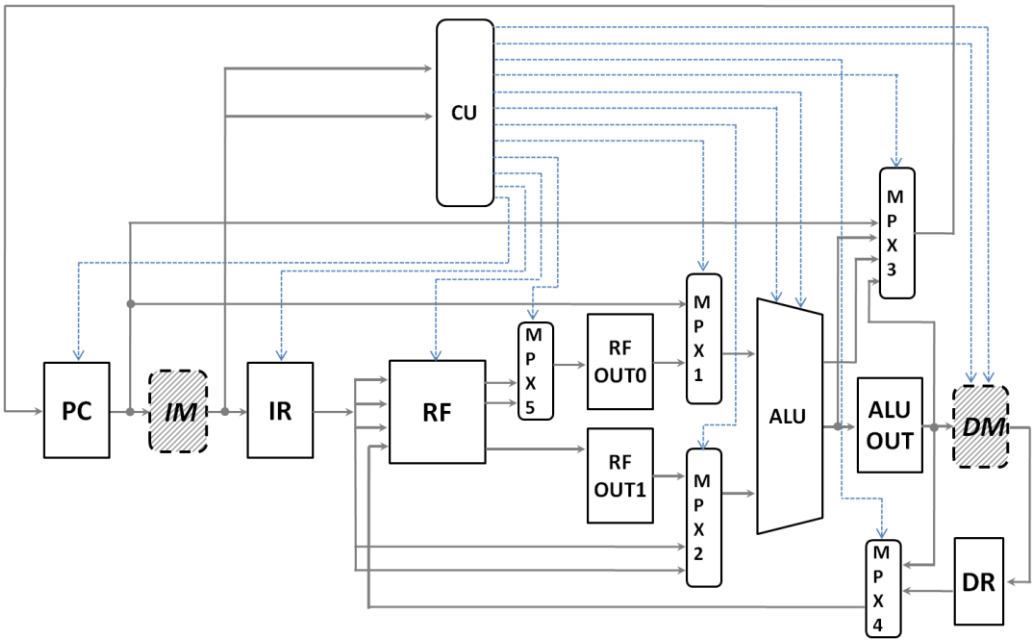

図 7：SAIX マルチのデータパス

(1)　シングルサイクルとの回路構造の違い

① 命令メモリの読み出し結果を保存するIRの追加.

② RFの読み出しデータを，MPX5の結果から保存するRFOUT0，RFOUT1の追加.

③ PCのアドレス加算と分岐アドレス算出をALUで行う.

④ ALUの演算結果を保存するALUOUTの追加.

⑤ LD命令でデータメモリから，読み出した結果を保存するDRの追加.

⑥ CUの入力データを命令メモリの出力データから受け取る．MIPSではIRの出力値を入力データとしている.

(2)　SAIXマルチの各ステップにおける動作

各ステップにおける命令タイプ別の動作を図8に示す. SAIXマルチは1命令を最大5ステップで行う．共通動作として，P0では，PCの更新と命令のフェッチを行い，P1では，命令のデコードとレジスタのフェッチを行う．P2以降は，命令タイプによって動作が異なる．条件分岐とジャンプ形式はP2で，RとI5形式はP3で，LD命令はP4でそれぞれ動作を完了する.

<table>
<tr><td></td><td>I8，J形式</td><td>R，I5形式</td><td>I5形式LD</td></tr>
<tr><td>P0</td><td colspan="3">PCの更新，命令フェッチ</td></tr>
<tr><td>P1</td><td colspan="3">命令デコード，レジスタフェッチ</td></tr>
<tr><td>P2</td><td>分岐，ジャンプの完了</td><td colspan="2">演算の実行</td></tr>
<tr><td>P3</td><td></td><td>RFへ演算結果の書き込み</td><td>メモリアクセス</td></tr>
<tr><td>P4</td><td colspan="2"></td><td>RFへデータの書き込み</td></tr>
</table>

図 8：各ステップにおける命令タイプ別の動作

2.4　パイプラインプロセッサ

2.2のSAIXアーキテクチャを元に設計したSAIXパイプラインのデータパスを図9に示す. パイプライン段数は5段である．分岐命令は分岐予測を行う機能を搭載していないため，分岐の不成立を前提としている．分岐が成立するとEXEで判定した場合は，先行投入されている各レジスタの中身をフラッシュする．以下にパイプラインの各ステージの動作を示す.

(1)　各ステージ動作

①IF：命令フェッチ

②ID：命令デコードとレジスタ・フェッチ

③EXE：命令実行/アドレス生成

④MEM：データ・メモリ・アクセス

⑤WB：データの書き込み

に分割されており，次のステージで使うデータを保存するためにIF/ID，ID/EX，EX/MEM,

MEM/WBレジスタを各ステージ間にはさんでいる. これは, 命令メモリから読み出した個々の命令を実現する値を，残りの4ステージで実行するためには，その値をレジスタに保存しておく必要があるためである．したがって，各ステージ動作を行うために，レジスタを配置することで，1つの回路で複数の命令を共有することが可能になる．

(2)　バイパシングユニット

パイプライン処理には命令を実行する際に，いくつかの問題が生じる．SAIXの演算部は第3ステージであり，パイプライン処理の特性上，MEM，WBステージで保持されている演算結果を用いる場合は，フォワーディングを行って，先行命令の結果を持ってくる必要がある．これがデータハザードであり，問題回避の手段としてバイパシングユニットを搭載している．図9に示すよう，ALUの入力データは2つのマルチプレクサによって命令ごとに選択される．このマルチプレクサに，フォワーディングされた2つのデータを，選択データとして追加する．バイパシングユニットは，演算ソースの依存性を判定して，制御信号をマルチプレクサに流す.

(3)　ハザードユニット

バイパシングユニットを使って解消できないデータハザードが1つある．それは，LD命令直後の命令が，DMから受け取ったデータを，ソースとして用いる場合である．LD命令は，MEMステージでDMにアクセスして, アドレスが指し示すデータを取り出す. このときに，次の命令はEXステージで演算しているため，フォワーディングによる演算ソースの受け渡しを行うことはできない．したがって，この問題を解消するには，LDの次の命令を，1クロックEXステージで待機させればよい．その結果，1クロック後にWBステージのレジスタに保存されたDMの値をフォワードできる．このデータハザードと分岐ハザードを検出するために，ハザード検出ユニットを搭載している．これまでのアーキテクチャでは，CUが回路全体の動作を制御していた．しかし，パイプラインでは，ハザード検出ユニット，CU，及びバイパシングユニットの3つに制御の働きを分散させて，回路を同時に動かしている．

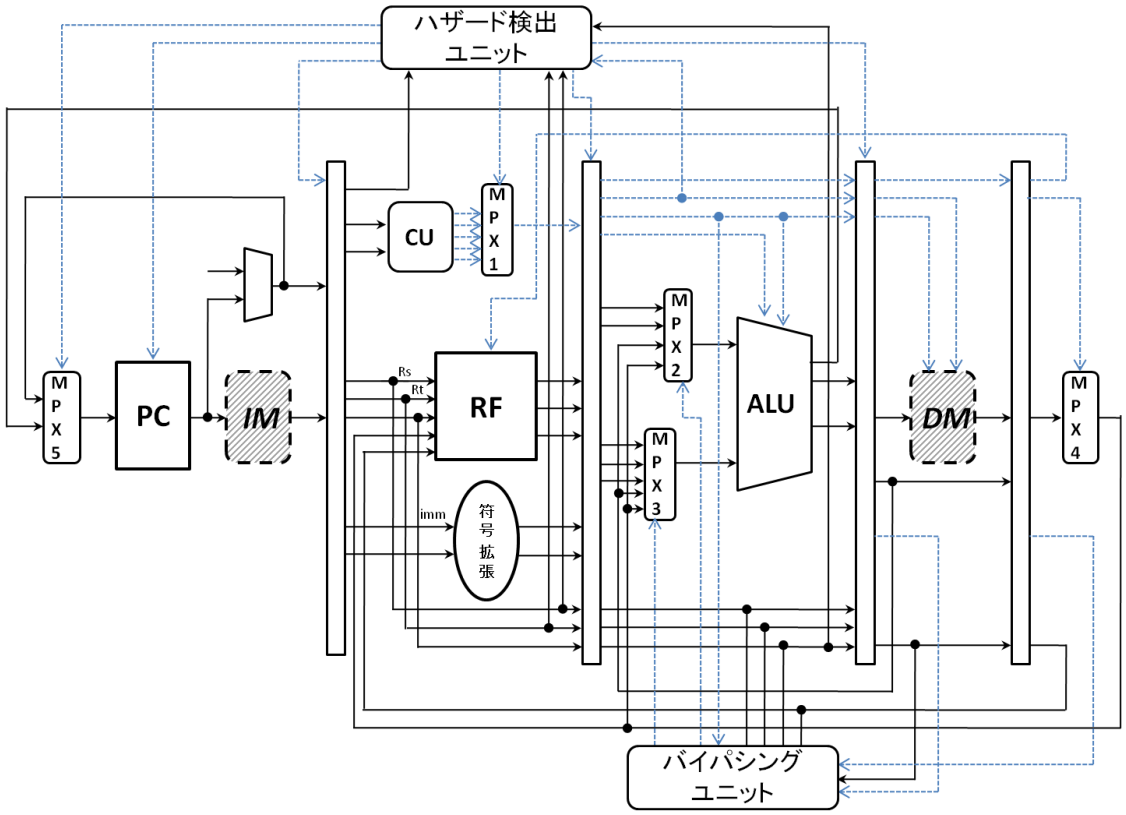

図 9：SAIX パイプラインのデータパス

(4) 設計の評価と考察

本研究では，HSCSを用いて，シングル，マルチ，及びパイプラインプロセッサを設計した．パイプラインではステージ段数を増やすほど，ハザードの回避に必要なコストが大きくなり，また，動作制御が複雑になる．ヘネパタにある「単純性は規則性につながる」ことがどのアーキテクチャの高速化，機能の追加を考えた場合でも，最終的に設計の念頭に置いておかなければならないものだと理解できた．しかし，アーキテクチャが複雑になるほど，単純性を維持することが難しい．

プロセッサ設計において，ソフトウェア学習では，アセンブリ言語を用いたアプリケーションを設計するため，C言語では意識しないレジスタなど，ハードウェア資源の管理を意識できた．シングルアーキテクチャを学習する際に，MONIシミュレータは，ヘネパタで学習するよりも，効率よくプロセッサの全体像を把握するのに役立った．しかし，パイプラインアーキテクチャを学習するための提供要素は，足りないと感じた．これは，いきなりMONIパイプラインのデータパスを示されて，回路構造が説明されないままにハザードの処理とフォワーディングが行われるからだと考える．パイプライン以上になると，回路構造の理解が，直接例外処理の理解に繋がると思われるため，今以上に回路の詳細が必要である．シミュレータの使い方としては，テストする命令とデータを入力することが難しい．これは，デバッグを進めることで解消できる．

