平成 29 年度　計算機科学実験及演習 3A
(3 回生前期学生実験　HW 最終報告)

機能設計仕様書

提出期限：6 月 16 日
提出日：6 月 16 日

第 22 班

1029272870　　谷　勇輝

目次

1 コンポーネント分割と担当

1.1 コンポーネント分割

プロセッサは、最上位レベルの分割として、図１に示すコンポーネントで構成される。各コンポーネント内部の設計（次レベルの分割等）はそれぞれの設計担当者が行う。

中間報告から大きな変更は無いが、パイプラインハザード処理に伴い EX フェーズ周辺の設計を一部最適化した。

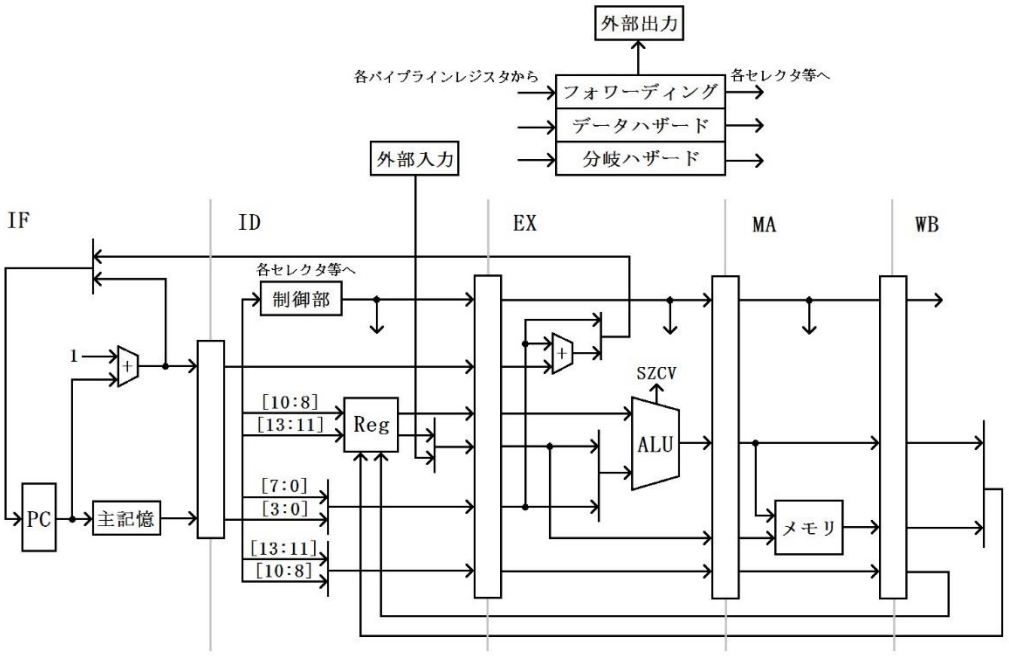

図 1　コンポーネント分割

1.2 担当

私が設計・実装を担当したプロセッサのコンポーネントは以下の通りである。

＜トップレベル＞

- MicroComputer_block (パイプライン主要配線 / 周辺配線)
  トップレベルの設計については班員 2 人で協力して行った

＜主要コンポーネント＞

- EX
- MA
- WB

＜周辺コンポーネント＞

- ALU (演算器)
- Forwarding (フォワーディング検知/処理)
- HazardDetection (データハザード検地/対処)
- TestEnvironment (FPGA ボード入出力用環境)
- Out (出力処理)

2017 年 5 月 11 日中間報告時点に設計済みであったモジュールのうち、最適化によって一部変更が行われたものを以下に示す。

- EX, MA, WB, TestEnvironment

中間報告時点から新たに設計を行ったモジュールを以下に示す。

- Forwarding, HazardDetection, Out

2 一部変更が行われたモジュール

2.1 EX

2.1.1 ALU モジュールへの役割委譲

EX モジュールに含めていた演算器（加算器、シフタ）を ALU モジュールとして分離、再構成した。この変更は verilog HDL 上の module でのまとめ方を変更したもので、EX の外部仕様を変更するものではない。ALU モジュールとして演算器を定義することで、コード上の不安定な記述を無くし、演算器を他の場所で再利用することができるようになった。

