課題研究報告書

FPGA用ソフトプロセッサのための自動最適化コンフィギュレータの構築

北陸先端科学技術大学院大学
情報科学研究科

宮内　哲夫

2015年3月

課題研究報告書

FPGA用ソフトプロセッサのための自動最適化コンフィギュレータの構築

指導教員　田中　清史 准教授

審査委員主査　田中　清史 准教授
審査委員　金子　峰雄 教授
審査委員　井口　寧 教授

北陸先端科学技術大学院大学
情報科学研究科

1310708 宮内　哲夫

提出年月: 2015 年 2 月

概 要

従来，組み込み向けアプリケーションに対して専用の集積回路を設計することが行われていたが，近年, FPGA(Field Programmable Gate Array) と呼ばれる書き換え可能な回路を持つデバイスが利用されるようになっている．FPGA は，専用の集積回路に比べて，集積度や速度面では劣るが，回路を書き換え可能なため，開発効率が向上する利点があり，組み込み向けアプリケーションに対して多く使われるようになっている．最近では，FPGA の集積度向上により，FPGA 内にプロセッサコアを構成することが可能となっている．SoC（System on Chip）を実現するためには総資源量の観点からプロセッサ規模が小さいことが望まれる．そこで本研究では FPGA 資源の有効活用のために，FPGA 上のプロセッサで実行対象となるアプリケーションプログラムで利用される命令を解析し，実際に使用される機能のみを自動的に選択してプロセッサ回路を生成するコンフィギュレータを構築した．いくつかのアプリケーションに対する本コンフィギュレータの効果が確認できた．

目 次

表 目 次

第1章　はじめに

1.1　背景と目的

一般に，組込みシステムと言われるものには，家電，携帯機器，医療ヘルスケア，車載，等様々なものがあるが，最近では，これらのシステムは複雑化し，機械的な部品のみで構成することは難しくなってきている．そのため，システムの制御のために，専用の半導体を内蔵するものが多くなってきている．

このような，アプリケーションに特化した形で組込みシステムに組み込まれる半導体は ASIC (Application Specific Integrated Circuit) と呼ばれる．また，複雑な処理をする回路を組み合わせて，ひとつの半導体デバイスとして構成する技術は SoC（System on Chip）と呼ばれる．

ASIC による構成は，アプリケーションに特化した専用のハードウエア回路で動作するため，デバイスの面積を小さくし，高い性能を出すことができる反面，通常，ASIC による半導体回路は，製造後には変更ができないため，システムのバージョンアップや，不具合などで回路を変更する必要がある場合には，ハードウエア的に回路自体を作り変える必要があるため，費用や工数が大きくかかり，容易に変更することができない．

そのため，汎用のプロセッサを搭載し，機器の制御はプロセッサ上のソフトウエアで実行させるシステムも多くみられる．この場合，ソフトウエアは ROM または，フラッシュメモリに置かれている．フラッシュメモリにソフトウエアを置けば，フラッシュメモリの書き換えでソフトウエアを変更することができるため，システムのバージョンアップや，不具合の修正に柔軟に対応することができる．

しかし，プロセッサ上のソフトウエアでの実行は，専用ハードウエアで動作させるより，一般には，速度面では遅いものとなる．

専用ハードウエアの持つ高速性と，プロセッサ上のソフトウエアの構成による柔軟性を併せ持つ環境として，FPGA が最近では利用されるようになってきている．

FPGA は，論理回路を構成するするが，その論理回路はソフトウエアのように書き換え可能であり，論理回路の構成によりプロセッサ上でソフトウエアを実行するよりシステムとして速く動作できるとともに，必要に応じて，構成する論理回路を変更可能となっている．

最近では，FPGA の大規模化により，FPGA 内にプロセッサコアを構成することが可能となっている．FPGA 内に構成されるプロセッサコアとしては，あらかじめ埋め込みハードウエアとして搭載されているものと，ハードウエア記述言語で書かれた回路を FPGA

内に動的に構成するソフトプロセッサがある．ソフトプロセッサは，一般に，ハードウエア埋め込み型より低速であるが，回路構成を変更することでFPGA 内の資源の最適な活用，周辺回路との柔軟な接続性があるといった利点がある．

アプリケーション毎に FPGA の限られた資源を最適に活用するために，アプリケーションに応じて最適なプロセッサを自動的に構成することができれば，FPGA 回路設計の効率化に寄与することができる．

1.2 研究方法

本研究は次のように行った．まず，Verilog HDL で，MIPS アーキテクチャの CPU を設計する．実装する命令は [1] の MIPS リファレンスデータのコア命令セットの命令を実装する．さらに，アプリケーションをコンパイルした際には，コンパイラは他の命令も出力するため，アプリケーションで使用される命令を追加で実装する．

実装した CPU に対し，テストプログラムを実行して動作を確認する．まず，アセンブリ言語で記述された各命令のテストを行う．次に，C 言語で作成したテストプログラムをコンパイラ (GCC) によりオブジェクトコードに変換し，オブジェクトコードからコンパイラのユーティリティ (`objdump`) により，命令コードを取り出したものを論理回路シミュレータ上で実行して，正しく実行されることを確認する．

次に，実装した CPU に対し命令／機能単位でのモジュール分けを行う．どの命令が，マルチプレクサ等のどの資源を使うかを命令毎に整理する．

コンフィギュレータの機能として，コンパイルされたアプリケーションプログラムから，生成された命令コードを取り出して，使われている命令の種類を調べる処理，命令に応じて，CPU のどの資源を使うかのリストを作る処理，そのリストに対応して，必要な機能を CPU の Verilog HDL のコードと対応させる処理を生成する部分を作成する．

コンフィギュレータの動作の検証のため，ソフトプロセッサコア上で動作させるアプリケーションプログラムを選定し，設計したソフトプロセッサコア上での動作を確認する．このアプリケーションプログラムに，本コンフィギュレータを適用し評価を行う．

Verilog HDL の開発環境としては，Xilinx 社の PlanAhead[13] を用いる．また，コンフィギュレータの開発には Python[12] を用いる．

1.3 本研究の貢献

限られた FPGA の資源を有効に活用するためには，CPU 上で実行するアプリケーションに応じて最適となる回路を生成する必要があるが，そのような回路の最適化を，アプリケーションがコンパイルされた命令コードを参照して，人手で行うのは労力と時間がかかる．本コンフィギュレータを用いて自動的に最適な回路を生成することにより，人手による労力と時間を削減することができる．

1.4　本論文の構成

まず，第 2 章で，本研究と関連する研究について述べる．
第 3 章では，本研究で実装した Verilog HDL による CPU の設計，フォワーディング，ストール，命令毎の CPU 資源の分類について述べる．
第 4 章では，自動最適化コンフィギュレータの構築について述べる．第 5 章では，いくつかのアプリケーションに対して，コンフィギュレータにより出力された回路を，規模，動作速度について評価する．

第2章　関連研究

2.1　FPGA 上のソフトプロセッサコア

FPGA 上のプロセッサコアとして，ユーザが作成した回路と共に FPGA 上に構成されるソフトマクロプロセッサコアと，ハードウエア的にあらかじめ FPGA 上に実装されたハードマクロプロセッサコアがある．

ハードマクロプロセッサコアの例としては，Xilinx 社の Virtex-5 上に搭載されている PowerPC 440 エンベデッドプロセッサ [15] がある．

ソフトマクロプロセッサコアの代表的なものとしては，Xilinx 社の MicroBlaze [10], Altera 社の NiosII [11] がある．それぞれ，次の表に挙げられるような特徴がある [7] [8].

表 2.1: ソフトマクロプロセッサコア

名称	MicroBlaze MCS	MicroBlaze	Nios II/e	Nios II/s	Nios II/f
メーカ	Xilinx	Xilinx	Altera	Altera	Altera
形式	32bit RISC	32bit RISC	32bit RISC	32bit RISC	32bit RISC
キャッシュ	なし	命令, データ	なし	命令	命令, データ
オプション	なし	FPU,MMU, 除算器	なし	乗算器, バレルシフタ, 分岐予測	乗算器, バレルシフタ, 分岐予測, 動的分岐予測, 除算器
費用	無償	有償	無償	有償	有償

2.2　ASIP

アプリケーションに特化した CPU 回路を構成する技術として ASIP (Application-domain Specific Instruction-set Processor) と呼ばれる技術がある [5]．ASIP は，ASIC(Application Specific Integrated circuit) の効率性と，汎用プロセッサのプログラム柔軟性の間を狙った技術である．

ASIP では，アプリケーションに対応して，プロセッサアーキテクチャの最適化を行う．

プロセッサ仕様として，

- パイプラインステージ数
- レジスタ長
- 実装する命令の種類

等を指定する．命令毎に仕様記述言語で記述し，それらの記述をデータフローグラフにし，命令毎のデータフローグラフをマージする．マージの際に必要な箇所にマルチプレクサを挿入する．このようにして，プロセッサの structure model を構成する．また，新規命令についても，定義することができる．

上記により生成されたプロセッサについて，GCC をベースにしたコンパイラと，そのプラグインによってコンパイル環境を生成する．

これらの一連の処理を行うツールとして ASIP Meister というツールが有料で提供されている．

第3章　CPU の設計

3.1　MIPS アーキテクチャの CPU 設計

本研究では，汎用 CPU をハードウエア記述言語により記述し，その構成要素を整理することを行う．そのため，組み込み用途などで広く使われている MIPS アーキテクチャ [1] の CPU コアを Verilog HDL で実装することとした．この CPU コアは，5 段パイプラインの構成とし，IF ステージ (命令フェッチ), ID ステージ (命令デコード), EX ステージ (命令実行), MEM ステージ (メモリアクセス), WB ステージ (レジスタへの書き戻し) からなる．

本研究の CPU の実装では，分岐命令の分岐条件の判断，分岐時の PC (プログラムカウンタ レジスタ) の設定は ID ステージで行うこととした．また，各分岐命令，ジャンプ命令は，ディレイスロットを 1 つ持つ．すなわち，分岐命令，ジャンプ命令直後の命令は分岐するかどうかにかかわらず，1 命令実行される．これにより，条件分岐する場合でも，パイプラインをフラッシュする必要がない．jump and link 命令 (`jal`，`jalr`) では，ジャンプ後の戻り先はディレイスロットの後とするため，PC+8 番地となる．

[1] の命令リファレンスにある CPU コア命令セットのうちマルチコア用の命令である load linked 命令 (`ll`) と store conditional 命令 (`sc`) を除くすべての命令と，その他に，アプリケーションプログラムをコンパイルした際に，コンパイラ (GCC) が出力する命令を実装した．実際に実装した命令は次の表 3.1 で示される命令である．

表 3.1: 命令セット

算術命令，論理命令	add, addu, addi, addiu, and, andi, nor, or, ori, sub, subu, xor, xori
定数操作命令	lui
比較命令	slt, sltu, slti, sltiu
シフト命令	sll, sra, srl, sllv, srav, srlv
分岐命令	beq, bne, bgez, bltz, blez, bgtz
ジャンプ命令	j, jal, jalr, jr
ロード命令	lb, lbu, lh, lhu, lw
ストア命令	sb, sh, sw
乗算命令	mult, multu
除算命令	div, divu
レジスタ転送命令	mfhi, mflo

また，乗算命令，除算命令を実装するにあたり，その結果を格納するレジスタとして，HI レジスタ，LO レジスタを実装した．他に，CPU 回路の生成で回路が追加されるわけではないが，後述する疑似命令が存在する．各命令の詳細は [2] を参照．

3.1.1 命令形式

MIPS アーキテクチャの命令形式は，次のように，R 形式, I 形式, J 形式から成る．

- R 形式

  R 形式の命令は図 3.1 のような形式となる．[31:26] ビットに，命令コード 0 が入り，その命令の機能は [5:0] ビットの funct で示される．使用するレジスタは，[25:21] ビット に rs レジスタ, [20:16] ビット に rt レジスタ, [15:11] ビット に rd レジスタのレジスタ番号 (0 〜 31) が割り当てられる．

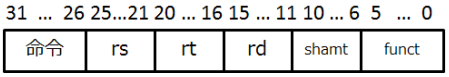


図 3.1: R 形式

ビット	内容
31:26 ビット	命令コード
25:21 ビット	rs レジスタ
20:16 ビット	rt レジスタ
15:11 ビット	rd レジスタ
10:6 ビット	シフト量
5:0 ビット	funct

使用する資源の観点から，R 形式の命令をさらに表 3.2 のように分類する．

表 3.2: R 形式の分類

形式	内容	命令
R	funct ビット [5:0] により命令の種類を識別	add など
R2	funct ビット [5:0] により命令の種類を識別し，シフト量ビット [10:6] を使用	sll など
M1	乗算器を使用	mult, multu
D1	除算器を使用	div, divu
MH	HI レジスタからの転送	mfhi
ML	LO レジスタからの転送	mflo
J3	rs の示すアドレスにジャンプし，戻り先のアドレスを `$31` に格納する	jalr
J4	rs の示すアドレスにジャンプする	jr

- I 形式

I 形式の命令は次のような形式となる．[31:26] ビットに命令コードが入り，[15:0] ビットに 16 ビットの即値が入る．使用するレジスタは，[25:21] ビット に rs レジスタ, [20:16] ビット に rt レジスタのレジスタ番号 (0 〜 31) が割り当てられる．

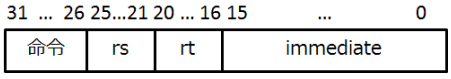


図 3.2: I 形式

ビット	内容
31:26 ビット	命令コード
25:21 ビット	rs レジスタ
20:16 ビット	rt レジスタ
15:0 ビット	immediate

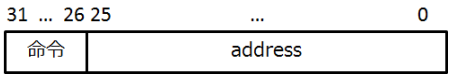
使用する資源の観点から，I 形式の命令をさらに表 3.3 のように分類する．

表 3.3: I 形式の分類

形式	内容	命令
I	16 ビットの即値を符号拡張して使用する	addi など
I2	16 ビットの即値をゼロ拡張して使用する	andi など
I3	16 ビットの即値を上位の 16 ビットにシフトして使用する	lui
L1	16 ビットの即値を符号拡張してアドレス計算に使用してメモリからワードデータをロード	lw
L2	16 ビットの即値を符号拡張してアドレス計算に使用してメモリからハーフワードデータをロード	lh, lhu
L3	16 ビットの即値を符号拡張してアドレス計算に使用してメモリからバイトデータをロード	lb, lbu
S1	16 ビットの即値を符号拡張してアドレス計算に使用してメモリへデータをストア	sw, sh, sb
B1	16 ビットの即値を符号拡張してアドレス計算に使用し, rs, rt を比較して分岐	beq, bne
B2	16 ビットの即値を符号拡張してアドレス計算に使用し, rs を使用して分岐 (>= の比較)	bgez, bltz
B3	16 ビットの即値を符号拡張してアドレス計算に使用し, rs を使用して分岐 (<= の比較)	blez, bgtz

- J 形式

  J 形式の命令は次のような形式となる．[31:26] ビットに命令コードが入り，[25:0] ビットにジャンプ先のアドレスが入る．



ビット	内容
31:26 ビット	命令コード
25:0 ビット	アドレス

図 3.3: J 形式

使用する資源の観点から，J 形式の命令をさらに，表 3.4 のように分類する．

表 3.4: J 形式の分類

形式	内容	命令
J1	[25:0] ビットのアドレスにジャンプする	j
J2	戻りアドレスを `$31` レジスタに格納し，[25:0] ビットのアドレスにジャンプする	jal

- 疑似命令

  コンパイラはアセンブリ言語のコードとして，次の命令を出力する．これらの命令は，CPU アーキテクチャに新しい命令を追加するものではないが，本コンフィギュレータで，使用命令を解析する際には，考慮する必要があり，表 3.5 の命令を解析可能とする．

表 3.5: 疑似命令

疑似命令	説明	実際の命令
move	Rsrc から Rdest へのレジスタの値を転送	addu
li	即値をレジスタへロード	addiu
nop	何もしない	sll
beqz	rs が 0 の場合分岐	beq
bnez	rs が 0 でない場合分岐	bne

3.1.2 CPU の実装

以下にパイプラインの各ステージ毎の CPU の実装を示す.

- IF, ID ステージ
  命令のフェッチ，命令のデコード，レジスタファイルからの読み出し，分岐条件のチェックなどを行う．図 3.4 に IF, ID ステージで実装した回路を示す.

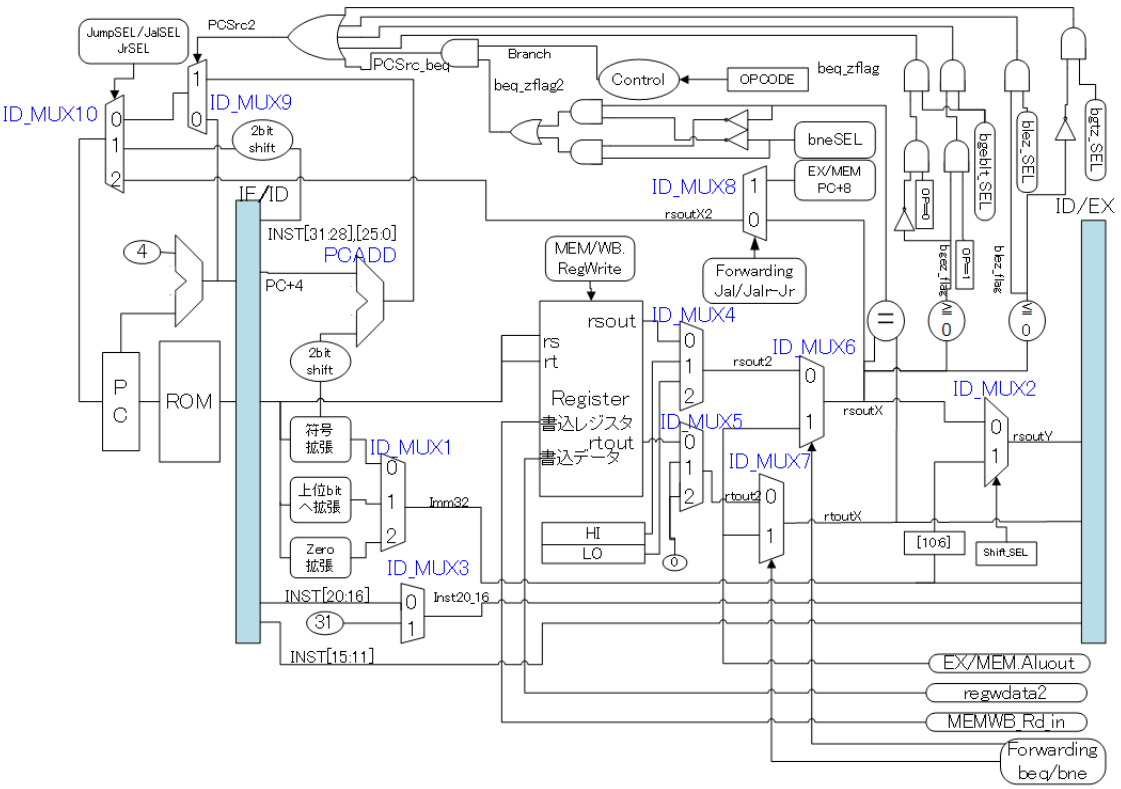


図 3.4: IF ID ステージ実装

IF, ID ステージでは表 3.6 のように資源を分類する.