3. プロセッサ用デザインパターンの設計

3.1 デザインパターンとは

従来のハードウェア，ソフトウェアデザインパターンは仕様モデル，実装モデル，そして設計結果の一部を取り出して再利用してきた．ハードウェアでは IP(Intellectual Property)，ソフトウェアではミドルウェアやライブラリと呼ばれる部分である．つまり，経験豊富な技術者は問題を抱えたときにゼロから解くのではなく，過去に解いた問題があれば答えを再利用しようと考える．開発者が抱える問題は共通なものが多く，解決方法を他者と共有することで, より柔軟に対応できる．

プロセッサ設計に用いるデザインパターンは，過去のプロセッサ設計データを仕様，実装に限らず，分類した構成要素を用いてプロセッサの学習，設計を補助する．本研究で用いるプロセッサ設計の再利用の形を図 10 に示す．HSCS を用いた学習では仕様の策定，アプリケーション，プロセッサの設計と幅広い領域を学習することになるため，デザインパターンで過去の設計を再利用することで，学生の負担を軽減でき，より高度なプロセッサの設計，学習に時間を充てることができる．

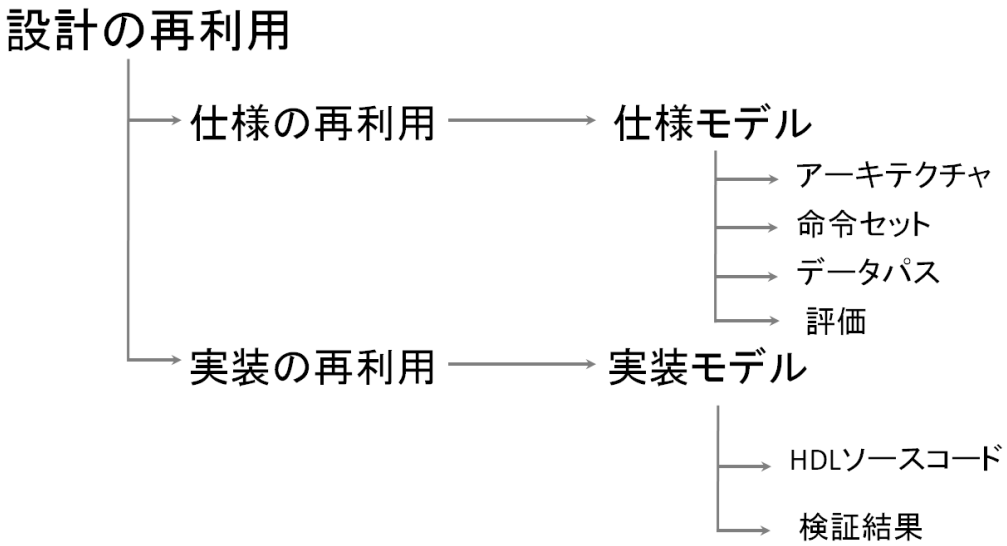

図 10：プロセッサ設計の再利用の形

デザインパターンの設計は次の点に重点を置いた.

(1) HSCS の MONI シミュレータと併用して学習しやすいように制御信号が観察できる要素を提供すること

(2) 設計を再利用するときに 1 つのプロセッサを，個々の命令動作などで分割して提供することによって，設計時間を短縮できること

(3) 過去の設計データを参考にすることで改善点や効率化のポイントが把握できること.

短期間に新たな仕様の策定が行えるように

(4) ソフトウェア指向プロセッサによってコンパイラに意識を持たせて，ハードウェアとソフトウェアのトレードオフの理解を促進すること

以上の点をふまえて，本研究におけるデザインパターンの特徴を以下に示す.

・制御信号を含めたデータパス、アニメーションにおいて，いくつかの命令動作でプロセッサの観察を行い，各種プロセッサアーキテクチャがどのように動作するか，ハザードなどの例外処理の機能ユニットを含めて理解できること
・データパスやプロセッサ，命令セットアーキテクチャを参考にすることで改善点や効率化のポイントが把握できるため短期間で効率的な仕様の策定が行えること
・現在の学習者がデザインパターンへ設計データを登録することで，次の学生へのデザインパターンにできること
・機能ユニット，ブロックのコードを再利用することによって，プログラミング能力の向上，設計期間の短縮ができること
・第三者が自分の設計データを再利用できるようにするため，他者の使用をイメージした設計が身につくこと

デザインパターンは MONI シミュレータと併用して各種プロセッサの動作理解を補助する．制御はプロセッサの性能を大きく左右する部分であり，信号をどのように割り当てればよいか学習する必要がある．MONI シミュレータは学生が MONI アーキテクチャを用いたアプリケーション設計を行い，そのプログラムを MONI シングル，マルチ，パイプラインプロセッサで動作を行いながら観察できる環境を提供している．しかし，アセンブリ言語によるプログラミング，デバッグは大きな負担になる．また，プログラムの動作はループ構造になっている部分が多くあり，各種アーキテクチャによるデータの格納，例外処理など個々の命令の制御と機能ユニットの動作が理解できればよい．そこで，デザインパターンではプロセッサの動作を理解するのに必要な命令を，あらかじめ提供する．そうすることで，命令ごとに制御信号がどのように変化するのか，アニメーションを用いて，最低限の命令動作を把握することで理解できる．

HSCS ではコラーニングを行うことでハードウェアとソフトウェア両方の知識を持つ人材の育成を目標としている．ソフトウェア学習は命令セットの策定とアセンブリプログラミングを行い, ハードウェア学習は各種アーキテクチャ学習と Verilog-HDL によるハードウェア設計を行う．現在ソフトウェア側でコンパイラの学習を行うツールの開発が進められている．プロセッサの性能を高めるにはハードウェア(アーキテクチャ)とソフトウェア(コンパイラ)両方の高機能化が必要である．

仕様の策定では，MONI が MIPS を参考にしたように，過去のプロセッサ設計データを設計見本の形で利用することで，そのプロセッサが持つ長所と短所が理解できる．従って，

設計するプロセッサの特徴と改善点や効率化のポイントを早い段階で把握できるため，同じ時間の中でも仕様の策定を効率的に進めて，より高度なアーキテクチャを採用できるなど内容を充実できる．

3.2　デザインパターンの構成

デザインパターンの構成と，それを用いたプロセッサ設計の手順を図 11 に示す．本研究で提案するデザインパターンは，学習者が MONI やオリジナルプロセッサの新規設計をするときに，ほしい情報のキーワードを入力し，デザインパターンのデータベースから検索する．そして，デザインパターン内にキーワードに類似したパターンが存在すれば学習者へ提供する．

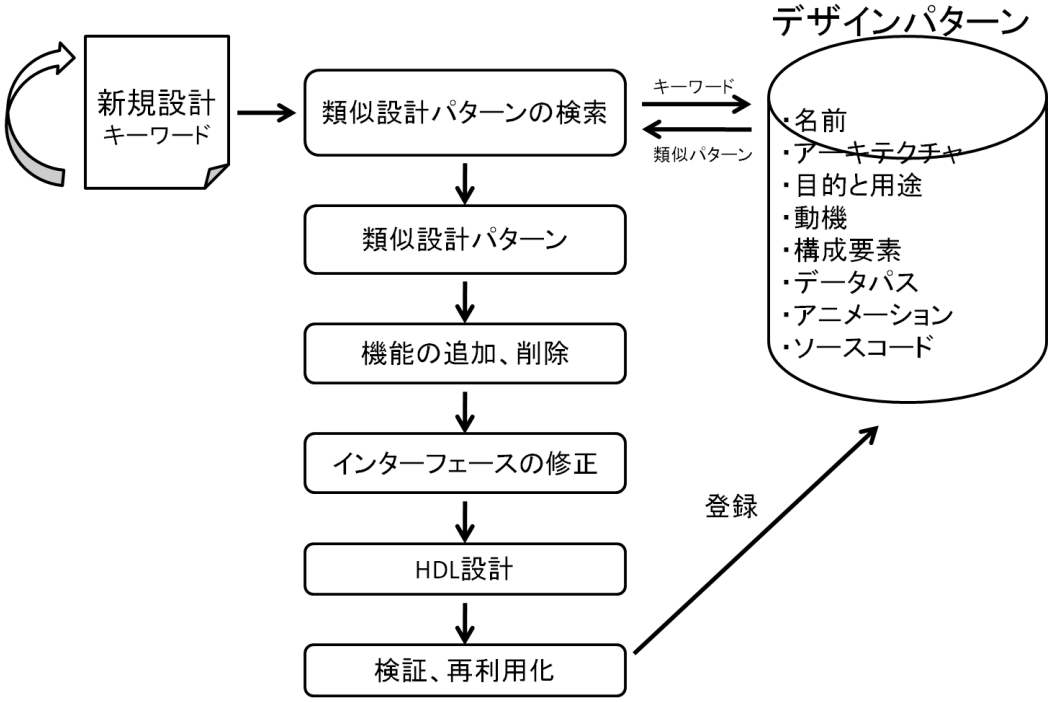

図 11：デザインパターンを用いたプロセッサ設計

本研究では，初めてプロセッサを設計する人，より高度なアーキテクチャを搭載したプロセッサを設計する人それぞれが望む活用しやすい設計資産のデザインパターンへの分割案を提案する．デザインパターンは過去の設計データから仕様と実装を再利用可能な要素ごとに分類することで，設計見本として有効的に使うことができる．設計と学習両面をサポートして，学習者がなるべく独習の形で最適なプロセッサの構成を導きだすための柔軟な設計能力を効率よく学習できるシステムを目指している．

これまで挙げたデザインパターンの設計思想と特徴に基づいて，構成は名前，目的と用途，動機，アーキテクチャ，構成要素，データパス，アニメーションと verilog-HDL ソー

スコードの 8 要素に分類した.

各設計段階において，学習者は求めるキーワードでデザインパターンに検索をかけてキーワードに類似したパターンがある場合に取り出せる.

新規設計を始めるときは，各種プロセッサアーキテクチャのアニメーションによる細かな回路動作の観察が行えるため，アーキテクチャ学習にかかる時間を短縮できる．また，アーキテクチャ，命令セットやデータパスからプロセッサの長所と短所を把握でき，効率化のポイントや改善点を発見しやすくなるのでプロセッサの仕様イメージをより明確にでき，仕様策定にかかる時間を効率的に短縮できる.

次に設計段階では，デザインパターン上にある単一機能モジュール，機能ブロック，命令形式ごとなどに分類された IP を用いることができる．ハードウェア設計の設計見本として用いるとプログラミング能力の向上と抱えている設計問題の解決策を探すことが可能である．また，設計の再利用では利用したい IP を選択してプロセッサの仕様にあったインターフェースの修正を行い，搭載することで 1 から機能を設計する必要がなくなり，設計期間の短縮ができる.

最後に検証段階では，オリジナルプロセッサと過去のプロセッサの性能比較と検証が行えるため，実際に設計したプロセッサが他のプロセッサと比べたときにどの程度の性能を持つか，また，特徴として挙げたポイントがきちんと設計できているか確認できる．そして，検証の終わったプロセッサをデザインパターンと比較して再利用可能なものがあれば，次の学習をサポートする設計見本として構成要素に分類し，デザインパターンへ登録する.

