ハードウェア記述言語によるアーキテクチャ生成の研究

名古屋大学 工学部
電子情報工学科 第４講座
石原 進

平成６年 5月 19日

Abstract

ハードウェア記述言語 VHDL を用いて、32 ビットオリジナルコンピュータ RS32 の設計を行なった。命令セットは Hennessy&Pattersonの提唱する RISC アーキテクチャ、DLX に準ずるものである。DLX の命令セットは、単純なロード／ストア命令、デコードの容易さを目指している。RS32 では３２ビット固定長命令を用い、低 CPI を目指した。また、性能評価として、ベンチマーク実行時の平均 CPI の測定を行なった。

目 次

1 背景と目的 **3**

2 言語よるハードウェア設計 **3**

2.1 特徴 3

2.2 設計の流れ 4

3 RS32 プロセッサの概要 **5**

4 命令セット **6**

4.1 命令セットの概要 6

4.2 状態遷移 6

4.3 ロード／ストア命令 6

4.4 算術論理演算命令 9

4.5 乗除算命令 9

4.6 セット命令 11

4.7 制御フロー命令 11

4.8 レジスタ転送命令 13

4.9 割り込み命令 13

4.10 命令符号 14

5 マシンの構造 **14**

5.1 基本構造 14

5.2 演算回路の VHDL 記述 17

5.3 ALU 18

5.4 乗算回路 19

5.5 除算回路 19

5.6 整列化ネットワーク 22

5.7 制御回路 22

5.7.1 制御回路の構成 22

5.7.2 ステートマシン 23

5.7.3 データパス制御信号 26

5.8 メモリシステム 27

5.8.1 概要 27

5.8.2 キャッシュメモリ 28

5.8.3 メモリの書き込み指定方式 30

6 性能評価 **31**

7 総括 **33**

7.1 RS32 の課題 33

7.2 言語記述設計による利点と問題点 33

8 謝辞 **33**

表 目 次

1 データ転送命令 . . . 7
2 制御フロー命令 . . . 7
3 算術論理演算命令 . . . 8
4 命令符号の OP コード部 . . . 15
5 ALU OP コード . . . 19
6 キャッシュのミスペナルティ . . . 30
7 メモリの書き込み指定方式 . . . 31
8 命令セットの CPI . . . 31
9 RS32 の平均 CPI . . . 32

図 目 次

1 従来手法（左）と今回用いたトップダウン手法（右） . . . 4
2 ブロック分割の例:VHDL と回路図モデルの混在 . . . 5
3 最上位の状態遷移 . . . 9
4 ロード／ストア命令の実行フェーズ . . . 10
5 算術論理演算命令の実行フェーズ . . . 10
6 乗除算命令の実行フェーズ . . . 11
7 セット命令の実行フェーズ . . . 12
8 制御フロー命令の実行フェーズ . . . 12
9 レジスタ転送命令の実行フェーズ . . . 13
10 割り込み命令の実行フェーズ . . . 14
11 命令符号の形式 . . . 15
12 基本構造 . . . 16
13 VHDL 記述での構成 . . . 17
14 除算回路の構成と演算実行過程 . . . 20
15 ロード時のデータの整列化（ LH 命令の時） . . . 22
16 ストア時のデータの整列化（ SB 命令でメモリのアドレスの下位 2 ビットが 01 のとき） . . . 23
17 制御回路の構成 . . . 24
18 ステートマシンの状態遷移 . . . 25
19 レジスタ読み込みサイクル . . . 27
20 キャッシュの構成 . . . 28
21 RS32 の平均 CPI 測定結果 . . . 32

1　背景と目的

半導体集積技術技術の進歩に伴い、マイクロプロセッサ回路は非常に大規模になってきている。このため開発期間は長期化し、プロセッサ開発における設計の比重は増加する一方である。これに伴い、多品種小量生産が求められる ASIC 開発において設計期間の短縮は大きな課題となっている。

しかし、従来の CISC アーキテクチャによるプロセッサ設計では、回路構成が複雑で開発期間が長くなり、最先端の技術、商品サイクルに追い付いていくのが困難になっている。これに対して最近では、RISC アーキテクチャにより設計されたプロセッサが台頭している。RISC アーキテクチャでは命令セットを簡素化して回路構成を単純なものにし、パイプライン、スーパースカラなどの手法により CPI の低減、クロックの高速化により、プロセッサの処理能力を上げるのであるが、回路構成が単純な分、開発期間は比較的短くて済むのである。

設計期間を短縮するには設計手法を変えることも必要である。従来のボトムアップ、ゲートレベル回路図に基づく設計方式は、煩わしく冗長でしかも遅く、エラーを起こしやすいため、万単位のゲート数を持つ回路を設計する必要のある現在、実用に耐えることは困難になっている。

これと逆の手法がハードウェア記述言語を用いたトップダウン設計である。近年では様々なハードウェア記述言語からの論理合成システムが発表されており、トップダウン設計はハードウェア設計の重要な役割を担いつつある。現在、VHDL などのハードウェア記述言語では、動作レベル、RT レベル、構造レベルという広範囲な記述レベルをサポートしている。ハードウェア記述言語で表現されたハードウェアモデルは、ゲートレベルから、システムレベルまでの各レベルでの動作シュミレーションが可能であり、また論理合成システムにより、言語レベルから論理合成を速やかにおこなえるため、システムレベルの設計から動作検証までの期間が大幅に縮小される。このため目標性能をクリアするための、設計から動作検証までの繰り返しを容易に行なうことが可能となる。

本研究ではハードウェア記述言語を用いたトップダウン設計により、32 ビットの RISC プロセッサ RS32 を設計した。設計に当たっては、低 CPI、クロックの高速化、デザインの変更の容易性を目指し、また、言語記述による設計の利点である、システムレベルでの変更を繰り返して、性能向上を試みた。

本稿では、まずハードウェア記述言語の特性について述べ、言語を用いたプロセッサ開発について述べる。次に、RS32 について命令セット、構造について説明する。最後にＲＳ３２の性能評価を行なった結果について述べる。

2　言語よるハードウェア設計

2.1　特徴

ハードウェア記述言語を用いたトップダウン設計による利点として、デザインの機能変更が容易にできるということが挙げられる。従来のボトムアップ方式やゲートレベル設計方式を使用した設計プロセス中の機能変更は、プロセスをかなり遅らせることになる。ハードウェア記述言語を用いたトップダウン設計方式で設計を行なえば、言語記述の一部を変更することにより、機能変更は速やかに実行されシュミレーションによって動作を確認することが可能になる。

本研究ではハードウェア記述言語として VHDL (VHSIC Hardware Description Language) を使用した。VHDL は、VHSIC (Very High Speed Integrated Circuit) の開発を推進するために、アメリカ合衆国国防総省のプロジェクトのもとで開発が進められ、1987 年、IEEE（アメリカ合衆国電気電子協会）によって、IEEE-1076 規格として正式に承認され、世界の標準としての位置を占めるまでになっている。

VHDL は次のような特徴を持つ。

設計手法およびテクノロジーのサポート　VHDL はトップダウン対ボトムアップなどの対照的な設計手法や、同期設計対非同期設計、PLA 対ランダムロジックなど様々な対照的な設計テクノロジーをサポートするように設計されている。これにより、様々な運用方法や設計ニーズを持った組織にとって設計の支援となり得る。

テクノロジーおよびプロセスからの独立　VHDL は、テクノロジーやプロセスに依存しない。どのような情報でも VHDL を使うことにより、記述可能である。シュミレーション可能なシステム動作の記述は、ゲートより上