表 3.6: IF, ID ステージの資源

資源名	内容
ID_MUX1	I 形式命令で用いる即値の形式の選択
ID_MUX2	ALU の第一入力に rs レジスタ値を用いるかシフト量を用いるかの選択
ID_MUX3	rd レジスタ番号として `$31` を用いるかの選択
ID_MUX4	rs レジスタ値, HI, LO レジスタ値を用いるかの選択
ID_MUX5	rt レジスタ値または 0 の選択
ID_MUX6	rs レジスタの値として EX ステージからフォワードされた値を用いるかの選択
ID_MUX7	rt レジスタの値として EX ステージからフォワードされた値を用いるかの選択
ID_MUX8	ジャンプ先の番地として，PC+8 を用いるか，rs レジスタ値を用いるかの選択
ID_MUX9	次の PC として，PC+4 か，即値フィールドの値を用いるかの選択
ID_MUX10	次の PC として用いる値の選択
PCADD	即値を用いて分岐する際に，相対アドレスとするための加算
=	`beq`，`bne` 命令での条件分岐のための比較
>=	`bgez`，`bltz` 命令での条件分岐のための比較
<=	`blez`，`bgtz` 命令での条件分岐のための比較

- EX ステージ
  演算器での演算などを行う．図 3.5 に EX ステージで実装した回路を示す．

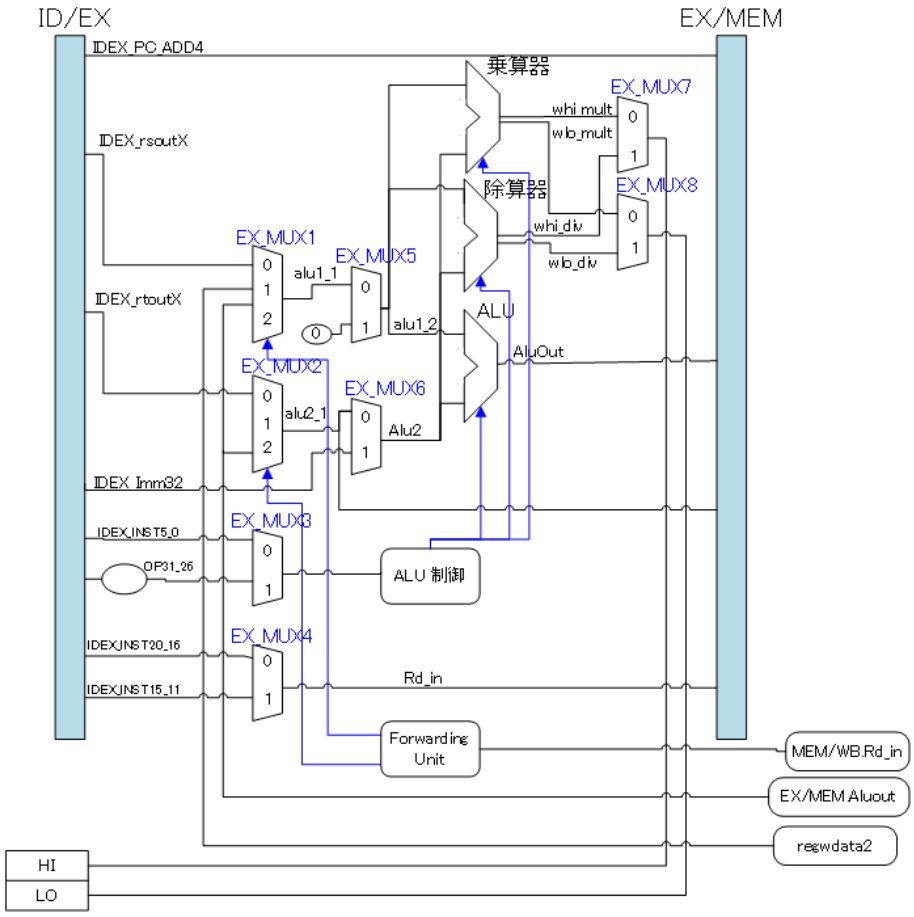


図 3.5: EX ステージ実装

EX ステージでは表 3.7 のように資源を分類する.

表 3.7: EX ステージの資源

資源名	内容
EX_MUX1	rs レジスタの ALU 入力値を EX ステージ，WB ステージからフォワーディングするかの選択
EX_MUX2	rt レジスタの ALU 入力値を EX ステージ，WB ステージからフォワーディングするかの選択
EX_MUX3	ALU 制御に [5:0] ビットの値を使うか [31:26] ビットの値を使うかの選択
EX_MUX4	rd レジスタのレジスタ番号として [20:16] ビットの値を使うか，[15:11] ビットの値を使うかの選択
EX_MUX5	rs レジスタの値として 0 を使うかの選択
EX_MUX6	rt レジスタの値としてレジスタ値を使うか，即値を使うかの選択
EX_MUX7	HI レジスタの値として乗算器の出力を使うか，除算器の出力を使うかの選択
EX_MUX8	LO レジスタの値として乗算器の出力を使うか，除算器の出力を使うかの選択
乗算器	乗算命令で使用
除算器	除算命令で使用

- MEM ステージ
  メモリからの読み出し，書き込みを行う．図 3.6 に MEM ステージで実装した回路を示す．

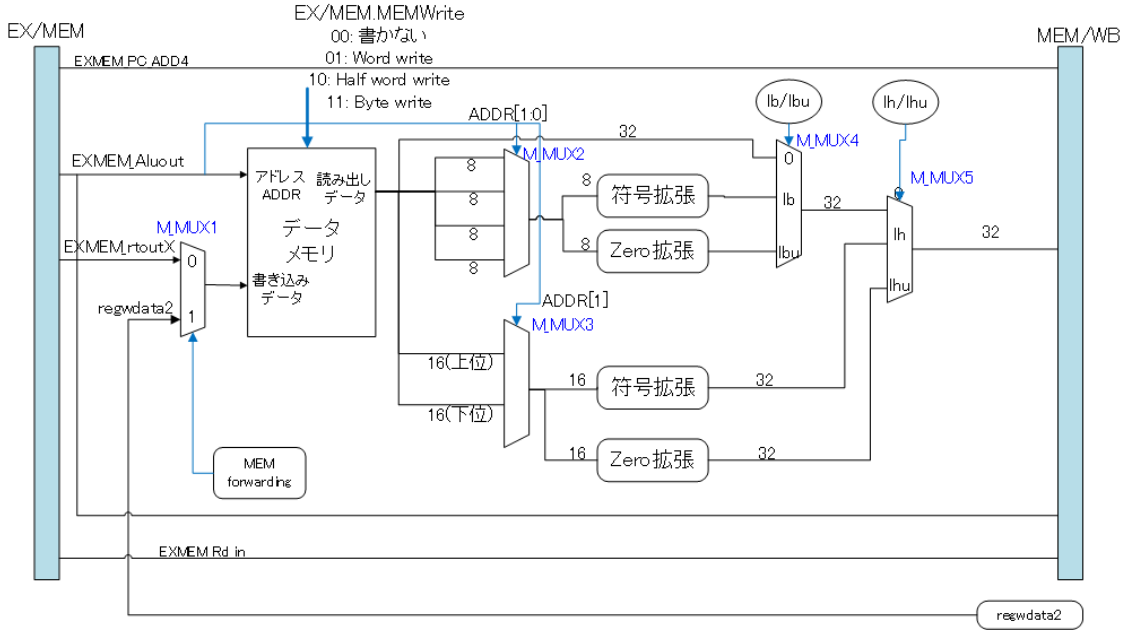


図 3.6: MEM ステージ実装

MEM ステージでは表 3.8 のように資源を分類する．

表 3.8: MEM ステージの資源

資源名	内容
MEM_MUX1	書き込みデータとして WB ステージからフォワーディングされた値を使うかを選択
MEM_MUX2	`lb` 命令でのバイト位置の選択
MEM_MUX3	`lh` 命令でのハーフワード位置の選択
MEM_MUX4	`lb` か `lbu` かの選択
MEM_MUX5	`lh` か `lhu` かの選択

- WB ステージ
  レジスタへの書き込みを行う． 図 3.7 に WB ステージで実装した回路を示す．

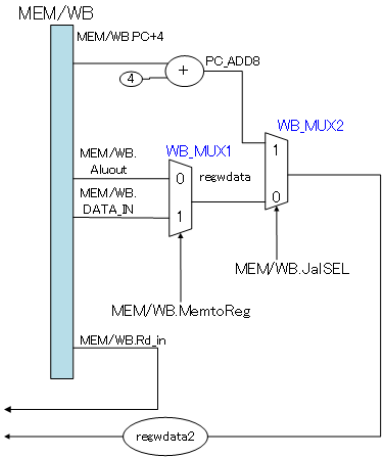


図 3.7: WB ステージ実装

WB ステージでは表 3.9 のように資源を分類する．

表 3.9: WB ステージの資源

資源名	内容
WB_MUX1	ALU 演算結果かメモリリード値かの選択
WB_MUX2	書き込みレジスタ値として PC+8 を用いるかの選択

3.2 フォワーディング

パイプラインで構成されるプロセッサでは，前の命令の実行結果を次の命令が使う場合，その結果がレジスタに格納されてから利用できるため，直後の命令は，結果がレジスタに格納されるまで待つ必要がある．そのような場合には，パイプラインをストールさせて，前の実行の結果が利用できるようになるまで待つ必要がある．命令の組み合わせの種類によっては，そのようなストールをさせることなく，前の命令の実行結果を，レジスタに格納する前に，直接後段のステージに渡して実行することができる．このような処理をフォワーディングという．フォワーディングを行うことにより，命令の実行を遅らすことなく，実行することができる．

本 CPU の実装でフォワーディングが行われる場合を以降の節で述べる．それぞれの場合に，フォワーディングを検出するユニットを有している．

3.2.1 ALU 計算結果のフォワーディング

図 3.8 で示すように，EX ステージの ALU 計算結果を直後の命令で使用する場合にパイプラインをストールさせずに実行するために，EX ステージの実行結果を EX ステージ

へフォワーディングする．

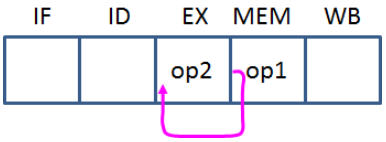


図 3.8: ALU 計算結果のフォワーディング

ALU 計算結果のフォワーディング

```
op1  $10, $8, $9  /  op1  $10, $8, imm16
op2  $11, $10, $9  /  op2  $11, $10, imm16

op1  $10, $8, $9  /  op1  $10, $8, imm16
op2  $11, $12, $10
```

op1, op2 : 上記のようなレジスタを使う適当な命令．

また，図 3.9 に示すように，2 つ前の ALU の実行結果が使用される場合もある．

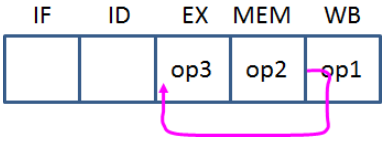


図 3.9: ALU 計算結果のフォワーディング (2 命令前)

ALU 計算結果のフォワーディング (2 つ前の結果)

```
op1  $10, $8, $9  /  op1  $10, $8, imm16
op2  ...
op3  $11, $10, $9  /  op3  $11, $10, imm16

op1  $10, $8, $9  /  op1  $10, $8, imm16
op2  ...
op3  $11, $12, $10
```

op1, op2 : 上記のようなレジスタを使う適当な命令．

上記のハザードの検出を行う Verilog HDL での回路記述は下記のようになる．

回路記述

```
if (EXMEMRegWrite && (EXMEMRd != 0) && (EXMEMRd == IDEXRs))
RForwardA = 2'b10;
else if (MEMWBRegWrite && (MEMWBRd != 0) && (MEMWBRd == IDEXRs))
RForwardA = 2'b01;
else
RForwardA = 2'b00;

if (EXMEMRegWrite && (EXMEMRd != 0) && (EXMEMRd == IDEXRt))
RForwardB = 2'b10;
else if (MEMWBRegWrite && (MEMWBRd != 0) && (MEMWBRd == IDEXRt))
RForwardB = 2'b01;
else
RForwardB = 2'b00;
```

EXMEMRegWrite : MEM ステージにあるレジスタライト信号
MEMWBRegWrite : WB ステージにあるレジスタライト信号
EXMEMRd : MEM ステージにある書き込みレジスタ番号
MEMWBRd : WB ステージにある書き込みレジスタ番号
IDEXRs : EX ステージにある Rs レジスタ番号
IDEXRt : EX ステージにある Rt レジスタ番号
RForwardA, RForwardB : マルチプレクサの選択

このフォワーディング検出条件は [1] にも述べられている.

3.2.2 分岐命令へのフォワーディング

図 3.10 で示すように，2 つ前の命令の実行結果を，分岐命令が使用する場合に，MEM ステージから，ID ステージへ実行結果をフォワーディングする必要がある．直前の命令の実行結果を，分岐命令が使用する場合には ID ステージの実行を 1 クロックストールさせる必要がある．ストールについては後述する．

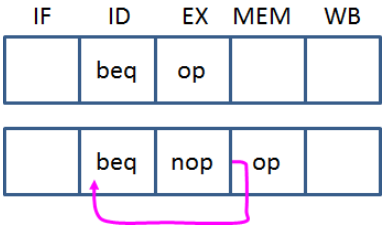


図 3.10: 分岐命令へのフォワーディング

分岐命令へのフォワーディング

```
op1 $10, $8, $9  / op1 $10, $8, imm16
op2 ...
beq $10, $11, label    : beq, bne

op1 $10, $8, $9  / op1 $10, $8, imm16
op2 ...
beq $11, $10, label    : beq, bne

op1  $10, $8, $9  / op1 $10, $8, imm16
op2
bgez $10, label  :  bgez, bgtz, blez, bltz
```

op1, op2 : 上記のようなレジスタを使う適当な命令.

上記のハザードの検出を行う Verilog HDL での回路記述は下記のようになる.

回路記述

```
if (M_Branch && M_EXEMEM_RegWrite && (M_EXEMEM_Rd != 5'b00000)
&& (M_EXEMEM_Rd == M_IFID_Rs))
RForward1 = 1'b1;
else
RForward1 = 1'b0;

if (M_Branch && M_EXEMEM_RegWrite && (M_EXEMEM_Rd != 5'b00000)
&& (M_EXEMEM_Rd == M_IFID_Rt))
RForward2 = 1'b1;
else
RForward2 = 1'b0;
```

M_Branch : ID ステージにある命令が分岐命令

M_EXEMEM_RegWrite：MEM ステージにある命令レジスタライト信号
M_EXMEM_Rd：MEM ステージにある書き込みレジスタ番号
M_IFID_Rs：ID ステージにある Rs レジスタ番号
M_IFID_Rt：ID ステージにある Rt レジスタ番号
RForward1, RForward2：マルチプレクサの選択

3.2.3 ストア命令へのフォワーディング

図 3.11 で示すように，直前の命令の実行結果を，ストア命令が使用する場合に，WB ステージから MEM ステージへ実行結果をフォワーディングする必要がある．

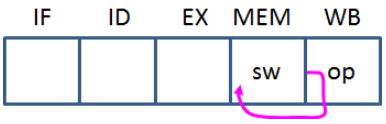


図 3.11: ストア命令へのフォワーディング

ストア命令へのフォワーディング

```
op $10, $8, $9
sw $10, offset($11)
```

op は適当な命令
上記のハザードの検出を行う Verilog HDL での回路記述は下記のようになる．

回路記述

```
assign Mout_Forward = ((M_MEMWB_RegWrite == 1'b1) & (M_Rd_in ==
M_EXMEMRt) & (M_EXMEM_MemWrite != 2'b00));
```

M_MEMWB_RegWrite：WB ステージにある命令レジスタライト信号
M_Rd_in：WB ステージにある書き込みレジスタ番号
M_EXMEMRt：MEM ステージにある Rt レジスタ番号
M_EXMEM_MemWrite：MEM ステージにあるメモリライト信号
Mout_Forward：マルチプレクサの設定

3.2.4 jal,jalr からのフォワーディング

図 3.12 で示すように，jal，jalr 命令のディレイスロットの直後の命令が戻り番地が格納されるレジスタ ($31) を使用する jr 命令の場合に，jr 命令の分岐先の設定は ID ステージで行われるため，戻り番地が格納されるレジスタ ($31) を ID ステージにフォワーディングする必要がある．

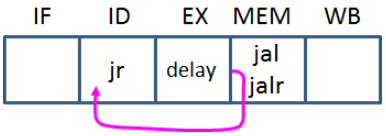


図 3.12: jal,jalr からのフォワーディング

jal,jalr からのフォワーディング

```
jal label / jalr rs  :  $31 に戻り値格納
delay slot
jr rs
```

上記のハザードの検出を行う Verilog HDL での回路記述は下記のようになる．

回路記述

```
assign J_forward = (((EXMEMopcode == 6'h3) /* jal */
| ((EXMEMopcode == 6'h0) & (EXMEMfunct == 6'h9))) /* jalr */
& ((IFIDopcode == 6'h0) & (IFIDfunct == 6'h8)) /* jr */
& (EXMEM_Rd == IFID_Rs));
```

EXMEMopcode : MEM ステージの命令の命令コード
EXMEMfunct : MEM ステージの命令の funct フィールド
IFIDopcode : ID ステージの命令の命令コード
IFIDfunct : ID ステージの命令の funct フィールド
EXMEM_Rd : MEM ステージの命令の書き込みレジスタ番号
IFID_Rs : ID ステージにある rs レジスタ番号
J_forward : マルチプレクサの設定

3.3　ストール

実行命令の依存関係により，前の命令の結果を利用できるようになるまでパイプラインをストールさせる必要がある場合がある．パイプラインをストールさせるためには，ストールを検出する回路が必要であるが，実行プログラムの出現命令を解析し，ストールする可能性がなければ，ストールを検出する回路を生成する必要がない．そのため，本コンフィギュレータでは，ストールする可能性の解析を行う．

パイプラインをストールさせる必要がある場合は次の通りである．

3.3.1　ロード命令の結果を続く命令が使う場合

図 3.13 で示されるように，ロード命令の結果を続く命令が使う際には，MEM ステージでメモリからロードした結果を EX ステージに渡すために，ID ステージの実行をストールさせる必要がある．ストールさせる必要があるのは次の場合である．

- ロード命令の直後の命令 (ストア命令以外) の rs レジスタがロード命令の結果レジスタと等しい場合．
- ロード命令の直後の命令 (ストア命令, および addi などの即値使用の算術論理演算命令以外) の rt レジスタがロード命令の結果レジスタと等しい場合．

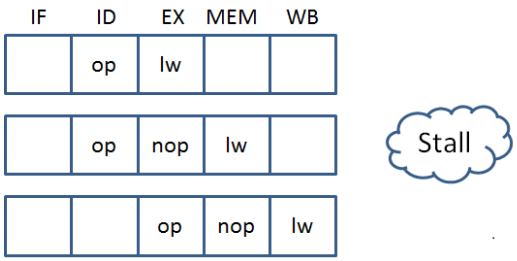


図 3.13: ロード命令によるストール

例：rs レジスタを使用

```
lw $8 offset($9)
add $9, $8, $10
```

例：rt レジスタを使用

```
lw $8 offset($9)
add $9, $10, $8
```

offset は rs に加算するオフセット値．

3.3.2　演算結果を続く分岐命令が使用する場合

図 3.14 で示されるように，EX ステージでの演算結果を，続く分岐命令が使用する場合にパイプラインをストールさせる必要がある．

(a) EX ステージの演算結果を直後の分岐命令 (beq，bne) が rs レジスタまたは rt レジスタで参照する場合．

(b) EX ステージの演算結果を直後の分岐命令 (bgez，bgtz，blez，bltz，beqz，bnez) が rs レジスタで参照する場合．

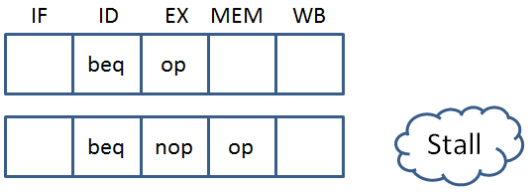


図 3.14: 演算結果を続く分岐命令が使用する場合 (op は適当な命令)

例：beq が直前の結果を使用

```
add $10, $8, $9
beq $10, $11, label
```

例：bgez が直前の結果を使用

```
add $10, $8, $9
bgez $10, label
```

3.3.3　ロード命令の結果を 2 命令後の分岐命令が使用する場合

図 3.15 で示されるように，ロード命令の結果を 2 命令後の分岐命令が使用する場合には，MEM ステージでのロード命令の結果を ID ステージで使用するために，ID ステージの実行をストールさせる必要がある (ロード命令の結果を 1 命令後の分岐命令が使用する場合は，前節の場合に含まれる)．

(a) ロード命令の演算結果を 2 命令後の分岐命令 (beq，bne) が rs レジスタまたは rt レジスタで参照する場合．

(b) ロード命令の演算結果を 2 命令後の分岐命令 (`bgez`, `bgtz`, `blez`, `bltz`, `beqz`, `bnez`) が rs レジスタで参照する場合.

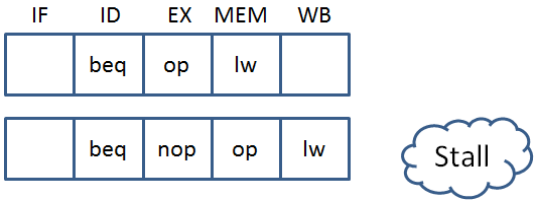


図 3.15: ロード命令の演算結果を 2 命令後の分岐命令が使用する場合

例：beq が lw の結果を使用

```
lw $10, offset($9)
op
beq $10, $11, label
```

op は適当な命令

例：bgez が lw の結果を使用

```
lw $10, offset($9)
op
bgez $10, label
```

op は適当な命令

3.3.4 直前の命令の演算結果を jr, jalr 命令が使用する場合

図 3.16 で示されるように，直前の演算結果を，`jr` 命令，`jalr` 命令で使用する場合には，EX ステージでの演算結果を ID ステージで使用するため，ID ステージにある命令の実行をストールさせる必要がある.