3.3 デザインパターンの例

プロセッサ設計に用いるデザインパターンは過去の設計データを用いて設計見本として分類を行う．本研究で分類したデザインパターンの構成は，図 8 が示す 8 個の要素によって構成される． デザインパターンの例として，MONI シングル R 形式と MONI シングル I5 形式の 2 つを以下に示す.

3.3.1 名前

設計見本としてデザインパターンを用いる場合，どのように求める要素，求めるものに近い要素を見つけ出すことができるかは重要である．パターン名はパターンの持つ本質を簡潔に連想させるものであり，わかりやすい良い名前を付けることはきわめて大切である.

3.3.2 アーキテクチャ

設計見本のプロセッサがどういった命令群、命令フィールドのビット長振り分けで構成されているかについて記述する。プロセッサの仕様参考とする場合に、アーキテクチャとデータパスを見れば大まかなプロセッサの長所と短所を把握することができ、効率化のポ

イントや改善点の把握に役立つ。MONI アーキテクチャは 2.1 に示した通りである.

3.3.3 目的と用途

そのデザインパターンがどのような動作をするのか、設計問題や学習課題を解決するためにどのようにパターンを分類したかの意図や動作原理などについて記述する。

MONI シングル R 形式

R 形式命令について、RF（状態記憶装置)、ALU(演算論理装置)、CU(制御論理装置)と MONI 命令セットのテストデータを合わせて提供することにより、1 クロックごとにどのタイミングで RF への読み出し、書き込み、ALU での演算が行われているか制御信号の流れを含め理解できるようにする。

MONI シングル I5 形式

即値演算命令について、MUX(信号選択装置)を追加することにより、R 形式命令とは異なる演算対象に即値を加える。実際にどの演算データを対象とするかは MUX によって選択される。MONI の命令セットに習って R 命令形式と I5 形式命令の動作の違い、また、MUX の追加による制御信号の追加について理解する。

3.3.4 動機

設計問題や学習課題に対して，提案されたデザインパターンがどのように課題を解消するかを記述する。目的と用途より細かい記述によって抽象的な内容把握に役立つ。

MONI シングル R 形式

プロセッサの動作として、現在実行している命令以前の演算結果は基本的に RF から命令フォーマットの Rs，Rt オペランドに対応するレジスタの内容が出力されて ALU の入力値となる。RF の内部状態が前後のクロックでどのように変化しているかを観察することで、ALU へ出力されるレジスタ値のクロックタイミングにおける有効性を検証することができる。しかし、ALU や RF などの論理要素を動作させるには制御信号を生成するユニットが必要になる。そこで R 形式命令ではこうした一連の動作に必要な機能をすべて提供している。このパターンを使うことで演算を行う前に命令フォーマットが指定する RF からの値の読み出し、命令の実行に必要な ALU の制御と動作、RF への書き込み制御、書き込み反映までを含めた流れを掴むことができる。

MONI シングル I5 形式

R 形式命令について、MUX1，2(マルチプレクサ)を追加することにより、命令形式ごとに異なる演算対象に即値を加える。ここで両方の命令形式を 1 つの回路で実現することで、回路資源の効率的な使用ができる。MUX2 は R 形式のソースレジスタに対応するデータと I5 形式フォーマットの即値フィールド両方から、命令形式ごとに必要とされるデータを選択して ALU の入力値とする。また、I5 形式命令は R 形式命令とは違う書き込みアドレスを対象とし、それは MUX1 の動作によって選択される。MUX2 の機能によって ALU のインターフェースの変更を小さくしつつ演算対象を増やすことができる。
プロセッサ全体としては、MONI の命令セットに習って R 命令形式と I5 形式命令の動作の違い、また、MUX1，2 の追加による CU の制御信号の追加、動作、制御理論について理解する。

3.3.5 構成要素

デザインパターンに使われている機能ユニットとそれらのプロセッサ上の動作で分担された機能の概要を記述する。
プロセッサを構成する機能単位で各ユニットを挙げており，これは学習者が設計を行う際の設計分割案である.

R 形式

PC-現在の命令のアドレスを保持しているレジスタである。IM(命令メモリ)上に格納されている命令にアクセスして取り出すために必要である。
RF-プロセッサ上で演算対象となる値を格納する。
ALU-opecode と function コードの組み合わせに応じた命令を実行する機能ユニット。
CU-命令メモリからデコードされた Opecode と Function 信号に応じて、各機能ユニットに対して動作を選択するため制御信号を定義、送信する。

I5 形式

R 形式を構成する機能ユニットに，以下のユニットを追加する.

MUX1-RF の書き込みアドレスの選択を行う。R 形式は Rd[4:2]、I5(即値)形式は Rt[7:5]がそれぞれ書き込みアドレスとなる。
MUX2-RF の出力値(RFOUT1)と命令フィールド[4:0]の即値から実行命令に応じた ALU の入力データを選択する。

3.3.6 データパス

デザインパターンで分類されたパターンがどのような回路構成なのか、制御信号を含めた回路図を記述する。パターンの構造を観察することで，回路を動作させるのに必要な機能ユニット，及び制御信号を把握することができる。また、構造を把握することは改善点や効率化のポイントを発見することにも役立つ。

MONI シミュレータではデータパス上のすべての機能ユニットに使用される制御信号を把握することはできない。デザインパターンのデータパスでは、学習者に対象のプロセッサ全体のすべての機能、制御信号と入出力の配線名を提供することで、より学習から設計段階へ移りやすいよう対応した。

ヘネパタに記載されている MIPS と研究室で定義した MONI プロセッサの大まかな違いは，命令セット，ALU の制御方法，PC アドレス加算の方法の 3 点である．図 12，図 13 に MONI シングル R 形式のデータパスと MONI シングル I5 形式のデータパスをそれぞれ示す．

MONI 命令形式の拡張による配線、機能ユニットで追加されたポイントは，データパス上で赤く強調されている．

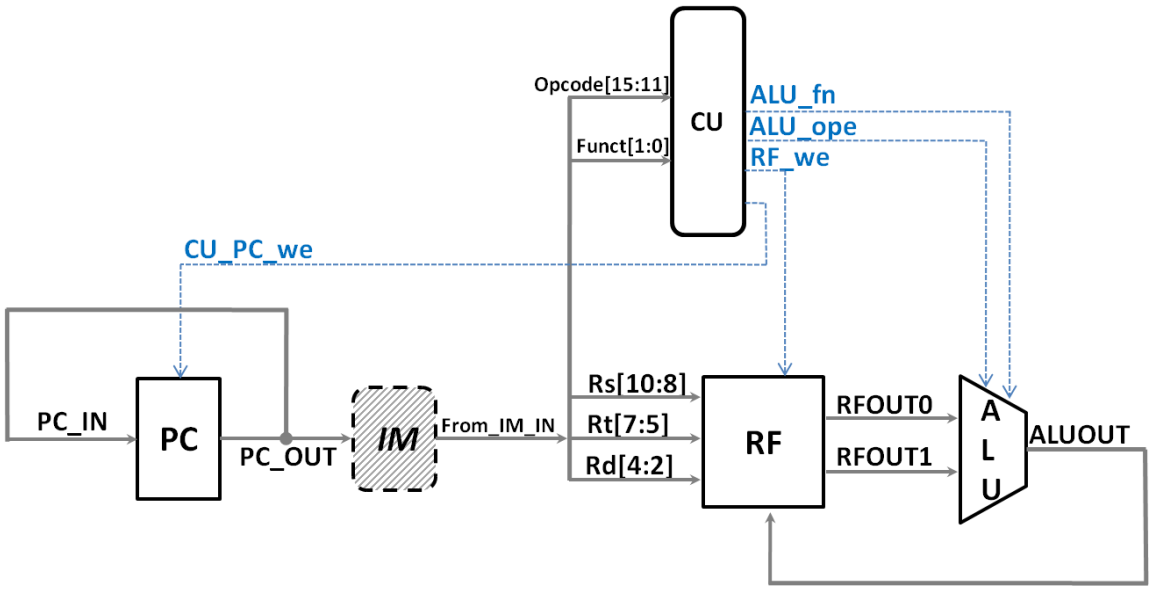

図 12：MONI シングル R 形式のデータパス

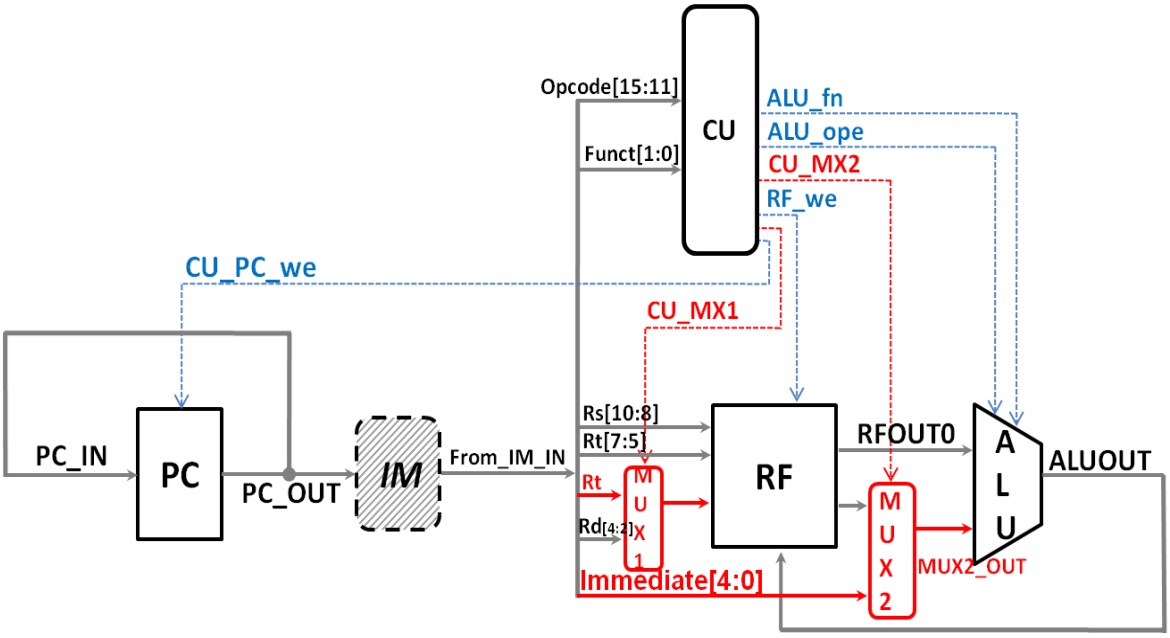

図 13：MONI シングル I5 形式のデータパス

3.3.7 アニメーション

学習者がプロセッサアーキテクチャを効率よく理解するためには、制御信号を含めたデータパスの動作を示す必要がある。アニメーションでは、1 クロックごとに命令の違いからくるプロセッサ上のデータの流れを観察できるため、アーキテクチャごとの制御信号を含めた動作と，アーキテクチャ固有の例外処理(ハザード)動作を理解しながらプロセッサの構造を体系的に学習できる。

アニメーションで取り上げる回路の要素についてはまだ改善の余地が大きくある。ここでは 1 つの案を提案したい。従来 HSCS のソフトウェア学習側では MONI シミュレータに MONI アーキテクチャに基づいたソフトウェアアプリケーションをアセンブリ言語で設計してプロセッサ全体のシミュレートによる動作の観察、理解をしていた。ただし、アプリケーションの動作には重複しており観察対象にならないものも多い。以上の理由から MONI シミュレータと併用して，より理解度を向上できるアニメーションの形が求められる。学習経験を元に学習者がプロセッサの動作を理解するときにアニメーションに求める部分を以下の 3 点に絞った。

① 簡略化されていないプロセッサの構造が参照できる
② 命令実行時の機能ユニット内部の値を観察できる
③ アーキテクチャごとの例外(ハザード)処理を理解できる

以上からアニメーションでは意味のあるプログラムの塊ではなく、ある一定量で全体の動

作、ハザード処理が観察できる命令を明示的に与える。例として挙げた R 形式は PC、CU、RF、ALU の内部動作とデータパス同様に入出力も含めて観察できる。RF は RST 状態で内部に RF[0:7]={0,1,2,3,4,5,6,7}を保持しており、次のクロックでどのように回路が動作しているか回路内で使用している部分は赤く強調することで把握できる。図 14，表 3~6 にそれぞれ MONI-R 形式のアニメーション，PC，CU，RF，ALU の動作を示す.