2.1.2 即値ジャンプ対応

拡張命令として JAL, JR を定義する際に、即値 immediate にプログラムカウンタを設定する動作（即値ジャンプ）が必要となった。これまでの EX の設計のままでは、PC+1 に immediate を加算した値（相対ジャンプ）しか分岐先として指定できない。そこで、分岐先を示す出力 IMnextPC は、control[0]（制御信号 Bradd）の値が 0 の際には PC+1+immediate を、値が 1 の際には immediate を流すように変更した。

2.1.3 halt 信号の伝播

halt 命令の処理を ID フェーズから WB フェーズに変更した影響で、halt 信号をそのまま前のフェーズから後ろのフェーズに伝播させる回路を追加した。具体的には、1bit 入力 halt と、1bit レジスタ出力 halt_を追加し、クロックと同期して halt を halt_に伝播させた。

2.2 MA

2.2.1 制御信号の仕様変更に伴う変更

拡張命令 JAL,JR に対応するため、control[0]に割り当てる役割が、レジスタ書き込みデータ制御信号 MemtoReg から戻り先アドレスレジスタを使用することを示す制御信号 Bradd に変更となった。この影響により、こ

れまで MemtoReg が担っていた役割を補完するため、control[3]（メモリ読込み制御信号 MemRead)を MA フェーズ以降でも使用することになった。これに伴って、今までの control 下 2bit に加え、control[3]を加えた 3bit を control_として出力する変更を行った。

2.2.2 halt 信号の伝播

halt 命令の処理を ID フェーズから WB フェーズに変更した影響で、halt 信号をそのまま前のフェーズから後ろのフェーズに伝播させる回路を追加した。具体的には、1bit 入力 halt と、1bit レジスタ出力 halt_を追加し、クロックと同期して halt を halt_に伝播させた。

2.3 WB

2.3.1 制御信号の仕様変更に伴う変更

拡張命令 JAL,JR に対応するため、control[0]に割り当てる役割が、レジスタ書き込みデータ制御信号 MemtoReg から戻り先アドレスレジスタを使用することを示す制御信号 Bradd に変更となった。この影響により、まず、MA フェーズから伝播されてくる入力信号 control の bit 幅が 2bit から 3bit に変更になっている。

また、ID フェーズが今までの control[1]（レジスタ書込み制御信号 RegWrite)に加え、新たに策定された control[0]（戻り先アドレスレジスタ使用制御信号 Bradd)を必要とするようになった。そこで、いままでの 1bit 出力 RegWrite_を廃止し、control_[1..0]を出力として上記の 2 制御信号を ID に伝播するよう変更した。

2.3.2 halt 信号の伝播

halt 命令の処理を ID フェーズから WB フェーズに変更した影響で、halt 信号をそのまま前のフェーズから後ろのフェーズに伝播させる回路を追加した。具体的には、1bit 入力 halt と、1bit レジスタ出力 halt_を追加し、クロックと同期して halt を halt_に伝播させた。

2.4 TestEnvironment

2.4.1 ステップカウンタの設置

TestEnvironment に、指定の clock 信号が隆起した数をカウントするステップカウンタを追加した。プログラムカウンタだけでは、ジャンプが起こった際に総ステップ数がカウントできないからである。ステップカウンタはプロセッサの stop 信号、halt 信号、reset 信号を共有して動作する。

具体的な動作としては、wire mainCK に接続されたクロック信号に合わせてカウントアップが行われる。出力は num[7]から num[0]としてカウントの 8 桁 10 進表示が与えられる。これを擬似出力に接続し表示することで利用できる。