- 直前の演算結果を `jr` 命令が rs レジスタで参照する場合.
- 直前の演算結果を `jalr` 命令が rs レジスタで参照する場合.

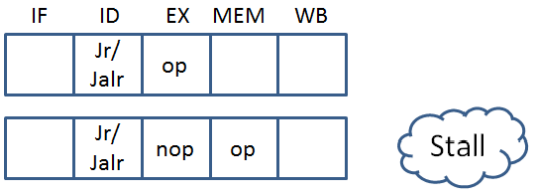


図 3.16: 直前の命令の演算結果を直後の jr, jalr 命令が使用する場合

例：jr が直前の演算結果を使用

```
add $10, $8, $9
jr  $10
```

3.3.5 2 命令前のロード命令の演算結果を jr, jalr 命令が使用する場合

図 3.17 で示されるように，ロード命令の結果を 2 命令後の `jr`，`jalr` 命令が使用する場合には，MEM ステージでのロード命令の結果を ID ステージで使用するために，ID ステージの実行をストールさせる必要がある (ロード命令の結果を 1 命令後の `jr`，`jalr` 命令が使用する場合は，前節の場合に含まれる).

- ロード命令の演算結果を 2 命令後の `jr` 命令が rs レジスタで参照する場合.
- ロード命令の演算結果を 2 命令後の `jalr` 命令が rs レジスタで参照する場合.

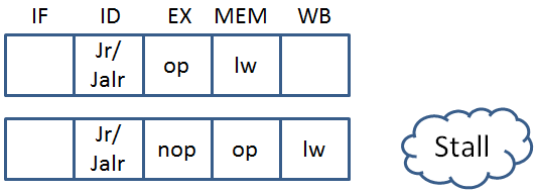


図 3.17: ロード命令の演算結果を 2 命令後の jr,jalr 命令が使用する場合

例：jr が lw の結果を使用

```
lw $10, offset($9)
op
jr $10
```

op は適当な命令

3.4　資源の分類

作成した CPU に対して，命令毎に，データパス，制御パスを整理して，使用する資源を分類する．CPU を構成する資源は，各マルチプレクサ要素，ストール検出ユニット，フォワーディング検出ユニット，条件分岐時の比較回路，乗算器，除算器等に分類する．この各々の要素に対して，回路構成を元に，各命令が実行される場合に，どの要素が使用されるかを命令毎に分類する．フォワーディング検出ユニットや，ストール検出ユニットは，アプリケーションプログラムの命令の並びによって，実際に使用されるかどうかが決まる．フォワーディング検出ユニットの使用の有無によって，関連するマルチプレクサを減らすことができる．

各パイプラインステージ毎に使用される資源は表 3.10 〜 表 3.13 のようになる．各命令で，マルチプレクサの 0 が選択される場合，1 が選択される場合，2 が選択される場合を表ではそれぞれ 0,1,2 と表す．また，その命令でマルチプレクサや，演算ユニットが使用されない場合に × で表し，使用される場合には 0,1,2 または ○ で表す．あるマルチプレクサについて，アプリケーションで使用される命令の組み合わせによって，例えば 0 側だけしか使わない場合，CPU 回路の構成では 0 側だけを配線するようにすることで，マルチプレクサを減らすことができる．また，フォワーディングについては，3.2.1 〜 3.2.4 節で説明される各フォワーディングで使用されるマルチプレクサを，節番号に対応した行に，選択されるマルチプレクサ (1,2) で表す．

表 3.10: 命令毎に使用する資源 (ID ステージ)

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5	MUX6	MUX7	MUX8	MUX9	MUX10	PCADD	=	>=	=<
add	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
addu	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
addi	I	0	0	0	0	×	0	×	×	0	0	×	×	×	×
addiu	I	0	0	0	0	×	0	×	×	0	0	×	×	×	×
and	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
andi	I2	2	0	0	0	×	0	×	×	0	0	×	×	×	×
nor	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
or	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
ori	I2	2	0	0	0	×	0	×	×	0	0	×	×	×	×
sll	R2	0	1	×	×	1	×	0	×	0	0	×	×	×	×
sllv	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
sra	R2	0	1	×	×	1	×	0	×	0	0	×	×	×	×
srav	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
srl	R2	0	1	×	×	1	×	0	×	0	0	×	×	×	×
srlv	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
sub	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
subu	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
xor	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
xori	I2	2	0	0	0	×	0	×	×	0	0	×	×	×	×
lui	I3	1	0	0	0	×	0	×	×	0	0	×	×	×	×
slt	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
sltu	R	×	0	×	0	0	0	0	×	0	0	×	×	×	×
slti	I	0	0	0	0	×	0	×	×	0	0	×	×	×	×
sltiu	I	0	0	0	0	×	0	×	×	0	0	×	×	×	×
beq	B1	×	×	×	0	0	0	0	×	1	0	○	○	×	×
bne	B1	×	×	×	0	0	0	0	×	1	0	○	○	×	×
bgez	B2	×	×	×	0	×	0	×	×	1	0	○	×	○	×
bltz	B2	×	×	×	0	×	0	×	×	1	0	○	×	○	×
blez	B3	×	×	×	0	×	0	×	×	1	0	○	×	×	○
bgtz	B3	×	×	×	0	×	0	×	×	1	0	○	×	×	○
j	J1	×	×	×	×	×	×	×	×	×	1	×	×	×	×
jal	J2	×	×	1	×	×	×	×	×	×	1	×	×	×	×
jalr	J3	×	×	0	0	×	0	×	0	×	2	×	×	×	×
jr	J4	×	×	×	0	×	0	×	0	×	2	×	×	×	×
lw	L1	0	0	0	0	×	0	×	×	0	0	×	×	×	×
lh	L2	0	0	0	0	×	0	×	×	0	0	×	×	×	×
lhu	L2	0	0	0	0	×	0	×	×	0	0	×	×	×	×
lb	L3	0	0	0	0	×	0	×	×	0	0	×	×	×	×
lbu	L3	0	0	0	0	×	0	×	×	0	0	×	×	×	×
sw	S1	0	0	×	0	0	0	0	×	0	0	×	×	×	×
sh	S1	0	0	×	0	0	0	0	×	0	0	×	×	×	×
sb	S1	0	0	×	0	0	0	0	×	0	0	×	×	×	×
mult	M1	×	0	×	0	0	0	0	×	0	0	×	×	×	×
multu	M1	×	0	×	0	0	0	0	×	0	0	×	×	×	×
div	D1	×	0	×	0	0	0	0	×	0	0	×	×	×	×
divu	D1	×	0	×	0	0	0	0	×	0	0	×	×	×	×
mfhi	MH	×	0	×	1	1	0	0	×	0	0	×	×	×	×
mflo	ML	×	0	×	2	2	0	0	×	0	0	×	×	×	×

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5	MUX6	MUX7	MUX8	MUX9	MUX10	PCADD	=	>=	=<
move	addu	×	0	×	0	0	0	0	×	0	0	×	×	×	×
li	addiu	0	0	0	0	×	0	×	×	0	0	×	×	×	×
nop	sll	0	1	×	×	1	×	0	×	0	0	×	×	×	×
beqz	beq	×	×	×	0	0	0	0	×	1	0	○	○	×	×
bnez	bne	×	×	×	0	0	0	0	×	1	0	○	○	×	×
3.2.2	–	×	×	×	×	×	1	1	×	×	×	×	×	×	×
3.2.4	–	×	×	×	×	×	×	×	1	×	×	×	×	×	×

表 3.11: 命令毎に使用する資源 (EX ステージ)

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5	MUX6	MUX7	MUX8	ALU	乗算器	除算器
add	R	0	0	0	1	0	0	×	×	add	×	×
addu	R	0	0	0	1	0	0	×	×	add	×	×
addi	I	0	×	1	0	0	1	×	×	add	×	×
addiu	I	0	×	1	0	0	1	×	×	add	×	×
and	R	0	0	0	1	0	0	×	×	and	×	×
andi	I2	0	×	1	0	0	1	×	×	and	×	×
nor	R	0	0	0	1	0	0	×	×	nor	×	×
or	R	0	0	0	1	0	0	×	×	or	×	×
ori	I2	0	×	1	0	0	1	×	×	or	×	×
sll	R2	0	0	0	1	0	0	×	×	sll	×	×
sllv	R	0	0	0	1	0	0	×	×	sll	×	×
sra	R2	0	0	0	1	0	0	×	×	sra	×	×
srav	R	0	0	0	1	0	0	×	×	sra	×	×
srl	R2	0	0	0	1	0	0	×	×	srl	×	×
srlv	R	0	0	0	1	0	0	×	×	srl	×	×
sub	R	0	0	0	1	0	0	×	×	sub	×	×
subu	R	0	0	0	1	0	0	×	×	sub	×	×
xor	R	0	0	0	1	0	0	×	×	xor	×	×
xori	I2	0	×	1	0	0	1	×	×	xor	×	×
lui	I3	×	×	1	0	1	1	×	×	nor	×	×
slt	R	0	0	0	1	0	0	×	×	slt	×	×
sltu	R	0	0	0	1	0	0	×	×	sltu	×	×
slti	I	0	×	1	0	0	1	×	×	slt	×	×
sltiu	I	0	×	1	0	0	1	×	×	sltu	×	×
beq	B1	×	×	×	×	×	×	×	×	×	×	×
bne	B1	×	×	×	×	×	×	×	×	×	×	×
bgez	B2	×	×	×	×	×	×	×	×	×	×	×
bltz	B2	×	×	×	×	×	×	×	×	×	×	×
blez	B3	×	×	×	×	×	×	×	×	×	×	×
bgtz	B3	×	×	×	×	×	×	×	×	×	×	×
j	J1	×	×	×	×	×	×	×	×	×	×	×
jal	J2	×	×	×	0	×	×	×	×	×	×	×
jalr	J3	×	×	×	0	×	×	×	×	×	×	×
jr	J4	×	×	×	×	×	×	×	×	×	×	×
lw	L1	0	×	0	0	0	1	×	×	add	×	×

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5	MUX6	MUX7	MUX8	ALU	乗算器	除算器
lh	L2	0	×	0	0	0	1	×	×	add	×	×
lhu	L2	0	×	0	0	0	1	×	×	add	×	×
lb	L3	0	×	0	0	0	1	×	×	add	×	×
lbu	L3	0	×	0	0	0	1	×	×	add	×	×
sw	S1	0	0	0	×	0	1	×	×	add	×	×
sh	S1	0	0	0	×	0	1	×	×	add	×	×
sb	S1	0	0	0	×	0	1	×	×	add	×	×
mult	M1	0	0	0	×	0	0	0	0	×	○	×
multu	M1	0	0	0	×	0	0	0	0	×	○	×
div	D1	0	0	0	×	0	0	1	1	×	×	○
divu	D1	0	0	0	×	0	0	1	1	×	×	○
mfhi	MH	0	0	0	1	0	0	×	×	add	×	×
mflo	ML	0	0	0	1	0	0	×	×	add	×	×
move	addu	0	0	0	1	0	0	×	×	add	×	×
li	addiu	0	×	1	0	0	1	×	×	add	×	×
nop	sll	0	0	0	1	0	0	×	×	sll	×	×
beqz	beq	×	×	×	×	×	×	×	×	×	×	×
bnez	bne	×	×	×	×	×	×	×	×	×	×	×
3.2.1 1 つ前	–	2	2	×	×	×	×	×	×	×	×	×
3.2.1 2 つ前	–	1	1	×	×	×	×	×	×	×	×	×

表 3.12: 命令毎に使用する資源 (MEM ステージ)

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5
add	R	×	×	×	×	×
addu	R	×	×	×	×	×
addi	I	×	×	×	×	×
addiu	I	×	×	×	×	×
and	R	×	×	×	×	×
andi	I2	×	×	×	×	×
nor	R	×	×	×	×	×
or	R	×	×	×	×	×
ori	I2	×	×	×	×	×
sll	R2	×	×	×	×	×
sllv	R	×	×	×	×	×
sra	R2	×	×	×	×	×
srav	R	×	×	×	×	×
srl	R2	×	×	×	×	×
srlv	R	×	×	×	×	×
sub	R	×	×	×	×	×
subu	R	×	×	×	×	×
xor	R	×	×	×	×	×
xori	I2	×	×	×	×	×
lui	I3	×	×	×	×	×
slt	R	×	×	×	×	×
sltu	R	×	×	×	×	×

命令 Forwarding Stall	種類	MUX1	MUX2	MUX3	MUX4	MUX5
slti	I	×	×	×	×	×
sltiu	I	×	×	×	×	×
beq	B1	×	×	×	×	×
bne	B1	×	×	×	×	×
bgez	B2	×	×	×	×	×
bltz	B2	×	×	×	×	×
blez	B3	×	×	×	×	×
bgtz	B3	×	×	×	×	×
j	J1	×	×	×	×	×
jal	J2	×	×	×	×	×
jalr	J3	×	×	×	×	×
jr	J4	×	×	×	×	×
lw	L1	×	×	×	0	0
lh	L2	×	×	○	×	1
lhu	L2	×	×	○	×	2
lb	L3	×	○	×	1	0
lbu	L3	×	○	×	2	0
sw	S1	0	×	×	×	×
sh	S1	0	×	×	×	×
sb	S1	0	×	×	×	×
mult	M1	×	×	×	×	×
multu	M1	×	×	×	×	×
div	D1	×	×	×	×	×
divu	D1	×	×	×	×	×
mfhi	MH	×	×	×	×	×
mflo	ML	×	×	×	×	×
move	addu	×	×	×	×	×
li	addiu	×	×	×	×	×
nop	sll	×	×	×	×	×
beqz	beq	×	×	×	×	×
bnez	bne	×	×	×	×	×
3.2.3	–	1	×	×	×	×

表 3.13: 命令毎に使用する資源 (WB ステージ)

命令 Forwarding Stall	種類	MUX1	MUX2
add	R	0	0
addu	R	0	0
addi	I	0	0
addiu	I	0	0
and	R	0	0
andi	I2	0	0
nor	R	0	0
or	R	0	0
ori	I2	0	0
sll	R2	0	0
sllv	R	0	0

命令 Forwarding Stall	種類	MUX1	MUX2
sra	R2	0	0
srav	R	0	0
srl	R2	0	0
srlv	R	0	0
sub	R	0	0
subu	R	0	0
xor	R	0	0
xori	I2	0	0
lui	I3	0	0
slt	R	0	0
sltu	R	0	0
slti	I	0	0
sltiu	I	0	0
beq	B1	×	×
bne	B1	×	×
bgez	B2	×	×
bltz	B2	×	×
blez	B3	×	×
bgtz	B3	×	×
j	J1	×	×
jal	J2	×	1
jalr	J3	×	1
jr	J4	×	×
lw	L1	1	0
lh	L2	1	0
lhu	L2	1	0
lb	L3	1	0
lbu	L3	1	0
sw	S1	×	×
sh	S1	×	×
sb	S1	×	×
mult	M1	×	×
multu	M1	×	×
div	D1	×	×
divu	D1	×	×
mfhi	MH	0	0
mflo	ML	0	0
move	addu	0	0
li	addiu	0	0
nop	sll	0	0
beqz	beq	×	×
bnez	bne	×	×
3.2.3	–	1	×

上述した資源に対して，Verilog HDL で記述した回路に，資源毎にマクロを作成する．

Verilog HDL で記述した回路に対して，下の例のようにあらかじめ，選択可能な資源に，Verilog HDL のコンパイラ指示子 ‘ifdef による記述を行っておく．

マルチプレクサを選択するためのマクロ例

```
/* MUX2 */
‘ifdef idmux_2_nouse
;
‘elsif idmux_2_sel0
assign rsoutY = rsoutX;
‘elsif idmux_2_sel1
assign rsoutY = Imm32_10_6;
‘elsif idmux_2_mux
MUX32 UMUX32_9(.A(rsoutX), .B(Imm32_10_6), .SEL(Shift_SEL), .Z(rsoutY));
‘else
/* ERROR */
Never Reached;
assign rsoutY = 32’h0bad2bad;
‘endif
```

これに対して，本コンフィギュレータは，次章で述べるようなマクロ定義ファイルを出力することで，最適な回路を選択することができるようになる．

第4章　自動最適化コンフィギュレータの構築

4.1　コンフィギュレータの構成

本コンフィギュレータは，図 4.1 で示されるように，コンパイラ起動処理，生成されたコードをダンプし逆アセンブルする処理，逆アセンブルされた結果から，使用命令を抽出する処理，フォワーディング，ストールをチェックする処理，これらの結果から，回路を構成するマクロ定義ファイルを出力する処理，および，コンパイル対象となるアプリケーションのソースファイルを選択して，これらの一連の処理を起動する GUI (Graphical User Interface) 部の処理から構成される．

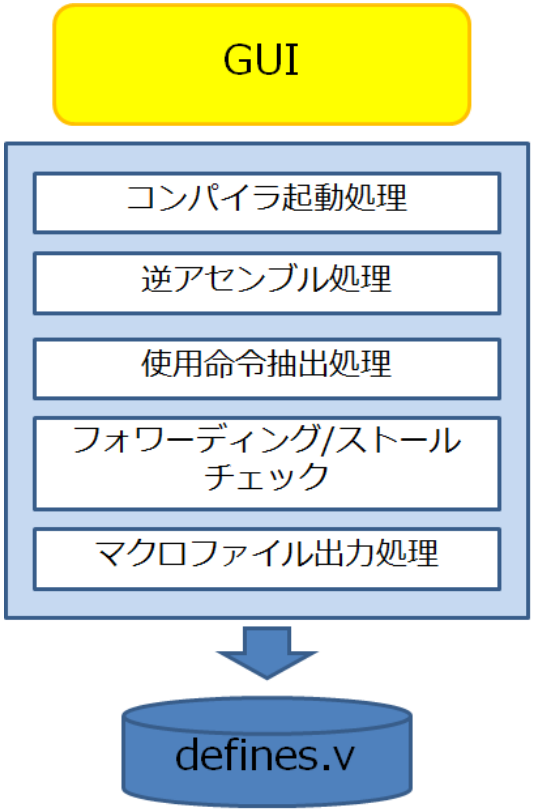


図 4.1: コンフィギュレータの構成

以下に各構成内容について述べる．

4.1.1 コンパイラ起動処理

指定されたC言語で書かれたアプリケーションのソースコードのファイルをコンパイラのコンパイル対象として，MIPS用のGCCコンパイラでコンパイルし実行形式のプログラムを生成する．

4.1.2 逆アセンブル処理

生成された実行形式のプログラムを GCC のユーティリティであるダンプコマンド (objdump) でダンプして，アセンブリ言語のコード及び，実行される命令の 16 進で表されたコードを抽出する．また，後述するフォワーディング，ストールの検出のためにラベルの情報が必要となるため，コンパイラによりアセンブリコードも生成する．

4.1.3 使用命令抽出処理

逆アセンブル結果から，使用する命令を抽出し，あらかじめ作成しておいた命令と使用する資源の一覧表から，アプリケーションプログラムで使用される資源を抽出する．

コンパイラで生成されたオブジェクトコードを，objdump コマンドで逆アセンブルすると，下の逆アセンブル例のような出力が得られる．ここから，命令コードのフィールドを抽出して，アプリケーションプログラムで使用される命令を抽出する．

逆アセンブル例

```
00000030 <main>:
  30: 27bdff58  addiu $29,$29,-168
  34: afbf00a4  sw $31,164($29)
  38: afb300a0  sw $19,160($29)
  3c: afb2009c  sw $18,156($29)
  40: afb10098  sw $17,152($29)
  44: afb00094  sw $16,148($29)
  48: 3c020000  lui $2,0x0
  4c: ac404020  sw $0,16416($2)
  50: 0c000068  jal 1a0 <SHA1Reset>
  54: 27a40010  addiu $4,$29,16
  58: 10400004  beqz $2,6c <main+0x3c>
  5c: 3c100000  lui $16,0x0
  60: 0c000008  jal 20 <error>
  64: 24040002  li $4,2
  68: 3c100000  lui $16,0x0
  6c: 0c00005a  jal 168 <strlen>
```

このような出力から，論理回路シミュレータ上で実行できるように，16 進の命令コードのフィールドを抽出する．

Verilog HDL では，組み込みファンクション `$readmemh()` により，指定されたファイルから 16 進で記述されたコードを読み込む機能がある．また，この際に `//` から始まる行はコメントとして扱われるため，上のようなアセンブリ言語のコードを下のように整形して，論理回路シミュレータ上で実行できる形式にする．