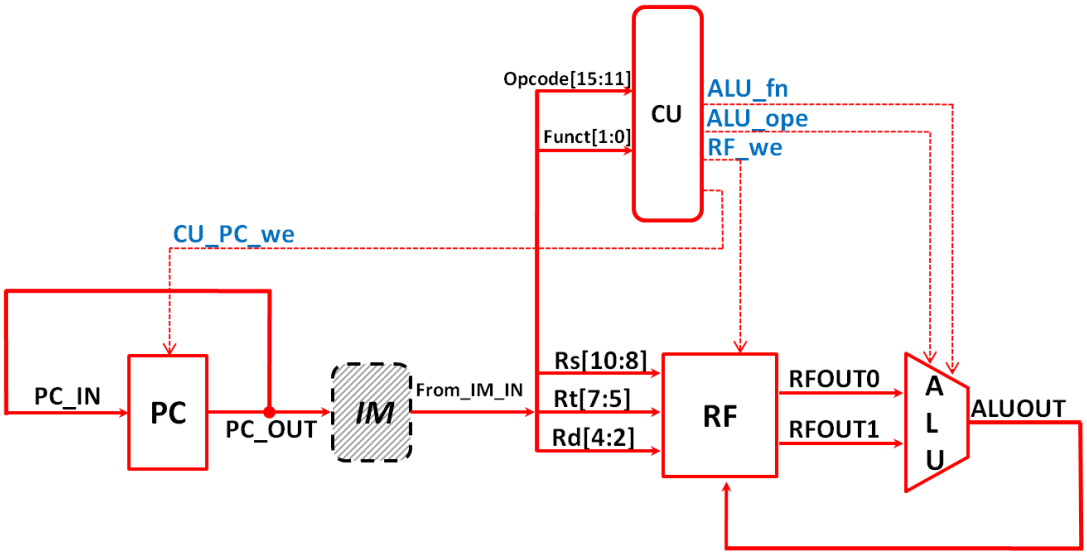

図 14：MONI-R 形式のアニメーション

表 3：PC の動作(ADD 命令)

PC	RST 状態	CLK1
PC_IN	0	1
PC_OUT	0	2
CU_PC_WE	0	1

表 4：CU の動作(ADD 命令)

CU	RST 状態	CLK1
Opecode	0	`R_type
Function	0	`Fn_ADD
CU_ALU_OPE	0	`R_type
CU_ALU_FN	0	`Fn_ADD
CU_RF_WE	0	1
CU_PC_WE	0	1

表 5：RF の動作(ADD 命令)

RF	RST 状態	CLK1
Rs[10:8]	0	1
Rt[7:5]	0	4
Rd[4:2]	0	3
RF_OUT0	0	1
RF_OUT1	0	4
RF_we	0	1
RF_INDATA	0	3
RF	{0,1,2,3,4,5,6,7}	{0,1,2,5,4,5,6,7}

表 6：ALU の動作(ADD 命令)

ALU	RST 状態	CLK1
ALU_IN0	0	1
ALU_IN1	0	4
ALU_ope	0	`R_type
ALU_fn	0	`Fn_ADD
ALU_OUT	0	5

3.3.8 Verilog-HDL ソースコード

デザインパターンの Verilog-HDL ソースコードを記載している．代表的なコードの書き方を手本とすることで効率的にコーディング能力を身に付けることができる．設計者によって機能ユニットのハードウェア動作が同じであっても，ソースコードの記述に違いが生じるため様々な記述方法で比較ができる．また、ソースコードには設計者が対面した問題に対処した解答のパターンが書かれており，同じ問題，類似の問題に直面した場合に積極的に参考、再利用することが可能である．

本体では入力，出力信号，レジスタや回路動作をコメント文で詳細に記している．モジュールの更新，デザインパターンとして再利用する場合は動作を細かくコメントして残すことで，他の人が見たときにユニットの構造理解を助けるようにする．また，他人との共有を意識することで丁寧な設計を心がけることができる．

本体の例として R 形式と I5 形式の CU は付録に載せる．R 形式から I5 形式に拡張されたときに追加される CU の信号，内部動作はオレンジ色で囲んでいる．

4. 各種アーキテクチャに基づいたデザインパターンの検討

4.1 各種プロセッサアーキテクチャ

本研究では，HSCS を用いて過去に設計したプロセッサに基づいてデザインパターンの検討を行う．デザインパターンは，2，3，及び 4.1 章で示す MONI，SAIX，SARIS アーキテクチャのシングル，マルチ，パイプラインアーキテクチャ毎に，命令形式，機能ブロックなどの要素を抽出し，3 章で示した 8 つの構成要素に分類する．SARIS の命令形式とシングルのデータパスを，それぞれ，図 15 と図 16 に示す．

ビット長 / 命令形式	5	2	3	3	3
R	Opecode	Fn	Rs	Rt	Rd
I5	Opecode	Immediate		Rt	Rd
I8	Opecode	Immediate/Address			(Rd)
J	Opecode	Target absolute address			

図 15：SARIS 命令形式

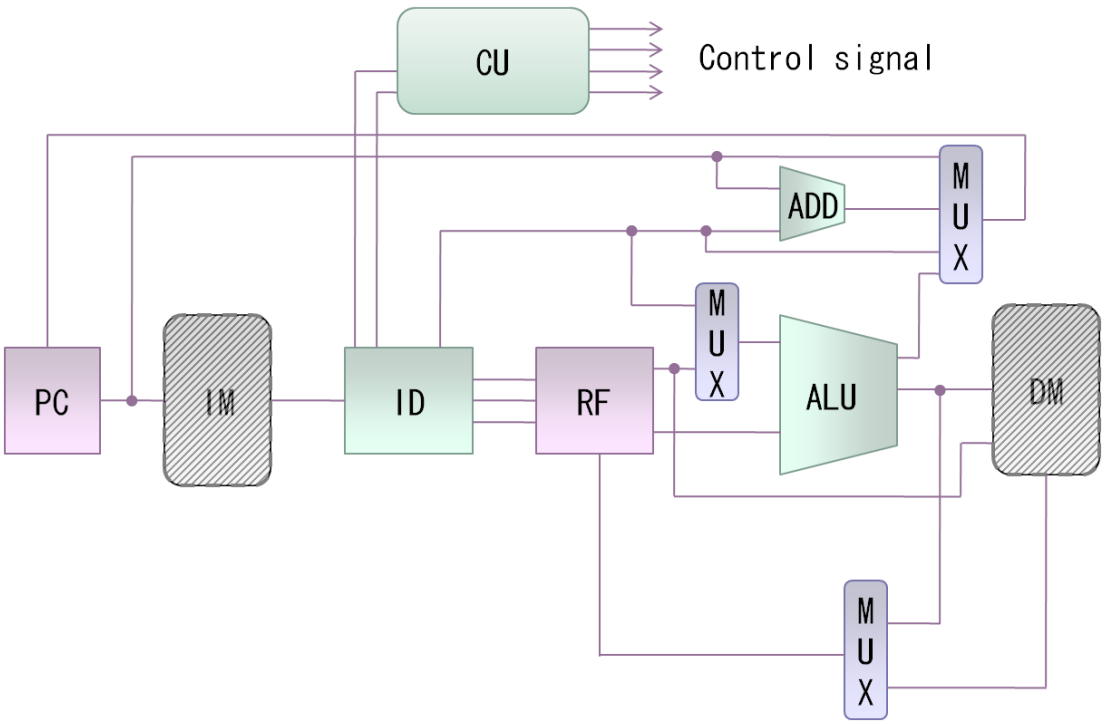

図 16：SARIS シングルのデータパス

4.2　シングルサイクル用

シングルサイクルでは，MONI，SAIX，SARIS は共通する部分が多い．これは，MONI のシングルを参考にして残りの 2 つが設計されているためである．従って，プロセッサの動作が命令フェッチ，デコード・レジスタフェッチ，命令操作の実行，メモリ・アクセス，ライトバックで成り立っており，回路を構成する機能ユニットもこの 5 つの動作を元に分割されている．そのため，デザインパターンの分割案として基本要素，命令形式毎に検討したものを表 7 に示す．3 章で説明した通り，I5 は R 形式に I5 形式の命令も実行できるように拡張したものである．したがって，J 形式の拡張ができたときに，MONI アーキテクチャすべての命令実行が可能なパターンとなる．

表 7：シングルサイクルのデザインパターン

シングルサイクル/MONI	構成要素，動作
R 形式	PC + IM + RF + ALU + CU → R 形式命令の実行
I5 形式	PC + IM + MUX1,2 + RF + ALU + CU → I5 形式命令の実行
LD 命令	PC + IM + MUX1,2,3 + RF + ALU + CU → LD 命令の実行
ST 命令	PC + IM + MUX1,2,3 + RF + ALU + CU → ST 命令の実行
I8 形式	PC + IM + MUX1,2,3 ,4+ RF + ALU + CU + PLUS → 条件分岐命令の実行
J 形式	PC + IM + MUX1,2,3 ,4+ RF + ALU + CU + PLUS → ジャンプ命令の実行
命令フェッチ	PC + IM → 実行命令の取り出し
デコード・レジスタフェッチ	IM + RF → 演算データの取り出し
命令操作の実行	RF + ALU → 命令操作の実行
メモリ・アクセス	RF + ALU + DM → アドレスが示す DM へアクセス
ライトバック	RF + ALU + DM + MUX3 → レジスタ・ファイルへ演算結果の格納

4.3 マルチサイクル用

マルチサイクルでは，SARIS と SAIX を用いてデザインパターン化を行う．SARIS は 4 ステップ，SAIX は 5 ステップで 1 命令を実行する．4.2 で示したように，5 つのプロセッサ動作を基本とするため，パターンのベースには，5 ステップの SAIX を採用した．マルチサイクルは，プロセッサの機能ユニットを，1 命令内で共有使用することが特徴である．2 つのプロセッサ動作の違いを，条件分岐命令の動作を例に挙げて示す．また，これにより分類したデザインパターンを表 8 に，SARIS マルチのデータパスを図 17 に示す．メモリ・アクセス，ライトバック，及び条件分岐命令以外は，基本動作は変わらない．しかし，動作全体で見たときには変わらないだけで，機能の割り振りに関しては違いがある．

SARIS は，ステップ 1 で PC 更新と命令フェッチを行い．ステップ 2 で命令デコードと RF 読み出しを行い．ステップ 3 では，ALU で分岐先のアドレス計算と分岐判定を行う．分岐が成立するときは，PC の値を再度分岐先のアドレスで更新する．不成立のときは，PC の値をそのまま次の命令呼び出しに使う．ステップ 3 で終了となるので，次はステップ 1 の処理を行う[22].

