16 bit super scaler processor $\mathcal{K}$amuri-chip の設計

川崎通　中江達哉　永見康一　中村寿彦　松岡裕人　木村晋二

1996 年 6 月 18 日 ($Revision : 1.5$)

目次

図一覧

表一覧

第 1 章

はじめに

このドキュメントは、$\mathcal{K}$amuri-chip という名の 16 bit スーパースカラ・マイクロプロセッサについて説明しています。このプロセッサは、われわれ $\mathcal{LD}$-kamuri が、1994 年 7 月 から 9 月にかけて設計したものであり、設計はハードウェア記述言語 SFL [1] で記述し、高位論理合成系 PARTHENON により自動合成しました。

$\mathcal{LD}$-kamuri というのは、設計当時、奈良先端科学技術大学院大学 情報科学研究科言語設計学講座 (渡邉勝正教授) の助教授であった

- 木村 晋二 (`kimura@aist-nara.ac.jp`)

と、当時木村助教授の下で論理 CAD に関わる研究を行う学生 (博士前期課程二年生) であった

- 川崎 通 (`tkawa@akm-dc.atsugi.asahi-kasei.co.jp`),
- 中江 達哉 (`nakae@fd.cad.fujitsu.co.jp`),
- 永見 康一 (`nagami@isl.ntt.jp`),
- 中村 寿彦 (`tosihiko@ccs.mt.nec.co.jp`),
- 松岡 裕人 (`hmatsu@aecl.ntt.jp`),

という合計 6 名からなるグループの名前です。$\mathcal{LD}$ は、言語設計学講座 (**L**anguage **D**esign Lab.) からとっています。kamuri とは、じつは kimura のアナグラム (並び換え) で適当に決まってしまった名前です。

この $\mathcal{K}$amuri-chip は、当初の目的通り、PARTHENON 研究会主催の「第 2 回 ASIC デザインコンテスト」の自由課題 (学生) 部門へ応募しました。プロセッサとしては数々の問題点を抱えていたものの [2] [1]、スーパースカラというハッタリ (?) が審査員にアピールしたのか、好運にも部門の最優秀賞をいただきました。この賞の副賞としていただいた、$\mathcal{K}$amuri-chip のマスクパターン図は、研究室に飾られています。また、PARTHENON 研究会と CQ 出版社のご厚意により、$\mathcal{K}$amuri-chip の設計に関する記事を Interface 誌上に 3 回に渡って掲載していただきました [3, 4, 5]。各記事をそれぞれ社会人となった中江、永見、中村が担当し、それぞれの視点で $\mathcal{K}$amuri-chip の設計を振り返っています。

以下、2 章で $\mathcal{K}$amuri-chip のアーキテクチャの概要を説明し、3 章で $\mathcal{K}$amuri アーキテクチャの SFL 記述について述べます。4 章ではシミュレーション・ベースの設計検証法について説明し、5 章で合成結果を示します。また、本文中で参照する図や表は、原則として付録 A にまとめてあります。付録 B は SFL で記述した各回路モジュールのリファレンス・マニュアルです。

[1] この文献 [2] には、$\mathcal{K}$amuri-chip に対するかなり的確なツッコミが展開されています。ちなみにタイトルの “教育用 16 ビット・マイクロプロセッサ” は、$\mathcal{K}$amuri-chip のことではありません。

第 2 章

$\mathcal{K}$amuri アーキテクチャの説明

2.1 基本アーキテクチャ

$\mathcal{K}$amuri アーキテクチャは、 load/store 型の RISC 命令セットを備えた、 16 bit の super scalar アーキテクチャであり、その super scalar 度は 2 です。また 16 bit のレジスタを 8 本備え、それらの役割はすべて等価です。 ゼロ・レジスタは特に用意してありません。

$\mathcal{K}$amuri アーキテクチャでは、メモリアドレスが指し示すデータの最小単位は byte (8 bit) であり、アドレス空間は 2^{16} です。また、アクセス単位となるデータ型は byte (8 bit), word (16 bit) の 2 つです。 word データは、メモリ上では、 word 境界上に整列された 'little endian' 方式で格納されていることを前提とします。すなわち下位 byte が偶数番地に、上位 byte が奇数番地に格納されます。

2.2 命令セット・アーキテクチャ

2.2.1 命令形式

すべての命令は 16 bit 固定長であり、その形式は図 A.1 に示される 5 つに分類されます。

R (Register) 型 2 つのレジスタの値を用いて演算を行ない、その結果をレジスタに格納する命令形式です。 rs1, rs2 がオペランドのレジスタ番号を示し、 rd が結果を格納するレジスタの番号を示します。

I5 (5 bit Immediate) 型 レジスタの値と即値を用いて演算を行ない、その結果をレジスタに格納する命令形式です。 rs, imm がそれぞれ、オペランドとなるレジスタの番号と 5 bit 即値を表します。 imm は、 16 bit に拡張されますが、符号つき拡張か符号なし拡張かは命令操作コードによります。 rd が結果を格納するレジスタの番号を表します。

ただし、命令操作コードが `sb`，`sw` (メモリへのストア命令) の場合は、 rs1, imm が格納先のアドレスを表し、 rs2 がストアする値をもつレジスタの番号を表します。

I8 (8 bit Immediate) 型 1 つのレジスタ番号 (rs または rd) と 8 bit の即値を持ちます。レジスタへの即値格納や、分岐命令などが該当します。

J (Jump) 型 11 bit の即値を持ちます。ジャンプ命令などが該当します。

2.2.2 命令セット

表 A.2 が全命令です。表中の、命令の形式的な記述については表 A.1 を参考にして下さい。また、これらの命令は表 A.4 に示される group に分類されます。

2.3 パイプライン・アーキテクチャ

$\mathcal{K}$amuri アーキテクチャは A pipe, B pipe の 2 本のパイプラインを持ち、各々は図 A.2 の 5 つのパイプライン・ステージからなります。

$\mathcal{K}$amuri アーキテクチャでは、命令の取り込みは常に 2 命令ずつ行なわれるため、制御の移行は 4 byte 境界に整列されます。以後、命令ペアという用語は A pipe と B pipe に同時に取り込まれる 2 命令、すなわち命令シーケンス上で 4 byte 境界に配置された 2 命令のことを差します。またこの場合 A pipe に取り込まれる命令を先

行命令、 B pipe に取り込まれる命令を後続命令とよぶことにします。また、命令ペアは、全てのステージにおいて、常に同時に次のステージに進みます。

2.4 メモリ・アーキテクチャ

$\mathcal{K}$amuri アーキテクチャは、 2 語の命令を取り込むための 32 bit 幅の命令バス、およびデータをアクセスするための 16 bit 幅のデータバスを独立に持ちます。命令の取り込み位置は、 14 bit の命令アドレス信号線によって指定されます。この信号の値は、下位 に 2 bit の 0 を補った 4 byte 境界アドレスを表します。データのアクセス位置は 15 bit のデータアドレス信号線および 2 bit のバイトイネーブル・マスク信号によって指定されます。データアドレス信号の値は下位に 1 bit の 0 を補った 2 byte 境界アドレスを表し、バイトイネーブル・マスク信号は、 2 byte のそれぞれについてアクセス対象かどうかを指定します。

第 3 章

SFL 記述の説明

3.1 $\mathcal{K}$amuri アーキテクチャのインプリメンテーション

3.1.1 命令操作コードの符号割り当て

本稿で示す $\mathcal{K}$amuri アーキテクチャの実装 $\mathcal{K}$amuri-chip では、命令ワードの命令操作コード (5 bit) は、表 A.3 のように符号割当を行なっています。

3.1.2 命令の並びに関する制約

$\mathcal{K}$amuri-chip では、次のような方針で各パイプライン・ハザード [6] に対応しています。

データハザード 命令ペア間のレジスタ・アクセス依存に関してはフォワーディングで対処します。ただし、命令ペア内のレジスタ・アクセス依存に関してはフォワーディングを行ないません。なお、load group に伴うレジスタ値決定の遅れについて、パイプライン・インターロックは行ないません。

構造ハザード controll/load/store group は、 A pipe のみで実行することができます。

制御ハザード 分岐命令は、 ID stage で実行を完了します。パイプライン・インターロックは行ないません。

したがって、投入される命令シーケンスは、以下の制約に従う必要があります。

1. 命令ペアの先行命令、後続命令の選択には、表 A.5 のような禁止組合せがあります。
2. レジスタ・アクセスの依存関係を持つような命令ペアはサポートされません。すなわち、先行命令の destination register を、後続命令が source register として参照するような命令ペアは、正しい結果をもたらしません (表 A.5)。
3. 分岐命令 (`beqz`, `bnez`, `j`, `jr`, `rst`) は、すべて遅延分岐であり、遅延スロットは 3 命令 (後続命令 + 次の命令ペア) です。ただし、分岐命令を遅延スロットに使うことは、正しい結果をもたらしません。
4. load group はすべて 遅延ロード であり、後に続く 3 命令は、 source register としてロード命令の destination register を参照することはできません。

3.1.3 メモリインタフェース

$\mathcal{K}$amuri アーキテクチャでは、命令メモリ空間とデータメモリ空間を分離しています[1]。メモリアクセスのインタフェースは次のようになってます。

命令メモリ　命令メモリのアクセスは読み込みのみです。 $\mathcal{K}$amuri-chip は、制御信号 `iread` の起動とともにデータ信号 `iaddr<14>` に命令メモリのアドレスの上位 14 bit を出力し、命令メモリの読み込みを要求します。命令メモリ側では、この要求を受けると、制御信号 `irdy` の起動とともにデータ信号 `ibus<32>` に読み込みデータを渡します (図 A.6)。このようなハンドシェークにより、メモリ側のタイミング制約から独立した記述としています。

[1] 実際の使い方としては、キャッシュコントローラを介して単一のメモリシステムを接続することになるでしょう。

データメモリ　データメモリのアクセスも、命令メモリのアクセスと同様のハンドシェークを使っています。 d access がアクセス要求で daddr<15> がアクセス・アドレスの上位 15 bit です。しかし、データメモリには読み込みと書き込みの双方向のアクセスがあるので、信号線 read_write の値によって区別しています (0 が読み込み、 1 が書き込み)。また、バイトアクセスをサポートするため、 dmask<2> がバイトイネーブル・マスクとなります。 daddr が表す 16 bit ワード境界アドレスに置かれたワードに対して、上位バイトをアクセスするかどうかを dmask<1> が 1 かどうかで表します。下位バイトが dmask<0> です。読み込みおよび書き込みアクセスに対する応答をそれぞれ drdy，dack が伝えます。それぞれのアクセスデータは dbus_in<16>，dbus_out<16> で受渡します[2]。