2.4.2 高周波クロックの設置

40MHz 以上のクロックを使用できるよう、Megafunction を利用した高周波クロックを各種設置した。

clock2_50on40	50MHz （50/40 倍）
clock2_60on40	60MHz （60/40 倍）
clock2_75on40	75MHz （75/40 倍）
clock2_80on40	80MHz （80/40 倍）
clock2_100on40	100MHz （100/40 倍）
clock2_ex	個別に設定

3 新たに設計したモジュール

3.1 Forwarding

3.1.1 外部仕様

Forwarding モジュールは、５段パイプラインを高速化するためのフォワーディング処理を行う。ID から EX に送られてくるレジスタデータは前２フェーズ分の命令で書き込まれるデータに対応していないので、これを補完する。(WB フェーズの書込みは ID フェーズの読込みよりも早いタイミングで行われるので拾う必要はない)

今回のアーキテクチャではその信号変換を EX フェーズ直前で行うことにした。５段パイプラインの ID モジュールと EX モジュールの間に挟む形で接続する。

全体内の位置づけを図２に、入力構造を表１に、出力構造を表２に示す。

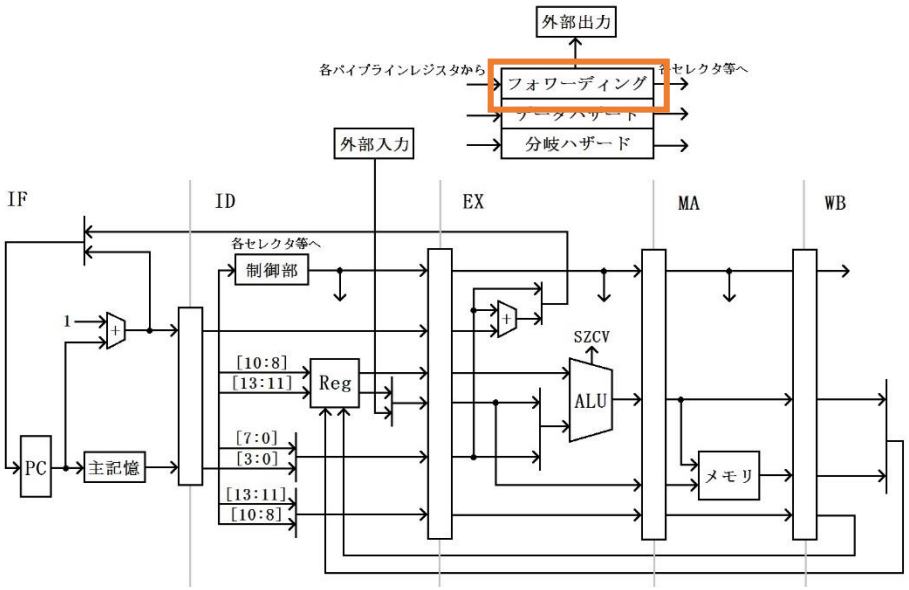

図 2　Forwarding モジュールの位置づけ

表 1　Forwarding モジュールの入力

入力信号名	bit 幅	接続	内容
In_RsRa	16	EX 前	ID から伝播された RsRa
In_RdRb	16	EX 前	ID から伝播された RdRb
EX_add_RsRa	3	EX 前	In_RsRa が参照したレジスタアドレス
EX_add_RdRb	3	EX 前	In_RdRb が参照したレジスタアドレス
MA_RegWrite	1	MA 前	MA にいる命令がレジスタ書込みを行うか
MA_add	3	MA 前	MA にいる命令が書込むレジスタアドレス
MA_data	16	MA 前	MA にいる命令が書込むデータ
WB_RegWrite	1	WB 後	WB にいる命令がレジスタ書込みを行うか
WB_add	3	WB 後	WB にいる命令が書込むレジスタアドレス
WB_data	16	WB 後	WB にいる命令が書込むデータ