SAIX は，ステップ 1 で PC 更新と命令フェッチを行い．ステップ 2 で命令デコード，分岐先アドレスの計算，及び RF 読み出しを行い．ステップ 3 では，フェッチされた Rd の内容から ALU で分岐の判定を行い，分岐を実行する場合は，ALUOUT に保存された分岐先アドレスを元に PC を更新する．

このように，2 つのプロセッサは，条件分岐命令を 3 ステップで行う．しかし，ステップごとの動作や回路機能の共有の仕方に違いがある．SARIS は，ステップ 1 と 3 で 2 回 ALU を使用しているが, SAIX ではすべてのステップで 3 回 ALU を使用している. この違いは，分岐命令の実行に必要な要素を，どのようなステップに分けて求めるかによる．これから，SARIS はステップ 3 で 2 つの分岐要素を求めるために, ALU の入力ポートを 3 つに拡張する必要があり，一方 SAIX はシングルで用いた 2 ポートのまま利用できる．同じ命令実行であっても，回路の機能をどう使って表現するかは，設計者によって違いが現れる．

表 8：マルチサイクルのデザインパターン

マルチサイクル/SAIX	構成要素，動作
R 形式	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1 + ALU + ALUOUT + CU → R 形式命令の実行
I5 形式	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1,2 + ALU + ALUOUT + CU → I5 形式命令の実行
LD 命令	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1,2,4,5 + ALU + ALUOUT + CU + DM + DR → LD 命令の実行
ST 命令	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1,2,4,5+ ALU + ALUOUT + CU + DM + DR → ST 命令の実行
I8 形式	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1,2,3,4,5 + ALU + ALUOUT + CU + DM + DR → 条件分岐命令の実行
J 形式	PC + IM + IR + RF + RFOUT0 + RFOUT1 + MPX1,2,3,4,5 + ALU + ALUOUT + CU + DM + DR→ ジャンプ命令の実行
命令フェッチ	PC + IM + IR + MPX1 + ALU → 実行命令の取り出し
デコード・レジスタフェッチ	IR + RF + RFOUT0 + RFOUT1 → 演算データの取り出し
命令操作の実行	RFOUT0 + RFOUT1 + MPX1,2 + ALU + ALUOUT → 命令操作の実行
メモリ・アクセス	RFOUT0 + MPX1 + ALU + ALUOUT + DM + DR → アドレスが示す DM へアクセス
ライトバック	DR + MPX4 + RF→ レジスタ・ファイルへ演算結果の格納
条件分岐命令	PC + IR + RF + MPX1,2,3,5 + RFOUT0 +ALU + ALUOUT → 条件分岐命令の実行
マルチサイクル/SARIS	**構成要素，動作**
メモリ・アクセス、ライトバック	ALUOUT + DM + MUX + RF → メモリアクセス、演算結果の格納
条件分岐命令	PC + IR + RF + ALU_DATA1 + MUX + ALU → 条件分岐命令の実行

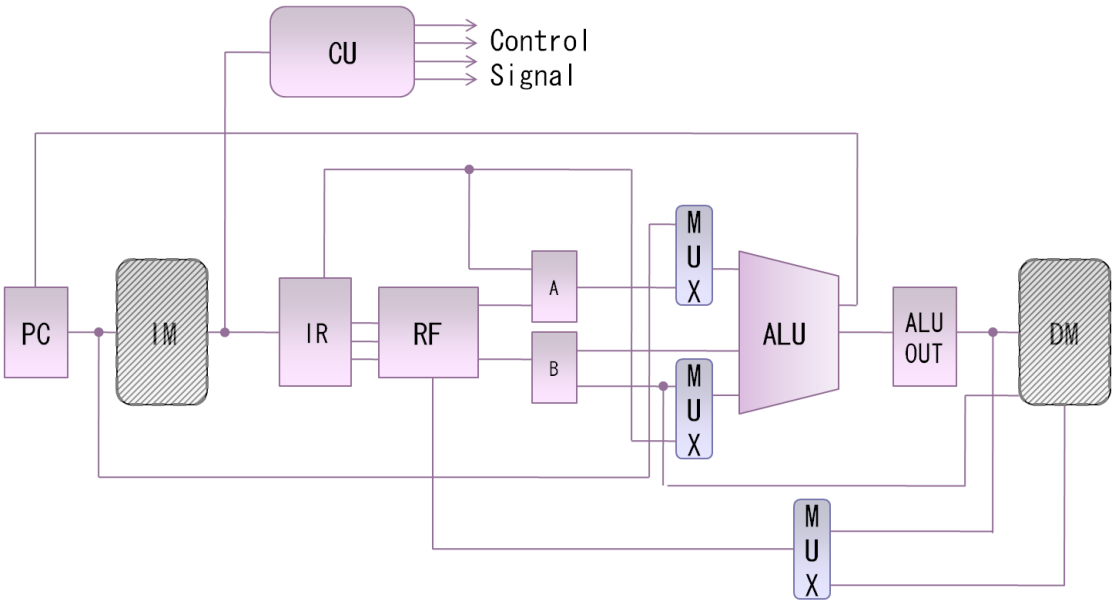

図 17：SARIS マルチのデータパス

4.4 パイプライン用

パイプラインでは，SARISとSAIXを用いてデザインパターン化を行う．パイプライン段数は5段である．パターンのベースには，ヘネパタに掲載されているMIPSパイプラインを参考にしたSAIXを採用した．パイプラインは，同時に実行可能な命令を増やすことで速度の向上を図っている．シングルでは，命令を実行するために必要な機能ユニットは使用する数だけ揃える必要があった．マルチでは，1つの回路で機能ユニットを共有して使用することができた．パイプラインでは，さらに各動作の間にレジスタを配置することで，1つの回路で複数の命令を共有することが可能になる．しかし，これまで発生しなかった複数命令実行によるハザード処理が必要になる．これに対応するため，制御信号の割り当てを3つのユニットに分担している．基本動作で分類したデザインパターンを表9に，SARISパイプラインのデータパスを図18に示す．

表 9 : パイプラインのデザインパターン

パイプライン /SAIX	構成要素，動作
R 形式	PC + IM + ADD + IF/ID + RF + MPX1 + ID/EXE + ALU + EXE/WB + BYPASS + CU → R 形式命令の実行
I5 形式	PC + IM + ADD + IF/ID + RF + 符号拡張 + MPX1,3 + ID/EXE + ALU + EXE/WB + BYPASS + CU → I5 形式命令の実行
LD 命令	PC + IM + ADD + IF/ID + RF + 符号拡張 + MPX1,2,3,4 + ID/EXE + ALU + EXE/WB + BYPASS + DM + CU → LD 命令の実行
ST 命令	PC + IM + ADD + IF/ID + RF + 符号拡張 + MPX1,2,3,4 + ID/EXE + ALU + EXE/WB + BYPASS + DM + CU → ST 命令の実行
I8 形式	PC + IM + ADD + IF/ID + RF + 符号拡張 + MPX1,2,3,4,5 + ID/EXE + ALU + EXE/WB + HAZARD + BYPASS + DM + CU → 条件分岐命令の実行
J 形式	PC + IM + ADD + IF/ID + RF + 符号拡張 + MPX1,2,3,4,5 + ID/EXE + ALU + EXE/WB + HAZARD + BYPASS + DM + CU→ ジャンプ命令の実行
命令フェッチ	PC + IM + ADD + IF/ID + MPX5+ HAZARD→ 実行命令の取り出し
デコード・レジスタフェッチ	IF/ID + RF + 符号拡張 + MPX1 + ID/EXE → 演算データの取り出し
命令操作の実行	ID/EXE + MPX2,3 + ALU + BYPASS + EXE/MEM → 命令操作の実行
メモリ・アクセス	EXE/MEM + DM + MEM/WB → アドレスが示す DM へアクセス
ライトバック	MEM/WB + RF + MPX5→ レジスタ・ファイルへ演算結果の格納
条件分岐命令	PC + ADD + IM + IF/ID + RF + 符号拡張 + ID/EXE + HAZARD + MPX1,2,3,5 + BYPASS +ALU → 条件分岐命令の実行

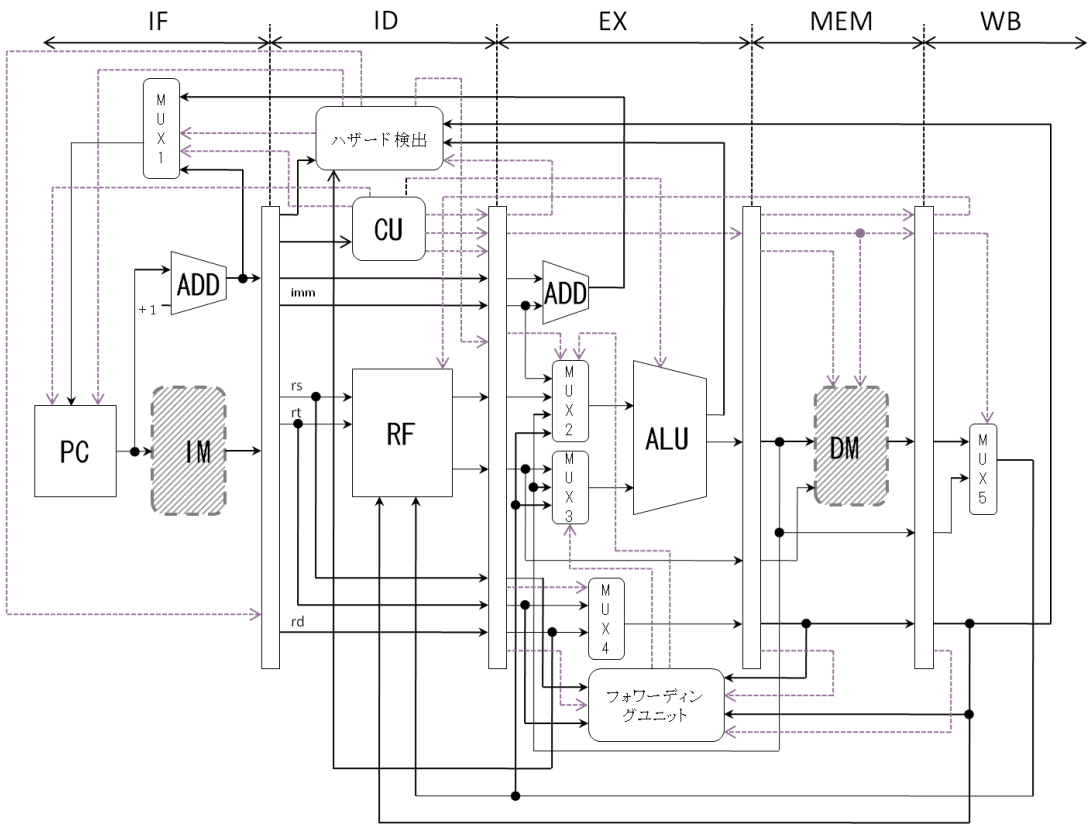

図 18：SARIS パイプラインのデータパス

5. デザインパターンを用いたプロセッサ設計

5.1 デザインパターンを用いた新規プロセッサの設計

デザインパターンがプロセッサ設計において有効であるかを評価するために, STRAD シングルプロセッサを設計した．STRAD は 2 章で示した MONI と SAIX を参考にしたプロセッサで，ハードウェア上での実行命令の動的な最適化を目指している．