3.1.4　割り込み動作

割り込み動作は、一切サポートされていません。したがって、メモリの不整列アクセスは、 nop 動作となります。

3.2　ブロック構成の概要

図 A.3 が、各機能ブロックとデータ・パスの概要です。

3.3　パイプライン動作の詳細化

3.3.1　パイプライン制御

A pipe, B pipe に投入された命令ペアは、先行命令と後続命令が常に同時にステージを進んで行きます。従って、 A pipe, B pipe の各ステージは、それぞれ 1 つの (SFL 記述 における) stage (IF，ID，EX，MEM，WB) 上で動作します (図 A.4)。
パイプラインのストールおよびフラッシュは、図 A.4 のような制御信号の連係によって実現します。

ストール　各 stage は、 busy 信号を出力することができます。これは、次のクロックで次の命令ペアを受けいれることができないことを伝達します。従って、後続の stage が busy 信号を出力している場合、その stage も busy 信号を出力することになり、 busy 信号の連鎖が形成されます。

フラッシュ　各 stage は、 active な状態にあるとき (= 命令ペアを保持しているとき) に flush 信号を受けとると、保持している命令ペアを廃棄して (= 後続の stage に引き渡さないで) finish します．これにより、パイプライン・バブルが発生します。

命令ペアの引き渡しおよび実行制御　命令ペアを後続の stage に引き渡す動作は、 relay によって記述しています。また、 CONTROLL stage が、外部からの制御信号および、内部制御信号 (halt) を監視し、 IF stage のタスクの generate を制御します。

なお、 CONTROLL stage は、図 A.5 のような有限状態機械に従って動作します。

3.3.2　データ転送

パイプラインの各ステージは、表 A.6 に示されるラッチの値によって動作を行ないます。各ステージにおけるデータ転送は、表 A.7 のように行なわれます。

3.4　module 構成

図 A.7, A.8 が、 𝒦amuri-chip の全モジュールの階層構造を示したものです．各 module の詳細については、付録 B の **SFL モジュール・リファレンス** を参照して下さい。

[2]当初はこの二つの信号を双方向信号として一つにまとめていたのですが、こうするとトライステート・バッファが挿入されるため合成結果の遅延が大きく、やむなく分離しました。

第 4 章

設計検証

4.1 シミュレーション

$\mathcal{K}$amuri-chip の動作確認は PARTHENON に含まれる機能レベルシミュレータである、 seconds を用い、シミュレーションによってよって行ないました。

ディジタルシステムの網羅的なシミュレーションを行なうことは、一般に繁雑で難しい問題であることが多いのですが、プロセッサのシミュレーションの場合には、プロセッサの命令セットを用いて機械語プログラムを記述し、そのプログラムをプロセッサが実行する動作をシミュレーションする、という有力な方法があります。すなわち、

- テストパターン → 機械語プログラム
- 期待値 → 機械語プログラムの期待動作 (を表すシミュレーション結果)

という図式でシミュレーションを行なうわけです。これにより、人間による期待値の確認作業をより直観的に行なうことができます。 $\mathcal{K}$amuri-chip に対しては、この戦略を採用することにしました。具体的には、次のようになります。

4.1.1 シミュレーション手法 1

最初に考えた方法は、プロセッサによるプログラム実行動作のシミュレーション結果を、人が目で見て確認する手法です。

1. $\mathcal{K}$amuri-chip のバスインタフェースに従って、シミュレーション用の外部メモリ `imem`, `dmem` を記述し、さらにそれらを結合する最上位モジュール `system` を用意しました (**SFL** モジュール・リファレンスを参照して下さい)。
2. $\mathcal{K}$amuri 命令セットのクロスアセンブラを作成し、アセンブリプログラム (`*.asm`) を `imem`, `dmem` に対する `memset` 文を含んだ seconds スクリプト (`*.bin`) に自動変換できるようにしました。
3. 主要モジュール (各 stage, フォワーディング回路、レジスタ・ファイルなど) の各種状態について、クロック毎にレポートさせるためのスクリプト (`*.rpt`: `rpt_add` 文の並び) を用意し、プログラム実行をシミュレートした結果のレポート出力により動作の確認を行ないました。

シミュレーションの流れは、図 A.9 のようになります。

4.1.2 シミュレーション手法 2

シミュレーション手法 1 では、シミュレーション規模が大きくなってくると人手による確認が困難になってくるという問題があります。そこで、動作確認の手法としてメモリイメージ比較による誤り検出も行ないました。これは、上述のシミュレーション・モジュールに加えて、 `tester` モジュール (**SFL** モジュール・リファレンスを参照して下さい) を用意し、これを $\mathcal{K}$amuri-chip のデータバスに接続します (図 A.10)。 `tester` は `dmem` と同じサイズのメモリを持っています．そして、シミュレーション用のアセンブリ・プログラムでは、チェックしたい値を、異なるアドレスのデータメモリに逐一ストアするようなプログラミングを行ない、さらに `tester` のメモリの該当するアドレスに期待値をあらかじめ `memset` しておきます。次に挙げる例を見て下さい。

```
 1  ; [TESTER を使用した実行状態の検査]
 2  ;
 3  ; 1 から 100 までの整数の総和を求める。
 4  ;
 5  ; [変数]
 6  ;    R0 <- 総和
 7  ;    R1 <- カウンタ (1,2,3,...)
 8  ;    R2 <- 終了値を指定 (100)
 9  ;    R3 <- 求めた総和の結果をデータメモリに書き出すアドレスを指定
10  ;
11  ; [結果]
12  ;    プログラムが実行されると、R1 に 1+2+3+... の結果が代入される。
13  ;    HALT 命令で停止した時、R1 に 10 進で 5050 (0x13ba) が格納される。
14  ;    また、R1 の値と同じものが、@LSUM 番値にも格納される。
15  ; (注) 100 までの総和を求めるのに、およそ 320 clock 必要。
16
17  LLI R0, 0x00
18  LLI  R1, 0x01 ; 0x01 = decimal 1
19  LLI R2, 0x64 ; 0x64 = decimal 100
20  LLI R3, 0x00
21  LHI R0, 0x00
22  LHI R1, 0x00
23  LHI R2, 0x00
24  LHI R3, 0x80
25  @L1: ADD R0, R0, R1 ; sum = sum + counter
26  SUBI R2, R2, 0x01
27  BNEZ R2, @L1
28  ADDI R1, R1, 0x01 ; counter++
29  NOP
30  NOP
31  SW 0x0(R3), R0 ; for tester
32  NOP
33  NOP
34  NOP
35  HALT
36  NOP
37
38  .TESTER 0x8000 0xba
39  .TESTER 0x8001 0x13
```

この $\mathcal{K}$amuri 命令セットのアセンブリ・プログラムは、レジスタ R0 に $\sum_{i=1}^{100} i$ (= 0x13ba) を計算した値を設定し、31 行でレジスタ R3 (= 0x8000) が示すアドレスのデータメモリに R0 の値をストアしています。38, 39 行で .TESTER 疑似命令によって、シミュレーション中にアドレス 0x8000 ~ 0x8001 へのストアが起きた場合には、そのストア・データが 0x13ba であるかどうかをチェックするよう指示しています。なお、プログラム中で生じる全てのストアについて、その期待値を .TESTER 文で指定してやる必要がありますので、同じアドレスに異なる値を何度もストアする必要があるようなプログラムはこのシミュレーション手法では扱えないという制約があります。

シミュレーション中に tester は、$\mathcal{K}$amuri-chip のデータアクセス (アドレス, バス) を常に監視し、データ・ストアが生じると、そのアドレスの tester メモリと、バスの値を比較し、動作の誤りを検出します。

4.1.3 シミュレーション用アセンブリ・プログラム

シミュレーション用に作成したアセンブリ・プログラムは、以下のようなものです。

[alu.asm] 各 ALU 演算命令のテスト。

[gcd.asm] ユークリッド互除法を用いて最大公約数を求めるプログラム。

[forward1.asm] 各 EX，MEM，WB ステージから、フォワーディング情報が ID ステージへ伝わっているかどうかを調べるプログラム。

[forward2.asm] フォワーディングが衝突した際に、正しい優先順位でフォワーディングが行なわれるかどうかを調べるプログラム。

[jump.asm] 分岐命令をテストするプログラム。

[prime.asm] 素数を列挙するプログラム。

第 5 章

設計結果の概要

5.1 設計期間

設計にかかった期間としては、 SFL ソース管理をはじめたのが 1994 年 7 月 18 日、コンテストに応募したのが 1994 年 10 月 4 日 (日曜日 !) ですので、アーキテクチャ決定なども含めると約 3 ヶ月ということになります。しかし、みんな $\mathcal{K}$amuri-chip にかかりきりというわけではなく、また、 SFL 記述の担当はほぼ一人で、 α-release が出来上がったのが 1994 年 8 月 30 日なので、工数としては 3 ヶ月 $\times$ 6 人よりははるかに少ないといえます。

5.2 SFL 記述

SFL の記述量は、合計 2,604 行、モジュール数 48 でした。ただし、 SFL の古い言語仕様 (PARTHENON Ver.2.2.1 ベース) に基づいて記述していますので、 Ver.2.3.0 ベースの仕様に基づいて記述すれば `declare` 文の使用などにより、若干記述量を減らせるかも知れません。

5.3 合成

論理合成に際しては、細かい合成パラメータの調整などは一切せず、 PARTHENON の自動合成スクリプトによる、全自動合成で行ないました (いろいろ味付けするのが筋なんでしょうが、時間も余裕もなかったのです)。

5.4 ゲート数・消費電力・動作周波数

PARTHENON Ver.2.2.1 で $\mathcal{K}$amuri-chip を合成した結果は以下のようになりました。

ゲート数 11491

消費電力 40975.6 μW/MHz

動作周波数 5.95745 MHz (最大遅延 $1.67857e + 02$ ns)

「ASIC デザイン・コンテスト」への応募にはこの結果を出しました。

試しに PARTHENON Ver.2.3.0 で合成を行なうと以下のような結果となりました。

ゲート数 15672

消費電力 3273.8 μW/MHz

動作周波数 9.57799 MHz (最大遅延 $1.04406e + 02$ ns)

速度的には、ずいぶん良くなっています。といっても、両方とも入出力ピンに対する物理制約をきちんと設定しているわけではないので、飽くまで目安であることには違いないのですが。

いずれにせよ、最大遅延パスが `EX stage` から `pcsv` へのフォワーディング・パスであることがわかっているので (直観的にも、 ALU とフォワーディング選択回路をまたぐこのパスが一番遅いのは自明なのですが)、この部分の合成を注意深く行なえば、多少パフォーマンスが上がるのではないかと予測しています。

参考文献

[1] 小栗清 他: “ はじめてのパルテノン”, PARTHENON/CQ 版 付属マニュアル, CQ 出版, 1994

[2] 山崎 傑 / 畔津 明仁: “ マンデルブロ集合描画装置と教育用 16 ビット・マイクロプロセッサに優秀賞”, Interface No.7, pp.168 – 174, CQ 出版, 1995 年 7 月