表 2　Forwarding モジュールの出力

出力信号名	bit 幅	接続	同期	内容
Out_RsRa_	1	EX 前	×	フォワーディングを行った RsRa
Out_RdRb_	3	EX 前	×	フォワーディングを行った RdRb

- MA_RegWrite が 1 で、EX_add_RsRa が MA_add に等しいならば、MA_data を Out_RsRa に出力
- WB_RegWrite が 1 で、EX_add_RsRa が WB_add に等しいならば、WB_data を Out_RsRa に出力。但し、上の MA が優先される。
- 上 2 つがいずれも当てはまらなければ In_RsRa を Out_RsRa に出力

- MA_RegWrite が 1 で、EX_add_RdRb が MA_add に等しいならば、MA_data を Out_RdRb に出力
- WB_RegWrite が 1 で、EX_add_RdRb が WB_add に等しいならば、WB_data を Out_RdRb に出力。但し、上の MA が優先される。
- 上 2 つがいずれも当てはまらなければ In_RdRb を Out_RdRb に出力

3.1.2 内部仕様

Forwarding モジュールのブロック図を図３に示す。

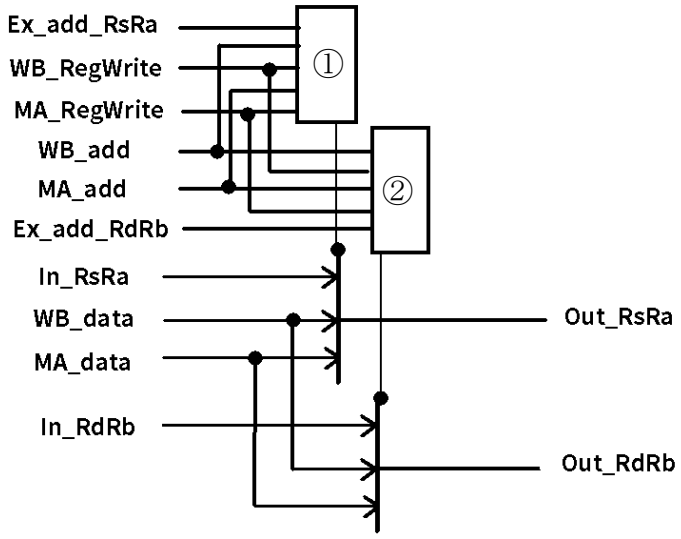

図 3　Forwarding モジュール（ブロック図）

① **RsRa の信号選択（*_RegWrite, *_RsRa → Out_RsRa）**

```
if (MA_RegWrite == 1 ∧ EX_add_RsRa == MA_add)
                                → Out_RsRa = MA_data
else if (WB_RegWrite == 1 ∧ EX_add_RsRa == WB_add)
                                → Out_RsRa = WB_data
else
                                → Out_RsRa = In_RsRa
```

② **RdRb の信号選択（*_RegWrite, *_RdRb → Out_RdRb）**

```
if (MA_RegWrite == 1 ∧ EX_add_RdRb == MA_add)
                                → Out_RdRb = MA_data
else if (WB_RegWrite == 1 ∧ EX_add_RdRb == WB_add)
                                → Out_RdRb = WB_data
else
                                → Out_RdRb = In_RsRa
```

3.2 HazardDetection

3.2.1 外部仕様

HazardDetection モジュールは、パイプライン化に伴って発生するデータハザードを検出し、それを検出した場合バブルを発生させてハザードが解消するまで動作をストールさせる役割を果たす。

フォワーディングによってレジスタ書込みの時間差によるデータハザードは取り除かれているが、EX フェーズの命令がメモリ読み込みで、そのデータを ID フェーズの命令が要求している場合のみ、パイプラインをストールさせてメモリ読込みを待つ必要がある。HazardDetection はこのデータハザードを検知し、ストールを ID フェーズに要求する。