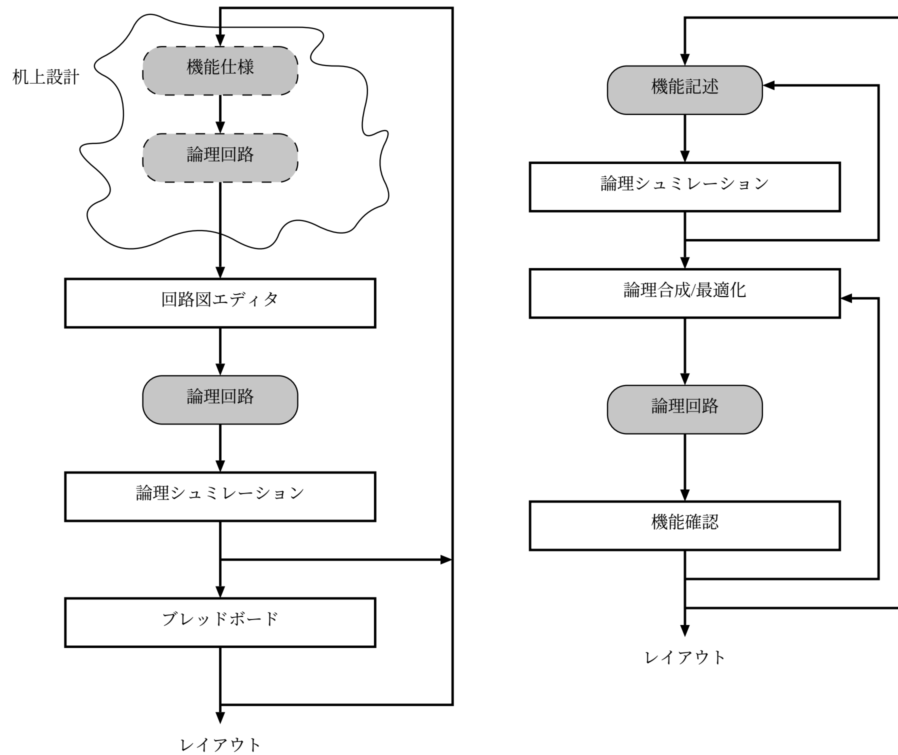

図 1: 従来手法（左）と今回用いたトップダウン手法（右）

位のレベルで開発され、その後種々のテクノロジー（例えば、CMOS、nMOS, GaAs）によって、ゲートレベルへの実現を可能としている。

広範囲な記述能力 VHDL はデジタルシステムレベルから、ゲートレベルまでのハードウェアの動作記述をサポートする。VHDLではそれぞれのレベルに適用できるような一貫した構文規則や意味づけを行ないながら、複数のレベルで記述されたデジタルシステム全体の動作を把握でき、またそれらの記述レベルが混在していてもシュミレーションが可能である。つまり、詳細に展開されたいくつかのサブシステムと上位レベルの機能記述を混在させてシュミレーションすることが可能である。

2.2 設計の流れ

今回の設計では Mentor Graphics社の CADシステム Design Architect と論理合成ツール Autologic VHDLを使用してトップダウン設計を行なった。図 1にハードウェア設計手法の従来手法と今回の設計手法における設計フローを示す。

従来手法による設計ではまず、机上における機能設計、論理回路エディタによる回路図の設計をおこない、論理シュミレーションをおこなう。次に、論理シュミレーションの結果が使用に見あったものならばブレッドボードでのシュミレーションを行なうという方法が行なわれていた。この手法では、論理シュミレーション、ブレッドボードでのシュミレーションの結果が意図と反するものであるとき、論理回路を修正しようとすると、修正作業には非常に長い時間がかかってしまう。また、回路使用の変更の際には回路図の大幅な変更が余儀なくされ、作業効率は悪くなる。

このような従来手法に対し, 今回行なった設計フローは以下のようなものである。

マシンの仕様を決定後、最上位の階層でマシンをいくつかの機能ブロックに分割し、それらの接続モデルを設計する。次に各機能ブロックの内部、またはその機能ブロックのさらに下の階層の機能ブロックの動作を VHDL で記述する。

それぞれの階層において、各機能ブロックのシュミレーションを行なうことが可能である。このとき、もしあ

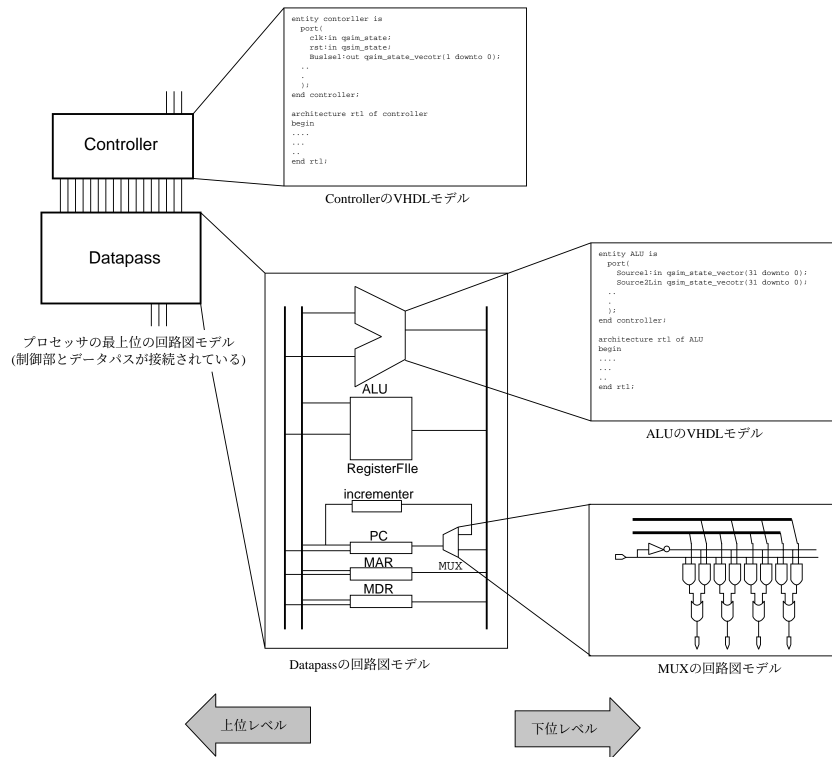

図 2: ブロック分割の例:VHDL と回路図モデルの混在

るブロックの下の階層で設計が完成していなくとも、論理合成可能な VHDL モデルの代わりに、論理合成することとは不可能ではあるが、シュミレーション動作をさせることが可能な抽象度の高いレベルの VHDL モデルを用いることによって上位のレベルのシュミレーションができる。また今回は使用しなかったが、VHDL 記述を用いない回路図モデルをデザイン中に混在させることも可能である。これは手作業で回路図を設計する必要があるブロックを、マシンの中に盛り込むことを可能にしている。この結果、各機能ブロックの設計が終了した時点で、システム全体の機能を確認することができ、回路の修正を迅速に行なうことができる。

シュミレーションでシステムの正常な動作が確認できたら、論理合成ツール (Autologic VHDL) を使用して、言語記述されたモデルをゲートレベルのネットリストに合成する。論理合成においては回路の制約条件を満たすため、キャリールックアヘッドビット数の指定、ステートマシン記述における各種ステート変数用に使用する各コードの設定、ラッチングに使用するフリップフロップの型の指定などを行なう。また論理合成ツールは効率の良い回路を合成するための数々の最適化を行なう。このあと、論理合成された結果に対し動作検証をおこなう。

本研究では、一般的な論理生成を行なうところまでをおこなっており、実際のデバイス上での素子遅延、配線遅延を含めた最適化は行なっていない。

3 RS32 プロセッサの概要

ＲＳ３２は単純なロードストアアーキテクチャを有するプロセッサである。データバスは３２ビット、アドレスバスは３２ビットであり３２ビットのアドレッシングが可能である。汎用レジスタは３２ビット×３２ワードであり、そのうち１ワードをゼロレジスタとして固定している。

浮動小数点数の演算は省略し、整数演算のみが可能である。算術論理演算命令では、加算、減算、乗算、除算、論理積、論理和、排他的論理和、論理左シフト、論理右シフト、算術右シフトが可能であり、命令総数は他の命令を含めて 57 である。アドレッシングモードは１種類しかなく、レジスタ相対のみであり、オフセットは符号つきの１６ビットまたは２６ビットの即値である。命令長は３２ビット固定長である。割り込み処理は現在のところオーバーフロー時のトラップのみをサポートしている。

キャッシュメモリを備えており、それぞれ書き込みアルゴリズムの違う４つのモデルを設計した。キャッシュヒット時の整数加算の CPI は 4 クロック／命令、整数乗算の CPI は 7 クロック／命令である。