[3] 中江 達哉: “ HDL 設計事例 PARHENON で 16 ビット・スーパースカラ CPU を設計する (前編)”, Interface No.3, pp.173 – 180, CQ 出版, 1996 年 3 月

[4] 永見 康一: “ HDL 設計事例 PARHENON で 16 ビット・スーパースカラ CPU を設計する (中編)”, Interface No.4, pp.215 – 222, CQ 出版, 1996 年 4 月

[5] 中村 寿彦: “ HDL 設計事例 PARHENON で 16 ビット・スーパースカラ CPU を設計する (完結編)”, Interface No.6, pp.207 – 214, CQ 出版, 1996 年 6 月

[6] John L.Hennessy and David A.Patterson; 富田 眞治 ・ 村上 和彦 ・ 新實 治男 訳: “コンピュータ・アーキテクチャ: 設計・実現・評価の定量的アプローチ”, 日経 BP, 1992 年

付録 A

図表

記法	意味	例	例の意味
R[ ]	レジスタ・ファイル. 16 bit レジスタの配列. レジスタ・アドレス (番号, 3bit) を添字で与え、レジスタを表す.	R[`rd`]	`rd` 番レジスタ
M[ ]	バイト・メモリ. アドレス (16bit) を添字で与え、バイト・データを表す.	M[`addr`]	アドレス `addr` のメモリ・データ
$\Leftarrow$	記憶素子へのデータ転送.	R[`rd`] $\Leftarrow$ `0x0000`	`rd` 番レジスタに、 `0x0000` を転送.
$\leftarrow$	信号線へのデータ転送.	`x` $\leftarrow$ M[0x0000]	信号線 `x` に 0x0000 番地のメモリ・データを転送.
$\mathtt{X}_0$ $\mathtt{Y}_{3..0}$ $\mathtt{Z}_{15,3..0}$	bit field 指定. 単一の数字は、その bit, $m..n$ は m 番 bit から n 番 bit までの bit field, ‘,’ は field の連結を表す.	$\mathtt{x}_{7..0} \leftarrow \mathrm{R}[\mathtt{rd}]_{15..8}$	`x` の下位バイトに `rd` 番レジスタの上位バイトを転送する.
$\mathrm{signed}^n(\mathtt{x})$	符号つき bit 拡張. `x` を 2 の補数表示の 2 進数と見て n bit に符号拡張する.	$\mathrm{signed}^{16}(\mathtt{x}_{7..0})$	`x` の下位バイトを符号つきでワード拡張したデータ.
$\mathrm{unsigned}^n(\mathtt{x})$	符号なし bit 拡張. `x` を絶対値表示の 2 進数と見て n bit に拡張する.	$\mathrm{unsigned}^{16}(\mathtt{x}_{7..0})$	`x` の下位バイトを符号なしでワード拡張したデータ.

表 A.1: 記法

	5 bit	3 bit	3 bit	3 bit	2 bit
R 型	命令操作コード	rs1	rs2	rd	未使用

	5 bit	3 bit	3 bit	5 bit
I5 型	命令操作コード	rs/rs1	rd/rs2	imm

	5 bit	3 bit	8 bit
I8 型	命令操作コード	rs/rd	imm

	5 bit	11 bit
J 型	命令操作コード	imm

図 A.1: 命令形式の分類

IF	**ID**	**EX**	**MEM**	**WB**
命令フェッチ	命令デコード、 オペランド・フェッチ	命令実行	メモリ・アクセス	レジスタ書き込み

図 A.2: パイプライン・ステージ

命令	型	動作	備考
`lb    rd, imm(rs )`	I5	$R[\mathtt{rd}] \Leftarrow \mathrm{signed}^{16}(M[\mathrm{signed}^{16}(\mathtt{imm})+R[\mathtt{rs}]])$	Load Byte
`lbu   rd, imm(rs )`	I5	$R[\mathtt{rd}] \Leftarrow \mathrm{unsigned}^{16}(M[\mathrm{signed}^{16}(\mathtt{imm})+R[\mathtt{rs}]])$	- Unsigned
`sb` $\mathtt{imm(rs_1), rs_2}$	I5	$M[\mathrm{signed}^{16}(\mathtt{imm})+\mathtt{rs}_1] \Leftarrow R[\mathtt{rs}_2]_{7..0}$	Store Byte
`lw    rd, imm(rs )`	I5	$R[\mathtt{rd}]_{15..8} \Leftarrow M[\mathrm{signed}^{16}(\mathtt{imm})+R[\mathtt{rs}]+1]$ $R[\mathtt{rd}]_{7..0} \Leftarrow M[\mathrm{signed}^{16}(\mathtt{imm})+R[\mathtt{rs}]]$	Load Word
`sw` $\mathtt{imm(rs_1), rs_2}$	I5	$M[\mathrm{signed}^{16}(\mathtt{imm})+\mathtt{rs}_1+1] \Leftarrow R[\mathtt{rs}_2]_{15..8}$ $M[\mathrm{signed}^{16}(\mathtt{imm})+\mathtt{rs}_1] \Leftarrow R[\mathtt{rs}_2]_{7..0}$	Store Word
`lhi   rd, imm`	I8	$R[\mathtt{rd}]_{15..8} \Leftarrow \mathtt{imm}$	Load High Imm.
`lli   rd, imm`	I8	$R[\mathtt{rd}]_{7..0} \Leftarrow \mathtt{imm}$	Load Low Imm.
`link  rd, imm`	I8	$R[\mathtt{rd}] \Leftarrow \mathrm{unsigned}^{16}(\mathrm{pc}) + \mathrm{signed}^{16}(\mathtt{imm})$	pc 退避
`add` $\mathtt{rd, rs_1, rs_2}$	R	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] + R[\mathtt{rs}_2]$	ADD
`addi  rd, rs , imm`	I5	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] + \mathrm{unsigned}^{16}(\mathtt{imm})$	ADD Imm.
`sub` $\mathtt{rd, rs_1, rs_2}$	R	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] - R[\mathtt{rs}_2]$	SUBtract
`subi  rd, rs , imm`	I5	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] - \mathrm{unsigned}^{16}(\mathtt{imm})$	
`and` $\mathtt{rd, rs_1, rs_2}$	R	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1]\ \&\ R[\mathtt{rs}_2]$	AND
`andi  rd, rs , imm`	I5	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1]\ \&\ \mathrm{unsigned}^{16}(\mathtt{imm})$	
`or` $\mathtt{rd, rs_1, rs_2}$	R	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \mid R[\mathtt{rs}_2]$	OR
`ori   rd, rs , imm`	I5	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \mid \mathrm{unsigned}^{16}(\mathtt{imm})$	
`xor` $\mathtt{rd, rs_1, rs_2}$	R	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \oplus R[\mathtt{rs}_2]$	XOR
`xori  rd, rs , imm`	I5	$R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \oplus \mathrm{unsigned}^{16}(\mathtt{imm})$	
`sl` $\mathtt{rd, rs_1, rs_2}$	R	**if** $(\mathrm{signed}^{16}(R[\mathtt{rs}_2]_{15,3..0}) \geq 0)$ **then** $R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \ll \mathrm{signed}^{16}(R[\mathtt{rs}_2]_{15,3..0})$ **else** $R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \gg -\mathrm{signed}^{16}(R[\mathtt{rs}_2]_{15,3..0})$	Shift Logical
`sli` $\mathtt{rd, rs_1, imm}$	I5	**if** $(\mathtt{imm}_{4..0} \geq 0)$ **then** $R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \ll \mathrm{signed}^{16}(\mathtt{imm}_{4..0})$ **else** $R[\mathtt{rd}] \Leftarrow R[\mathtt{rs}_1] \gg -\mathrm{signed}^{16}(\mathtt{imm}_{4..0})$	
`slt` $\mathtt{rd, rs_1, rs_2}$	R	**if** $(R[\mathtt{rs}_1] < R[\mathtt{rs}_2])$ **then** $R[\mathtt{rd}] \Leftarrow$ 0x0001 **else** $R[\mathtt{rd}] \Leftarrow$ 0x0000	Set Less Than
`slti` $\mathtt{rd, rs_1, imm}$	I5	**if** $(R[\mathtt{rs}_1] < \mathrm{signed}^{16}(\mathtt{imm}))$ **then** $R[\mathtt{rd}] \Leftarrow$ 0x0001 **else** $R[\mathtt{rd}] \Leftarrow$ 0x0000	
`sgt` $\mathtt{rd, rs_1, rs_2}$	R	**if** $(R[\mathtt{rs}_1] > R[\mathtt{rs}_2])$ **then** $R[\mathtt{rd}] \Leftarrow$ 0x0001 **else** $R[\mathtt{rd}] \Leftarrow$ 0x0000	Set Greater Than
`sgti` $\mathtt{rd, rs_1, imm}$	I5	**if** $(R[\mathtt{rs}_1] > \mathrm{signed}^{16}(\mathtt{imm}))$ **then** $R[\mathtt{rd}] \Leftarrow$ 0x0001 **else** $R[\mathtt{rd}] \Leftarrow$ 0x0000	
`beqz  rs , imm`	I8	**if** $(R[\mathtt{rs}] == 0)$ **then** $\mathrm{pc} \Leftarrow \mathrm{pc} + \mathrm{signed}^{14}(\mathtt{imm})$	0 分岐
`bnez  rs , imm`	I8	**if** $(R[\mathtt{rs}]\ \mathbf{!=}\ 0)$ **then** $\mathrm{pc} \Leftarrow \mathrm{pc} + \mathrm{signed}^{14}(\mathtt{imm})$	非 0 分岐
`j     imm`	J	$\mathrm{pc} \Leftarrow \mathrm{pc} + \mathrm{signed}^{14}(\mathtt{imm})$	rerative Jump
`jr    rs`	I8	$\mathrm{pc} \Leftarrow (\mathrm{pc} + R[\mathtt{rs}]) \ \mathbf{mod}\ 2^{14}$	indirect Jump
`rst`	J	$\mathrm{pc} \Leftarrow 0$	ReSeT
`halt`	J	halt execution	HALT
`nop`	J	no operation	No OPeration

注: プログラム カウンタ pc は 14 bit の値を保持し、 2 命令語境界 (4 byte) を指示単位とする

表 A.2: 命令セット

命令	符号	命令	符号	命令	符号	命令	符号
`lb`	00000	`add`	01000	`xor`	10000	`beqz`	11000
`lbu`	00001	`addi`	01001	`xori`	10001	`bnez`	11001
`sb`	00010	`sub`	01010	`sl`	10010	`j`	11010
`lw`	00011	`subi`	01011	`sli`	10011	`jr`	11011
`sw`	00100	`and`	01100	`slt`	10100	`rst`	11100
`lhi`	00101	`andi`	01101	`slti`	10101	`halt`	11101
`lli`	00110	`or`	01110	`sgt`	10110	`nop`	11110
`link`	00111	`ori`	01111	`sgti`	10111		