命令コードの抽出

```
//00000030 <main>:
//  30: 27bdff58  addiu $29,$29,-168
27bdff58
//  34: afbf00a4  sw $31,164($29)
afbf00a4
//  38: afb300a0  sw $19,160($29)
afb300a0
//  3c: afb2009c  sw $18,156($29)
afb2009c
//  40: afb10098  sw $17,152($29)
afb10098
//  44: afb00094  sw $16,148($29)
afb00094
//  48: 3c020000  lui $2,0x0
3c020000
//  4c: ac404020  sw $0,16416($2)
ac404020
//  50: 0c000068  jal 1a0 <SHA1Reset>
0c000068
//  54: 27a40010  addiu $4,$29,16
27a40010
//  58: 10400004  beqz $2,6c <main+0x3c>
10400004
//  5c: 3c100000  lui $16,0x0
3c100000
//  60: 0c000008  jal 20 <error>
0c000008
//  64: 24040002  li $4,2
24040002
//  68: 3c100000  lui $16,0x0
3c100000
//  6c: 0c00005a  jal 168 <strlen>
```

このように整形されたコードから，アプリケーションで使用されている命令を抽出する．

4.1.4 ディレイスロットによるフォワーディング・ストール

前述した，フォワーディングやストールは，ディレイスロットにある命令を実行した後に，ジャンプして，ジャンプ先でフォワーディングやストールが起こる可能性がある．そのため，フォワーディングやストールのチェックは，連続した命令の並びだけでなく，分岐先の命令も考慮する必要がある．

本コンフィギュレータでは，C 言語のソースコードから，コンパイラによりアセンブリ言語のコードのファイルを生成させて，ラベルのある命令を調べることにより，ディレイスロットの影響を考慮している．但し，ラベルがついている命令をすべて調べているため，実際にその命令にジャンプされない可能性もある．また，レジスタ番号の組み合わせまでは確認していないが，下に挙げる命令の組み合わせを確認している．

前章で挙げたフォワーディングの可能性がある場合に対応して，本コンフィギュレータでは，次の場合の検出を行う．

- ALU 計算結果のフォワーディング
  レジスタへ書き込む命令がディレイスロットにあり，ラベルの先頭の命令が rs または rt レジスタを参照する場合，フォワーディングを行う可能性がある．下に例を示す．

ALU 計算結果のフォワーディング (ディレイスロット)

```
        op1  $10, $8, $9  /  op1  $10, $8, imm16    //ディレイスロットにある命令
        ...

label:
        op2  $11, $10, $9  /  op2  $11, $10, imm16  //ラベルの先頭が rs または rt を参照
```

  op1,op2：適当な命令

- 分岐命令へのフォワーディング
  ディレイスロットに演算結果をレジスタに書き込む命令があり，ラベルの先頭が分岐命令である場合に，フォワーディングする可能性がある．下に例を示す．

分岐命令へのフォワーディング 1(ディレイスロット)

```
        op1 $10, $8, $9  / op1 $10, $8, imm16    //ディレイスロットにある命令
        ...
label:
        beq $10, $11, label                      //ラベルの先頭が分岐命令
```

op1：適当な命令

分岐命令へのフォワーディング 2(ディレイスロット)

```
        op1 $10, $8, $9  / op1 $10, $8, imm16    //ディレイスロットにある命令
        ...
label:
        op2
        beq $10, $11, label                      //ラベルの先頭の次の命令が分岐命令
```

op1, op2：適当な命令

- ストア命令へのフォワーディング
  ディレイスロットに演算結果をレジスタに書き込む命令があり，ラベルの先頭にストア命令がある場合に，フォワーディングする可能性がある．

ストア命令へのフォワーディング (ディレイスロット)

```
        op $10, $8, $9          //ディレイスロットにある命令
        ...
label:
        sw $10, offset($11)     //ラベルの先頭がストア命令
```

op：適当な命令

- jal,jalr からのフォワーディング
  `jal`，`jalr` 命令がディレイスロットにあることはないので，この場合は起こらない．

前章で挙げたパイプラインをストールさせる可能性がある場合に対応して，本コンフィギュレータでは，次の場合の検出を行う．

- ロード命令の結果を続く命令が使う場合

ロード命令がディレイスロットにあり，ラベルの先頭が rs または rt レジスタを参照する場合，ストールする可能性がある．下に例を示す．

ロード命令の結果を続く命令が使う場合 (ディレイスロット)

```
        lw  $8, offset($9)  //ディレイスロットにある命令
        ...

label:
        add $9, $8, $10     //ラベルの先頭が rs または rt を参照
```

- 演算結果を続く分岐命令が使用する場合
  演算結果をレジスタへ書き込む命令がディレイスロットにあり，ラベルの先頭が，beq，bne 命令や，bgez，bgtz，blez，bltz，beqz，bnez 命令の場合，ストールする可能性がある．下に例を示す．

演算結果を続く分岐命令が使用する場合 (ディレイスロット)

```
        add $10, $8, $9        //ディレイスロットにある命令
        ...

label:
        beq $10, $11, label2  //ラベルの先頭が分岐命令
```

- ロード命令の結果を 2 命令後の分岐命令が使用する場合
  ロード命令がディレイスロットにあり，ラベルの先頭の次の命令が，beq，bne 命令や，bgez，bgtz，blez，bltz，beqz，bnez 命令の場合，ストールする可能性がある．下に例を示す．

ロード命令の結果を 2 命令後の分岐命令が使用する場合 (ディレイスロット)

```
        lw $10, offset($9)    //ディレイスロットにある命令
        ...

label:
        op
        beq $10, $11, label2   //ラベルの先頭の次の命令が分岐命令
```

op は適当な命令

- 直前の命令の演算結果を直後の jr，jalr 命令が使用する場合
  演算結果をレジスタに書き込む命令がディレイスロットにあり，ラベルの先頭の命令が jr，jalr の場合, ストールする可能性がある．下に例を示す.

直前の命令の演算結果を jr, jalr 命令が使用 (ディレイスロット)

```
        add $10, $8, $9     //ディレイスロットにある命令
label:
        jr  $10             //ラベルの先頭が jr，jalr
```

- 2 命令前のロード命令の演算結果を jr，jalr 命令が使用する場合
  演算結果をレジスタに書き込む命令がディレイスロットにあり，ラベルの先頭の次の命令が jr，jalr の場合, ストールする可能性がある．下に例を示す.

2 命令前のロード命令の演算結果を jr, jalr 命令が使用 (ディレイスロット)

```
        lw $10, offset($9)      //ディレイスロットにある命令
        ...
label:
        op
        jr $10                  //ラベルの先頭の次の命令が jr，jalr
```

op は適当な命令

4.1.5 マクロファイル出力処理

アプリケーションプログラムで使用される資源を選択する Verilog HDL のマクロ定義ファイルを生成する．使用命令抽出処理およびフォワーディング・ストールチェックの解析により，実行される各命令が使用する資源に対応したマクロ定義を出力する．前章で述べたように本 CPU の実装では，選択される資源毎に‘ifdef マクロ名, ‘else, ‘endif によるコンパイル指示子が記述されており，それらのマクロ名に対応して，実際に使用される資源に対応したマクロ定義が出力される．

表 3.10 ～ 表 3.13 で示された資源に対応して，各資源が使用される場合，コンフィギュレータは,

‘define ステージ_資源番号_選択

の形式のマクロを出力する．ここで，ステージはそれぞれ,

ID ステージ	`idmux`
EX ステージ	`exmux`
MEM ステージ	`memmux`
WB ステージ	`wbmux`

で表される．また，資源番号は，マルチプレクサの番号 MUX1, MUX2, ... に対応し，ID ステージの PCADD は 11, = は 12, >= は 13, =< は 14, EX ステージの ALU は 9, 乗算器 は 10, 除算器 は 11 にそれぞれ対応する．選択については，マルチプレクサが選択されるとき `mux` または `mux3` (3 入力のとき), 資源が使用されないときは `nouse`, マルチプレクサの 0 側のみが選択されるときは `sel0`, 1 側のみが選択されるときは `sel1` など選択の組み合わせが示される．マルチプレクサの一方の側しか選択されない場合には，一方の入力が出力に直接配線されればよく，マルチプレクサを実装する必要がない．

フォワーディングの検出は，3.2 節で述べた各フォワーディングに該当する命令の並びかを調べて，該当する場合には対応するマルチプレクサを有効にするマクロを出力する．

ストールの検出回路については，3.3 節で説明されたストールの分類に対応して，該当するストールが必要な場合には，それぞれ次のようなマクロが出力される．

ストールの種類	マクロ
3.3.1	`stall1`
3.3.2 (a)	`stall2a`
3.3.2 (b)	`stall2b`
3.3.3 (a)	`stall2_2a`
3.3.3 (b)	`stall2_2b`
3.3.4	`stall4`
3.3.5	`stall5`

さらに，4.1.4 節で述べたディレイスロットによる，フォワーディング，ストールの検出を行っている．フォワーディングにおいては，表 3.10 〜 表 3.13 で示されるように，ID ステージの MUX6, MUX7, EX ステージの MUX1, MUX2, MEM ステージの MUX1 が使用される可能性がある (ID ステージの MUX8 はディレイスロットによるフォワーディングでは使われない)．これらのマルチプレクサが使用される場合には，それぞれ，次のようなマクロが出力される．

ステージ	マルチプレクサ	マクロ
ID	MUX6	idmux_6_mux_delaylabel
ID	MUX7	idmux_7_mux_delaylabel
EX	MUX1(0,1 使用) MUX1(0,2 使用) MUX1(0,1,2 使用)	exmux_1_mux_sel01_delaylabel exmux_1_mux_sel02_delaylabel exmux_1_mux_sel012_delaylabel
EX	MUX2(0,1 使用) MUX2(0,2 使用) MUX2(0,1,2 使用)	exmux_2_mux_sel01_delaylabel exmux_2_mux_sel02_delaylabel exmux_2_mux_sel012_delaylabel
MEM	MUX1	memmux_1_mux_delaylabel

さらに，ディレイスロットにより，ストールの可能性がある場合には，3.3 節で説明したストールの分類に対して，次のようなマクロが出力される．

ストールの種類	マクロ
3.3.1	stall1delayslot
3.3.2 (a)	stall2adelayslot
3.3.2 (b)	stall2bdelayslot
3.3.3 (a)	stall2_2adelayslot
3.3.3 (b)	stall2_2bdelayslot
3.3.4	stall4delayslot
3.3.5	stall5delayslot

次にマクロ定義ファイルの出力例を示す．

マクロ定義ファイル出力例

```
‘define idmux_1_mux3
‘define idmux_2_mux
‘define idmux_3_mux
‘define idmux_4_mux3
‘define idmux_5_mux3
‘define idmux_9_mux
‘define idmux_10_mux3
‘define idmux_11_sel0
‘define idmux_12_sel0
‘define idmux_13_nouse
‘define idmux_14_nouse
‘define exmux_3_mux
‘define exmux_4_mux
‘define exmux_5_mux
‘define exmux_6_mux
‘define exmux_7_sel0
‘define exmux_8_sel0
‘define exmux_9_mux
‘define exmux_10_sel0
‘define exmux_11_nouse
‘define memmux_2_nouse
‘define memmux_3_nouse
‘define memmux_4_sel0
‘define memmux_5_sel0
‘define wbmux_1_mux
‘define wbmux_2_mux
 /* forward stall check */
‘define idmux_8_mux
‘define idmux_6_mux
‘define idmux_7_mux
‘define memmux_1_sel0
‘define exmux_1_sel012
‘define exmux_2_sel02
‘define stall2a
‘define stall2_2a
‘define stall4
‘define stall5
 /* Delay slot check */
‘define memmux_1_mux_delaylabel
‘define exmux_2_sel012_delaylabel
```

このマクロ定義と，Verilog HDL で記述された CPU 回路を組み合わせて，回路の合成，インプリメントを行うことにより，アプリケーションプログラムに応じて最適な回路が得られる．

4.1.6 GUI

GUI 画面で，コンパイル対象となるアプリケーションのファイルを選択し，上記の一連の動作を実行する．

コンフィギュレータの GUI 操作画面を図 4.2 に示す．

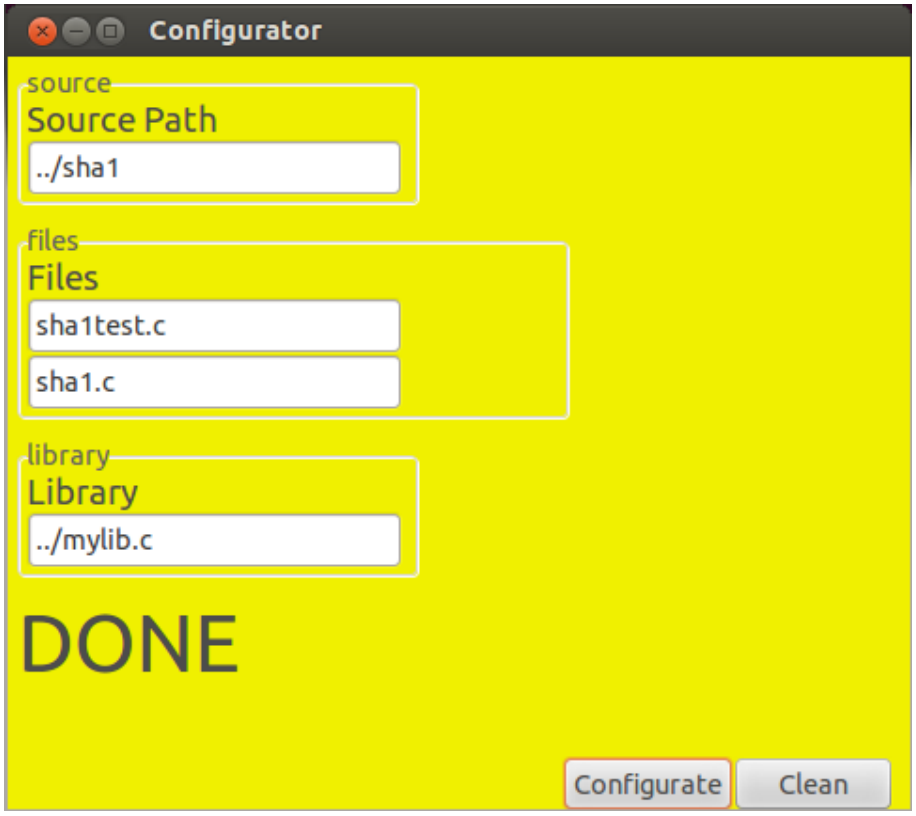


図 4.2: コンフィギュレータ GUI 画面

この画面で，アプリケーションプログラムのソースコードのファイルが置かれているディレクトリのパス，アプリケーションプログラムのソースコード，関連するライブラリファイルを指定する．これらの指定後に，Configure ボタンを押すことで，ここまでの一連の流れの動作が実行でき, 回路を構成するためのマクロ定義ファイルが生成される．また，Clear ボタンを押すことで，生成されたマクロ定義ファイルとその生成に使用された中間ファイルが削除される．

4.2 動作概要

4.2.1 実行環境

本コンフィギュレータは，Linux(Ubuntu 12.04.4_64) の上で実行される．本コンフィギュレータの開発は Python を用いた．また，CPU 回路の合成にあたっては，Xilinx 社製の開発環境 ISE Design Suite 14.7 [14] を用いて FPGA として Sprtan-6 を対象として合成およびインプリメントを行った．

アプリケーションファイルから CPU 回路の生成までは下のような流れで実行される．

- コンフィギュレータの処理
  - C 言語で書かれたアプリケーションプログラムを MIPS 用の GCC コンパイラでコンパイルし実行形式のプログラムを生成.
  - 生成された実行形式のプログラムを GCC のユーティリティである `objdump` コマンドで逆アセンブルを行う (ツールパッケージは [3] を利用). また，コンパイラによりアセンブリコードも出力する.
  - 逆アセンブル結果から，使用する命令を抽出.
  - あらかじめ作成しておいた，命令と使用する資源の一覧表から，アプリケーションプログラムで使用される資源を抽出.
  - アプリケーションプログラムで使用される資源を選択.
  - Verilog HDL のマクロ定義ファイルを生成. (Verilog HDL で記述した CPU の回路には，あらかじめ，上記マクロ定義により資源が選択できるようにされている．)
- 回路の合成
  - 上記マクロ定義ファイルを含む CPU の回路を ISE にて合成およびインプリメント.
  - ISE 付属の回路シミュレーションツールで波形を確認.
  - ISE のインプリメントレポートにより，回路規模, 実行可能周波数を確認.

図 4.3 に，実行の流れを示す.

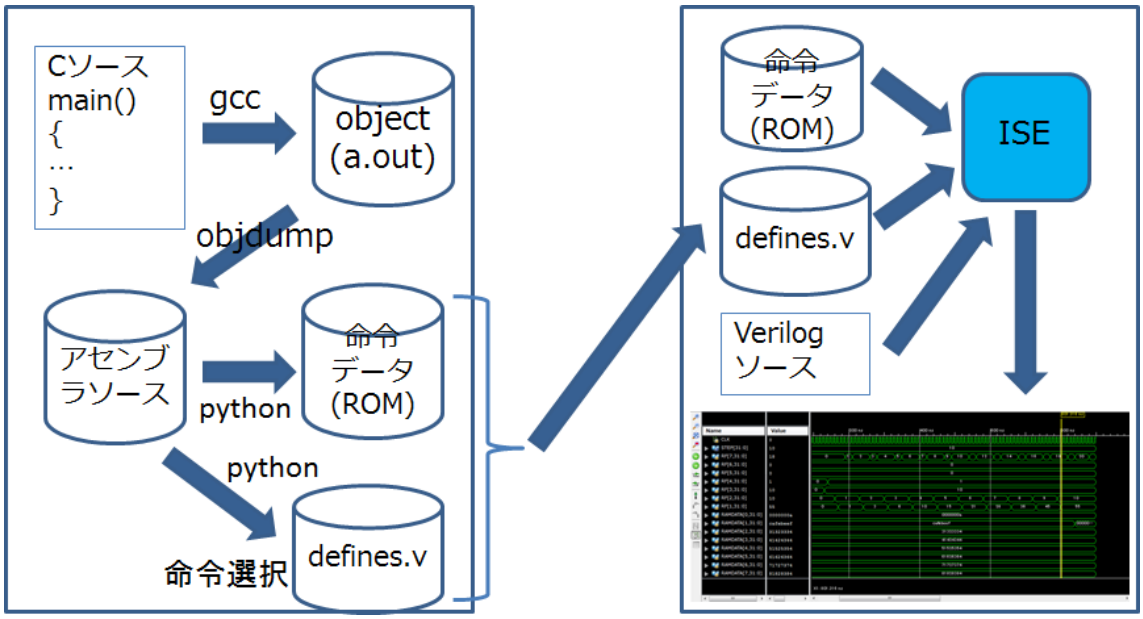

Cソース
main()
{
…
}
gcc
object
(a.out)
objdump
アセンブ
ラソース
python
命令
データ
(ROM)
python
命令選択
defines.v
命令
データ
(ROM)
defines.v
Verilog
ソース
ISE

図 4.3: 実行の流れ

第5章　評価と考察

5.1　アプリケーションの実行

C 言語で記述されたアプリケーションとして，行列積，クイックソート，SHA1 を実行した．それぞれのプログラムに対応した回路を本コンフィギュレータを用いて生成し，CPU の動作と，生成された回路の規模，速度を確認した．

5.1.1　行列積

4 行 x 4 列の行列の積を行った．行列のサイズはさらに大きいものでも実行可能であるが, 行列のサイズを増やしても出力される命令の違いはなく，生成される CPU 回路は変わらないため，実行結果の確認が可能なサイズとして 4 行 x 4 列のサイズとした．

実際に，次のような行列の積を求めて，その結果を確認した．

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{pmatrix} = \begin{pmatrix} 130 & 140 & 150 & 160 \\ 306 & 332 & 358 & 384 \\ 482 & 524 & 566 & 608 \\ 658 & 716 & 774 & 832 \end{pmatrix} \tag{5.1}$$