4　命令セット

4.1　命令セットの概要

RS32 の命令セットは、Hennesy と Patterson が提唱する RISC アーキテクチャDLX の命令セットのサブセットを用いている。すなわち DLX の命令セットから浮動小数点演算命令を省略したものが RS３２の命令セットである。DLX アーキテクチャは最新のロード／ストアアーキテクチャで設計された実験機や商用機の多くを平均化したアーキテクチャであり、単純なロード／ストア命令セット、パイプライン効率を追求した設計、デコードの容易な命令セット、コンパイラを意識した効率化などの点を重視して設計されている。RS32ではパイプライン処理は行なっていない。

命令セットを表 1から表 3に示す。表中で#と name は即値を表し、## はデータの接続を表す。また、M はメモリ内のデータを示している。

RS32の命令セットの特徴として NOP 命令と汎用レジスタ間のデータ転送命令を持っていないことがあげられる。これらの命令は、ADD 命令でオペランドの一つをゼロレジスタである R0 にすることにより、代替可能であるためである。また、DLX には存在しないが RS32には存在する命令として、コネクト命令がある。これは、３２ビット即値のロードを符号拡張操作の影響を受けないで行なえるように、あるレジスタの上位と別のレジスタの下位、および即値をそれぞれレジスタの上位と下位に格納する命令である。

4.2　状態遷移

RS32の命令実行時の状態遷移は図 3のようになる。

最初の状態で、PC で示されたアドレスのメモリ内データを IR に格納する。このときメモリアクセスが完了していなければ、この状態を繰り返す。次の状態で PC を 4 だけインクリメントし、レジスタファイルから Rs1,Rs2 をそれぞれラッチ A、B に格納する。RS32では命令長は３２ビット固定で、オペランドは命令１語内に含まれるため、デコードと同時にオペランドのフェッチが可能になっている。

第 3 フェーズ以降の状態は各命令別に説明するが、符号つきの整数演算命令および乗除算演算命令実行時にオーバーフロー、またはゼロによる除算が起きた場合についてここで説明する。符号つき演算実行時にオーバーフローが発生するか、ゼロによる除算が行なわれると、演算実行後に演算の結果を格納するフェーズには遷移しないで、割り込み処理フェーズに遷移する。割り込み処理フェーズでは PC の値を割り込みレジスタ (IAR)に格納する。次に、OS の割り込み処理用ルーチンの示されたアドレスを PC に格納し、最初の状態に遷移する。つまり、オーバーフロー発生時には演算結果は格納されず、現在の PC を退避してから割り込み処理ルーチンにジャンプするのである。

4.3　ロード／ストア命令

ロード、ストアともにアドレッシングモードは一種類であり、レジスタ相対のみである。ロード命令では、バイト、半語、語それぞれに対しアクセスが可能であり、バイト、半語に対してはそれぞれ符号なし、符号つきでのロードが可能である。ストア命令では、バイト、半語、語それぞれに対しアクセスが可能である。

実行フェーズは図 4のようになる。

最初の状態で、ラッチ A の値と符号拡張された１６ビットの即値を加算して実効アドレスを生成し、これを MAR にロードする。この後ロード命令は、メモリ内の MAR で示されたアドレスからのデータを MDR に移し、次にこれをラッチ C に移して、最後にレジスタファイルの Rd に格納する。

ストア命令では実行アドレス生成後、ラッチ B のデータを MDR に移す。次に MDR のデータをメモリの MAR で示されたアドレスに格納する。ロード／ストアともにメモリアクセスが必要なフェーズでメモリアクセスが完了していない時には、そのフェーズを繰り返す。

表 1: データ転送命令

名前	意味	例	例の意味
LB	１バイトロード 符合つき	LB R1,40(R3)	$R1 \leftarrow_{32} (M[40+R3]_0)^{24}$ $\#\# M[40+R3]$
LBU	１バイトロード 符合なし	LBU R1, 40(R3)	$R1 \leftarrow_{32} 0^{24} \#\# M[40+R3]$
LH	半語ロード 符合つき	LH R1, 40(R3)	$R1 \leftarrow_{32} (M[40+R3]_0)^{16}$ $\#\# M[40+R3] \#\# M[41+R3]$
LHU	半語ロード 符合なし	LHU R1, 40(R3)	$R1 \leftarrow_{32} 0^{24} \#\# M[40+R3]$ $\#\# M[41+R3]$
LW	１語ロード	LW R1, 40(R3)	$R1 \leftarrow_{32} M[40+R3]$
SB	１バイトストア	SB 40(R3), R2	$M[40+R3] \leftarrow_8 R3_{24-31}$
SH	半語ストア	SH 40(R3), R2	$M[40+R3] \leftarrow_{16} R3_{16-31}$
SW	１語ストア	SW 40(R3), R2	$M[40+R3] \leftarrow_{32} R3$
MOVI2S	汎用レジスタから割り込み用レジスタへの転送	MOVI2SR2	$IAR \leftarrow R2$
MOVS2I	割り込み用レジスタから汎用レジスタへの転送	MOVS2I R2	$R2 \leftarrow IAR$

アドレッシングモードは１６ビットディスプレースメントと汎用レジスタを加算して得られる。

表 2: 制御フロー命令

名前	意味	例	例の意味
BEQZ	ゼロの時分岐（PC 相対）	BEQZ R4, name1	$if(R4=0)PC \leftarrow$ $(PC+4)+name1$
BNEZ	ゼロでない時分岐（PC 相対）	BNEZ R4, name1	$if(R4\neq 0)PC \leftarrow$ $(PC+4)+name1$
J	ジャンプ（PC 相対）	J name2	$PC \leftarrow (PC+4)+name2$
JR	ジャンプ（レジスタ相対）	JR R1, name1	$PC \leftarrow R1+name1$
JAL	ジャンプアンドリンク（PC 相対）	JAL name2	$R31 \leftarrow PC+4$ $PC \leftarrow (PC+4)+name2$
JALR	ジャンプアンドリンク（レジスタ相対）	JALR R1, name1	$R31 \leftarrow PC+4$ $PC \leftarrow R1+name1$
TRAP	トラップ発生	TRAP name2	$IAR \leftarrow PC; PC \leftarrow name2$
RFE	ユーザモードへの復帰	RFE	$PC \leftarrow IAR$

name1は１６ビットオフセット。name2は２６ビットオフセット。

表 3: 算術論理演算命令

名前	意味	例	例の意味
ADD ADDU	加算 符合つき 加算 符合なし	ADD R1,R2, R3 ADDU R1 , R2 , R3	$R1 \leftarrow R2 + R3$
ADDI ADDUI	加算 即値 符合つき 加算 即値 符合なし	ADDI R1 , R2 , #3 ADDUI R1 , R2 , #3	$R1 \leftarrow R2 + 3$
SUB S UBU	減算 符合つき 減算 符合なし	S UBR1 , R2 , R3 S UBUR1 , R2 , R3	$R1 \leftarrow R2 - R3$
S UBI S UBUI	減算 即値 符合つき 減算 即値 符合なし	S UBIR1 , R2 , #3 S UBUIR1 , R2 , #3	$R1 \leftarrow R2 - 3$
MULT MULTU	乗算 符合つき 乗算 符合なし	MULT R1 , R2 , R3 MULTU R1 , R2 , R3	$R1 \leftarrow R2R3$
DI V DI VU	除算 符合つき 除算 符合なし	DI VR1 , R2 , R3 DI VUR1 , R2 , R3	$R1 \leftarrow R2/R3$
AND ANDI	論理積 論理積 即値	AND R1 , R2 , R3 ANDI R1 , R2 , #3	$R1 \leftarrow R2 \wedge R3$ $R3 \leftarrow R2 \wedge 3$
OR ORI	論理和 論理和 即値	OR R1 , R2 , R3 ORI R1 , R2 , #3	$R1 \leftarrow R2 \vee R3$ $R1 \leftarrow R2 \vee 3$
XOR XORI	排他的論理和 排他的論理和 即値	XOR R1 , R2 , R3 XORI R1 , R2 , #3	$R1 \leftarrow R2 \oplus R3$ $R1 \leftarrow R2 \oplus 3$
LHI	レジスタの上位半分に即値をロード	LHIR1 , #42	$R1 \leftarrow 42\#\#0^{16}$
S LL S RL S RA	変数 論理左シフト 変数 論理右シフト 変数 算術右シフト	S LLR1 , R2 , R3 S RLR1 , R2 , R3 S RAR1 , R2 , R3	$R1 \leftarrow R2 \ll R3$ $R1 \leftarrow R2 \gg R3$ $R1 \leftarrow R2_0\#\#R2_{1-31} \gg R3$
S LLI S RLI S RAI	定数 論理左シフト 定数 論理右シフト 定数 算術右シフト	S LLIR1 , R2 , #4 S RLIR1 , R2 , #4 S RAIR1 , R2 , #4	$R1 \leftarrow R2 \ll \#4$ $R1 \leftarrow R2 \gg \#4$ $R1 \leftarrow const_0\#\#R2_{1-31} \gg \#4$
CHI CHL	レジスタの上位半語と１６ビット即値を結合する レジスタの上位半語とレジスタの下位半語を結合する	CHI R1 , R2 , const CHL R1 , R2 , R3	$R1 \leftarrow R2_{0-15}\#\#const$ $R1 \leftarrow R2_{0-15}\#\#R3_{16-31}$
S LT S GT S LE S GE S EQ S NE	コンディションセット	S LT R1 , R2 , R3 S GTR1 , R2 , R3 S LER1 , R2 , R3 S GER1 , R2 , R3 S EQR1 , R2 , R3 S NER1 , R2 , R3	$if(R2 < R3)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 > R3)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \leq R3)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \geq R3)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 = R3)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \neq R3)R1 \leftarrow 1\ else R1 \leftarrow 0$
S LTI S GTI S LEI S GEI S EQI S NEI	コンディションセット即値	S LTI R1 , R2 , #7 S GTIR1 , R2 , #7 S LEIR1 , R2 , #7 S GEIR1 , R2 , #7 S EQIR1 , R2 , #7 S NEIR1 , R2 , #7	$if(R2 < 7)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 > 7)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \leq 7)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \geq 7)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 = 7)R1 \leftarrow 1\ else R1 \leftarrow 0$ $if(R2 \neq 7)R1 \leftarrow 1\ else R1 \leftarrow 0$