表 A.3: 命令操作コードの符号割り当て

load	`lb, lbu, lw`
store	`sb, sw`
loadi	`lhi, lli`
alu	`add, sub, and, or, xor, sl, slt, sgt,` `addi, subi, andi, ori, xori, sli, slti, sgti`
controll	`beqz, bnez, j, jr, rst, halt`
link	`link`
nop	`nop`

表 A.4: 命令グループ

		B: 後続命令						
		load	store	loadi	alu	controll	link	nop
A: 先行命令	load	×	×	raw	raw	×	○	○
	store	×	×	○	○	×	○	○
	loadi	×	×	raw	raw	×	○	○
	alu	×	×	raw	raw	×	○	○
	controll	×	×	○	○	×	○	○
	link	×	×	raw	raw	×	○	○
	nop	×	×	○	○	×	○	○

×: ペアリング禁止, ○: ペアリング可能, raw: レジスタの組合せに制限あり

表 A.5: 命令ペアのペアリング規則

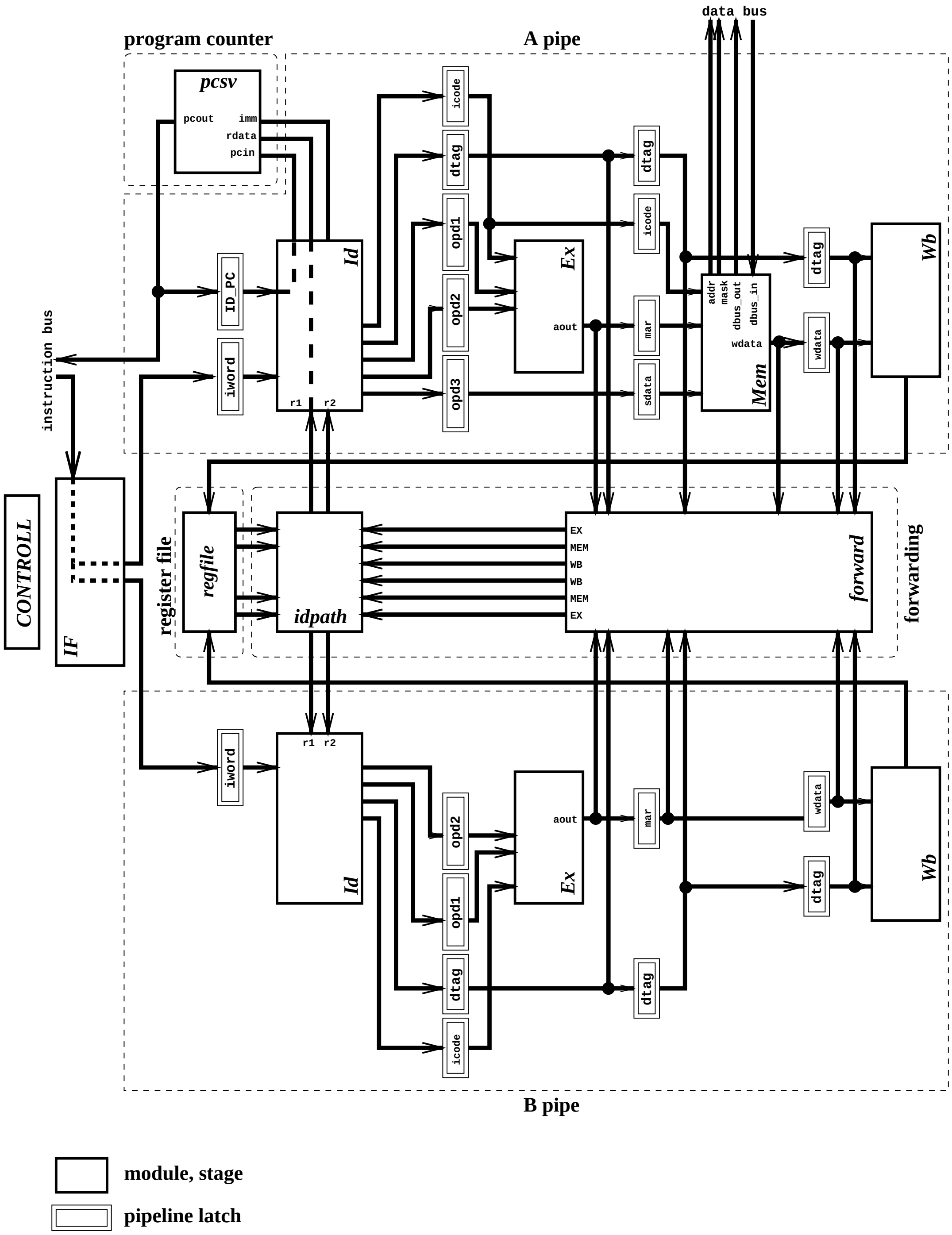
data bus
program counter
A pipe
pcsv
pcout
imm
rdata
pcin
icode
dtag
opd1
opd2
opd3
Id
Ex
aout
mar
sdata
addr
mask
dbus_out
dbus_in
wdata
Mem
Wb
ID_PC
iword
instruction bus
r1
r2
CONTROLL
IF
register file
regfile
idpath
EX
MEM
WB
WB
MEM
EX
forward
forwarding
B pipe
module, stage
pipeline latch

図 A.3: ブロックの概要

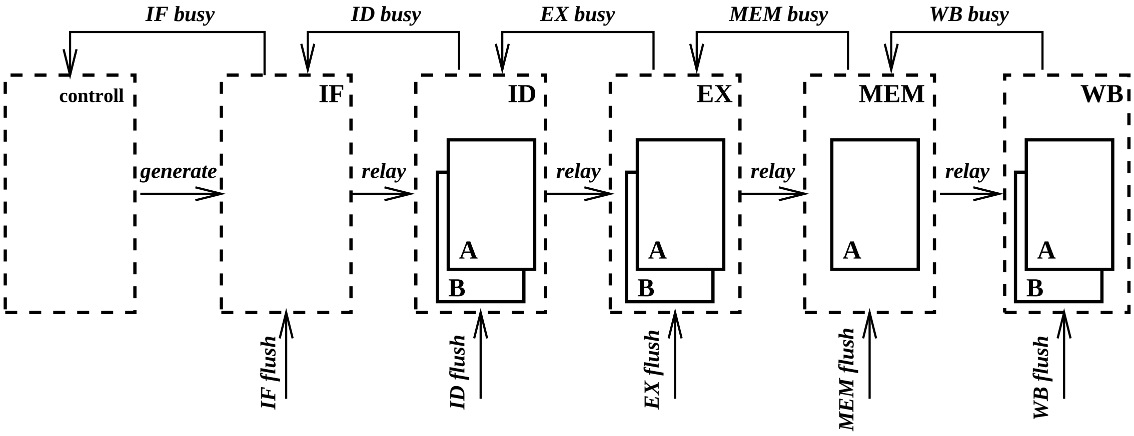

図 A.4: パイプラインの制御

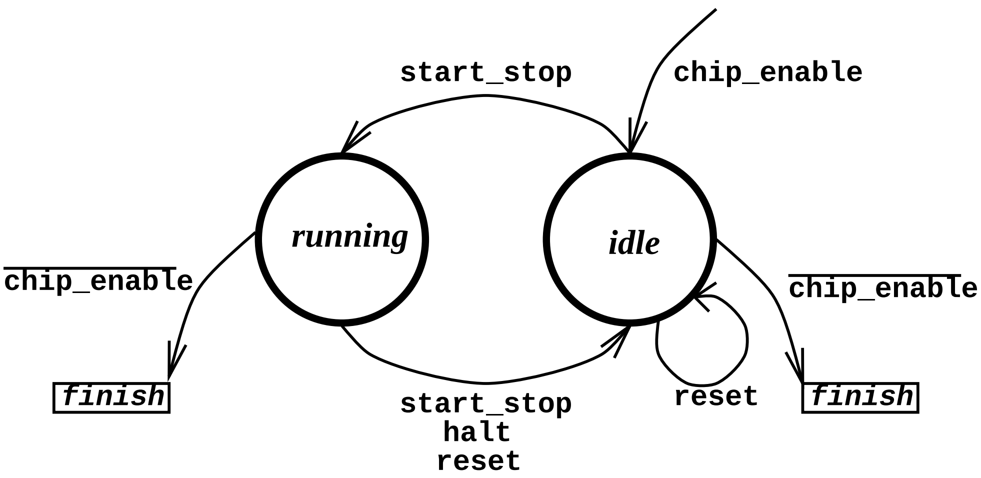

図 A.5: CONTROLL stage の状態遷移

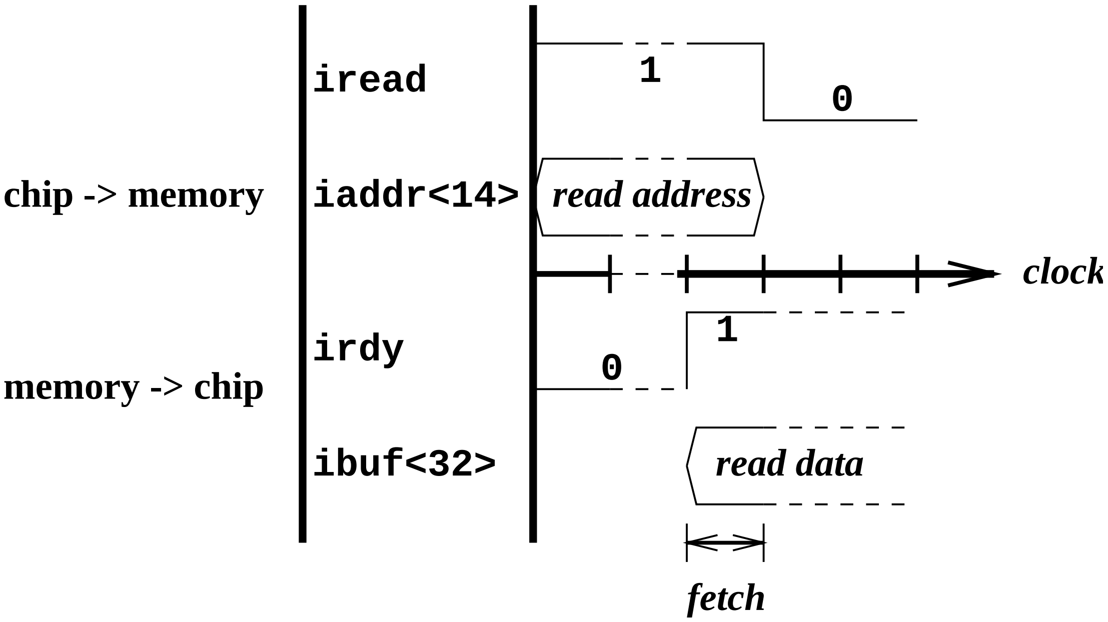

図 A.6: 命令メモリのアクセス・インタフェース

stage	A/B	name	descrption
IF	A	-	
	B	-	
ID	A	ID_PC_A<14> IR_A<16>	先行命令のアドレス 命令ワード
	B	IR_B<16>	後続命令の命令ワード
EX	A	EX_ICODE_A<5> EX_DTAG_A<4> OPD1_A<16> OPD2_A<16> OPD3_A<16>	先行命令の操作コード destination tag オペランド 1 (ALU 入力) オペランド 2 (ALU 入力) オペランド 3 (ストアデータ)
	B	EX_ICODE_B<5> EX_DTAG_B<4> OPD1_B<16> OPD2_B<16>	後続命令の操作コード destination tag オペランド 1 (ALU 入力) オペランド 2 (ALU 入力)
MEM	A	MEM_ICODE_A<5> MEM_DTAG_A<4> SDATA_A<16> MAR_A<16>	先行命令の操作コード destination tag ストアデータ ライトバックデータ
	(B)	MEM_DTAG_B<4> MAR_B<16>	後続命令の destination tag ライトバックデータ
WB	A	WB_DTAG<4> WDATA_A<16>	先行命令の destination tag ライトバックデータ
	B	WB_DTAG<4> WDATA_A<16>	後続命令の destination tag ライトバックデータ