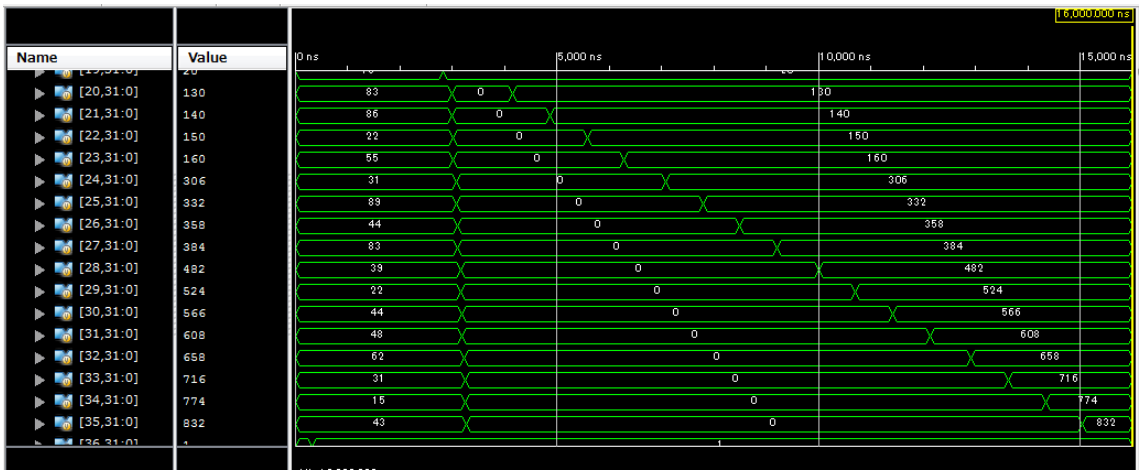


図 5.1: 演算結果

図 5.1 のように，行列積の結果 (130，140，...) がメモリに出力されていることを確認した．

5.1.2 クイックソート

C 言語で書かれたクイックソートプログラムを実行した．実行にあたっては，あらかじめ 100 個のランダムなデータを作成しておき，それらが期待通りにソートされたことを確認した．実行したクイックソートプログラムは [3] の C 言語で書かれたプログラムを使用した．

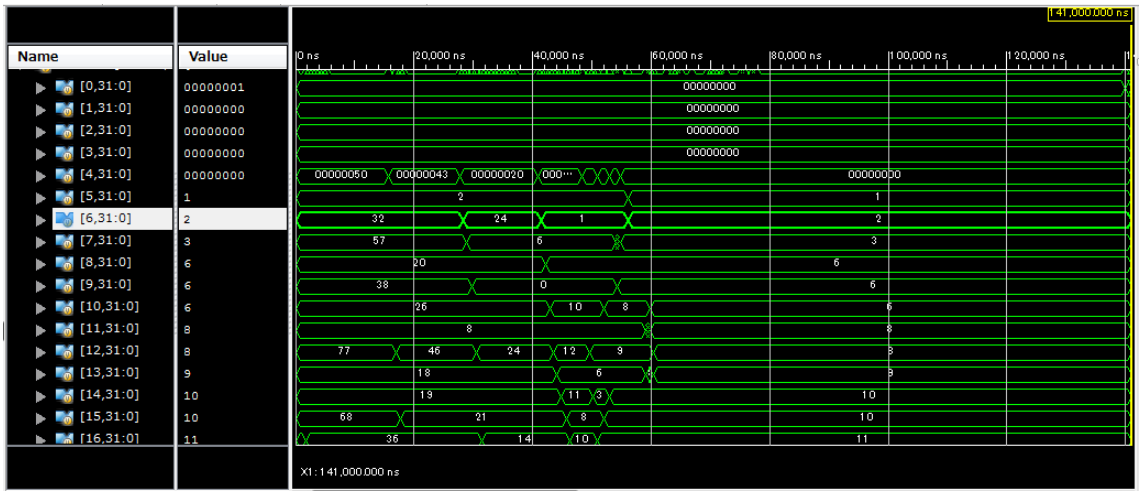


図 5.2: 演算結果

図 5.2 のように，データがソートされた結果 (1,2,3...) がメモリに出力されていることを確認した．

5.1.3 SHA1

C 言語で書かれた SHA1 プログラムを実行した．SHA1 プログラムについては [4] を使用した．

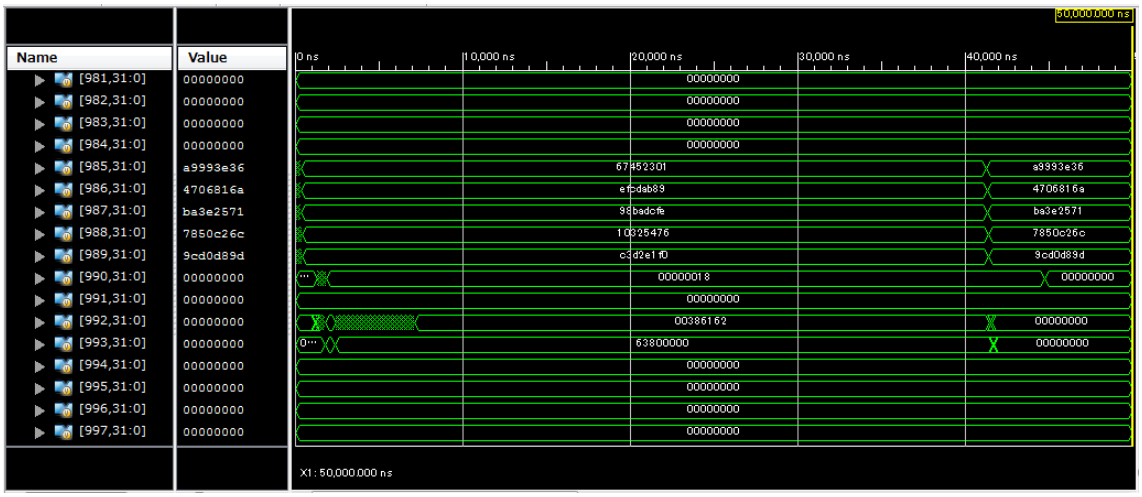


図 5.3: 演算結果

図 5.3 のように，入力値 (`"abc"`) に対して，期待された出力値 (`a9993e36`，`4706816a`，`ba3e2571`，`...`) がメモリに出力されていることを確認した．

5.2 生成回路の評価

前節の各アプリケーションに対して，本コンフィギュレータにより CPU 回路のコンフィギュレーションを行った．ISE により Sprtan-6 FPGA を対象としてインプリメントした CPU 回路の規模と，実行可能な最小周期，周波数について表 5.1 のような結果が得られた．

表 5.1: CPU インプリメント結果

	Matrix	Qsort	SHA1	Full
Register	653	589	593	738
LUT	1202	1795	1814	9269
Slice	399	555	569	3238
IO	167	167	167	167
DSP48A1	8	-	-	8
Minimum Period(ns)	13.040	11.870	10.964	13.920
Maximum Frequency(MHz)	76.687	84.246	91.358	71.839

この結果から，アプリケーションにより使用される CPU 資源が異なっており，フル実装の CPU に対して，アプリケーションで使用する資源のみを用いた回路を生成することが，回路規模，動作速度の面からも効果があることがわかった．また，アプリケーション

で使用される命令を解析して，その命令に応じた回路を自動的にツールで生成できることも確認できた．

アプリケーションプログラムで使用される命令を人手で解析して，最適な回路構成にすることは時間がかかるが，自動的に行うコンフィギュレータにより，人手をかけずに最適な回路を作成できることがわかった．

第6章　まとめ

本研究では，専用ハードウエアの持つ高速性と，プロセッサ上のソフトウエアで実現するアプリケーションの柔軟性を併せ持つ環境である FPGA を対象とした．FPGA の限られた資源を有効に活用するために，FPGA 上にソフトプロセッサコアを実現する際に，アプリケーションのプログラムが利用する命令を解析し，実際に利用される命令に対応して最適となるようなプロセッサの回路を実現することを目的とした．

プロセッサの回路を実装する際に，命令毎に利用される資源を分類し，それらの資源が選択可能なように実装した．また，コンパイラが出力する命令コードを調べて，アプリケーションで実際に利用される命令に対応した資源のみを利用するプロセッサの回路を生成するコンフィギュレータを作成することができた．

実際に，行列積，クイックソート，SHA1 のプログラムで，本研究で作成したコンフィギュレータの効果を確認することができた．

今後は，マルチプロセッサを対象として，さらに詳細な資源の分類を行い，最適な構成を自動生成するコンフィギュレータへ発展させる予定である．

参考文献

[1] パターソン, ヘネシー: コンピュータの構成と設計 上, 下, 第 4 版, 日経 BP 社, ISBN978-4-8222-8478-7, ISBN978-4-8222-8479-4.

[2] MIPS®Architecture For Programmers Volume II-A: The MIPS32®Instruction Set http://www.imgtec.com/downloads/factsheets/MD00086-2B-MIPS32BIS-AFP-05.03.pdf (2015 年 1 月 12 日閲覧)

[3] 第 2 回 ARC/CPSY/RECONF 高性能コンピュータシステム設計コンテスト アプリケーションプログラム: http://aquila.is.utsunomiya-u.ac.jp/contest/ (2014 年 12 月 27 日閲覧)

[4] SHA1: http://tools.ietf.org/html/rfc3174 (2014 年 12 月 27 日閲覧)

[5] Imai, M., Takeuchi, Y., Sakanushi, K., Ishiura, N.: Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP) IPSJ Transactions on System LSI Design Methodology Vol.3, pp.161-178, (Aug. 2010)

[6] 桜井 至: HDL 設計入門 (改定版) テクノプレス, ISBN4-924998-24-9

[7] 小林 優: FPGA ボードで学ぶ組込みシステム開発入門 [Xilinx 編] ISBN978-4-7741-5641-4

[8] 小林 優: FPGA ボードで学ぶ組込みシステム開発入門 [Altera 編] ISBN978-4-7741-4839-7

[9] 末吉 敏則, 天野 英晴 : リコンフィギャラブルシステム オーム社, ISBN4-274-20071-X

[10] MicroBlaze: http://japan.xilinx.com/tools/microblaze.htm (2015 年 1 月 17 日閲覧)

[11] Nios II: http://www.altera.co.jp/devices/processor/nios2/ni2-index.html (2015 年 1 月 17 日閲覧)

[12] Python: http://www.python.jp/ (2015 年 1 月 10 日閲覧)

[13] PlanAhead: http://japan.xilinx.com/tools/planahead.htm (2015 年 1 月 18 日閲覧)

[14] ISE アドバンスチュートリアル: http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx12_1/ise_tutorial_ug695.pdf (2015 年 1 月 18 日閲覧)

[15] Embedded Processor Block in Virtex-5 FPGAs Reference Guide: UG200 (v1.8) February 24, 2010, http://japan.xilinx.com/products/intellectual-property/ppc440_virtex5.html#documentation (2015 年 1 月 17 日閲覧)

謝辞

本研究を進めるにあたり，入念なご指導をいただきました，田中 清史准教授に心から感謝致します．また，課題研究計画提案発表で適切なご指摘をしていただきました，井口 寧教授, 金子 峰雄教授, 副テーマのご指導をいただきました青木 利晃准教授，さらに，非常に興味深い講義をしてくださった各先生方に深く感謝致します．