全体内の位置づけを図 4 に、入力構造を表 3 に、出力構造を表 4 に示す。

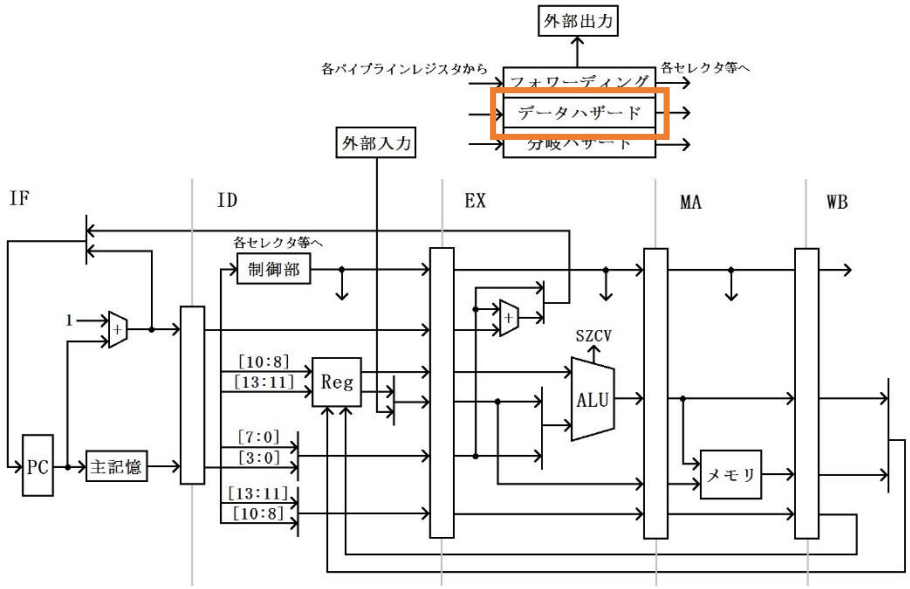

図 4　HazardDetection モジュールの位置づけ

表 3　HazardDetection モジュールの入力

入力信号名	bit 幅	接続	内容
EX_control	6	EX 前	現 EX に流れる制御信号
EX_WBaddress	3	EX 前	現 EX に流れるレジスタ書込みアドレス

ID_RsRa	3	ID	現 ID が RsRa として参照している レジスタアドレス
ID_RdRb	3	ID	現 ID が RsRa として参照している レジスタアドレス

表 4　HazardDetction モジュールの出力

出力信号名	bit 幅	接続	同期	内容
stall	1	ID,外部	×	ストール信号

- EX_control[3]（メモリ読込み制御信号 MemRead) が 1 で、EX_WBaddress が ID_RsRa か ID_RdRb のいずれかと一致した場合、stall に 1 を出力
- 上記以外は stall に 0 を出力

3.2.2 内部仕様

HazardDetection モジュールのブロック図を図 5 に示す。

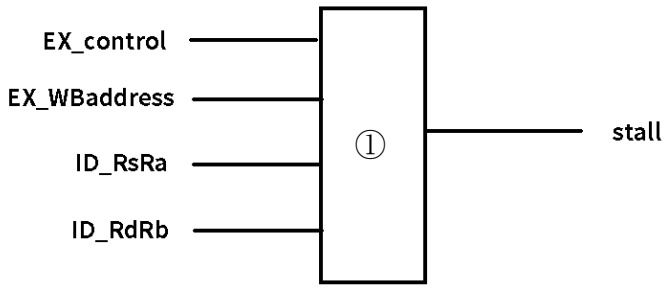

図 5　HazardDetection モジュール（ブロック図)

① データハザードの検知（ 全入力 → stall ）

```
if (EX_MemRead == 1 ∧
        EX_WBaddress == (ID_RsRa or ID_RdRb)
                                        →  stall = 1
else
                                        →  stall = 0
```