表 A.6: 各ステージが参照するラッチ

stage	A	B	
IF	ID_PC_A $\Leftarrow$ current program counter value IR_A $\Leftarrow$ ibus$_{15..8}$	IR_B $\Leftarrow$ ibus$_{7..0}$	
ID	EX_ICODE_A $\Leftarrow$ IR_A$_{15..11}$ *r1, r2* $\leftarrow$ forwarding data (if needed) EX_DTAG_A, OPD*_A $\Leftarrow$ (表 A.8)	EX_ICODE_B $\Leftarrow$ IR_B$_{15..11}$ *r1, r2* $\leftarrow$ forwarding data (if needed) EX_DTAG_A, OPD*_B $\Leftarrow$ (表 A.8)	
	r1 $\leftarrow$ forwarding data consult PCSV to branch (with *r1*)		*1
EX	MEM_ICODE_A $\Leftarrow$ EX_ICODE_A *func* $\leftarrow$ decode EX_ICODE_A MAR_A $\Leftarrow$ ALU(OPD1_A, OPD2_A, func) SDATA_A $\Leftarrow$ *opd3*	*func* $\leftarrow$ decode EX_ICODE_A MAR_B $\Leftarrow$ ALU(OPD1_B, OPD2_B, func)	*1
MEM	WB_DTAG_A $\Leftarrow$ MEM_DTAG_A	WB_DTAG_B $\Leftarrow$ MEM_DTAG_B	
	daddr, mask $\leftarrow$ align MAR_A WDATA_A $\Leftarrow$ unalign and expand dbus_in		*2
	daddr, mask $\leftarrow$ align MAR_A dbus_out $\leftarrow$ align SDATA_A		*3
	WDATA_A $\Leftarrow$ MEM_MAR_A	WDATA_B $\Leftarrow$ MEM_MAR_B	*4
WB	*is_reg, addr* $\leftarrow$ decode WB_DTAG_A if (*is_reg*) then R[*addr*] $\Leftarrow$ WDATA_A	*is_reg, addr* $\leftarrow$ decode WB_DTAG_B if (*is_reg*) tnen R[*addr*] $\Leftarrow$ WDATA_B	

controll group *2 load group *3 store group *4 load/store group 以外

表 A.7: 各ステージにおけるデータ転送

命令 group	DTAG	OPD1	OPD2	OPD3
load	0b0||rd	R[rs]	imm5	0x0000
store	0b1111	R[rs$_1$]	imm5	R[rs$_2$]
loadi	0b0||rd	R[rs$_1$]	imm8	0x0000
alu (RR)	0b0||rd	R[rs$_1$]	R[rs$_2$]	0x0000
(RI)	0b0||rd	R[rs$_1$]	imm5	0x0000
controll	(受け渡しなし)			
link	0b0||rd	pc	imm8	0x0000
nop	0b1111	0x0000	0x0000	0x0000

表 A.8: ID $\rightarrow$ EX の各受け渡しデータ

- **kamuri**
  - **regfile** RF
  - **forward** FWD
  - **Id** ID_A
  - **Id** ID_B
  - **Wb** WB_A
  - **Wb** WB_B
  - **pcsv** PCSV
  - **Mem** MEM_A
  - **Ex** EX_A
  - **Ex** EX_B
  - **idpath** IPATH_A
  - **idpath** IPATH_B

- **regfile**
  - **reg16cell8** R
    - **dec3to8** WAD0
    - **reg16** R4
    - **dec3to8** WAD1
    - **reg16** R5
    - **reg16** R6
    - **reg16** R7
    - **dec3to8** RAD0
    - **dec3to8** RAD1
    - **reg16** R0
    - **dec3to8** RAD2
    - **reg16** R1
    - **dec3to8** RAD3
    - **reg16** R2
    - **reg16** R3
  - **eq3** EQ

- **forward**
  - **eq3** EA_EB
  - **tagdec** EA_D
  - **tagdec** EB_D
  - **eq3** MA_EA
  - **eq3** MB_EA
  - **eq3** MA_EB
  - **tagdec** MA_D
  - **eq3** MB_EB
  - **tagdec** MB_D
  - **eq3** MA_MB
  - **eq3** WA_EA
  - **eq3** WB_EA
  - **eq3** WA_EB
  - **tagdec** WA_D
  - **eq3** WB_EB
  - **tagdec** WB_D
  - **eq3** WA_MA
  - **eq3** WB_MA
  - **eq3** WA_MB
  - **eq3** WB_MB
  - **eq3** WA_WB

- **Id**
  - **idecoder** D
    - **icodedec** D
      - **dec5to32** D
    - **expand11to16** E11
    - **expand5to16** E5
    - **expand8to16** E8

- **idpath**
  - **eq3** R1_EA
  - **eq3** R2_EA
  - **eq3** R1_EB
  - **eq3** R2_EB
  - **eq3** R1_MA
  - **eq3** R2_MA
  - **eq3** R1_MB
  - **eq3** R2_MB
  - **eq3** R1_WA
  - **eq3** R2_WA
  - **eq3** R1_WB
  - **eq3** R2_WB

- **pcsv**
  - **eq16** E
    - **eq4** E0
    - **eq4** E1
    - **eq4** E2
    - **eq4** E3
  - **adder16** Ai
    - **cla4** CLA
    - **adder4** A0
    - **adder4** A1
    - **adder4** A2
    - **adder4** A3
  - **adder16** Ar
    - **cla4** CLA
    - **adder4** A0
    - **adder4** A1
    - **adder4** A2
    - **adder4** A3

- **Mem**
  - **mdecoder** D
    - **icodedec** D
      - **dec5to32** D
  - **expand8to16** E
  - **align** A

- **Wb**
  - **tagdec** TD

- **eq4**
  - **eq2** EH
  - **eq2** EL

- **adder4**
  - **cla4** CLA

図 A.7: $\mathcal{K}$amuri-chip の module 階層構造

- **Ex**
  - **edecoder** D
    - **icodedec** D
      - **dec5to32** D
  - **alu16** ALU
    - **adder16** ADD
      - **cla4** CLA
      - **adder4** A0
      - **adder4** A1
      - **adder4** A2
      - **adder4** A3
    - **dec4to16** FD
      - **dec2to4** DH
      - **dec2to4** DL
      - **gate4** G0
      - **gate4** G1
      - **gate4** G2
      - **gate4** G3
    - **band16** AND
      - **band4** BA0
      - **band4** BA1
      - **band4** BA2
      - **band4** BA3
    - **eq16** EQ
      - **eq4** E0
      - **eq4** E1
      - **eq4** E2
      - **eq4** E3
    - **shift16** SL
      - **dec5to32** D
    - **bor16** OR
      - **bor4** BO0
      - **bor4** BO1
      - **bor4** BO2
      - **bor4** BO3
    - **bnot16** NOT
      - **bnot4** BN0
      - **bnot4** BN1
      - **bnot4** BN2
      - **bnot4** BN3
    - **bxor16** XOR
      - **bxor4** BX0
      - **bxor4** BX1
      - **bxor4** BX2
      - **bxor4** BX3

- **dec5to32**
  - **dec3to8** DH
    - **dec2to4** D
    - **gate4** GH
    - **gate4** GL
  - **dec2to4** DL
  - **gate4** G0
  - **gate4** G1
  - **gate4** G2
  - **gate4** G3
  - **gate4** G4
  - **gate4** G5
  - **gate4** G6
  - **gate4** G7

図 A.8: $\mathcal{K}$amuri-chip の module 階層構造 (続き)

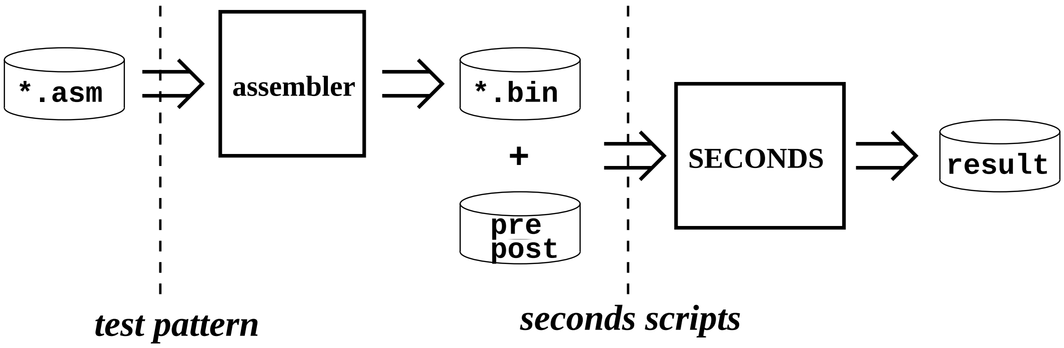

図 A.9: プログラム実行のシミュレーション

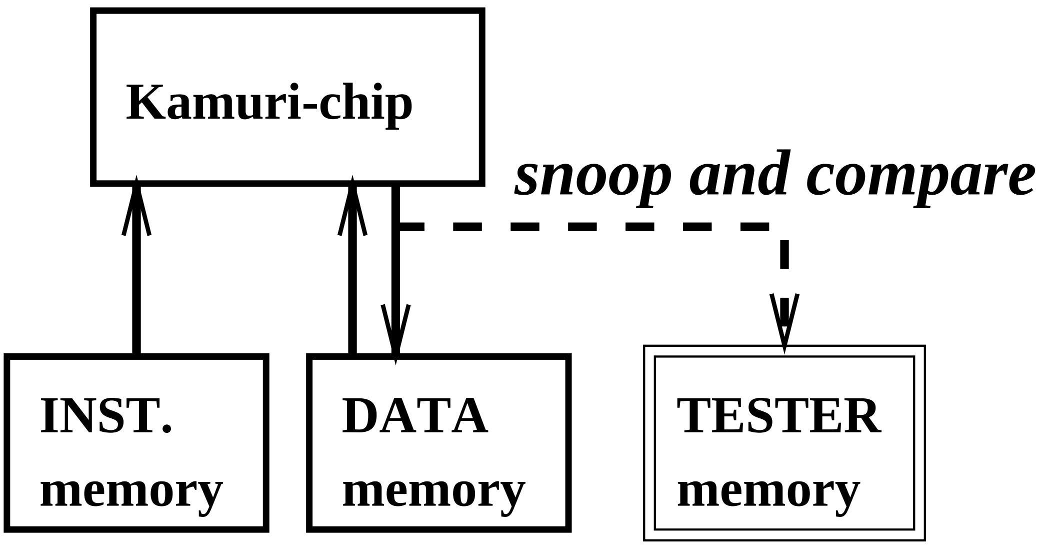

図 A.10: `tester` モジュールによる、データアクセスの監視

付録 B

SFL モジュール・リファレンス

凡例

(module の名称)

interface

(外部端子のリスト)

instruct signals

(制御端子と引数のリスト)

description
(module の動作の概要説明)