付録

各命令毎に利用する資源と対応づけるため，命令毎のデータパスの図を添付する．

- add 命令

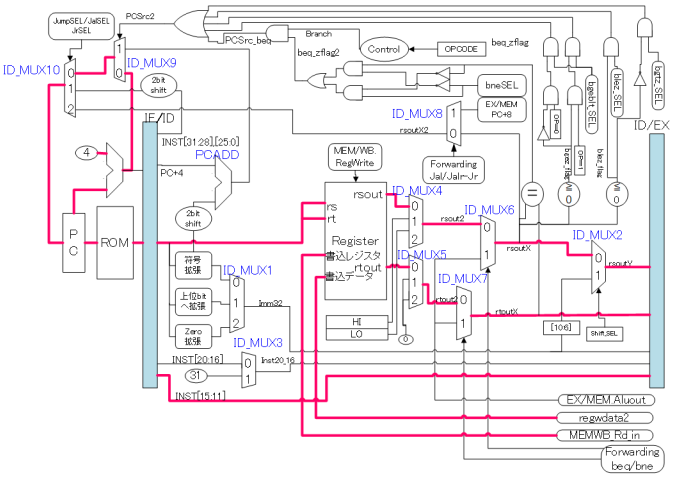


図 A.1: IF,ID ステージ

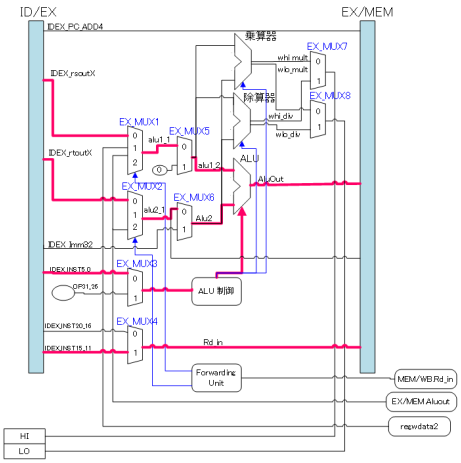


図 A.2: EX ステージ

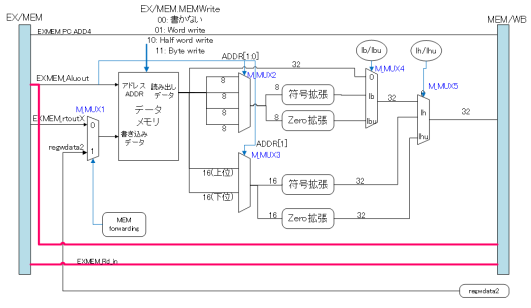


図 A.3: MEM ステージ

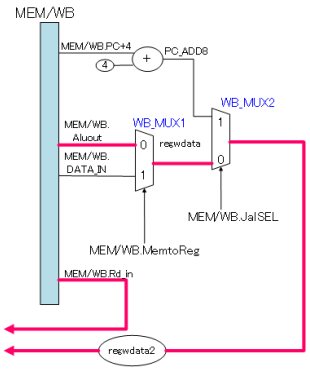


図 A.4: WB ステージ

- addi 命令

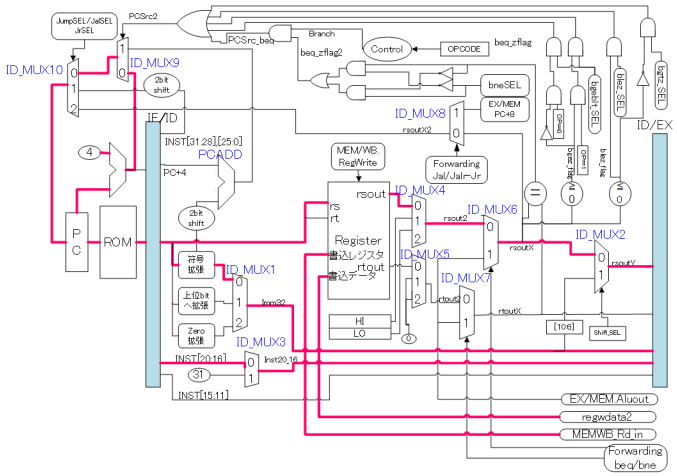


図 A.5: IF,ID ステージ

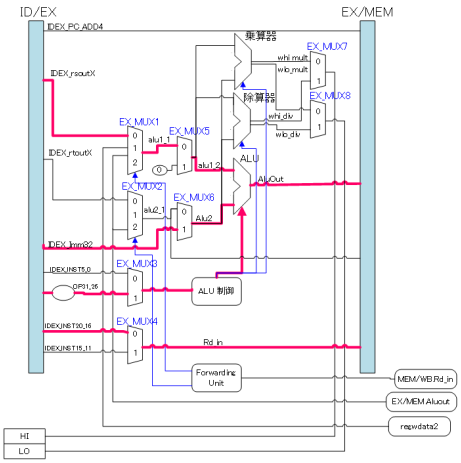


図 A.6: EX ステージ

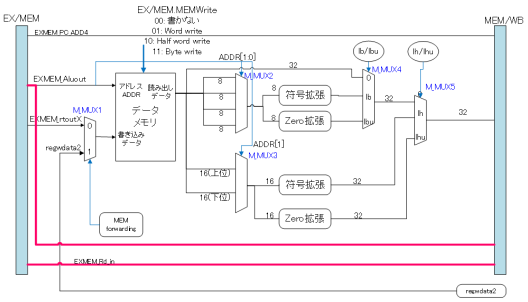


図 A.7: MEM ステージ

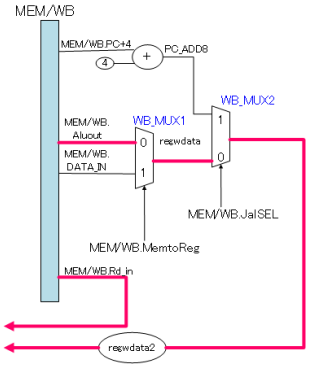


図 A.8: WB ステージ

- andi 命令

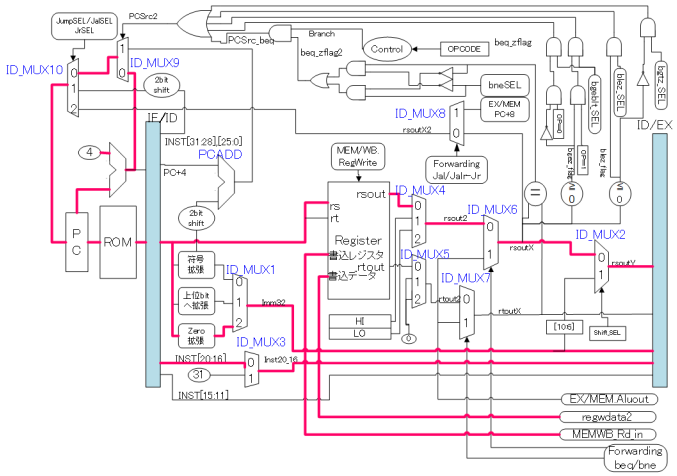


図 A.9: IF,ID ステージ

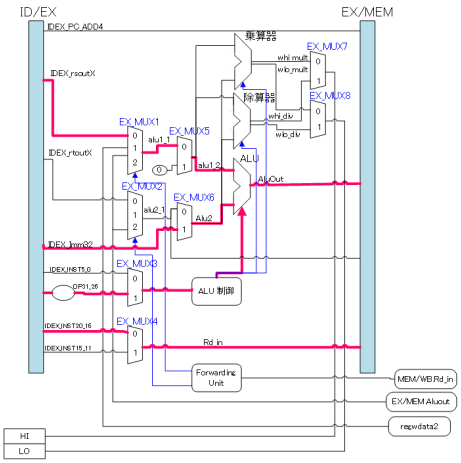


図 A.10: EX ステージ

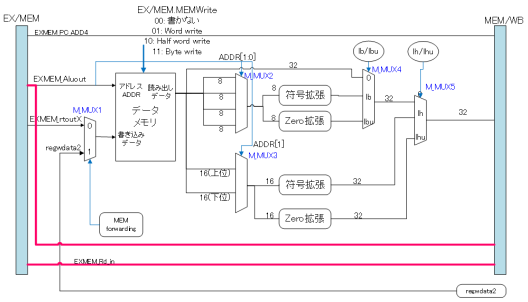


図 A.11: MEM ステージ

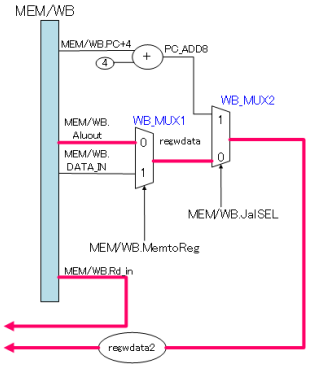


図 A.12: WB ステージ

- sll 命令

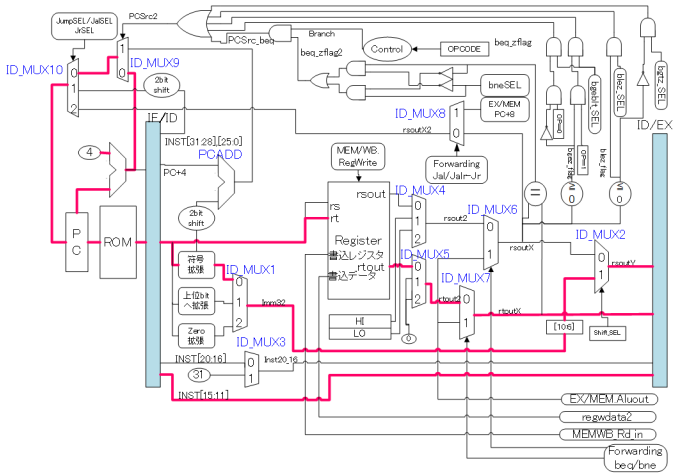


図 A.13: IF,ID ステージ

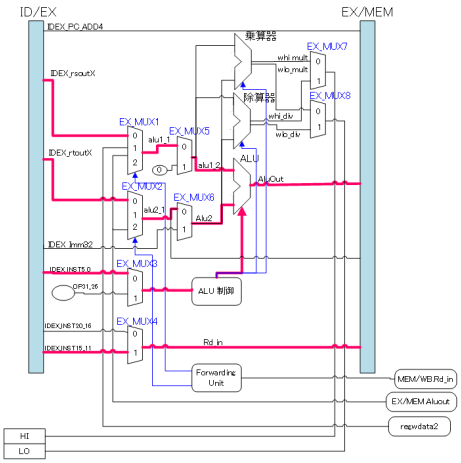


図 A.14: EX ステージ

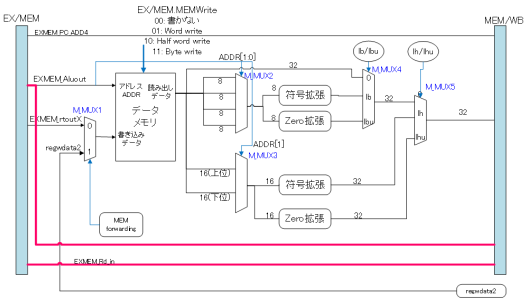


図 A.15: MEM ステージ

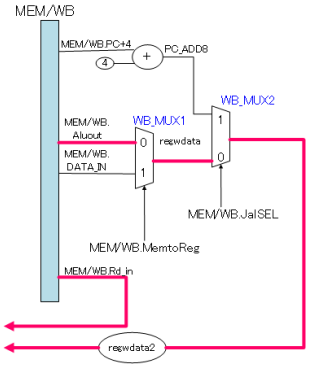


図 A.16: WB ステージ

- sllv 命令

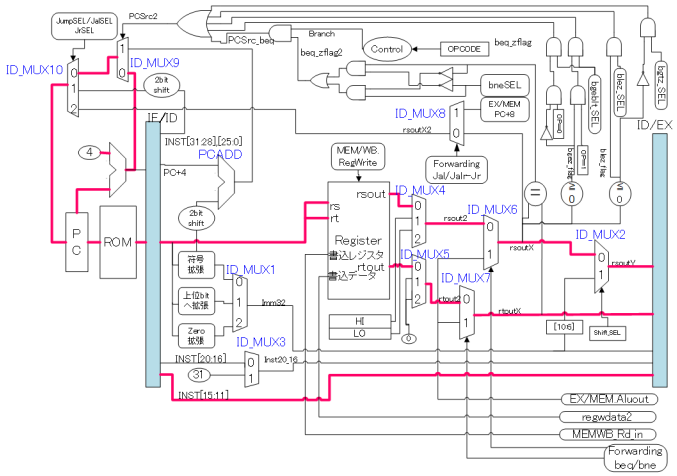


図 A.17: IF,ID ステージ

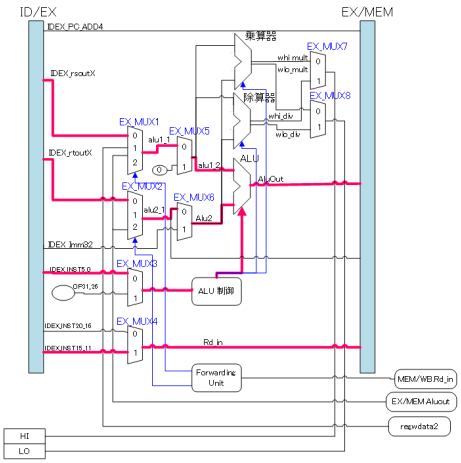


図 A.18: EX ステージ

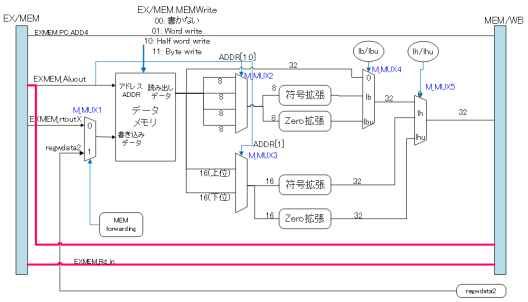


図 A.19: MEM ステージ

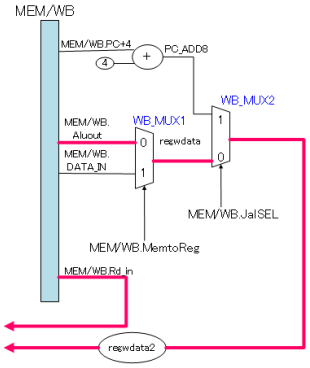


図 A.20: WB ステージ

- lui 命令

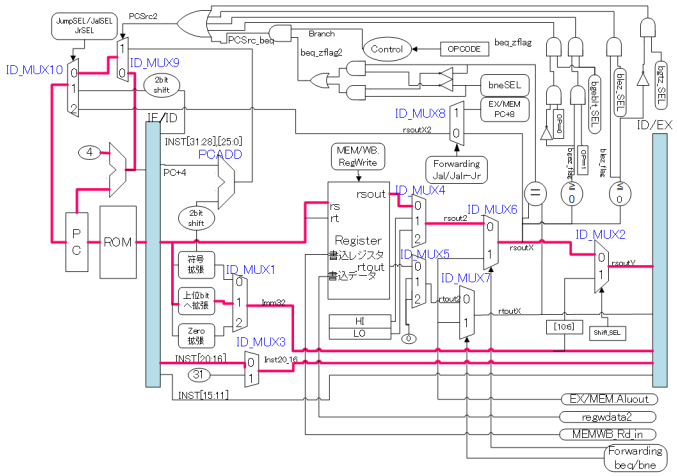


図 A.21: IF,ID ステージ

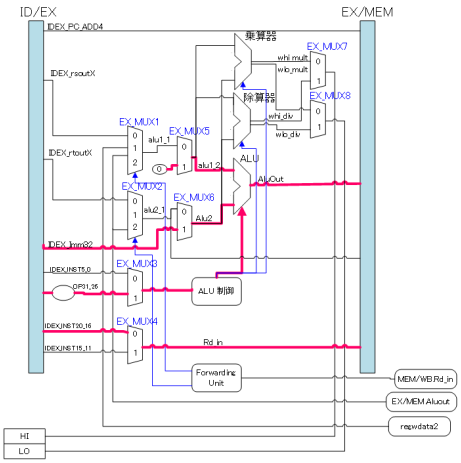


図 A.22: EX ステージ

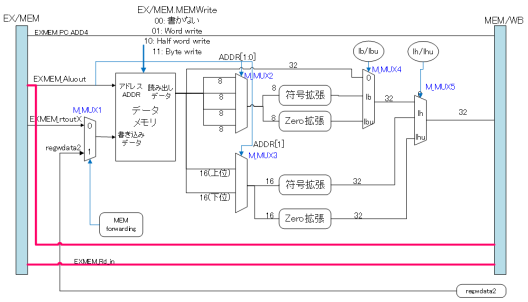


図 A.23: MEM ステージ

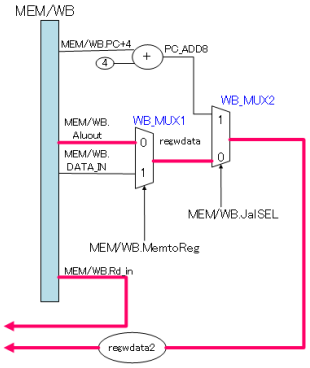


図 A.24: WB ステージ

- beq 命令

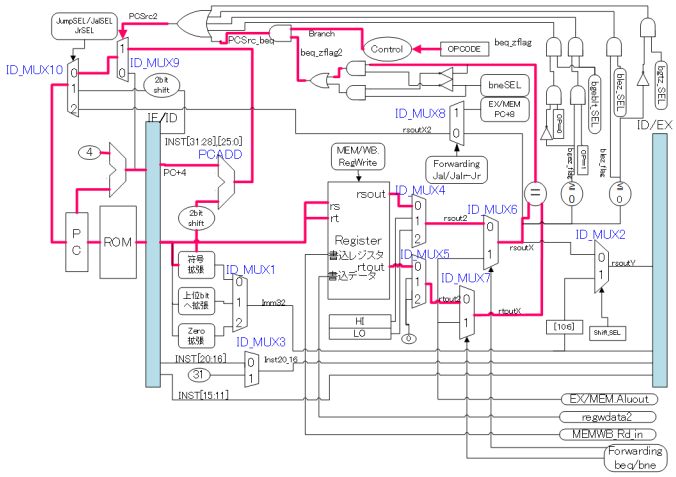


図 A.25: IF,ID ステージ

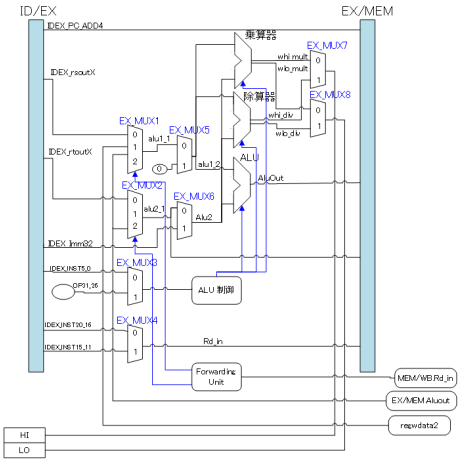


図 A.26: EX ステージ

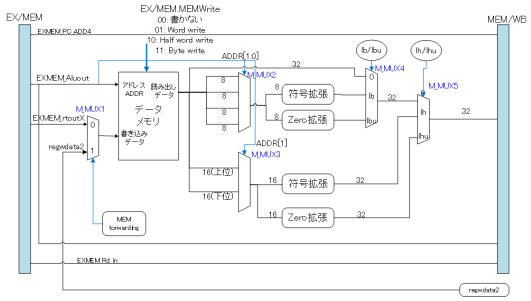


図 A.27: MEM ステージ

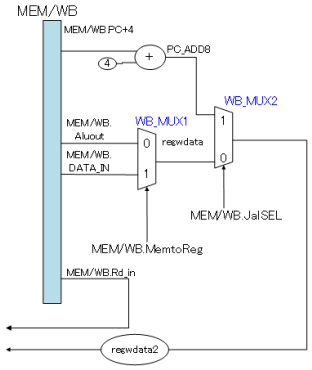


図 A.28: WB ステージ

- bgez 命令

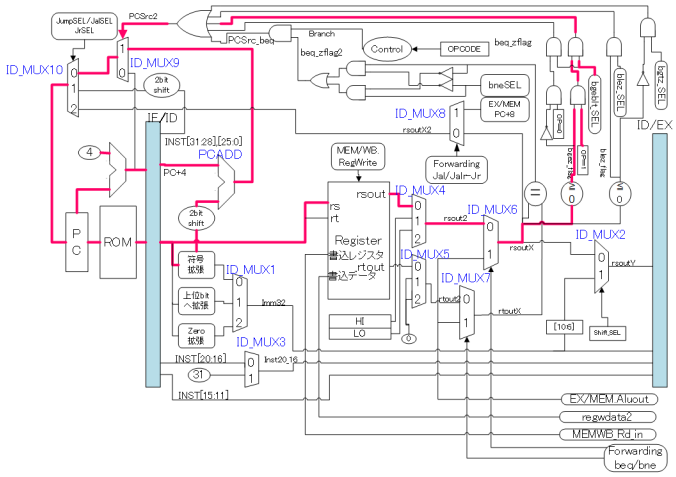


図 A.29: IF,ID ステージ

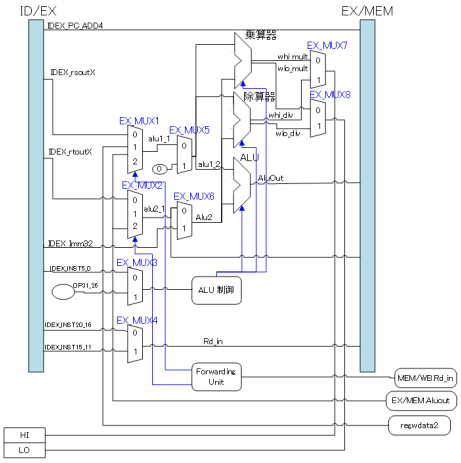


図 A.30: EX ステージ

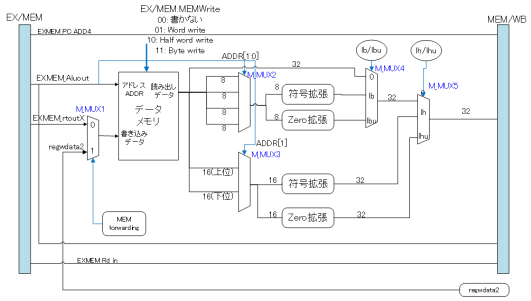


図 A.31: MEM ステージ

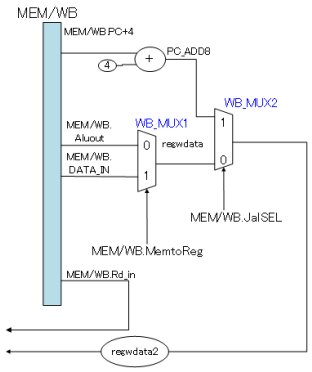


図 A.32: WB ステージ

- bltz 命令

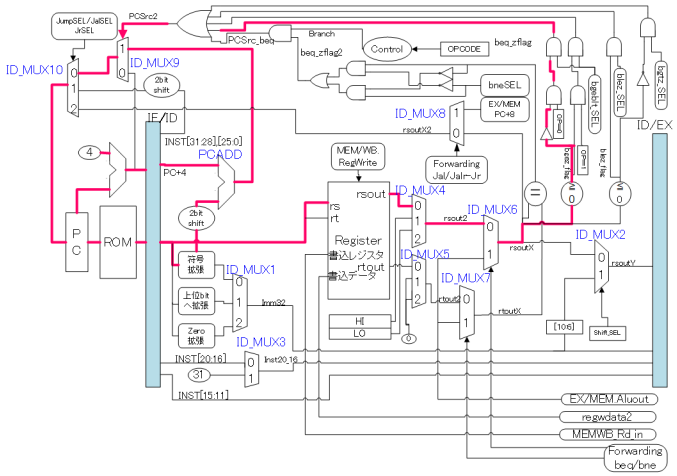