符合つき演算で、オーバーフロー時トラップ

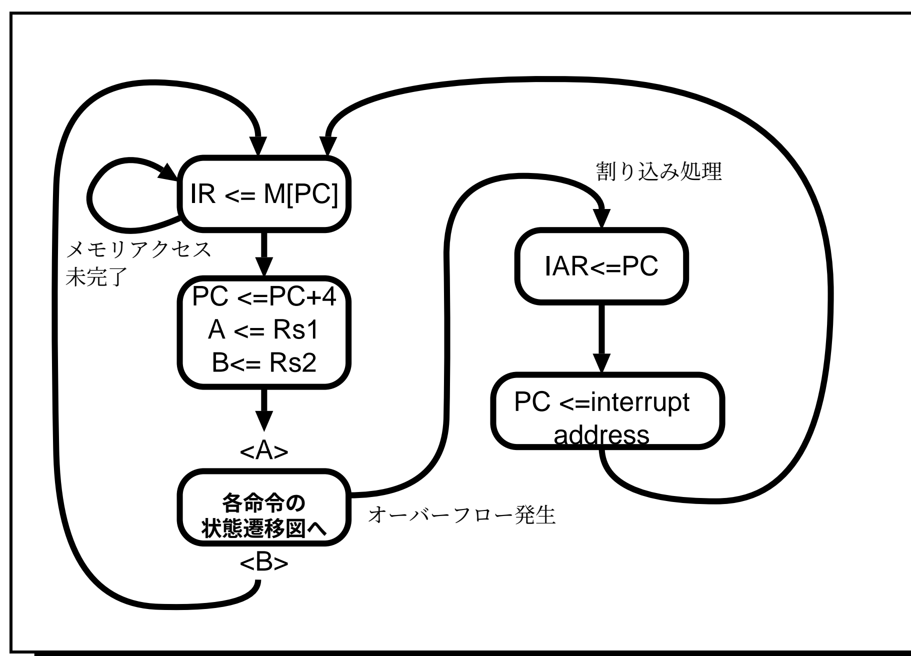

図 3: 最上位の状態遷移

4.4 算術論理演算命令

RS32 では算術論理演算として、加算、減算、論理積、論理和、排他的論理和が可能である。また、シフトは論理左シフト、論理右シフト、算術右シフトが可能である。シフト幅は 31 ビットまで自由に指定可能である。これらの演算はそれぞれ、即値と汎用レジスタ、または汎用レジスタ同士の間で実行可能である。即値は 16 ビット値であり、演算時にはこの 16 ビット値を符号拡張した３２ビット値を用いる。加算命令と減算命令の符号つき演算では、オーバーフローを検出するとトラップを発生する。

算術論理演算を行なうものではないが、算術論理演算命令と同様の実行フェーズをもつ命令として、LHI 命令と CHI 、CHL 命令がある。LHI 命令は即値をレジスタの上位半分にロードする命令である。RS32 では即値ロード命令の働きは、ADDI 命令でオペランドの一つをゼロレジスタである R0 とすることにより行なうが、この方法だと即値はレジスタの下位 16 ビットにしかロードすることができない。このため ADDI ではロードできない上位 16 ビットにロードを行なう命令が LHI である。LHI 命令と ADDI 命令の併用により 32 ビット即値のデータをロードすることができる。

CHL 、CHI 命令はレジスタの上位 16 ビットとレジスタの下位 16 ビット、または即値 16 ビットを結合する命令である。LHI 命令とともに用いることにより、符号拡張による影響を受けることなく 32 ビット即値のロードが可能になる。

実行フェーズは図 5のようになる。最初の状態でラッチ A とラッチ B 、または符号拡張された 16 ビットの即値との演算を行ない、結果をラッチ C に格納する。このとき、符号つき演算でオーバーフローが発生したら、割り込み処理の状態に移る。オーバーフローが発生しなかったら、次の状態でラッチ C のデータをレジスタファイルの Rd に格納する。

4.5 乗除算命令

32 ビットの汎用レジスタ内の整数同士の符号つき、および符号なしの乗除算が可能である。結果は 32 ビット整数として出力される。除算時の余りは出力されない。符号つきの演算時のオーバーフロー発生時、またはゼロによる除算が行われたときにはトラップを発生させる。