B.1　基本モジュール

dec2to4 (dec3to8, dec4to16)

interface

input	`in` $< 2 >$	2進データ
output	`out` $< 4 >$	デコード結果

instruct signals

instrin	`do`(in)	module 起動

description
いわゆる 2 bit decoder です。制御入力 `do` の起動により、 `in` の値を 2 bit binary と見て、その値に対応する bit のみ 1 とした 4 bit binary を `out` に出力します。例えば、 `do(0b10).out == 0b0100` です。 `dec3to8`, `dec4to16`, `dec5to32` も、 bit 幅の違いを除いて `dec2to4` に準じます。

eq2 (eq3, eq4, eq16)

interface

input	`x` $< 2 >$	2 進データ
output	`y` $< 2 >$	デコード結果

instruct signals

instrin	do(x, y)	module 起動
instrout	eq()	x == y のとき起動される

description

制御入力 `do` の起動により、 2 つの 2 bit binary `x`, `y` が等しいかどうかを比較します。等しい時に `eq == 1` となります。例えば、 `do(0b11, 0b10).eq == 0`, `do(0b01, 0b01).eq == 1` です。 `eq3`, `eq4`, `eq16` も、 `x`, `y` の bit 幅が異なるだけで、 `eq2` に準じます。

expand5to16 (expand8to16, expand11to16)

interface

input	in $< 5 >$	5 bit 2 進
output	out $< 16 >$	16 bit への拡張結果

instruct signals

instrin	signed(in)	符号つき拡張
instrin	unsigned(in)	符号なし拡張

description

5 bit binary を 16 bit binary へ拡張します。 `signed`, `unsigned` は、 `in` をそれぞれ 2 の補数表示、絶対値表示 と見て拡張を行ないます。したがって例えば

```
  signed(0b10000).out  ==  0b1111111111110000
unsigned(0b10000).out  ==  0b0000000000010000
```

となります。 `signed`, `unsigned` を同時に起動することは禁止されています。 `expand8to16`, `expand11to16` も `in` の bit 幅が違うだけで、 `expand5to16` に準じます。

gate4

interface

input	in $< 4 >$	4 bit data 入力
input	enable $< 1 >$	through / zero
output	out $< 4 >$	4 bit data 出力

instruct signals

instrin	do($in, enable$)	module 起動

description

`do(in, enable)` が起動されると、 `enable` の値にしたがって `out` に `in` または `0b0000` を出力します。例えば `do(x, 0b1).out == x`, `do(x, 0b0).out = 0b0000` です。

tagdec

interface

input	tag $< 4 >$	destination tag
output	raddr $< 3 >$	register address

instruct signals

instrin do(*tag*)	module 起動
instrout is_reg(*raddr*)	destination が register のとき起動される

description

destination tag とは、命令の書き込み先を表現する 4 bit data で、次のように定義されます。

`0b0***`	書き込み先は register であり、下位 3 bit が register address を表す
`0b1***`	書き込み先は register 以外である

`do(tag)` が起動されると、この定義にしたがって、書き込み先が register である場合 `is_reg` が起動されます。またその時、 register address が `raddr` に出力されます。

B.2 ALU

adder4 (adder16)

interface

input x < 4 >	被加数
input y < 4 >	加数
input cin < 1 >	キャリー入力
output z < 4 >	加算結果
output cout < 1 >	キャリー出力
output sovf < 1 >	符号つき加算オーバーフロー
output pout < 1 >	キャリー伝搬信号
output gout < 1 >	キャリー生成信号

instruct signals

instrin do(*in*)	*module*起動

description

4 bit の加算器です。 `do(x, y, cin)` の起動により、 `z` に `x + y + cin` の結果が、 `cout` に加算のキャリーが出力されます。また、 `pout`, `gout` には、桁上げ先見回路 (`cla` module) のための信号が出力されます。 sovf は、 `x`, `y` をともに 2 の補数表示の binary と見た場合に、加算で overflow または underflow が生じているならば 1 となります. 絶対値表示の 2 数の加算に用いる場合には意味を持ちません。 `adder16` は、 `adder4` を 4 個と `cla` を 1 個用いた 16 bit 加算器であり、 `x`, `y` の bit 幅が異なるのみで、 `adder4` に準じます。

cla4

interface

input pin < 4 >	各桁のキャリー伝搬信号入力
input gin < 4 >	各桁のキャリー生成信号入力
input cin < 1 >	キャリー入力
output cout < 4 >	各桁のキャリー出力
output pout < 1 >	キャリー伝搬信号出力
output gout < 1 >	キャリー生成信号出力

instruct signals

$$\textbf{instrin}\ \ \texttt{do}(pin, gin, cin) \qquad module\text{起動}$$

description
4 bit の キャリー生成器です。いわゆる桁上げ先見回路です。

band4 (band16, bnot4, bnot16, bor4, bor16, bxor4, bxor16)

interface

input	x < 4 >	オペランド
input	y < 4 >	オペランド
output	z < 4 >	演算結果

instruct signals

$$\textbf{instrin}\ \ \texttt{do}(in) \qquad \text{module 起動}$$

description
4 bit の 論理積を求めます。 `do(x, y)` の起動により、 `z` に `x`, `y` の bit 毎の論理積をとった結果を出力します。 `band4` を 4 個 用いて 16 bit の演算器にしたのが `band16` です。
　同様に bit 毎の論理演算を行なう module として

bnot4, bnot16	論理否定
bor4, bor16	論理和
bxor4, bxor16	排他的論理和

があります。

shift16

interface

input	x < 16 >	被シフトデータ
input	y < 16 >	シフト量
output	z < 16 >	シフト結果

instruct signals

$$\textbf{instrin}\ \ \texttt{do}(x, y) \qquad \text{module 起動}$$

description
16 bit の両方向シフト回路です。シフトの方向および量は、 `y<15>||y<3:0>` の 5 bit を 2 の補数表示と見て、正ならば左シフト (≪)、負ならば右シフト (≫) とし、絶対値をシフト量と定めます。例えば

$$\begin{aligned}
\texttt{do(0b0000000000000001, 0b0}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}\texttt{0100).z} &\quad == \quad \texttt{0b0000000000010000} \\
\texttt{do(0b1000000000000000, 0b1}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}\texttt{1100).z} &\quad == \quad \texttt{0b0001000000000000} \\
\texttt{do(x, 0b1}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}{*}\texttt{0000).z} &\quad == \quad \texttt{0x0000}
\end{aligned}$$

となります。

alu16

interface

input	x $<16>$	被シフトデータ
input	y $<16>$	シフト量
input	func $<4>$	演算コード
output	z $<16>$	演算結果

instruct signals

instrin	do$(x, y, func)$	module 起動

description

16 bit の算術論理演算回路 (ALU) です。 12 種の演算が用意されており、それらは、 do(x，y，func) の起動とともに渡される func の値によって選択され、演算結果が z に出力されます. func の符号割り当ては以下のようになっています。

0b0000	加算	$z = x + y$
0b0001	減算	$z = x - y$
0b0010	論理積	$z = x\&y$
0b0011	論理和	$z = x \vert y$
0b0100	排他的論理和	$z = x \oplus y$
0b0101	シフト	$z = x \ll\gg y$
0b0110	小なり比較	$z = x > y ? 0x0001 : 0x0000$
0b0111	大なり比較	$z = x < y ? 0x0001 : 0x0000$
$0b1*00$	バイト連結	$z = x$
$0b1*01$	バイト連結	$z = x < 15:8 > \vert\vert y < 7:0 >$
$0b1*10$	バイト連結	$z = y < 15:8 > \vert\vert x < 7:0 >$
$0b1*11$	バイト連結	$z = y$

シフト演算の定義については shift16 を参照して下さい。

B.3 レジスタ・ファイル

reg16

interface

input	din $<16>$	書き込みデータ
output	dout $<16>$	読み出しデータ

instruct signals

instrin	read()	読みだし要求
instrin	write(din)	書き込み要求

description

16 bit の register 回路です。 `write(din)` の起動によって、 `din` の値が 内部の register に取り込まれます。また、 `read()` の起動によって `dout` に register の値が出力されますが、同時に `write` も起動されていた場合、バイパスによって書き込み後の register の値と同一の値が `dout` に出力されます。例えば、内部の register の値が `0xffff` であったところに、 `write(0x1122)` が起動されていたとすると、 `read().dout == 0x1122` となり、次のクロックで register の値も `0x1122` となります。

reg16cell8

interface

input	`raddr0` $<3>$	読み出しポート 0 の register address
input	`raddr1` $<3>$	1
input	`raddr2` $<3>$	2
input	`raddr3` $<3>$	3
input	`waddr0` $<3>$	書き込みポート 0 の register address
input	`waddr1` $<3>$	1
input	`din0` $<16>$	書き込みポート 0 の 書き込みデータ
input	`din1` $<16>$	1
output	`dout0` $<16>$	読み出しポート 0 の読み出しデータ
output	`dout1` $<16>$	1
output	`dout2` $<16>$	2
output	`dout3` $<16>$	3

instruct signals

instrin	`read0`(*raddr0*)	読みだしポート 0 の読み出し要求
instrin	`read1`(*raddr1*)	1
instrin	`read2`(*raddr2*)	2
instrin	`read3`(*raddr3*)	3
instrin	`write0`(*waddr0*, *din0*)	書き込みポート 0 の書き込み要求
instrin	`write1`(*waddr1*, *din1*)	1