(1) 設計思想

本来 RISC 型が優位だと言われた理由は, コードの大部分が単純な命令で占められており, 複雑な命令を単純な命令に置き換えた方が高速化につながったからである．しかし，命令セットの変遷を見るとおり, RISC 型はより複雑な命令を搭載する方向へ進んでいる. また, CISC 型は内部で RISC 型に命令変換して実行するため, 命令によっては固定長である RISC 型よりも短いデータ幅で同一の演算が可能である．過去の設計仕様を見ると，演算器の個数が 1 クロックあたりの演算回数を決定している．従って，複雑な命令を追加しても，うまく使用率を高めることができればメリットの方が大きいと考えた．

そこで，STRAD 命令セットには複数データを同時実行可能な SIMD 形式を，ハードウェアには命令変換機能，データ幅を訂正する機能，及び SIMD 演算器を追加した．命令変換機能は 2 つの単純命令を SIMD 命令に変換することで命令の実行効率を上げる機能であり，ハードウェア側で実行命令を増せる可能性がある．

(2) 命令セットアーキテクチャ

STRAD 命令形式を図 19 に示す. Opecode は共通 4 ビット, 各レジスタ指定幅は R と I5 形式では 2 ビット，I9 形式は 3 ビット，MD 形式は 4 ビット指定である．I5 形式では Sel ビットによってアクセスするレジスタファイル領域を上位，下位に分けて利用でき, MONI では R 形式のみにあった機能コードを I5 形式にも割り振っている. MD 形式では Rd, Rs, Rt がそれぞれ 4 ビット幅をもち，2 つの命令が同時実行できる．これらから改善された要素を以下に示す.

① MONI のフォーマットを参考にした新規命令の追加領域を確保
② I9 形式が 9 ビット分岐先アドレスを指定可能
③ MD 命令形式の追加により，ループ構造で依存関係のない ADD，SUB 命令に関して SIMD 命令を用いて，1 クロックで 2 命令実行が可能

ビット長 / 命令形式	4	2	2	2	6	
R	Opecode	Rd	Rs	Rt	Fn	
I5	Opecode	Rd	Rs	Immediate	Sel	Fn
I9	Opecode	Rd	Immediate/Address			
MD	Opecode	Rd	Rs	Rt		
J	Opecode	Target absolute address				

図 19：STRAD 命令形式

(3) SIMD 演算とは

SIMD とは Single Instruction Multiple Data(単一命令，複数データ)の略で，1 つの命令で複数のデータに対して処理を行う演算方式である．以下に SIMD 演算の例を示す．従来の命令では以下のように 1 命令ごとに 1 つの演算を行うため，レジスタファイルの中身を[1.2.3.4.5.6.7.8]とすると

ADD $0 $2 $3;　　//$0 <- $2 + $3

ADD $1 $3 $3;　　//$1 <- $3 + $3

$0 <- $2 + $3 = 3 + 4…RF[7.2.3.4.5.6.7.8]

$1 <- $3 + $3 = 4 + 4…RF[4.8.3.4.5.6.7.8]となる．これを同じ条件のもとで SIMD 命令を用いると

SIMADD ($0,$1))　($2,$3)　+　($3,$3);

$0 -> $2 + $3 = 3 + 4 ,$1 ->$3 + $3 = 4 + 4 の計算が 1 命令で行えるため，うまく使えば動作に必要な命令数を減らせる．

(4) 命令変換機能とは

命令の実行効率を上げるためにハードウェア上で動的に命令実行の最適化を図る機能である．命令変換機能はループ内で命令の圧縮が可能な場合に，2 つの ADD，SUB 命令を 1 つの SIMD 命令に変換をする．

命令変換ユニットの内部には，LR（ループレジスタ），PIR(past instruction：過去命令レジスタ)，PCR(プログラムカウンタレジスタ)と CIR(変換命令レジスタ)がある．LR はアプリケーションのループの回数，PIR は実行命令の 1 つ前の命令，PCR は変換可能な命令の PC アドレスを保持している．

変換動作はループ回数が 1 回以上でループ構造の確定と，実行命令と PIR の比較による依存関係の判定を行う．今回の機能では ADD or SUB 命令が連続かつ，実行命令の Rs,

Rt と過去命令の Rd が違う場合の 2 つの条件を満たすときに命令の変換を行って CIR に変換した命令を書き込む．そして，2 回目以降のループにおいて変換した命令アドレス PCR とループ時の PC が同じであれば命令を入れ替える．図 20 に命令変換の例を示す．変換前と変換後を見て分かるように，ループ内で 1 命令短縮できた．これによりループ回数が多いほど実行命令数を減らせることがわかる．現状は SIMD 演算への変換しかできないが，積和演算などを含めた複合演算命令への最適化が考えられる．

(5) STRAD アーキテクチャのメリットとデメリット

MONII5 形式と STRADI5 形式を比較したときのメリットとデメリットを以下に示す．

メリット：MONI では I5 形式命令 1 つの全体に占める割合は，Opecode5 ビットを用いるので命令全体の 1/32 である．STRAD では Opecode4 ビットと機能コード 2 ビットで 1/16*1/4=1/64 となり，1 命令の全体に占める割合は半分になる．よって，STRAD の方が I5 形式の命令を多く持て，プログラマーの視点から，使いやすい命令を増やせる．

デメリット：MONI ではレジスタアドレスに 3 ビットを割り当てるため 8 個のレジスタすべてを直接選択できるが，STRAD は 2 ビット幅のため 4 個のレジスタしか選択できない．今回 STRAD は Sel ビットを与えることで，上下 4 つずつ 8 個のレジスタすべてにアクセスできるが LD，ST 命令は扱いにくい．

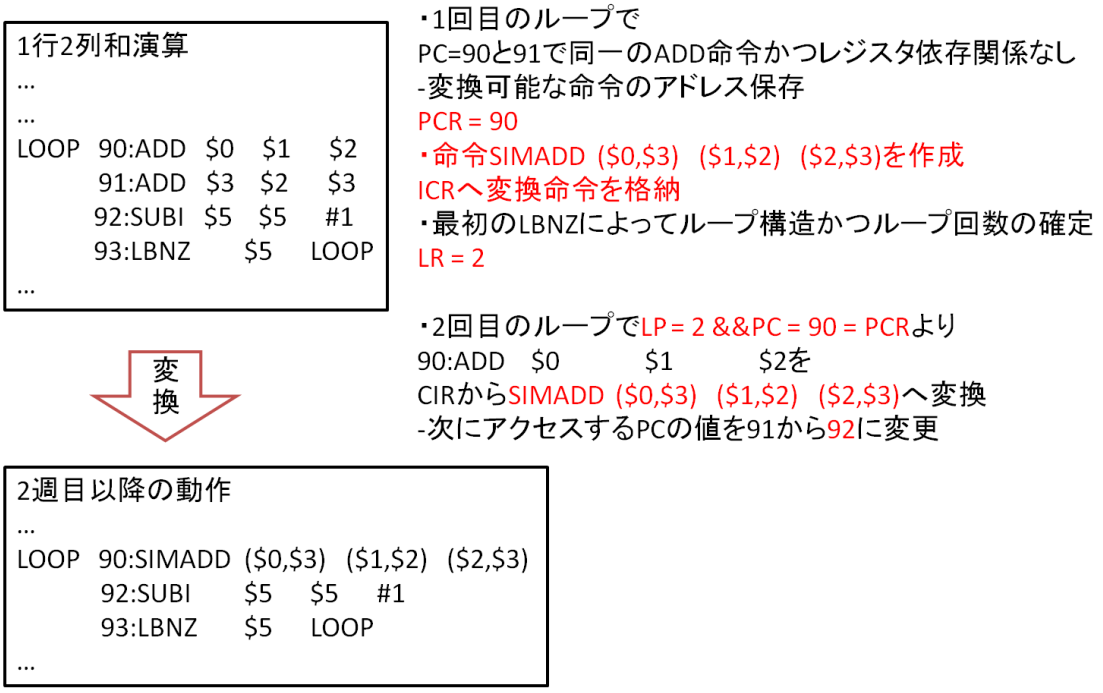

図 20：命令変換の例

(6) デザインパターンを用いた設計

今回の設計ではデザインパターンから MONI，SAIX の命令セット，データパス，サンプルプログラムを用いた．設計した STRAD シングルのデータパスを図 21 に示す．STRAD は従来のプロセッサの回路構造や特定のアプリケーションにおける命令動作によって，改善点や効率化のポイントを把握して仕様を固めた．また，設計段階では SAIX プロセッサをベースに Correct_Unit，RF の入出力ポートの追加・修正を行い，回路全体を設計した．回路機能のほとんどが SAIX から再利用できたため，設計時間の大幅な短縮ができた．

SIMD 内部には, 2 つの ALU を保持しており, MD 形式は SIMD のみを用いて行う. SIMD を使用しているときには ALU は利用していない. そのため, 改良点としては, ALU と SIMD 両方を用いた実現が考えられる．そうすることにより，SIMD を小さく設計でき，ALU を有効活用できるため，回路面積の削減が行える．また，命令変換ユニットの動的な最適化をコンパイラで静的に行うことで，命令の並列性を確定できれば，さらなる削減が望める．