図 6に実行フェーズを示す。乗除算命令ではラッチ A とラッチ B との演算をそれぞれ乗算回路、除算回路で行

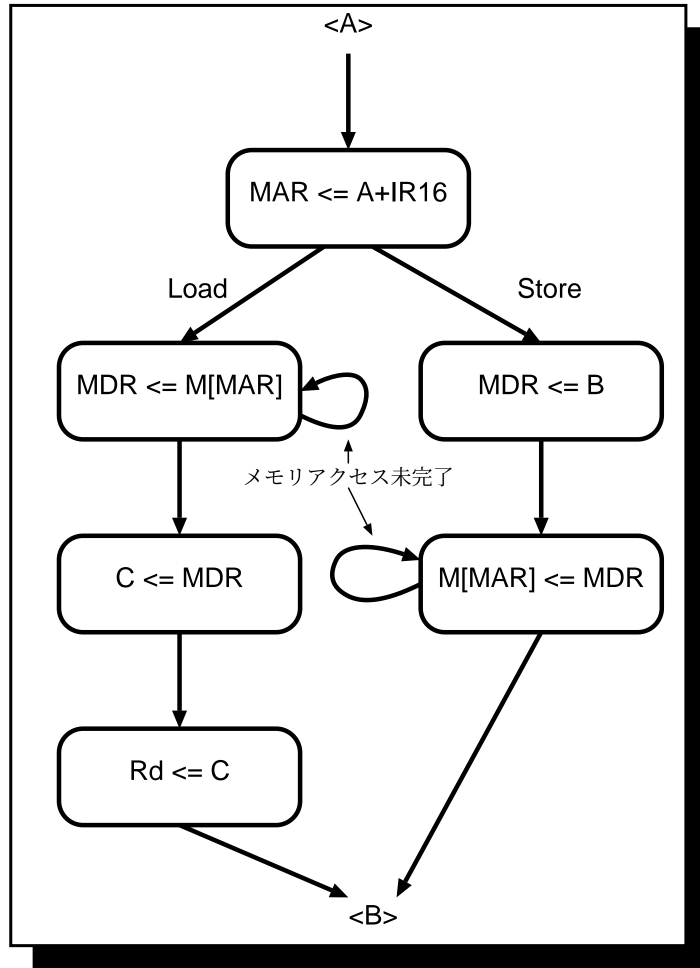

図 4: ロード／ストア命令の実行フェーズ

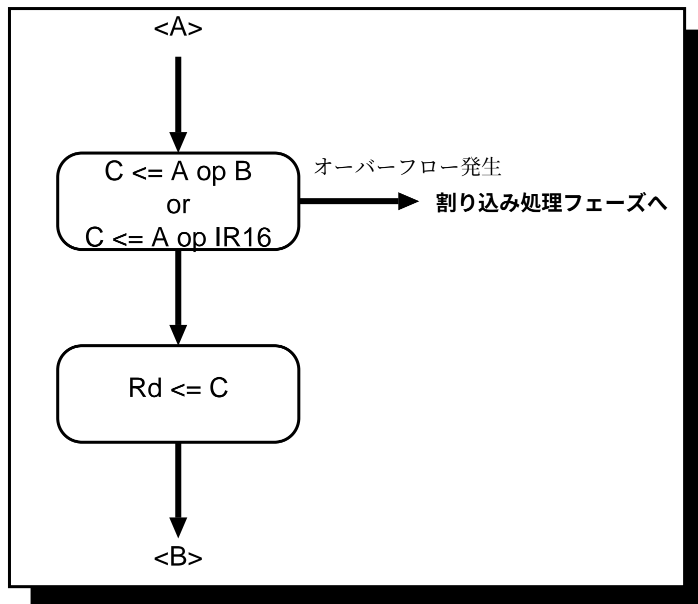

図 5: 算術論理演算命令の実行フェーズ

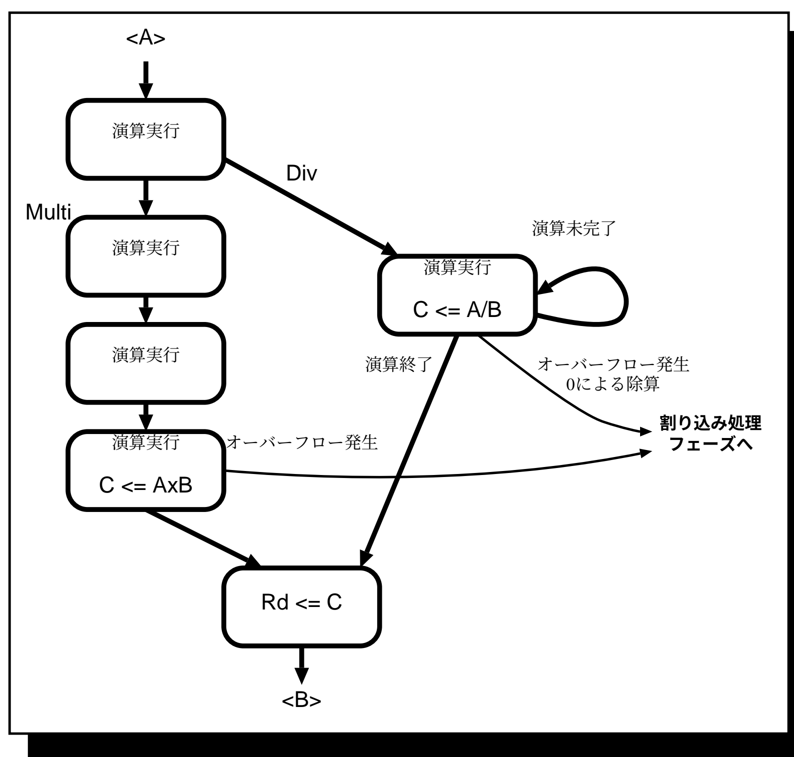

図 6: 乗除算命令の実行フェーズ

なう。乗算回路は配列乗算器であり、組合せ回路のみで構成されているが、現在、RS32 に対して素子遅延、配線遅延の情報を含んだシュミレーションが行なえていないため、処理速度は ALU に対して 4 倍ほどであるとの仮定を行なっている。従ってラッチ A とラッチ B の値を乗算回路に入力してから結果を得るまでには 4 クロックを要するということになる。4 クロックかかって演算が終了すると演算結果をラッチ C に格納する。次の状態でラッチ C の値をレジスタファイルの Rd に格納する。

除算回路はメインクロックに対し 2 倍の周波数の専用クロックにより駆動され、専用クロックの１クロックごとに１ビットずつ演算を行なう。また演算実行中では、メインクロックの立ち下がりでラッチ C に除算回路の出力をセットし続ける。除算回路は演算が終了すると、プロセッサの制御回路に対し演算終了信号を返す。この信号が返されると制御回路の内部状態は次のフェーズに遷移し、ラッチ C の値をレジスタファイルの Rd に格納する。

4.6 セット命令

セット命令は条件分岐のためのコンディションをセットするための命令である。値の比較評価は汎用レジスタ同士、および汎用レジスタと 16 ビット即値に対して可能である。16 ビットの即値は 32 ビットに符号拡張されて比較される。比較の基準は、less than, greater than, less equal, greater equal, equal, not equal の６種類である。

実行フェーズは図 7のようになる。

ラッチ A とラッチ B または、ラッチ A と符号拡張された 16 ビット即値の比較を行なうと ALU はコンディション出力に比較の結果を出力する。次の状態では、この値によりラッチ C に格納する値を判断し、C に値を格納する。ここで評価の結果が真であればが１が格納され、偽であれば０が格納される。最後に C の値をレジスタファイルの Rd に格納する。