description

`reg16` を 8 個集めた、 register 集合です。アクセスポートとして、 4 個の読み出しポートと 2 個の書き込みポートをもっており、これらは独立にアクセスすることができます。ただし、 2 個の書き込みポートが同一の register に書き込み要求を出すことは禁止されています (この module ではチェックしていません)。アクセスのセマンティクスについては `reg16` を参照して下さい。

regfile

interface

input	`raddrA1` $<3>$	読み出しポート A1 の register address
input	`raddrA2` $<3>$	A2
input	`raddrB1` $<3>$	B1
input	`raddrB2` $<3>$	B2
input	`waddrA` $<3>$	書き込みポート A の register address
input	`waddrB` $<3>$	B

input dinA $< 16 >$	書き込みポート A の 書き込みデータ	
input dinB $< 16 >$	B	
output doutA1 $< 16 >$	読み出しポート A1 の読み出しデータ	
output doutA2 $< 16 >$	A2	
output doutB1 $< 16 >$	B1	
output doutB2 $< 16 >$	B2	

instruct signals

instrin readA1($raddrA1$)	読みだしポート A1 の読み出し要求
instrin readA2($raddrA2$)	A2
instrin readB1($raddrB1$)	B1
instrin readB2($raddrB2$)	B2
instrin writeA($waddrA, dinA$)	書き込みポート A の書き込み要求
instrin writeB($waddrB, dinB$)	B

description
reg16cell8 の アクセスポートを A, B の 2 つにわけ、順序つきの書き込みアクセスセマンティクスを実現します。 A, B の両方の書き込み要求が同時に起動された場合、 register および 同時に起動された読み出し要求は、A $\rightarrow$ B の順で書き込みが行なわれた場合の値を受け取ることになります。

B.4 プログラム・カウンタ

pcsv

interface

input rdata $< 16 >$	分岐命令のレジスタオペランド 入力
input pcin $< 14 >$	分岐命令の program counter 値 入力
input imm $< 16 >$	分岐命令の即値オペランド 入力
output pcout $< 14 >$	現時刻の program counter 値 出力

instruct signals

instrin reset()	program couter リセット
instrin eqz($rdata, pcin, imm$)	ゼロ分岐要求
instrin eqz($rdata, pcin, imm$)	非ゼロ分岐要求
instrin abs($rdata$)	絶対ジャンプ要求
instrin rel($pcin, imm$)	相対ジャンプ要求
instrin inc()	program couter インクリメント要求 (+1)
instrin read()	program couter 値の読み出し要求

description
14 bit の register を program counter として持ち、分岐、インクリメントなどを制御する回路です。分岐条件を判定するための 16 bit ゼロ比較器を 1 個備えており、条件の成立 / 不成立は pcsv で判定することができます。また、実効アドレスを生成するための加算器も備えています。分岐要求 (eqz, nez, abs, rel) の起動は排他的である必要があり、同時に起動することは禁止されています。 inc, read とそれらのうちの一つを同時に起動することは構いません。その場合、分岐が成立すると、 inc は無視され、また同時に read が起動されていると pcout には、 (分岐先の program couter 値ではなく) その時刻の program couter 値が出力されます。

B.5　フォワーディング

forward

interface

input exA_dtag $< 4 >$	A pipe:EX stage の命令の destination tag
input exB_dtag $< 4 >$	B pipe:EX stage
input exA_aout $< 16 >$	A pipe:EX stage の命令の ALU 出力
input exB_aout $< 16 >$	B pipe:EX stage
input memA_dtag $< 4 >$	A pipe:MEM stage の命令の destination tag
input memB_dtag $< 4 >$	B pipe:MEM stage
input memA_mar $< 16 >$	A pipe:MEM stage の命令の MAR 出力
input memB_mar $< 16 >$	B pipe:MEM stage
output wbA_dtag $< 4 >$	A pipe:WB stage の命令の destination tag
output wbB_dtag $< 4 >$	B pipe:WB stage
output wbA_wdata $< 16 >$	A pipe の WB stage の命令の WDATA の値
output wbB_wdata $< 16 >$	B pipe:WB stage
output fwdEAaddr $< 3 >$	A pipe:EX stage のフォワード・アドレス
output fwdMBaddr $< 3 >$	B pipe:EX stage
output fwdEAdata $< 16 >$	A pipe:EX stage のフォワード・データ
output fwdEBdata $< 16 >$	B pipe:EX stage
output fwdMAaddr $< 3 >$	A pipe:MEM stage のフォワード・アドレス
output fwdMBaddr $< 3 >$	B pipe:MEM stage
output fwdMAdata $< 16 >$	A pipe:MEM stage のフォワード・データ
output fwdEBdata $< 16 >$	B pipe:MEM stage
output fwdWAaddr $< 3 >$	A pipe の WB stage のフォワード・アドレス
output fwdWBaddr $< 3 >$	B pipe
output fwdWAdata $< 16 >$	A pipe の WB stage のフォワード・データ
output fwdWBdata $< 16 >$	B pipe

instruct signals

instrin do()	module 起動
instrin exA_active(exA_dtag, exA_aout)	A pipe:EX stage は命令を処理中
instrin exB_active(exB_dtag, exB_aout)	B pipe:EX stage
instrin memA_active($memA\_dtag, memA\_mar$)	A pipe:MEM stage は命令を処理中
instrin memB_active($memB\_dtag, memB\_mar$)	B pipe:MEM stage
instrin wbA_active(wbA_dtag, wbA_wdata)	A pipe:WB stage は命令を処理中
instrin wbB_active(wbB_dtag, wbB_wdata)	B pipe:WB stage
instrout fwdEA($fwdEAaddr, fwdEAdata$)	A pipe:EX stage のフォワード起動
instrout fwdEB($fwdEBaddr, fwdEBdata$)	B pipe:EX stage
instrout fwdMA($fwdMAaddr, fwdMAdata$)	A pipe:MEM stage のフォワード起動
instrout fwdMB($fwdMBaddr, fwdMBdata$)	B pipe:MEM stage
instrout fwdWA($fwdWAaddr, fwdWAdata$)	A pipe:WB stage
instrout fwdWB($fwdWBaddr, fwdWBdata$)	B pipe:WB stage

description

フォワーディング・データの ‘ 供給回路’ です。 {A, B} pipe それぞれの {EX, MEM, WB} ステージという計 6 つのステージの命令の destination tag をデコードし、書き込み先がレジスタの場合は、そのアドレスおよび書き込みデータをのせて `fwd{E, M, W}{A, B}` を起動します。ただし、 6 つのステージのいくつかが同じレジスタへの書き込みデータをフォーワードしようとした場合、 **B pipe:EX A pipe:EX B pipe:MEM A pipe:MEM B pipe:WB A pipe:WB** という優先順位のもと、優先順位の一番高いステージのフォワーディングのみが有効となり、その他は取り消されます。従って、いくつかの `fwd{E, M, W}{A, B}` が同時に起動されている場合、それらはすべて別々のレジスタに関するフォワーディングとなります。

idpath

interface

input `r1addr` $<3>$	レジスタ・アドレス 1
input `r1in` $<16>$	レジスタ・ファイルから読み出したレジスタ値 1
input `r2addr` $<3>$	レジスタ・アドレス 2
input `r2in` $<16>$	レジスタ・ファイルから読み出したレジスタ値 2
input `fwdEAaddr` $<3>$	A pipe:EX stage のフォワードアドレス
input `fwdEAdata` $<16>$	データ
input `fwdEBaddr` $<3>$	B pipe:EX stage のフォワードアドレス
input `fwdEBdata` $<16>$	データ
input `fwdMAaddr` $<3>$	A pipe:MEM stage のフォワードアドレス
input `fwdMAdata` $<16>$	データ
input `fwdMBaddr` $<3>$	B pipe:MEM stage のフォワードアドレス
input `fwdMBdata` $<16>$	データ
input `fwdWAaddr` $<3>$	A pipe:WB stage のフォワードアドレス
input `fwdWAdata` $<16>$	データ
input `fwdWBaddr` $<3>$	B pipe:WB stage のフォワードアドレス
input `fwdWBdata` $<16>$	データ
output `r1out` $<16>$	レジスタ・オペランド 1 の valid な値
output `r2out` $<16>$	レジスタ・オペランド 2 の valid な値

instruct signals

instrin `r1req`($r1addr, r1in$)	レジスタ・オペランド 1 が要求された
instrin `r2req`($r2addr, r2in$)	レジスタ・オペランド 2 が要求された
instrin `fwdEA`($fwdEAaddr, fwdEAdata$)	A pipe:EX stage からフォワード起動
instrin `fwdEB`($fwdEBaddr, fwdEBdata$)	B pipe:EX stage
instrin `fwdMA`($fwdMAaddr, fwdMAdata$)	A pipe:MEM stage からフォワード起動
instrin `fwdMB`($fwdMBaddr, fwdMBdata$)	B pipe:MEM stage
instrin `fwdWA`($fwdWAaddr, fwdWAdata$)	A pipe:WA stage からフォワード起動
instrin `fwdWB`($fwdWBaddr, fwdWBdata$)	B pipe:WB stage

description

`forward` が生成したフォワーディング情報を参照し、 ID ステージによるレジスタ・オペランドのフェッチ要求に対して valid なレジスタ値を渡します。

B.6 パイプライン・ステージ

icodedec

interface

input icode $<5>$	命令操作コード

instruct signals

instrin do($icode$)	モジュール起動
instrout lb∼nop()	デコード結果

description
5 bit の命令操作コードのデコード回路です．各命令ニーモニック (lb ∼ nop) と同一の名前の制御出力を持ち、do(icode) の起動とともに icode をデコードし、該当する命令の制御出力を起動します．命令操作コードの符号割当の変更の際に必要な修正は、この module で閉じています．

idecoder

interface

input iword $<16>$	命令ワード
output r1addr $<3>$	レジスタ・オペランド 1 のアドレス
output r2addr $<3>$	2
output imm $<16>$	符号拡張された即値オペランド
output tag $<4>$	命令の destination tag

instruct signals

instrin do($iword$)	モジュール起動
instrout rr(tag)	命令が R[ ] ⇐ R[ ], R[ ] 形式
instrout ri(tag, imm)	命令が R[ ] ⇐ R[ ], imm 形式
instrout pi(tag, imm)	命令が R[ ] ⇐ pc, imm 形式
instrout rir(imm)	命令が M[ ] ⇐ R[ ], imm, R[ ] 形式
instrout beqz($r1addr, imm$)	beqz 命令
instrout bnez($r1addr, imm$)	bnez 命令
instrout j(imm)	j 命令
instrout jr(imm)	jr 命令
instrout rst, halt, nop()	それぞれ rst，halt，nop 命令
instrout r1req($r1addr$)	レジスタ・オペランド 1 の参照を示す (分岐命令は除く)
instrout r2req($r2addr$)	レジスタ・オペランド 2 の参照を示す

description
ID stage の動作のためのデコーダです．主な動作は、レジスタ・アドレス抽出、即値オペランド生成、命令形式分類、分岐命令抽出、などです．