図 A.33: IF,ID ステージ

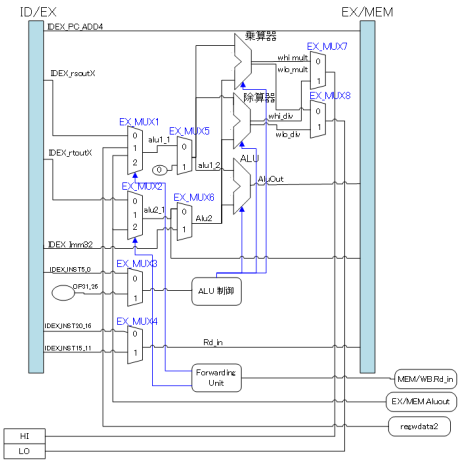


図 A.34: EX ステージ

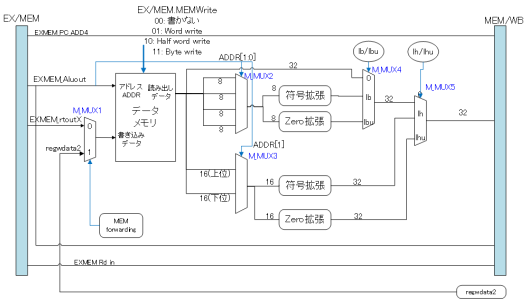


図 A.35: MEM ステージ

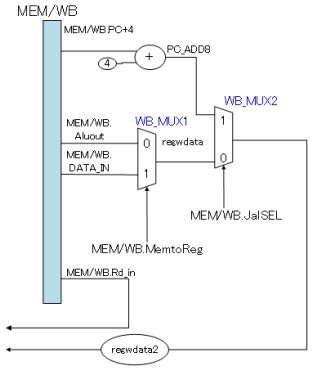


図 A.36: WB ステージ

- blez 命令

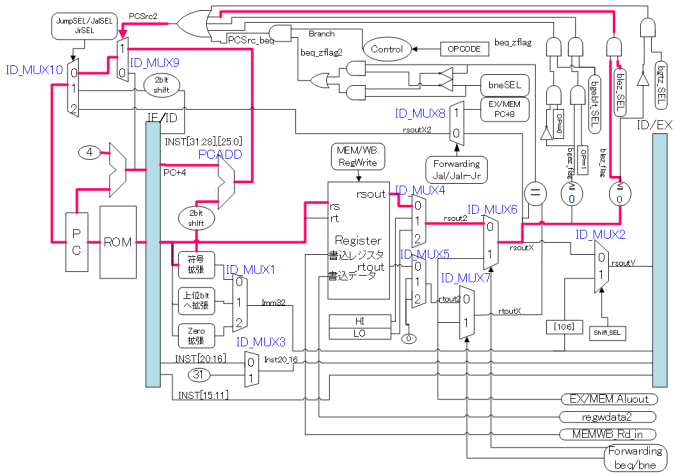


図 A.37: IF,ID ステージ

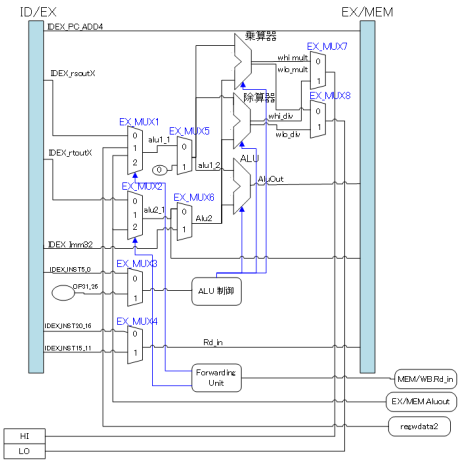


図 A.38: EX ステージ

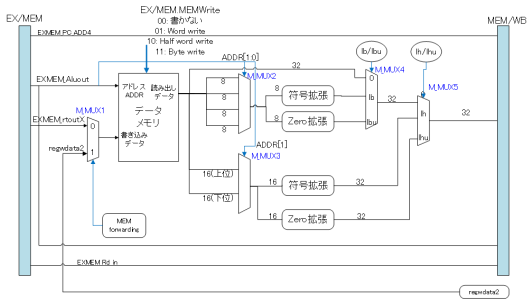


図 A.39: MEM ステージ

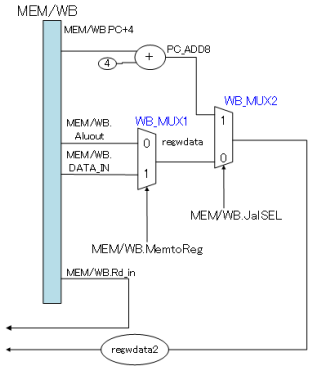


図 A.40: WB ステージ

- bgtz 命令

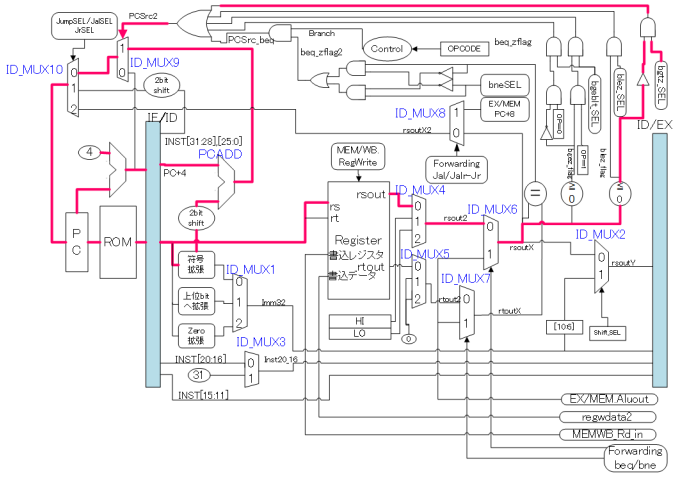


図 A.41: IF,ID ステージ

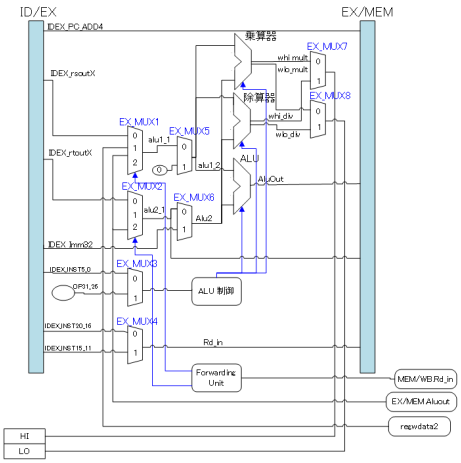


図 A.42: EX ステージ

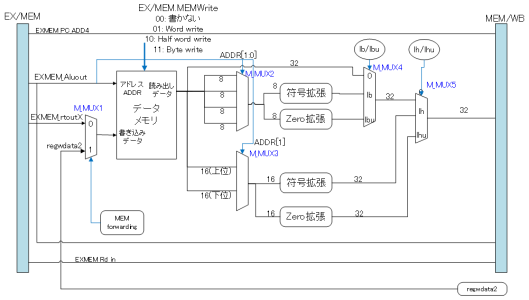


図 A.43: MEM ステージ

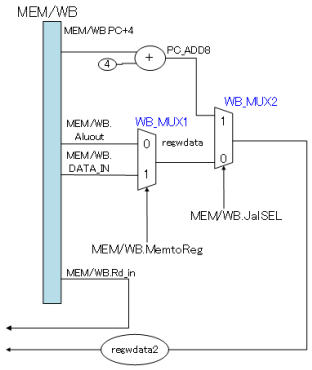


図 A.44: WB ステージ

- jump 命令

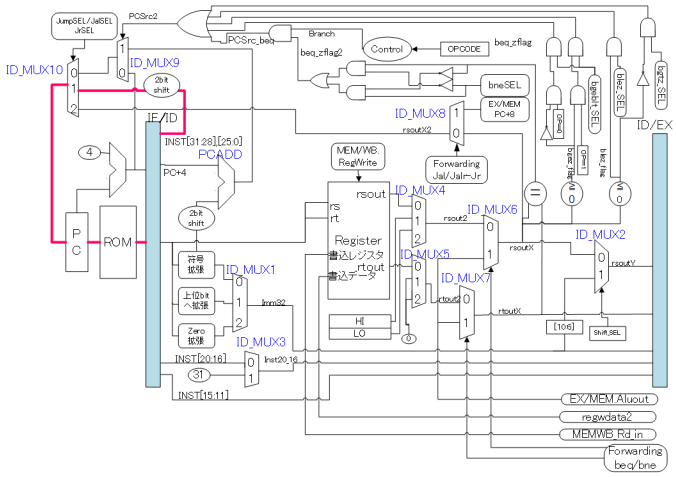


図 A.45: IF,ID ステージ

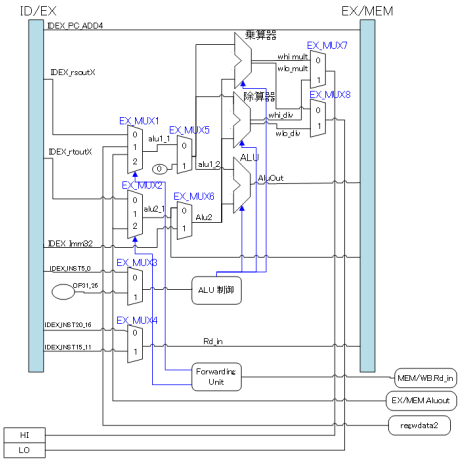


図 A.46: EX ステージ

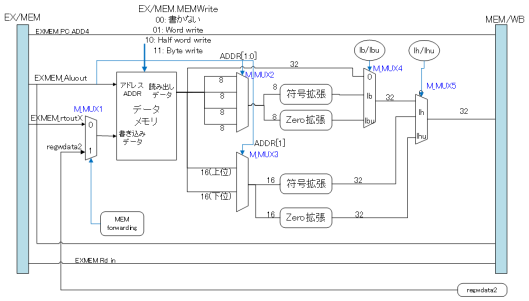


図 A.47: MEM ステージ

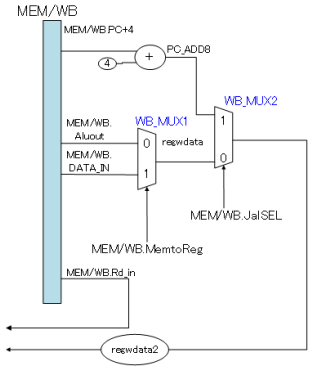


図 A.48: WB ステージ

- jal 命令

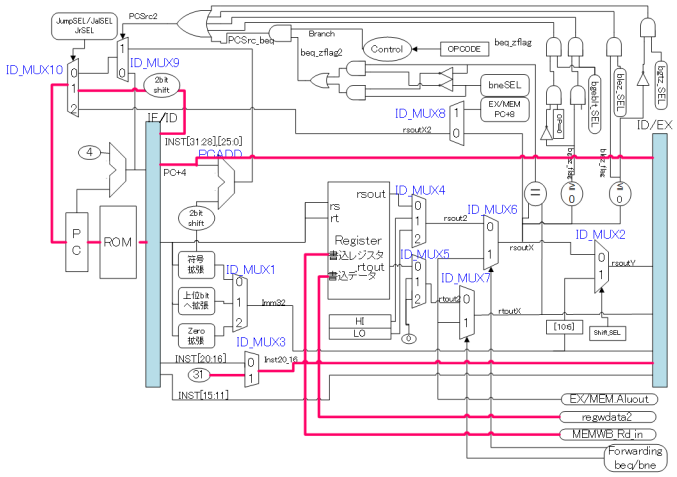


図 A.49: IF,ID ステージ

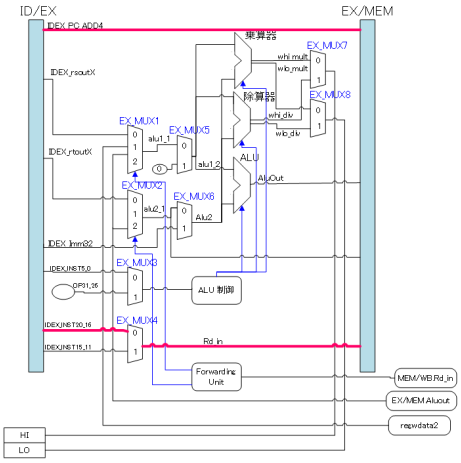


図 A.50: EX ステージ

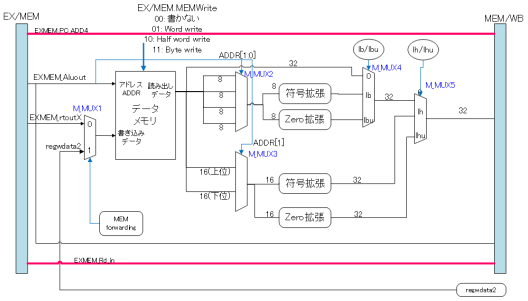


図 A.51: MEM ステージ

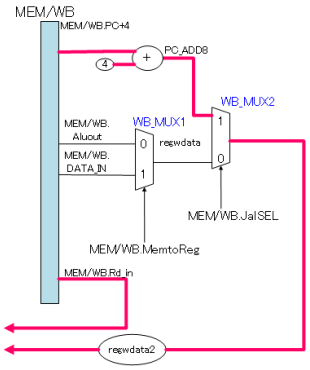


図 A.52: WB ステージ

- jalr 命令

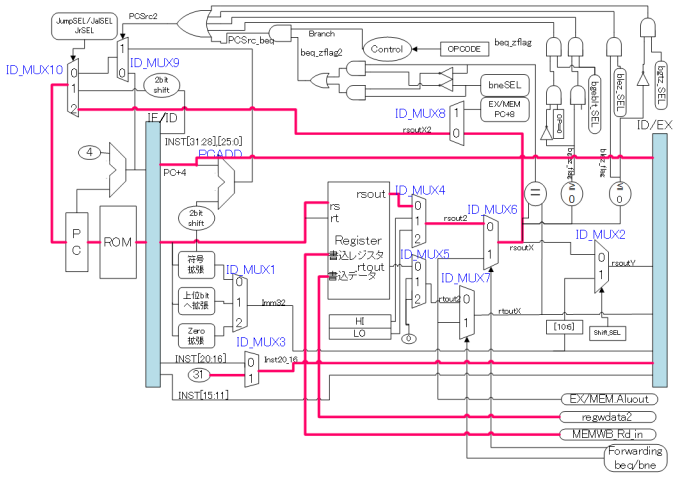


図 A.53: IF,ID ステージ

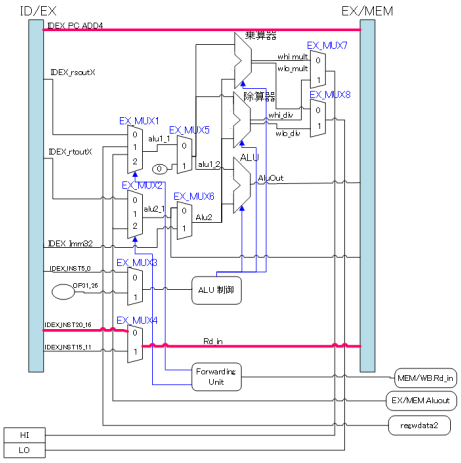


図 A.54: EX ステージ

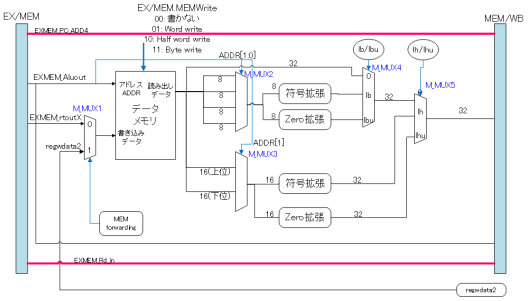


図 A.55: MEM ステージ

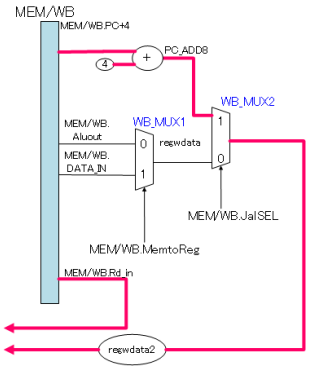


図 A.56: WB ステージ

- jr 命令

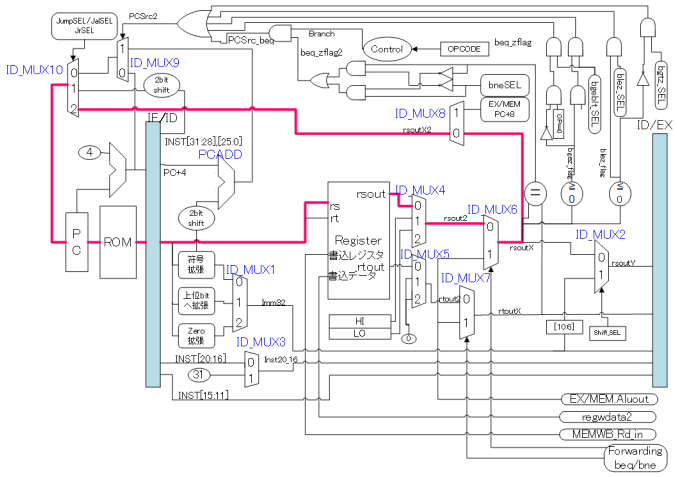


図 A.57: IF,ID ステージ

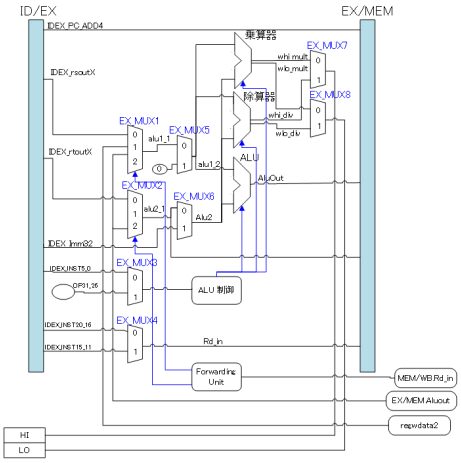


図 A.58: EX ステージ

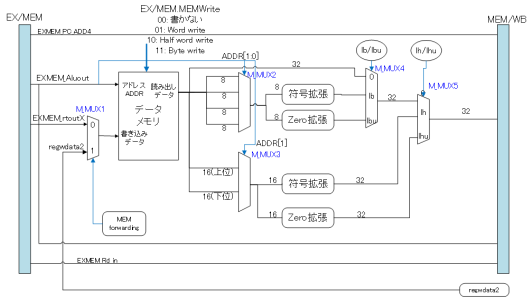


図 A.59: MEM ステージ

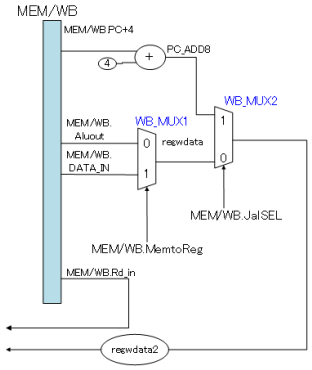


図 A.60: WB ステージ

- lw 命令

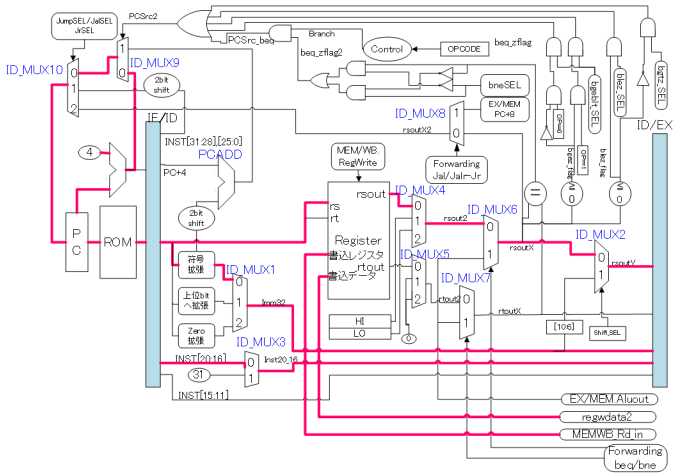


図 A.61: IF,ID ステージ

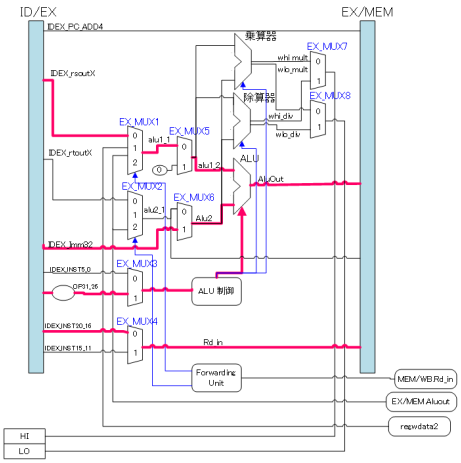


図 A.62: EX ステージ

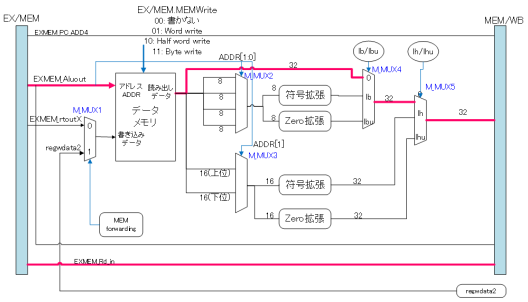


図 A.63: MEM ステージ

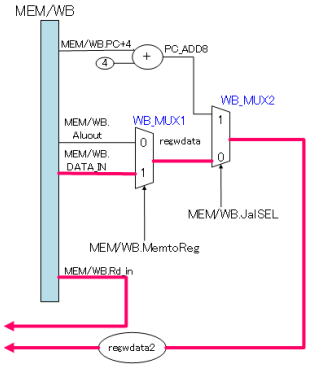


図 A.64: WB ステージ

- lh 命令

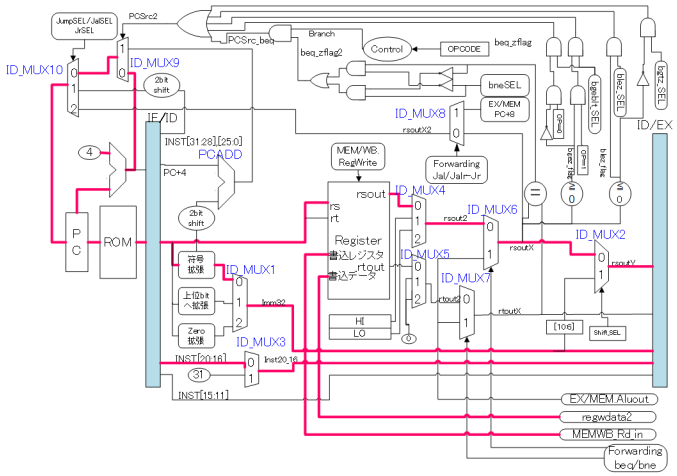


図 A.65: IF,ID ステージ

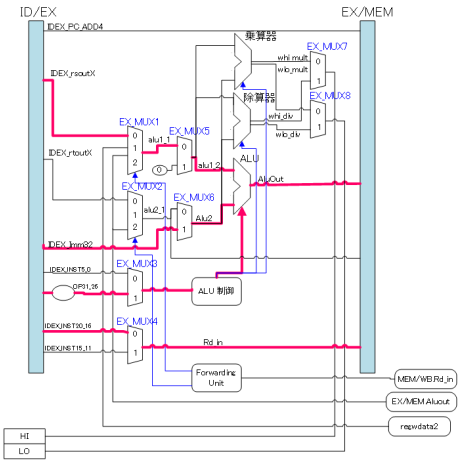


図 A.66: EX ステージ

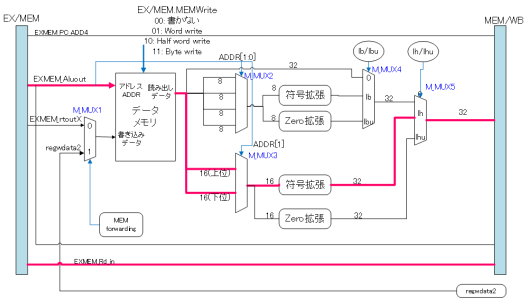