3.3 Out

3.3.1 外部仕様

Out モジュールは、Out 命令によって外部に出力される値を保持するためのレジスタを担当する。このレジスタは、Out 命令に出力を依頼されるとレジスタの値を更新し、次の Out 命令が来るまでその値を出力し続ける役割を果たす。

全体内の位置づけを図６に、入力構造を表５に、出力構造を表６に示す。

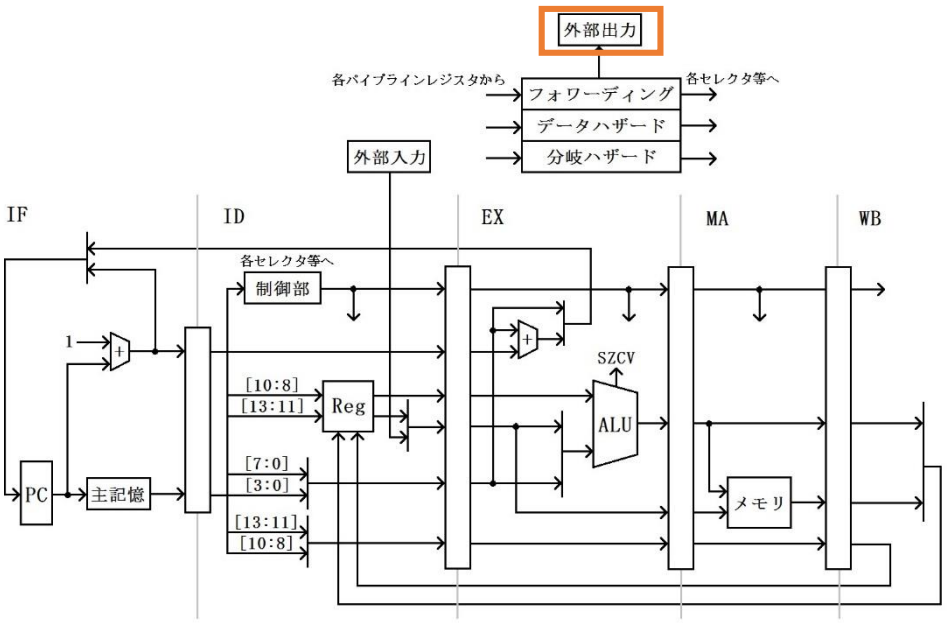

図 6　Out モジュールの位置づけ

表 5　Out モジュールの入力

入力信号名	bit 幅	接続	内容
clock	1	clock 信号	clock 信号
stall	1	HazardDetecter	stall 信号
change	1	ID	Out 出力制御信号
data	16	EX 前（Forwarding 後）	出力データ

表 6　Out モジュールの出力

出力信号名	bit 幅	接続	同期	内容
out	16	外部	○	出力

- change が 1 （Out1 命令）で stall が 1 でない（ストールしていない）時、data を out として clock 同期で出力

3.3.2 内部仕様

Out モジュールのブロック図を図 7 に示す。

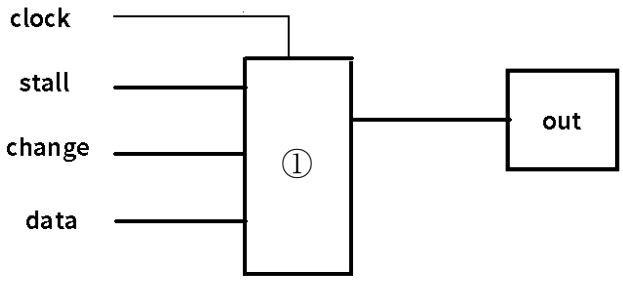

図 7　Out モジュール（ブロック図）

① 出力変更（ 全入力 → out ）

```
when (clock が立ち上がった)
        if (change  == 1 ∧ stall == 0 )        →  out = data
```

4 性能評価

4.1 EX (+ALU)

回路規模 :　　441 論理素子(2%)
最大動作周波数:　210.86 MHz
クリティカルパス: Rs_Ra→ALU→ConditionCode_[2]