Id

interface

input `iword` $<16>$	`ID_IR` ラッチからの入力
input `pcin` $<14>$	`ID_PC` ラッチからの入力
input `r1in` $<16>$	レジスタ・ファイルからの入力 1
input `r2in` $<16>$	レジスタ・ファイルからの入力 2
output `dtag` $<4>$	EX stage へ引渡すデータ: 命令の destination tag
output `opd1` $<16>$	: 命令の オペランド 1
output `opd2` $<16>$	: 命令の オペランド 2
output `opd3` $<16>$	: 命令の オペランド 3
output `r1addr` $<3>$	レジスタ 1 のフェッチアドレス. レジスタファイルへ渡す.
output `r2addr` $<3>$	レジスタ 2 のフェッチアドレス. レジスタファイルへ渡す.

instruct signals

instrin `decode`$(iword, pcin)$	モジュール起動
instrout `r1read`$(r1addr)$	レジスタ 1 のフェッチ要求
instrout `r2read`$(r2addr)$	レジスタ 2 のフェッチ要求
instrout `done`$(dtag, opd1, opd2, opd3)$	EX stage へのデータ引渡し可能
instrout `halt`$()$	実行停止
instrout `beqz`$(r1addr)$	0 分岐要求: 即値オペランドは `opd2<15:0>`.
instrout `bnez`$(r1addr)$	非 0 分岐要求: 即値オペランドは `opd2<15:0>`.
instrout `j`$()$	相対ジャンプ要求: 即値オペランドは `opd2<15:0>`.
instrout `jr`$(r1addr)$	間接ジャンプ要求
instrout `rst`$()$	0 番値ジャンプ要求

description

ID stage の動作を定義した回路です. 命令語をデコードし、 `regfile` へのフェッチ要求、 `pcsv` への分岐要求、 `halt` 信号の起動、 EX stage への引渡しデータの生成などを行ないます.

edecoder

interface

input `icode` $<5>$	命令操作コード
output `func` $<4>$	ALU function code

instruct signals

instrin `do`$(icode)$	モジュール起動

description

EX stage の動作のためのデコーダです. `icode` をデコードし、 ALU の function code を生成します.

Ex

interface

input `icode` $<5>$	命令操作コード

input	alu1 $< 16 >$	ALU 入力 1
input	alu2 $< 16 >$	ALU 入力 2
output	aout $< 16 >$	ALU の出力

instruct signals

instrin	issue($icode, alu1, alu2$)	モジュール 起動

description
EX stage の動作を定義した回路です. `edecoder` によって ALU の function を定め、 ALU 出力を `aout` に出力します.

mdecoder

interface

input	icode $< 5 >$	命令操作コード

instruct signals

instrin	do($icode$)	モジュール起動
instrout	lb, lbu, lw, sb, sw()	load/store group の命令について、各命令を抽出.

description
MEM stage の動作のためのデコーダです. `icode` をデコードし、 load/store group の命令を分類します.

align

interface

input	addr $< 16 >$	アクセス・アドレス (16 bit)
input	data $< 16 >$	ストア・データ
input	mode $< 1 >$	アクセス・データ型. 0:byte, 1:word
output	mask $< 2 >$	ストア・データの byte enable mask
output	aligned_addr $< 15 >$	word 境界に整列されたアクセス・アドレスの上位 15 bit
output	aligned_addr $< 15 >$	整列に応じて byte 移動されたストア・データ

instruct signals

instrin	read($mode, addr$)	ロード用処理起動
instrin	write($mode, addr, data$)	ストア用処理起動
instrout	misalign()	不整列アクセス発生

description
メモリ・アクセスの整列化 (word 境界) を行なう回路です。 load/store アクセスにしたがって `read`, `write` を起動することによって回路は動作しますが、これらの起動は排他的である必要があります。回路の動作は、例えば byte load の場合

$$\texttt{read(0b0, 0b1111000011110001)} \rightarrow \begin{array}{l}\texttt{aligned\_addr == 0b111100001111000} \\ \texttt{mask == 0b10}\end{array}$$

となります。また byte store の場合

$$\texttt{write}(\texttt{0b0}, \texttt{0b1111000011110001}, \texttt{0x1122}) \rightarrow \begin{array}{l} \texttt{aligned\_addr} == \texttt{0b111100001111000} \\ \texttt{mask} == \texttt{0b10} \\ \texttt{aligned\_data} == \texttt{0x2200} \end{array}$$

となり、ストア・データの byte の移動が行なわれます。 `mode = 0b1` (word), `addr =` 奇数アドレス として アクセスを行なうと `misalign` が起動されます。この場合、各出力信号には何も出力されません。

Mem

interface

input `icode` < 5 >	命令操作コード
input `sdata` < 16 >	`SDATA` ラッチからの入力: ストア・データ
input `mar` < 16 >	`MAR` ラッチからの入力: メモリ・アドレス / ライトバック・データ
input `dbus_in` < 16 >	`dbus` からの入力: ロード・データ
output `addr` < 15 >	整列化されたメモリ・アドレス
output `mask` < 2 >	ストア・データの byte enable mask. 1:enable
input `dbus_out` < 16 >	`dbus` への出力: ストア・データ
output `wdata` < 16 >	WB stage への引渡しデータ: ライトバック・データ

instruct signals

instrin `access`(*icode*, *sdata*, *mar*)	モジュール 起動
instrin `rdy`(*dbus_in*)	ロード・データ ready
instrin `ack`()	ストア・データ acknowledged
instrout `done`(*wdata*)	WB stage へのデータ引渡し可能
instrout `read`(*addr*, *mask*)	ロード要求
instrout `write`(*addr*, *mask*, *dbus_out*)	ストア要求
instrout `misalign`()	不整列アクセス発生

description

MEM stage の動作を定義した回路です。 `mdecoder` によって load/store group の抽出とアクセスの整列化を行ない、それに応じてデータバス・アクセスの制御を行ないます。

Wb

interface

input `dtag` < 4 >	`MEM_DTAG` ラッチからの入力：命令の destination tag
input `wdata` < 16 >	`WDATA` ラッチからの入力：ライトバック・データ
output `addr` < 3 >	書き込みレジスタ・アドレス
output `data` < 16 >	ライトバック・データ

instruct signals

instrin `do`(*dtag*, *wdata*)	モジュール 起動
instrout `wback`(*addr*, *data*)	レジスタ書き込み要求

description
WB stage の動作を定義した回路です。 `dtag` をデコードし、書き込み先がレジスタの場合には `wback` を起動します。またこのとき、 `addr`, `data` にレジスタ・アドレスとライトバック・データが出力されます。

B.7 $\mathcal{K}$amuri

kamuri

interface

input `ibus` $< 32 >$	命令バス
input `dbus_in` $< 16 >$	データバス (入力)
output `dbus_out` $< 16 >$	データバス (出力)
output `iaddr` $< 14 >$	命令アドレス
output `daddr` $< 15 >$	データアドレス
output `dmask` $< 2 >$	ストア・データ の byte enable mask
output `read_write` $< 1 >$	`dbus` アクセス・モード. 0:load, 1:store

instruct signals

instrin `chip_enable()`	プロセッサ activation
instrin `reset()`	リセット
instrin `start_stop()`	実行の開始 / 停止トグル
instrin `irdy`$(ibus)$	`ibus` ready
instrin `drdy`$(dbus\_in)$	`dbus` ready
instrin `dack()`	`dbus` acknowledged
instrout `iread`$(iaddr)$	`ibus` のアクセス要求
instrout `daccess`$(read\_write, daddr, dmask)$	`dbus` のアクセス要求

description
マイクロ・プロセッサ $\mathcal{K}$amuri-chip の動作を記述した最上位 module です。

B.8 シミュレーション

imem

interface

input `addr` $< 14 >$	命令アドレス
output `bus` $< 32 >$	命令バス

instruct signals

instrin `read`$(addr)$	命令フェッチ要求
instrin `wait()`	命令バス・ウェイト
instrin `rdy`(bus)	命令バス・レディ

description
$\mathcal{K}$amuri の命令バス・インタフェースに合わせた、シミュレーション用命令メモリ・モジュールです。

dmem

interface

input read_write $<1>$	アクセス・モード. 0:load 1:store
input addr $<15>$	データ・アドレス
input mask $<2>$	ストアデータ の byte enable mask
bidirect dbus $<16>$	データバス

instruct signals

instrin access$(read_write, addr, mask)$	データバス・アクセス要求
instrin wait()	データバス・ウェイト
instrin rdy(bus)	データバス・レディ
instrin ack(bus)	データバス acknowledged

description

$\mathcal{K}$amuri-chip のデータバス・インタフェースに合わせた、シミュレーション用データメモリ・モジュールです。

tester

interface

input read_write $<1>$	アクセス・モード. 0:load 1:store
input addr $<15>$	データ・アドレス
input mask $<2>$	ストアデータ の byte enable mask
input bus $<16>$	データバス

instruct signals

instrin access$(read_write, addr, mask)$	データバス・アクセス要求
instrin wait()	データバス・ウェイト
instrin reset()	リセット
instrin ack(bus)	データバス acknowledged

description

メモリイメージの比較に基づく、シミュレーション用の誤動作検出回路です。シミュレーション用の $\mathcal{K}$amuri プログラムで、チェックしたいレジスタ値などを、異なるアドレスに逐一ストアするようにしておきます。

さらに、この module は `dmem` と同一の大きさのメモリ `cell[65536]` を備えていますので、プログラムをシミュレーションする前に、そのプログラムが正しく動作した場合に得られるデータメモリのイメージを、`cell[65536]` の上に `memset` などによってあらかじめ書き込んでおきます。

こうしておくと、シミュレーション中にこの module は $\mathcal{K}$amuri のデータバスを監視し、`store` 命令が実行されると `cell` 上にある正しい値とストアデータとを比較し、それらが異なる場合に `Fail` というレジスタに `0b1` をセットします (`Fail` の値は、`reset` の起動によって `0b0` にクリアされます)。したがって、プログラムの要するクロック数だけシミュレーションを進めた後、`Fail` の値が 0b1 か 0b0 かによって、動作の正しさを確認することができます。

system

interface

instruct signals

instrin	`chip_enable()`	→ `kamuri` の `chip_enable`
instrin	`reset()`	→ `kamuri` の `reset`
instrin	`iwait()`	命令バス・ウェイトを引き起こす
instrin	`dwait()`	データバス・ウェイトを引き起こす

description

$\mathcal{K}$amuri と同様の制御信号をもち、 `kamuri` と `imem`, `dmem`, `tester` とをつないだ、シミュレーション用の最上位モジュールです。データバス・インターフェースには、 `dmem` と `tester` をつないでありますので、メモリイメージに基づく動作確認が可能です。