図 A.67: MEM ステージ

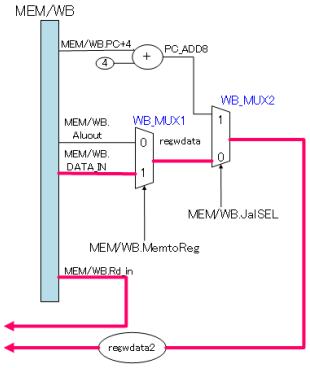


図 A.68: WB ステージ

- lhu 命令

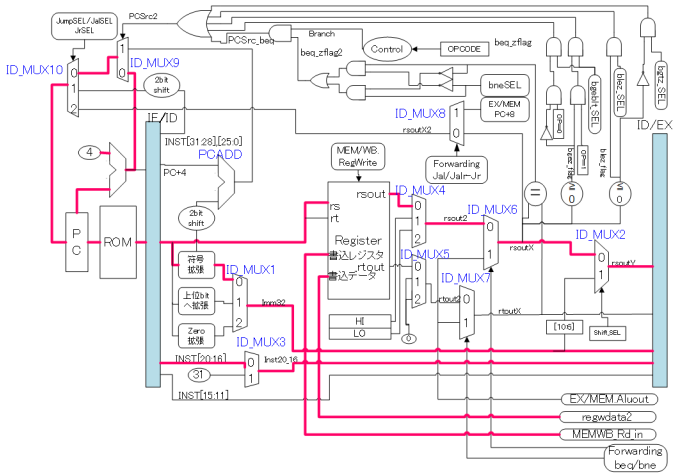


図 A.69: IF,ID ステージ

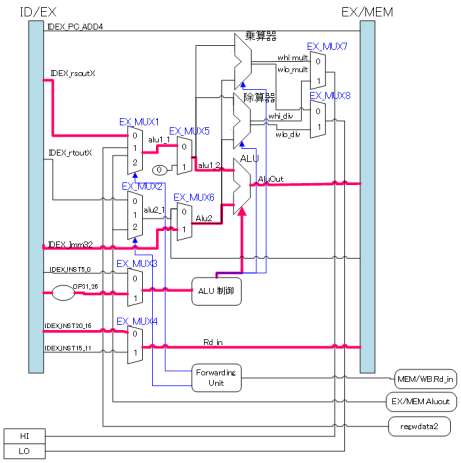


図 A.70: EX ステージ

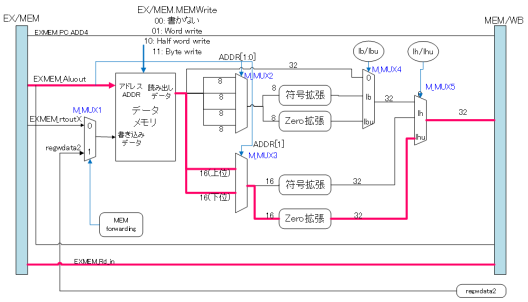


図 A.71: MEM ステージ

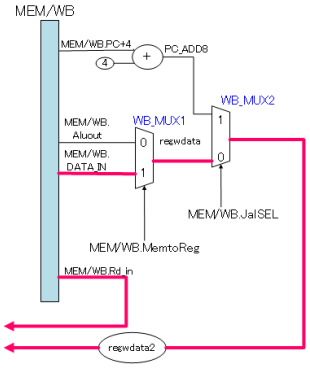


図 A.72: WB ステージ

- lb 命令

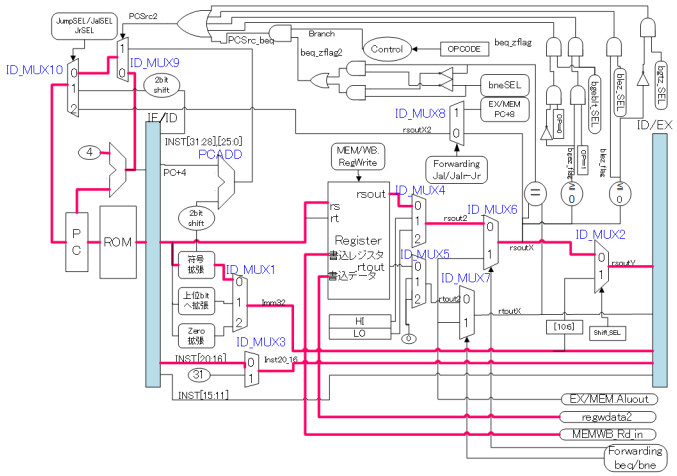


図 A.73: IF,ID ステージ

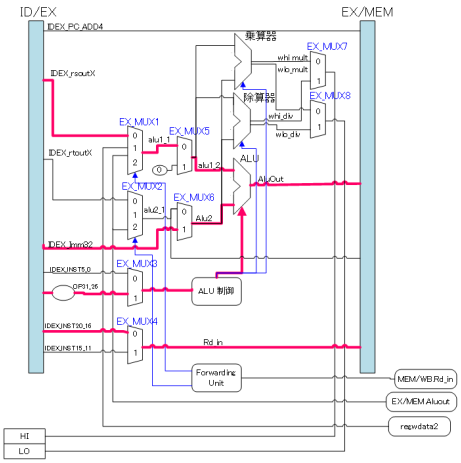


図 A.74: EX ステージ

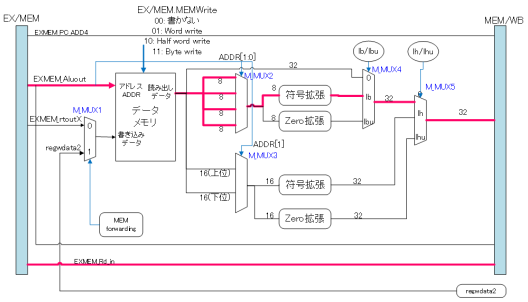


図 A.75: MEM ステージ

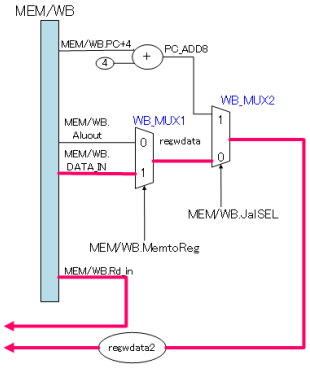


図 A.76: WB ステージ

- lbu 命令

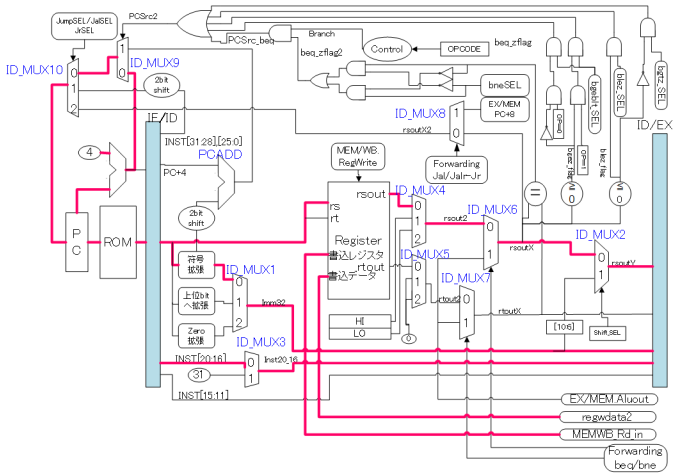


図 A.77: IF,ID ステージ

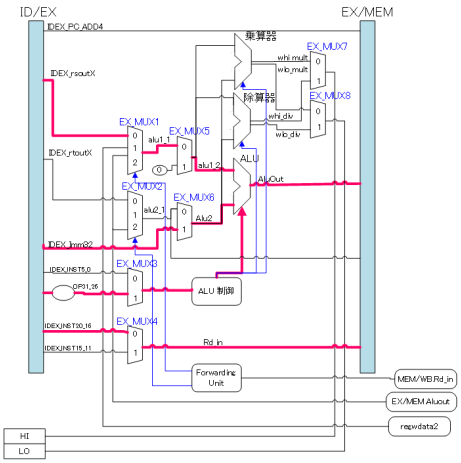


図 A.78: EX ステージ

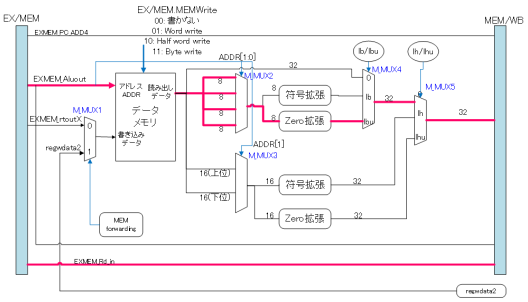


図 A.79: MEM ステージ

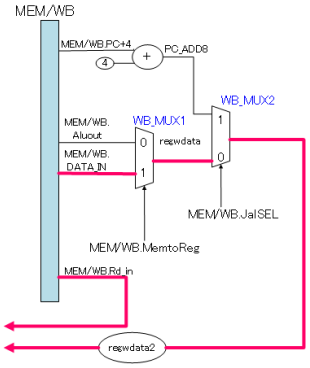


図 A.80: WB ステージ

- sw 命令

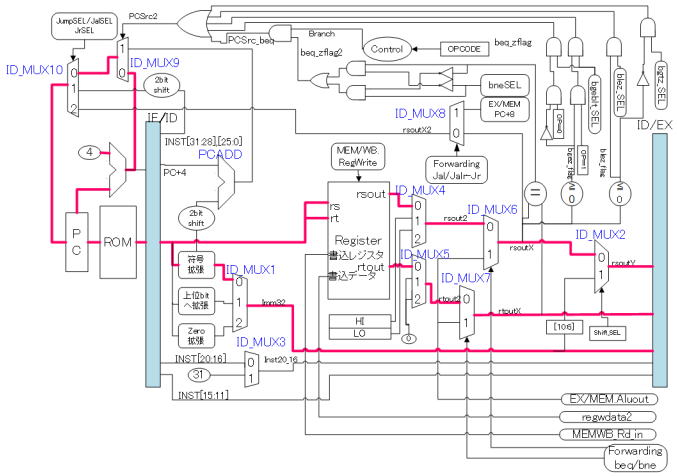


図 A.81: IF,ID ステージ

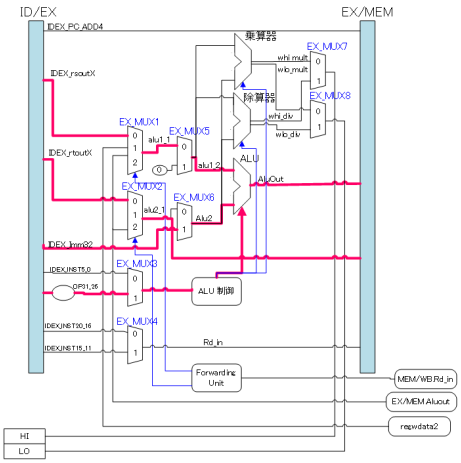


図 A.82: EX ステージ

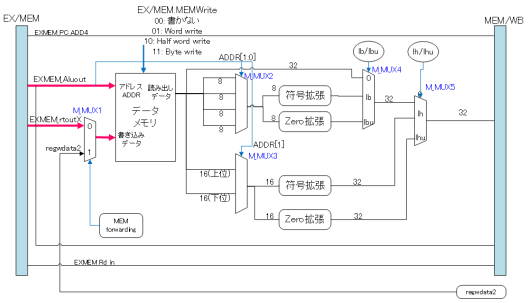


図 A.83: MEM ステージ

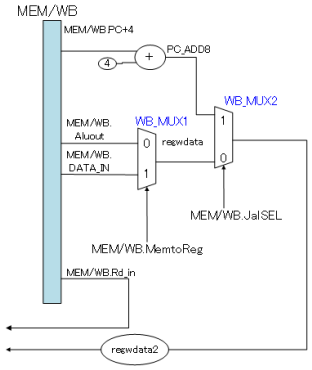


図 A.84: WB ステージ

- mult 命令

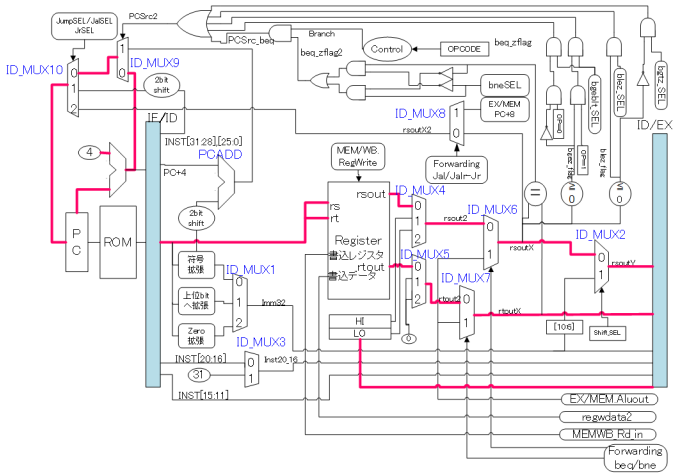


図 A.85: IF,ID ステージ

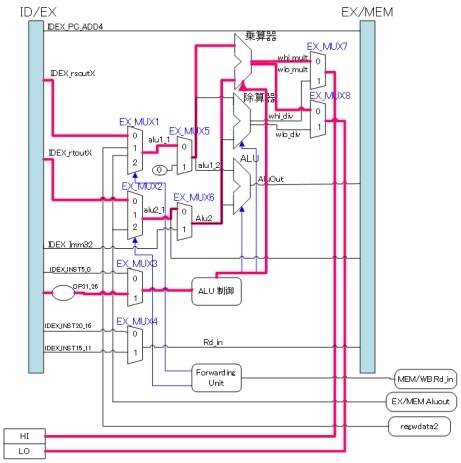


図 A.86: EX ステージ

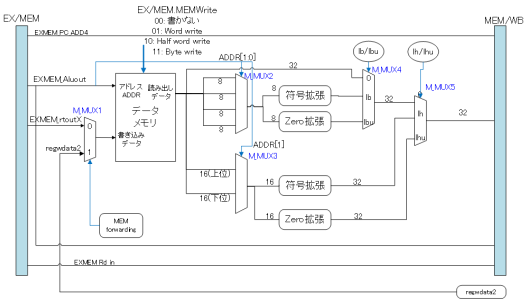


図 A.87: MEM ステージ

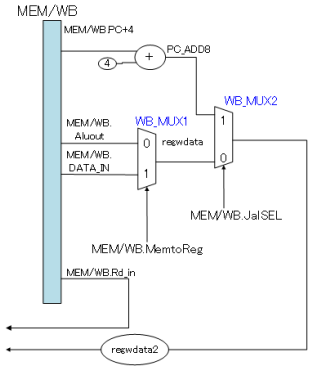


図 A.88: WB ステージ

- div 命令

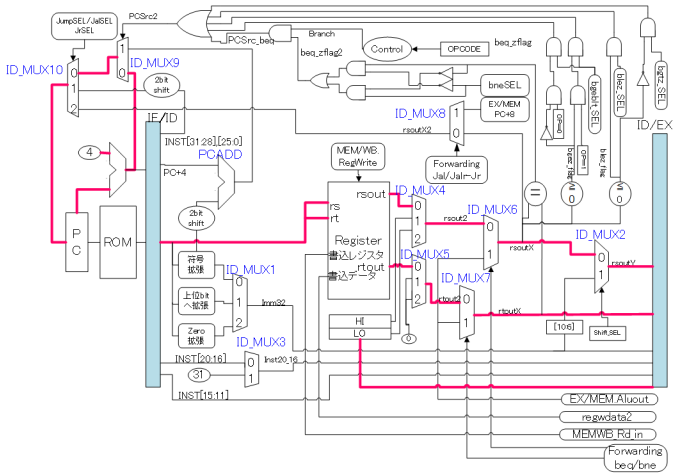


図 A.89: IF,ID ステージ

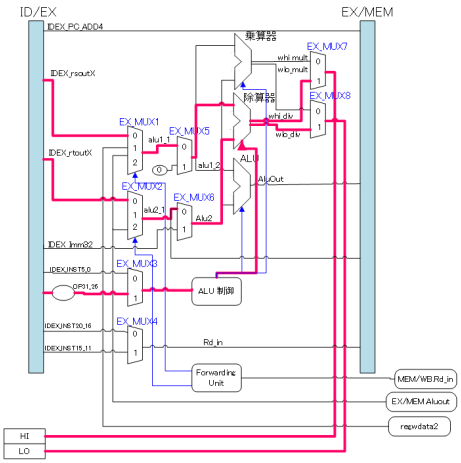


図 A.90: EX ステージ

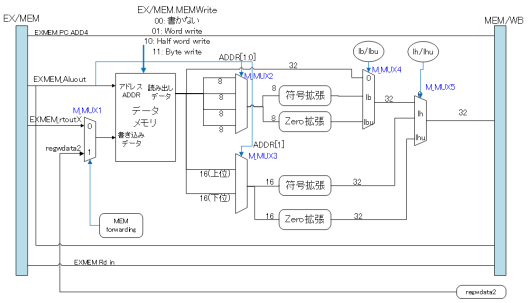


図 A.91: MEM ステージ

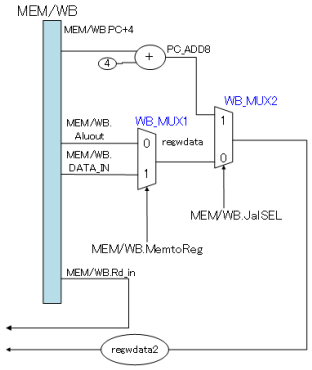


図 A.92: WB ステージ

- mfhi 命令

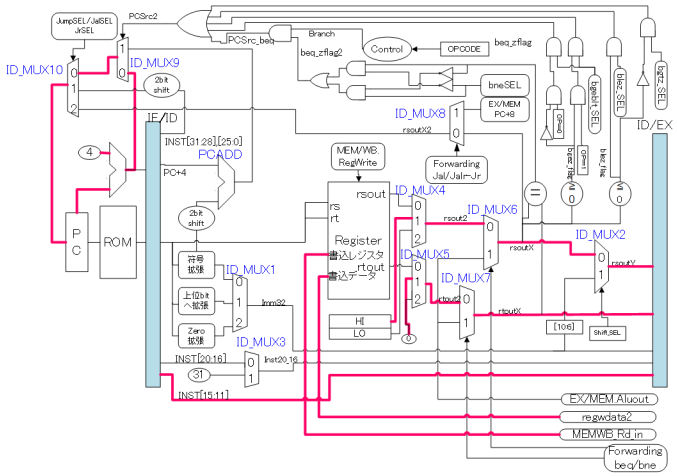


図 A.93: IF,ID ステージ

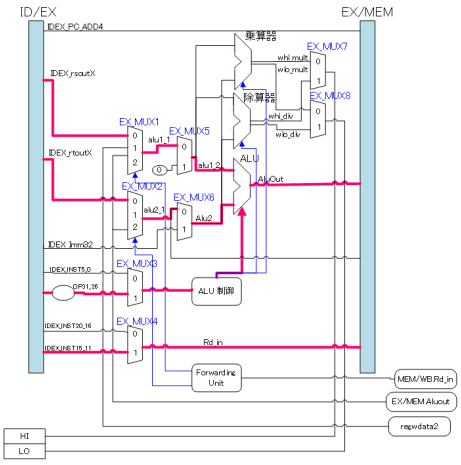


図 A.94: EX ステージ

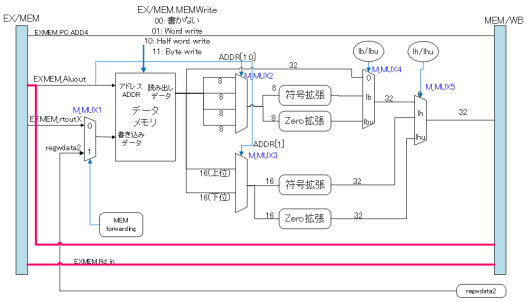


図 A.95: MEM ステージ

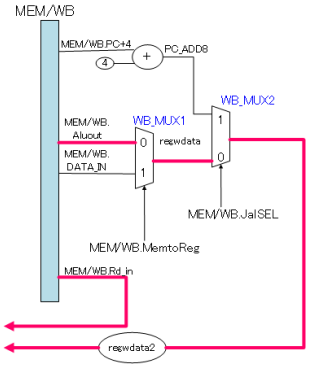


図 A.96: WB ステージ

- mflo 命令

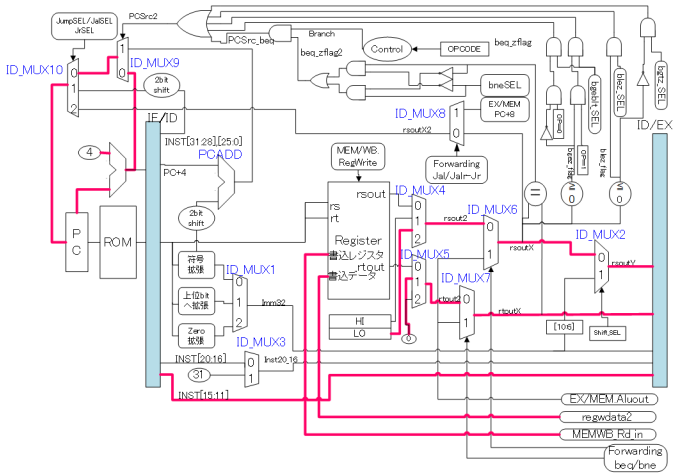


図 A.97: IF,ID ステージ

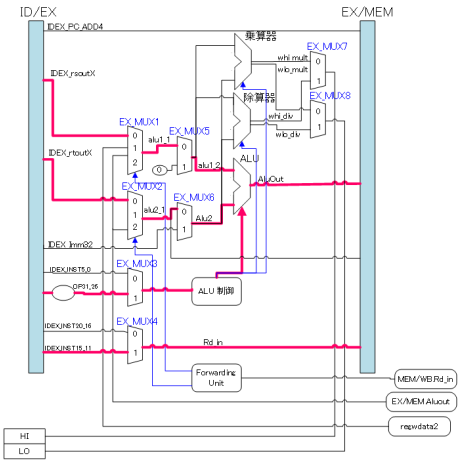


図 A.98: EX ステージ

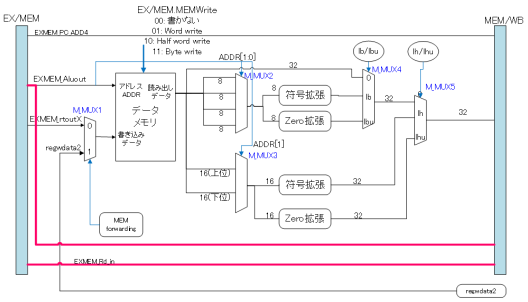


図 A.99: MEM ステージ

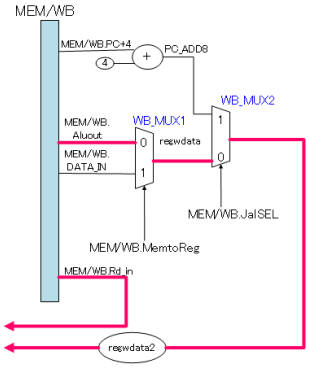


図 A.100: WB ステージ