4.7 制御フロー命令

制御フロー命令には、無条件分岐命令、条件分岐命令、ジャンプアンドリンク命令がある。実行フェーズを図 8に示す。

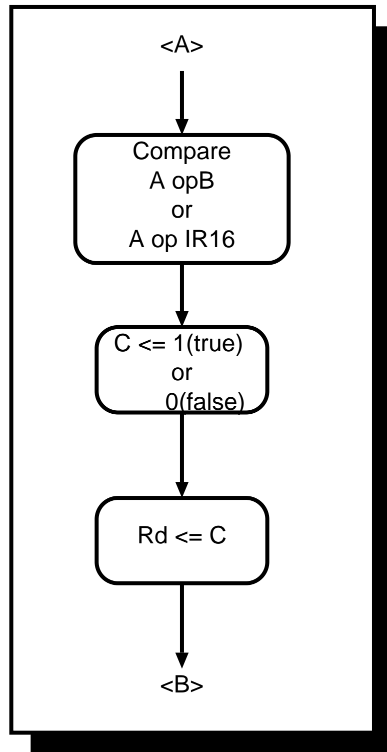

図 7: セット命令の実行フェーズ

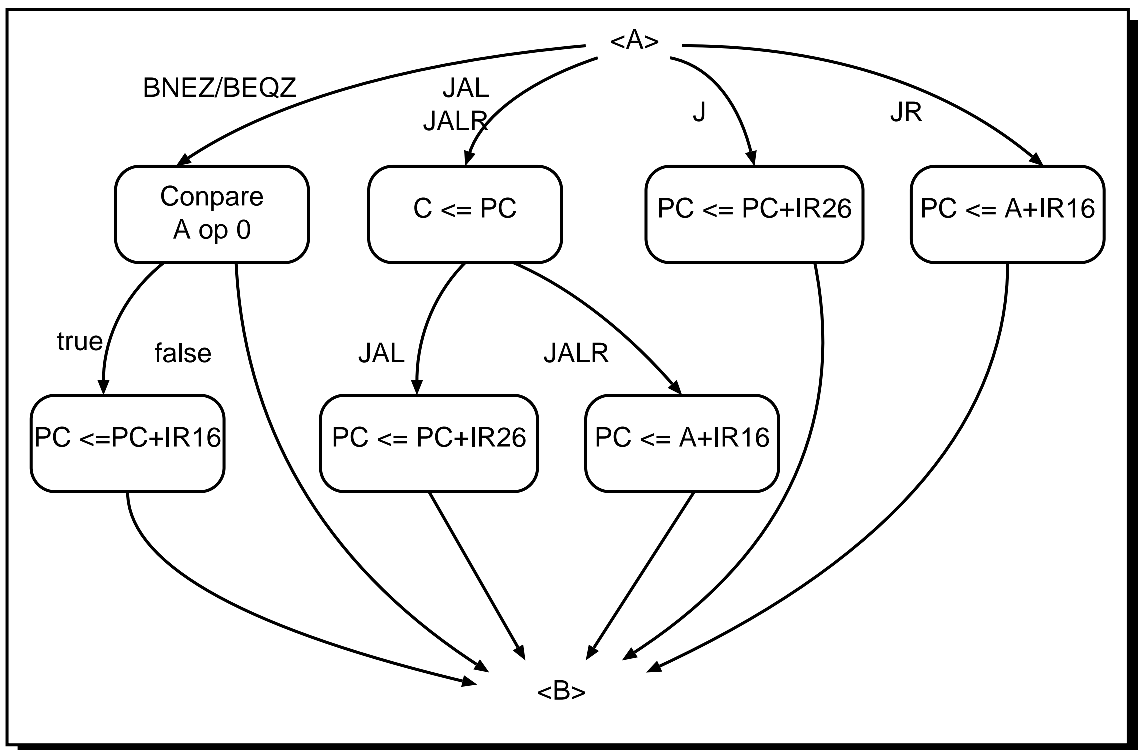

図 8: 制御フロー命令の実行フェーズ

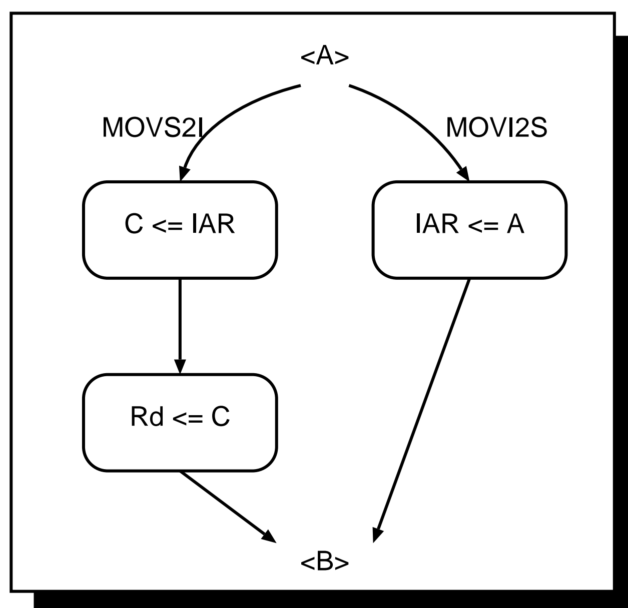

図 9: レジスタ転送命令の実行フェーズ

無条件分岐命令には PC 相対アドレッシングの J 命令と、汎用レジスタ相対アドレッシングの JR 命令がある。それぞれ、PC またはラッチ A の値に符号拡張された即値を加算した結果を PC に格納する。

条件分岐命令にはゼロと等しい時に分岐する BEQZ とゼロと異なる時に分岐する BNEZ がある。それぞれアドレッシングは PC 相対である。まず最初の状態で、ラッチ A とゼロとの比較を行う。次の状態で、比較の結果が真の場合は、16 ビット即値のオフセット（32 ビットに符号拡張したもの）を PC に加える。

ジャンプアンドリンク命令ではアドレッシングが PC 相対の JAL 命令と汎用レジスタ相対である JALR 命令がある。それぞれ最初の状態で、PC の値を汎用レジスタ R31 にロードする。汎用レジスタ R31 はジャンプアンドリンクのときの PC 退避用として固定である。次の状態で、オフセットを PC またはラッチ A の値に加算し、PC に格納する。オフセットは、JAL 命令では 26 ビット即値、JALR 命令では 16 ビット即値であり、加算時にはそれぞれ 32 ビットに符号拡張されている。

4.8 レジスタ転送命令

割り込みレジスタから汎用レジスタにデータを転送する MOVS2I 命令と、汎用レジスタから割り込みレジスタにデータを転送する MOVI2S 命令がある。

実行フェーズを図 9に示す。

MOVS2I 命令では最初の状態で、割り込みレジスタ IAR のデータをラッチ C に格納する。次の状態で、ラッチ C のデータを汎用レジスタ Rd に格納する。

MOVI2S 命令ではラッチ A のデータを IAR に格納する。

4.9 割り込み命令

トラップを意図的に発生させる TRAP 命令と、OS の割込処理用ルーチンからユーザモードに復帰する RFE 命令がある。

実行フェーズを図 10に示す。

TRAP 命令では最初の状態で PC のデータを IAR に退避させる。次の状態で、指定された 26 ビット即値を PC に加算した値を PC に格納する。26 ビット即値は 32 ビットに符号拡張されている。

RFE 命令では、IAR のデータを PC に格納する。

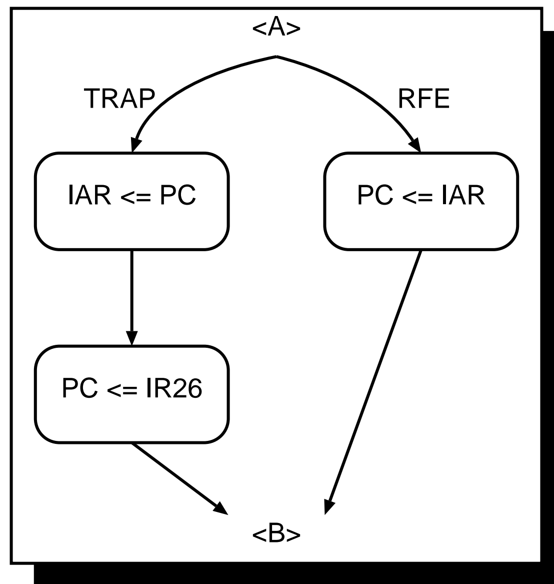

図 10: 割り込み命令の実行フェーズ

4.10 命令符号

RS32 の命令はすべて 32 ビット長である。命令符号のタイプは 3 種類に分けられる。図 11に命令符号の形式を示す。

すべての命令符号形式において、上位 6 ビットが命令コードである。またソースレジスタのアドレスも同じ位置に置かれる。これにより、命令コードのデコードと同時に、レジスタファイルのデータをラッチA、Bにロードすることが可能である。

I 型は即値とレジスタをオペランドとして用いる命令が使用する。即値の算術論理演算命令、即値のセット命令、ロードストア命令、条件分岐命令、JALR 、JR 命令がこれに相当する。即値には 16 ビットの値を使用できる。

R 型はオペランドとして即値を用いない命令が使用する。レジスタ算術論理演算命令、レジスタセット命令、レジスタ転送命令がこれに相当する。命令符号の下位１１ビットは機能コードである。この機能コード部分は、算術論理演算命令、セット命令では ALU の操作コード指定、乗除算命令では乗算器、除算器に対する符合演算指定などに使用される。従って、命令コード部分には算術論理演算命令とセット命令、乗除算命令、MOVI2S,MOVS2I の区別だけが示される。

J 型は即値のみをオペランドに持つ命令が使用する。JAL 、J 、TRAP 、RFE の各命令がこれに相当する。即値には 26 ビットの値を使用できる。

5 マシンの構造

5.1 基本構造

RS32 の基本的な構造を図 12に示す。

バス構成は、S1 、S2 、Dst の 3 バス構成である。レジスタファイルはクロックサイクルごとに読み書きする必要がないので、このシーケンスを複数サイクルに分割し、かつサイクル時間を短くして基本動作を高速化する。すなわち、レジスタファイルの出力ポートに２個のラッチ（A、B）、入力ポートに１個のラッチ（C）を備える。

レジスタファイルは 32 ビット汎用レジスタ 32 個からなる。レジスタ 0 は常にゼロが格納される。またレジスタ 31 はジャンプアンドリンク時のプログラムカウンタ退避用として用いられる。その他のレジスタ類は、プログラムカウンタ、割り込みレジスタ、命令レジスタ、メモリアドレスレジスタ、メモリデータレジスタがあり、すべて 32 ビットである。割り込みレジスタはオーバーフロー発生時の割り込み処理を行なう際に、現在の PC の値を