4.2 MA

回路規模 :　　248 論理素子(1%未満)
最大動作周波数:　131.96 MHz
クリティカルパス: control[3] → mem2 → LDresult_

4.3 WB

回路規模 :　　17 論理素子（1%未満）
遅延時間:　　10.036 (ns)
クリティカルパス: ALUresult[6] → WBdata_[6]

4.4 Forwarding

回路規模 :　　72 論理素子（1%未満）
遅延時間:　　16.353 (ns)
クリティカルパス: MA_add[0] → Out_RdRb[14]

4.5 HazardDetection

回路規模:　　5 論理素子（1%未満）
遅延時間:　　10.065 (ns)
クリティカルパス: ID_RsRa[0] → stall

5 考察・感想

5.1 考察

5.1.2　全体設計に関する考察

プロセッサの全体設計はチーム２人でアイデアを出し合いながら行った。全体設計は後のプロセッサの性能と開発の効率に大きく影響すると考え、細かい点まで綿密に行ったが、それが後に功を成すことになった。以下に全体設計に関する考察を記す。

第一に、モジュール分割の方法について考察する。今回の設計では、主に１フェーズ１モジュールとして分割をし、担当モジュールを先に決定して、開発終了までそのモジュールを責任をもって創り上げるという体制をとった。結果としてこの体制はうまく機能し、効率よく開発を進めることができた。この方針の利点としては、①**全体のプロセッサの構成がわかりやすく、開発、改良が容易である点**、②**担当範囲が明確で、開発衝突が起こりにくい点**が挙げられる。逆に欠点としては、**モジュール内部とモジュール外部の両方に関連する動作が実装しにくい点**を挙げることができる。後者の欠点は、データハザード検出の際に ID フェーズの出力増加という手数増加に繋がった。

第二に、ID フェーズと EX フェーズの役割分担について考察する。全体設計の際、決定するのを苦労した項目の１つがフェーズの分担である。特に ID フェーズと EX フェーズ間の機能分担をどちらにするかは難しい課題であった。全体設計の際は**機能はできるだけ ID フェーズに分担し、EX フェーズの演算の前に他の機能を持たせない**方針としたが、これは結果的に正解だったと考える。なぜなら、完成したプロセッサのクリティカルパスは、WB フェーズからのフォワーディング入力を経て EX フェーズの演算を行うパスになったからである。もしも EX フェーズの演算の前に他の機能を持たせていた場合、このクリティカルパスを伸ばしてしまうことになる。従って、フォワーディングを除く演算前の処理はすべて ID フェーズで行うべきであると考察できる。

第三に、メモリの配置位置について考察する。今回の設計はハーバードアーキテクチャを採用しているため、メモリは IF モジュールと MA モジュールに対し１つずつ用意された。しかし問題は、このメモリをモジュール内部

に配置するか、外部に配置し適宜アクセスするかという２つの設計方針である。今回の全体設計ではわかり易いという理由で前者を選択し、モジュール内部にメモリを隠す設計としたが、**これはあまり良くなかった**と言える。主な理由は、コア化を画策する際の障害となった点である。パイプラインのモジュールの中にメモリを入れてしまっていたので、いざコア化してみようとなった際に**メモリを共有メモリとして外に出さなければならない**手間が生じたのである。結果的にコア化させる時間は無かったが、この設計方針の誤選択は並列化を大きく阻害してしまったと言わざるを得ない。

その他にも、分岐判断の位置を EX フェーズまで繰り上げたり、WB フェーズのクロック制御を ID フェーズに委託したりと様々な工夫を凝らした設計方針を立てた。上記したメモリの配置位置の方針を除き、それらその他の方針は総じて良く機能したと考えられる。

5.1.3　個人設計に関する考察

個人的に設計を受け持ったのは EX(ALU)、MA、WB、Forwarding、HazardDetection、Out、TestEnvironment、の７つのモジュールである。このうち、特に EX モジュールと TestEnvironment モジュールの２設計について考察する。