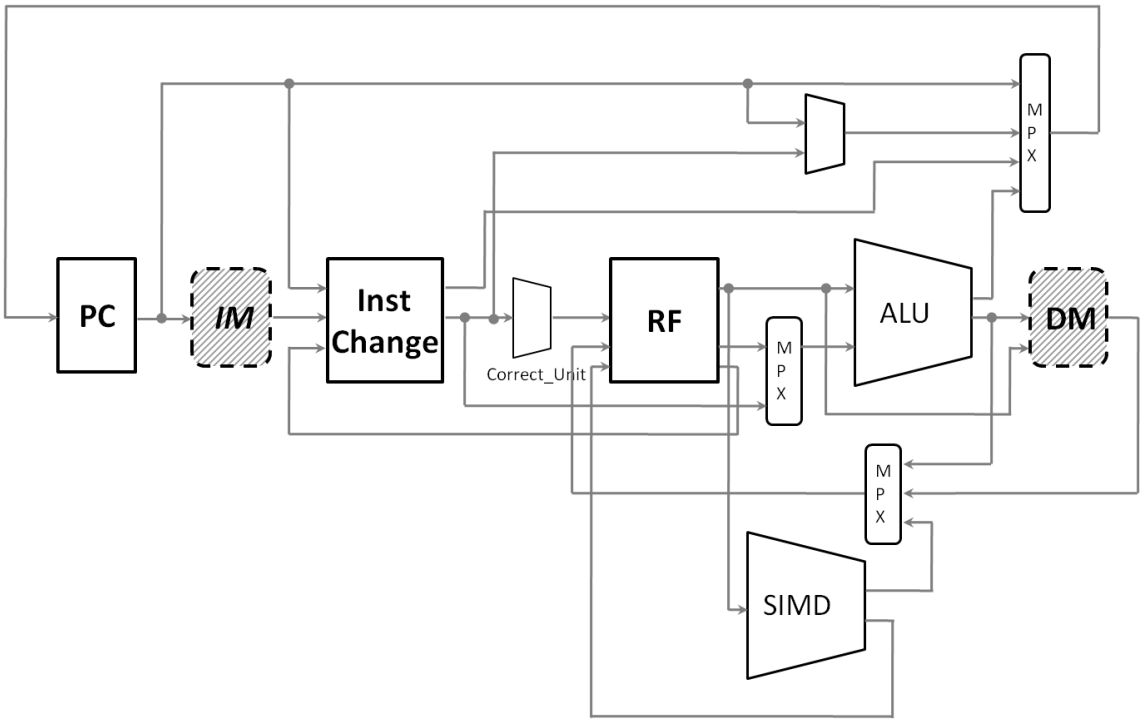

図 21：STRAD シングルのデータパス

5.2 4 回生によるプロセッサの設計

学生がデザインパターンを用いて, 設計した MAP プロセッサの命令形式とデータパスをそれぞれ，図 22 と図 23 に示す[22]．MAP は，内部に 2 つの ALU を持っており，1 クロックで 2 命令の同時実行が可能なプロセッサである．MAP プロセッサには以下のような特

徴がある.

MAP アーキテクチャ

・命令語長は 64 ビット固定

・演算部は 32 ビット、2-ALU

・3 オペランド命令方式

・全 37 命令

・4 命令形式

・R 形式と I 形式は組み合わせ可能

命令語調	64											
命令形式	6	5	5	5	5	6	6	5	5	5	5	6
R形式	Opecode	Rs	Rt	Rd	Shamt	Fn	Opecode	Rs	Rt	Rd	Shamt	Fn
I形式	Opecode	Rs	Rt	即値			Opecode	Rs	Rt	即値		
L形式	Opecode	Rs	address/immediate									
J形式	Opecode	address										

図 22 : MAP 命令形式

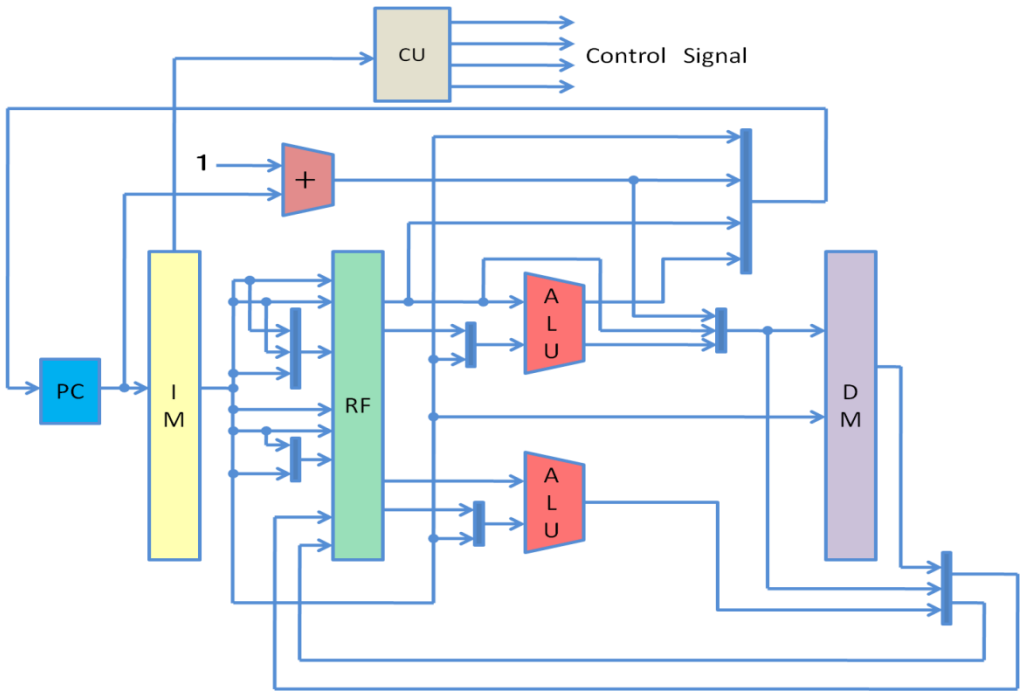

図 23 : MAP のデータパス

5.3 考察

デザインパターンを用いたプロセッサの設計時間を表 10 に示す. ソフトウェアの要素は，新たな機能の追加と改善による仕様の考察，それを元にした命令セットの定義，及びアプリケーションの設計である．ハードウェアの要素は，アーキテクチャの学習，verilog-HDL による設計，及びシミュレーションである．

STRAD の仕様は，過去のプロセッサの命令実行動作から，回路の特徴となる改善点を発見できた．SIMD 命令では，1 クロックで 2 つの命令実行を行うために，演算器を 2 つ並べた構成となる. また, ハードウェア設計は, SAIX とのアーキテクチャの違いを Correct_Unit で修正することで，SIMD を除く機能ユニットを再利用できた．そのため，設計時間の大部分を Inst_Change に充てることができ，短時間で STRAD を完成できた．プロセッサの動作としては，ADD か SUB が連続で依存関係がない場合のみ SIMD 命令へ変換可能である. プロセッサの改善点は 5.1 で示している. また, 命令変換機能は, SIMD だけではなく，積和や積差といった複合命令形式へ拡張することで，スーパースカラなど動的にスケジューリングを行い，複数命令を実行するアーキテクチャで効果が発揮できると考える．

MAP では，命令セットにおける 64 ビットの機能振り分けが難しい．これは，過去の設計データの中に参考例がなく，効果的に使用できる形を定義するために時間がかかる．また，設計者は MAP の 2 命令間の依存関係をソフトウェア側でスケジューリングし，次に，プログラムのハンドアセンブルを行う．このため，ソフトウェアに要した時間が 2.1 の従来の結果と比べて増加している．ハードウェア設計では，回路全体は MONI や SAIX と比べて多くの機能ユニットが必要である．しかし，デザインパターンの資産を利用して，2-ALU や RF などの機能ユニットを修正して搭載したため, 設計とテストの時間が短縮できている.

HSCSとデザインパターンを利用した結果，シングルサイクルの学習には十分に対応できていると考える．それは，制御信号の流れと複数命令形式を実行するために，必要な機能ユニットが形式ごとに把握できるためである．また，トレードオフをより理解できる環境としては，ソフトウェア側でコンパイラ学習をできる必要がある．パイプライン，スーパスカラの学習でわかったことは，いかに最適な命令をパイプラインに流せるかを考察することである．これは規模が大きくなるほど，設計者の手に余る部分である．ソフトウェア側で静的に，いかにうまく命令の並列性を検出して，スケジューリングを行ってからハードウェアで実行できるか．プロセッサの性能を高めるには，コンパイラの性能がどれだけ高められるか．そして，効率的な演算構造で，命令実行できるハードウェアの設計を目指す必要がある．命令実行の効率化を，ハードウェアだけに求めるのではなく，ソフトウェアにうまく分業することで，ハードウェアとソフトウェアのトレードオフの理解が促進できる．作り終わったプロセッサで，共通使用できるコンパイラがあると便利だと思う．多くの論文が出されているように，コンパイラの設計もプロセッサのハードウェア設計と並ぶ大きな学習要素になると考えられる．

表 10：プロセッサの設計時間　　単位：時間

プロセッサ	MAP	STRAD
アーキテクチャ	シングル	シングル
仕様の考察 命令セットの定義 ソフトウェア設計	100	10
アーキテクチャの学習 ハードウェア設計 シミュレーション	60	10

6. おわりに

本研究では，シングル，マルチ，及びパイプラインプロセッサの設計と HSCS を利用した過去の設計データに基づいたプロセッサ用デザインパターンの検討を行った．

HSCS を利用したプロセッサ設計によって，現れたシステムの問題に対し，デザインパターンとして，設計データを設計見本の形で再利用するために，分類した構成要素を示した．また，デザインパターンを用いて，オリジナルプロセッサの設計を行った．ハードウェア設計においては，過去の設計時間と比べて短縮ができた．仕様の策定においては，効率化のポイントや改善点を把握しやすいため，短期間で中身を充実できた．

今後の課題として，ソフトウェア指向プロセッサの VLIW やハードウェア指向のスーパースカラプロセッサのデザインパターン登録が考えられる．コンパイラはソフトウェア側の働きにおいて，重要な要素であり，ハードウェア側とうまく役割を分割できれば，より高性能なプロセッサを設計することができて，ハードウェアとソフトウェアのトレードオフの理解を促進できる．また，プロセッサアーキテクチャにおいて，マルチコアや演算器，内部動作の多段数化によるスーパーパイプラインといった，高度な設計能力の習得にも対応できるように，デザインパターンの拡張が必要である．

HSCS は，効果的に利用できればハードウェア・ソフトウェアの知識と技術を幅広く習得できる．しかし，学習範囲が広いために，設計手順の繁雑化と特定のアーキテクチャのサポート不足を招いている．本研究で提案したプロセッサ用デザインパターンが，プロセッサ設計の支援システムとして用いられ，上記の問題を解決する手段となることを望む．

謝辞

本研究の機会を与えてくださり，貴重な助言，ご指導を頂きました山崎勝弘教授に深く感謝いたします．また，本研究に関して様々な相談に乗って頂き，貴重な助言，ご意見を頂きました境氏，そして高性能計算研究室の皆様に深く感謝いたします．

参考文献

[1] David A Patterson，John L Hennessy 著，成田光彰 訳：コンピュータの構成と設計(上)(下) 第2版，日経ＢＰ社，2003.