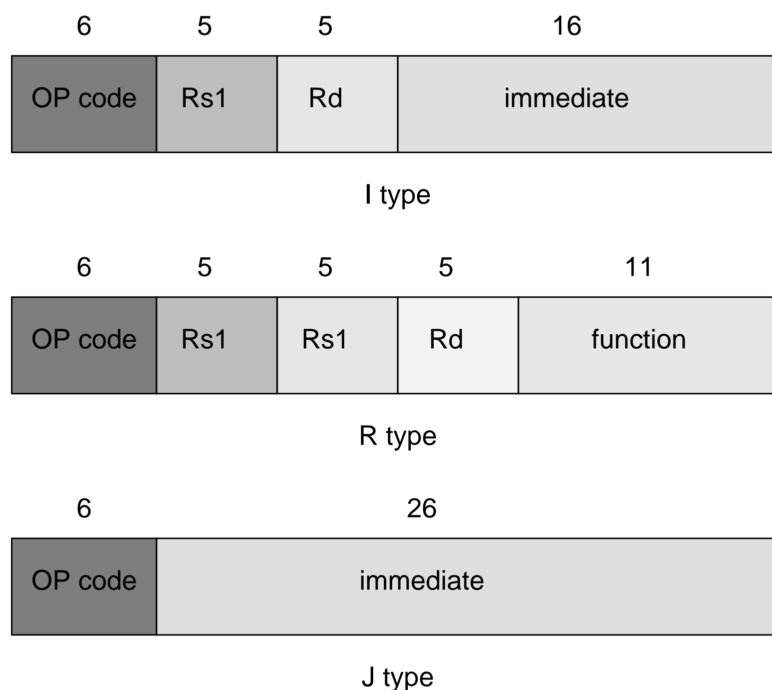

図 11: 命令符号の形式

表 4: 命令符号の OP コード部

I 型命令		R型命令	
命令操作コード	名称	命令操作コード	名称
11[ALU contorl code]	即値 ALU 命令	001111	ALU 演算
101[ALU controlcode]	即値 SET 命令	001110	セット命令
100111	JALR	001011	MULT
100110	JR	001010	DIV
100011	BEQZ	001001	MOVEI2S
100010	BNEZ	001000	MOVES2I
011111	LB	J 型命令	
011110	LBU	命令操作コード	名称
011101	LH	000011	JAL
011100	LHU	000010	J
011011	LW	000001	TRAP
011000	LHI	000000	RFE
010111	SB		
010101	SH		
010011	SW		

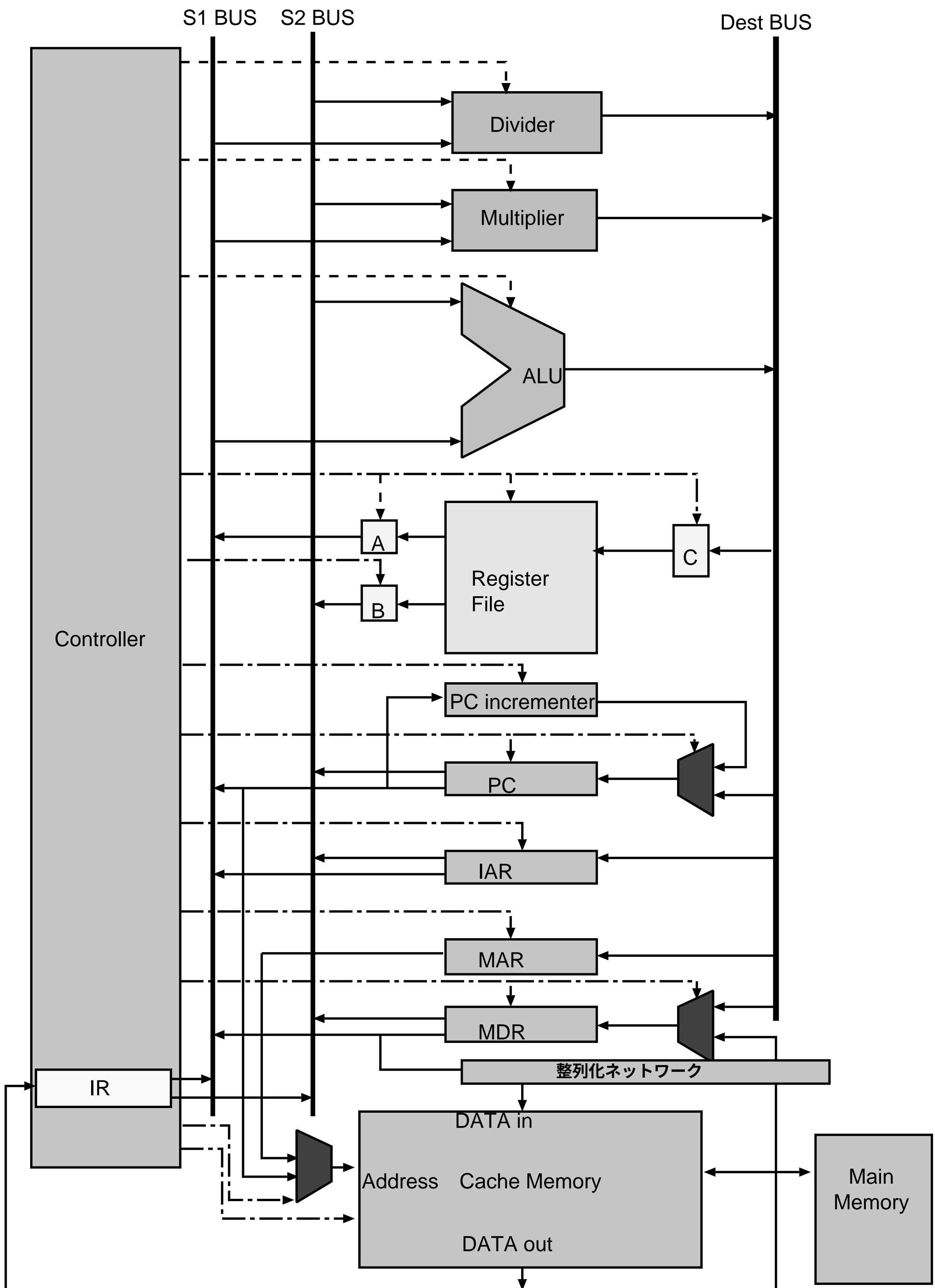

図 12: 基本構造

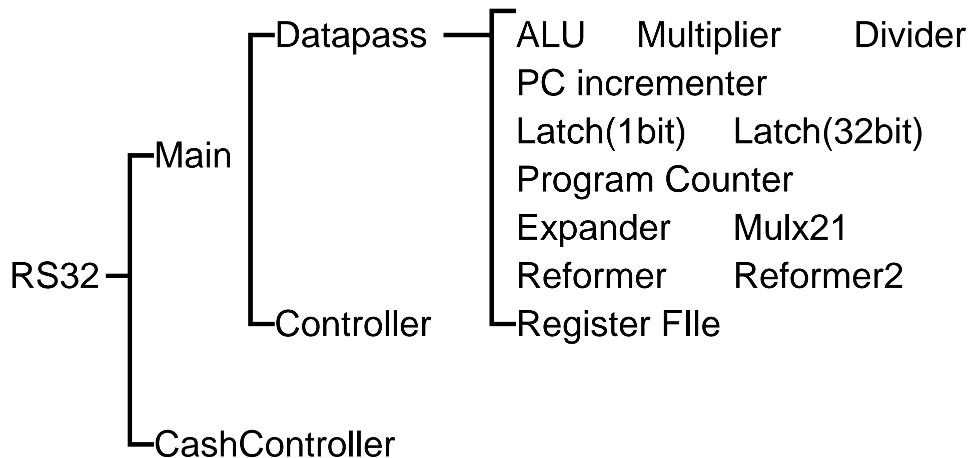

図 13: VHDL 記述での構成

退避するために用いる。割り込みレジスタの値は、PC または汎用レジスタに転送可能である。

プログラムカウンタには専用のインクリメンタを備える。これは PC の値を 4 ずつ加算する。PC への入力は、マルチプレクサで Dst バスと PC インクリメンタを切替える。