EX モジュールは、演算と分岐判断を行うモジュールである。演算器は ALU モジュールとして纏められ、EX の内部で使用される。EX モジュールは演算と共にコンディションコードと呼ばれる値を保持し、分岐判断に用いる。今回の設計では、演算を行うまで判断が付かない３つのコード C、V、S については、ALU モジュールの出力として扱うことにした。特に、主に繰り上がりの有無をしめす C については、**演算結果を 1bit 拡張した出力**の最上位 bit として取り出す設計とした。この方針により S を自然に実装することができ、回路のシンプル化、高速化に寄与したと考えられる。

また、シフタについては高速動作するバレルシフタを採用した。シフトの種類によって個別のシフタを自前で用意し、計算結果が高速に算出できるようにした。

以上のような小さな工夫を積み重ねつつ、全体としてはシンプルな回路が出来上がるよう意識した。結果として EX フェーズは単体で 200MHz を越える最大動作周波数を達成し、プロセッサの高速化に貢献できた。

TestEnvironment モジュールは、プロセッサを実機上で動作させるための付属モジュールである。実機上で自由かつ円滑にテストを行うため、全てのピンを刺して、全入力、出力を利用可能な状態を提供した。

特に LED 出力については信号線名を特定の括弧の中に記述するだけで簡単に外部表示できるよう設計した。この機能の利用価値は高く、このモジュールの設計が**開発を円滑にすすめられた一番の要因**となったと考える。

やはり、質のよい道具を初期段階で作成することが、あらゆる開発を行う上で大切なことなのだろう。

また、中間発表時、LED 表示がチラつく不具合が確認された。全ての LED を使用可能とするために、カウンタを用いてセレクタと表示させるデータを遷移させていたのだが、タイミングに問題があったようだ。カウンタの細かさを３倍にし、セレクタの変更タイミングと表示データの変更タイミングをずらすことで解決したので、同じタイミングでセレクタと表示を切り替えてしまうと、実機上ではタイミングにわずかな差異が生じてしまうのだと考察できる。

コードや回路がシンプルになるように設計することは、開発の効率をあげるだけでなく、見えないところで性能に少しずつ寄与するのだと感じた。設計段階では目的の機能をいかにシンプルに実現するかが最重要であると今回の実験を受けて私は考えた。

5.2 感想

今回の実験は、実に沢山のものが得られる有意義なものだった。また、目標以上のものが達成でき、とても満足のいくものだった。

まず第一に、初のペアでの実験をうまく行うことができた。ペアであった白石君とのコミュニケーションも密に行え、２人であることのメリットを十分に活かせたと感じている。連携も驚くほど滑らかで、お互い気分よく実験を楽しめたと思う。これは今回の実験で最も満足している点である。

第二に、ソートコンテストで第２位という良い成績を残せた。当初の目標であった 0.5ms の実に 10 分の 1 である 0.5062ms を達成できた。これは、実験初期の段階で２人でアイデアを出し合って独自のアーキテクチャを研鑽できたことと、プロセッサ本体以外のテスト環境やアセンブラ等の道具をうまく利用できた結果である。しっかりと足場を固めながら結果を伸ばせたという結果に達成

感を覚えた。

第三に、いままで知識だけだったプロセッサの基本形を作成することで PC の理解が深まった。マイクロプロセッサの中身という今までブラックボックスだったものを自ら設計、作成することでプロセッサに親近感を感じることができるようになったのは嬉しかった。

欲を言えば、せっかく良い環境を整えた上で開発ができていたので、コア化まで拡張してソートコンテスト１位のグループに挑戦してみたかった。作業効率の向上は私の永遠の課題である。

総じてとても実りのある学生実験であった。今後も実験に精力的に参加して今回のように楽しんでいきたいと思う。

以上