[2] 馬場敬信 著：コンピュータアーキテクチャ，オーム社，1994.

[3] デイビッド・マネー・ハリス・L・ハリス 著，天野英晴・鈴木貢・中條拓伯・永松礼夫 訳：ディジタル回路設計とコンピュータアーキテクチャ，翔泳社，2009.

[4] 中森章 著：マイクロプロセッサ・アーキテクチャ入門，CQ出版，2006.

[5] 坂井修一 著：コンピュータアーキテクチャ，コロナ社，2004.

[6] マイコミジャーナル，コンピュータアーキテクチャの話：
http://journal.mycom.co.jp/column/architecture/150/index.html

[7] 石井忠俊：D4章システムアーキテクチャ設計技術5, STARC, 2008

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 著，本位田真一・吉田和樹 監訳：オブジェクト指向における再利用のためのデザインパターン，ソフトバンク, 1995.

[9] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton “Design Patterns for Reconfigurable Computing” in FCCM, IEEE April 2004.

[10] F. Rincon, F. Moya, J. Barba, and J. C. Lopez “Model Reuse through Hardware Design Patterns” Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2005

[11] 桜井祐一，長澤龍，宮内新，石川知雄：教育用RISC型マイクロプロセッサMITEC-IIを用いた演習環境の開発及びMITEC-IIを用いた演習の実施, 情報処理学会研究報告, Vol.2001, No.101，pp.47-54，2001.

[12] 末吉敏則，久我守弘，紫村英智：KITEマイクロプロセッサによる計算機工学教育支援システム，電子情報通信学会論文誌，Vol.J84-D-1，No.6，pp.917-926，2001.

[13] 西村克信, 額田多政, 天野英晴 : 教育用パイプライン処理マイクロプロセッサ PICO^2 の開発，情報処理学会研究報告，Vol.2000，No.2，pp.141-148，2000.

[14] 池田修久，中村浩一郎，大八木睦，Hoang Anh Tuan，山崎勝弘，小柳滋：ハード/ソフト・コラーニングシステムにおけるFPGAボードコンピュータの設計，情報処理学会，第66回全国大会講演論文集，5T-5，2004.

[15] 大八木睦，池田修久，山崎勝弘，小柳滋：ハード/ソフト・コラーニングシステムにおけるアーキテクチャ選択可能なプロセッサシミュレータの設計，情報処理学会，第66回全国大会講演論文集，5T-6，2004.

[16] 中村浩一郎，池田修久，山崎勝弘，小柳滋：プロセッサアーキテクチャ教育用FPGAボードコンピュータシステムの開発，情報科学技術レターズ，FIT2004，LC-008，2004.

[17] 大八木睦：ハード/ソフト・コラーニングシステム上でのアーキテクチャ可変なプロセッサシミュレータの設計と試作，立命館大学理工学研究科修士論文，2004.

[18] 難波翔一朗：FPGAボード上での単一サイクルマイクロプロセッサの設計と検証，立命館大学理工学部卒業論文，2005.

[19] 中村浩一郎：命令定義可能なハード/ソフト・コラーニングシステム上でのプロセッサデバッガの設計と実装，立命館大学理工学研究科修士論文，2006.

[20] 難波翔一朗：プロセッサ設計支援ツールの実装とハード/ソフト協調学習システムの評価，立命館大学理工学研究科修士論文，2007.

[21] 志水建太：ハード/ソフト協調学習システム上でのプロセッサ設計とプロセッサデバッガによる検証，立命館大学理工学部卒業論文，2007.

[22] 井出純一：ハード/ソフト協調学習システムを用いたプロセッサ設計と評価，立命館大学理工学部卒業論文，2008.

[23] 宮崎匡史：プロセッサ設計支援ツールを用いた独自プロセッサの設計，立命館大学理工学部卒業論文，2009.

[24] PISHVA JOHN CYRUS P：ハード/ソフト協調学習システムを用いた割り込みプロセッサの設計，立命館大学理工学部卒業論文，2010.

[25] 安倍厚志，山崎勝弘：プロセッサ設計におけるデザインパターンの利用の検討，情報処理学会，第71回全国大会論文集，2K-4，2009.

付録　デザインパターンのソースコード

(1)R 形式 : CU

```
`timescale 1ns / 1ps     //最初にこの一行を書いておく

`define Rr          5'b00000
`define Rrr         5'b00001
`define Rrrr        5'b00010
`define Rrrrr       5'b00011
`define NOP         5'b11110
`define HALT        5'b11111

module CU(
      I_CU_OPE,
      I_CU_FN,

      CU_PC_CE,
      CU_RF_WE,

      CU_ALU_OPE,
      CU_ALU_FN,

      START
);

      input [4:0] I_CU_OPE;   //オペコード : From_IM_data[15:11]
      input [1:0] I_CU_FN;   //機能コード : From_IM_data[1:0]

      output CU_PC_CE;   //PC 書き込みイネーブル信号
      output CU_RF_WE;   //RF 書き込みイネーブル信号

      output [4:0] CU_ALU_OPE;   //実行命令選択信号
      output [1:0] CU_ALU_FN;   //R 命令形式選択信号

      input START;
```

```
//pc:HALT 以外の命令で PC へ書き込みイネーブルを出力する。
    assign CU_PC_CE = (START == 1'b1 && I_CU_OPE != `HALT)? 1:1'b0;

//rf:HALT NOP JUMP 以外の命令で RF へ書き込みイネーブルを出力する。
    assign CU_RF_WE = (START == 1'b1 && I_CU_OPE != `JUMP)
                    && (START == 1'b1 && I_CU_OPE != `NOP)
                    && (START == 1'b1 && I_CU_OPE != `HALT) ? 1:1'b0;

//alu:NOP HALT JUMP 以外の命令ではデコードされた命令のオペコードを出力する。
NOP HALT 命令は 5'b00000 を出力する。ALU では ADD 命令が演算されるが CU_RF_WE
信号によって書き込みが否定される。
    assign CU_ALU_OPE = (START == 1'b1 && I_CU_OPE != `JUMP )
                      && (START == 1'b1 && I_CU_OPE != `NOP)
                      && (START == 1'b1 && I_CU_OPE != `HALT) ? I_CU_OPE:
                         5'b00000;
//alu:オペコードが R 形式の場合に入力された機能コードを出力する。それ以外の命令形式
では 2'b00 を出力する。ALU では ADD 命令が演算されるが CU_RF_WE 信号によって書
き込みが否定される。
    assign CU_ALU_FN = (START == 1'b1 && I_CU_OPE == `Rr)
                     | | (START == 1'b1 && I_CU_OPE == `Rrr)
                     | | (START == 1'b1 && I_CU_OPE == `Rrrr)
                     | | (START == 1'b1 && I_CU_OPE == `Rrrrr) ? I_CU_FN:
                         2'b00;

endmodule
```

⑵I5 形式：CU

```
`timescale 1ns / 1ps

`define Rr          5'b00000
…
`define ADDI        5'b00100
`define SUBI        5'b00101
`define ANDI        5'b00110
`define ORI         5'b00111
`define XORI        5'b01000
`define SLLI        5'b01001
`define SRLI        5'b01010
`define SRAI        5'b01011
`define SLTI        5'b01100
`define SGTI        5'b01101
`define SLEI        5'b01110
`define SGEI        5'b01111
`define SEQI        5'b10000
`define SNEI        5'b10001

`define JUMP        5'b11100
`define NOP         5'b11110
`define HALT        5'b11111

module CU(
    I_CU_OPE,
    I_CU_FN,

    CU_PC_CE,
    CU_RF_WE,

    CU_ALU_OPE,
    CU_ALU_FN,

    CU_MX1,
    CU_MX2,
```

```
        START
);

        input [4:0] I_CU_OPE;   //オペコード：From_IM_data[15:11]
…
        output [1:0] CU_MX1; //書き込みアドレスの選択 rd or rt
        output [1:0] CU_MX2; //ALU の演算データ選択　RFdata or 即値

        input START;

…

        //mx1:R 形式命令 rd 選択信号 2'b00、 I5 形式命令 rt 選択信号 2'b01 を出力
        assign CU_MX1 =
(START == 1'b1 && I_CU_OPE == `Rr) || (START == 1'b1 && I_CU_OPE == `Rrr)||
(START == 1'b1 && I_CU_OPE == `Rrrr) || (START == 1'b1 && I_CU_OPE ==
`Rrrrr) ? 2'b00:

(START == 1'b1 && I_CU_OPE == `ADDI) || (START == 1'b1 && I_CU_OPE ==
`SUBI)
|| (START == 1'b1 && I_CU_OPE == `ANDI) || (START == 1'b1 && I_CU_OPE ==
`ORI)
|| (START == 1'b1 && I_CU_OPE == `XORI) || (START == 1'b1 && I_CU_OPE ==
`SLLI)
|| (START == 1'b1 && I_CU_OPE == `SRLI) || (START == 1'b1 && I_CU_OPE ==
`SRAI)
|| (START == 1'b1 && I_CU_OPE == `SLTI) || (START == 1'b1 && I_CU_OPE ==
`SGTI)
|| (START == 1'b1 && I_CU_OPE == `SLEI) || (START == 1'b1 && I_CU_OPE ==
`SGEI)
|| (START == 1'b1 && I_CU_OPE == `SEQI) || (START == 1'b1 && I_CU_OPE ==
`SNEI)2'b01:
2'bx;

        //mx2:R 形式命令では RF 出力データ選択 2'b00、 I5 形式命令では命令フィールドの
```

即値幅を選択 2'b10

```
      assign CU_MX2 = (START == 1'b1 && I_CU_OPE == `Rr)
| | (START == 1'b1 && I_CU_OPE == `Rrr)
| | (START == 1'b1 && I_CU_OPE == `Rrrr)
| | (START == 1'b1 && I_CU_OPE == `Rrrrr)? 2'b00:

(START == 1'b1 && I_CU_OPE == `ADDI) || (START == 1'b1 && I_CU_OPE ==
`SUBI)
| | (START == 1'b1 && I_CU_OPE == `ANDI) || (START == 1'b1 && I_CU_OPE ==
`ORI)
| | (START == 1'b1 && I_CU_OPE == `XORI) || (START == 1'b1 && I_CU_OPE ==
`SLLI)
| | (START == 1'b1 && I_CU_OPE == `SRLI) || (START == 1'b1 && I_CU_OPE ==
`SRAI)
| | (START == 1'b1 && I_CU_OPE == `SLTI) || (START == 1'b1 && I_CU_OPE ==
`SGTI)
| | (START == 1'b1 && I_CU_OPE == `SLEI) || (START == 1'b1 && I_CU_OPE ==
`SGEI)
| | (START == 1'b1 && I_CU_OPE == `SEQI) || (START == 1'b1 && I_CU_OPE ==
`SNEI) ? 2'b10:
2'bx;

endmodule
```