メモリへのアドレスの入力は MDR および PC から直接指定できる。メモリはこれらのデータをマルチプレクサで切替えて使用する。MDR への入力はマルチプレクサにより、Dst バスおよびメモリからのデータ出力を切替える。

演算部はＡＬＵと乗算回路、乗算回路を備えている。

アドレスバスは３２ビットであり、３２ビットのバイトアドレッシングが可能であるが、メモリへのアクセスはワード単位（３２ビット単位）で行なう。このため、ロード／ストア命令での半語、バイトのアクセスを可能とするために整列化ネットワークがメモリ部と MDR の間に設けられている。

VHDL 記述では図１３に示すような構成で設計を行なった。Controller には制御回路部分の機能をを記述してある。Datapass では下位コンポーネントを接続した構造記述がなされている。これに含まれる各コンポーネントは ALU、レジスタファイルなどであり、それぞれの機能記述がなされている。また、Reformer および Reformer2 はロード／ストア時の整列化ネットワークである。なお、キャッシュメモリはプロセッサ外部に SRAM で構成するようにしてあり、プロセッサ内部にはキャッシュコントロール部を内蔵するようにした。

5.2 演算回路の VHDL 記述

VHDL では演算子として以下の演算子をサポートしている。

論理 and,or,nand,nor,not,xor

関係 =,/=,<,<=,>,>=

加算 +,-,&

符号 +,-

乗算 *,sll,srl,sra,rll,rrl,/,mod,rem

その他 **,abs

これらの演算子を使用することにより演算部、とくに ALU の機能記述は簡単に行なうことが出来た。ALU の機能記述の一部を以下に示す。case 文により、ALU 制御コード OP に従って演算結果を選択する記述である。ここで、s1ext,s2ext は１ビット拡張されて３３ビットになった ALU 入力データであり、f_cout は演算結果を示す、ALU 内の内部変数である。

```
case OP(6 downto 3)  is
  when "0000" => fct_out := s1_ext + s2_ext ;
  when "0001" => fct_out := s1_ext + s2_ext ;
  when "0010" => fct_out := s1_ext + not_s2_ext ;
  when "0011" => fct_out := s1_ext + not_s2_ext ;
  when "0100" => fct_out := s1_ext and s2_ext;
  when "0101" => fct_out := s1_ext or s2_ext;
  when "0110" => fct_out := s1_ext xor s2_ext;
  when "0111" => fct_out := s1_ext(32 downto 16) & s2_ext(15 downto 0);
  when "1000" => fct_out := (others => '0');
  when "1001" => fct_out := (others => '1');
  when "1010" => fct_out := SLL_ext;
  when "1011" => fct_out := SRL_ext;
  when "1100" => fct_out := s1_ext;
  when "1101" => fct_out := s2_ext;
  when "1110" => fct_out := SRA_ext;
  when "1111" => fct_out := (others => 'X');
  when others => fct_out := (others => 'X');
end case;
```

加算器のキャリールックアヘッドビット数は、回路規模、処理速度に深く関係してくるパラメータであるが、これは論理合成時に指定可能であるので、VHDL 記述中で明示する必要はない。また、論理合成時には同じデータパスに存在する、すべての演算しに対してリソースシェアリングが行なわれ、回路面積を小さくするような論理合成が行なわれる。

以上のように ALU の設計は簡単に行なうことが出来たが、乗算器、除算器に関しては特別の配慮が必要になった。本来 VHDL でサポートされている乗算演算子のうち'/' と' mod' と' rem' は論理合成ツールによりサポートされていないので、除算器は他の演算子を用いてアルゴリズムを記述しなければならなかった。また、乗算器に関しては VHDL でサポートされている' *'演算子を使用して 32 ビット x32 ビットの整数乗算を行なうと出力は 64 ビットとなってしまい、設計用計算機およびソフトウェア上の制約により、VHDL ソースをコンパイルすることが出来ないため、他の演算子を用いて計算アルゴリズムを記述しなければならなかった。

5.3 ALU

ALU の動作表を表 5に示す。

入力は３２ビットデータ入力２、操作コード 7 ビットである。出力は３２ビットデータ出力、コンディション出力、オーバーフロー出力である。オーバーフローは符号つき加算、減算の実行時にオーバーフローがおきた場合のみセットされる。オーバーフローおよびアンダーフローは次のようにして得られる。

加算 ($S1+S2$) **を行なった時** S1 と S2 の符号が同じで加算結果の符号が S1 の符号と異なる時オーバーフロー。

減算 ($S1-S2$) **を行なった時** S1 と S2 の 2 の補数の符号が同じで、S1 と S2 の補数の加算の結果、符号が S1 の符号と異なる時アンダーフロー。

こうして得られるオーバーフローとアンダーフローを符号つき演算の時にのみ出力する。オーバーフロー、アンダーフロー発生で、オーバーフロー出力は 1 が出力され、オーバーフロー、アンダーフローがない時には 0 が出力される。

コンディションのセットは次のように行なわれる。$S1-S2$ という減算を行なった時に、減算結果の符号ビットとアンダーフローの排他的論理和をとり、これが 1 であったとすると $S1<S2$ である。また、演算結果がゼロならば $S1=S2$ である。これらの結果を組み合わせることにより、EQ,NEQ, LT, GT, LE, GE の判定ができる。コ

表 5: ALU OP コード

OP (6 downto 3)	出力	OP (2 downto 0)	セット指定
0 0 0 0	$S1+S2$	0 0 0	
0 0 0 1	$S1+S2\ (unsigned)$	0 0 1	EQ
0 0 1 0	$S1-S2$	0 1 0	LT
0 0 1 1	$S1-S2\ (unsigned)$	0 1 1	LE
0 1 0 0	$S1 and S2$	1 0 0	GT
0 1 0 1	$S1 or S2$	1 0 1	GE
0 1 1 0	$S1 xor S2$	1 1 0	NEQ
0 1 1 1		1 1 1	
1 0 0 0	0		
1 0 0 1	1		
1 0 1 0	$S1 sll S2$		
1 0 1 1	$S1 srl S2$		
1 1 0 0	$S1$		
1 1 0 1	$S2$		
1 1 1 0	$S1 sra S2$		
1 1 1 1			

ンディション出力は、符号なし減算を行なった時に有効な結果を得られる。比較結果が真の時は 1 が出力され、偽の時は 0 が出力される。

シフト演算に対しては 3 1ビットまでの左右論理シフト、算術右シフトが可能である。このとき S2 の値を正の値として要求する。また 3 1ビットまでのシフトにシフト幅を制限したことから、シフト幅の判定には S 2の下位 5 ビットしか参照していない。

5.4 乗算回路

乗算回路には Design Architect のVHDL ライブラリを流用した。回路構成は 3 桁オーバーラップ法を用いた wallace木構造をした配列乗算器となっている。入力は 3 2ビットデータ入力 2 、符号指定信号 1 ビットであり、出力は 3 2ビットデータ出力 1 、オーバーフロー出力である。回路内部では積の計算を 6 4ビットで行っている。これはオーバーフローの出力を積の上位 3 2ビットが全て 0 、または全て 1 ではないときに 1 とするようにしているためである。

5. 5 除算回路

3 2ビット数同士の符号つきおよび符号なしの除算が可能である。符号つき演算でのオーバーフローが起きた時、およびゼロによる除算が行なわれた時にはオーバーフロー信号が出力される。入力は被除数 3 2ビット、除数 3 2ビット、除算開始信号 1 ビット、符号指定信号 1 ビット、クロック入力 1 ビットである。出力は商 3 2ビット、オーバーフローおよびゼロによる除算を示す信号 1 ビット、除算終了信号 1 ビットである。

VHDL では除算演算子が存在するが、論理合成ツールではこれをサポートしていないので、言語記述での設計時には計算アルゴリズムを記述する必要があった。計算手法は 3 2ビットを引き離し法により逐次計算するものである。

回路は演算部と制御部により構成される。演算部の主要部は 1 ビットレジスタ (s i g n)、3 3 ビットレジスタ (P)、 3 2 ビットレジスタ (A)、および 1 つの全加算器よりなる。

図 1 4に回路構成および演算実行過程を示す。図中の丸数字は以下に述べる説明の番号に対応している。

1 符号処理を行う。符号つき演算を正確に行なうため、除算回路に入力された除数と非除数は、符号ビット

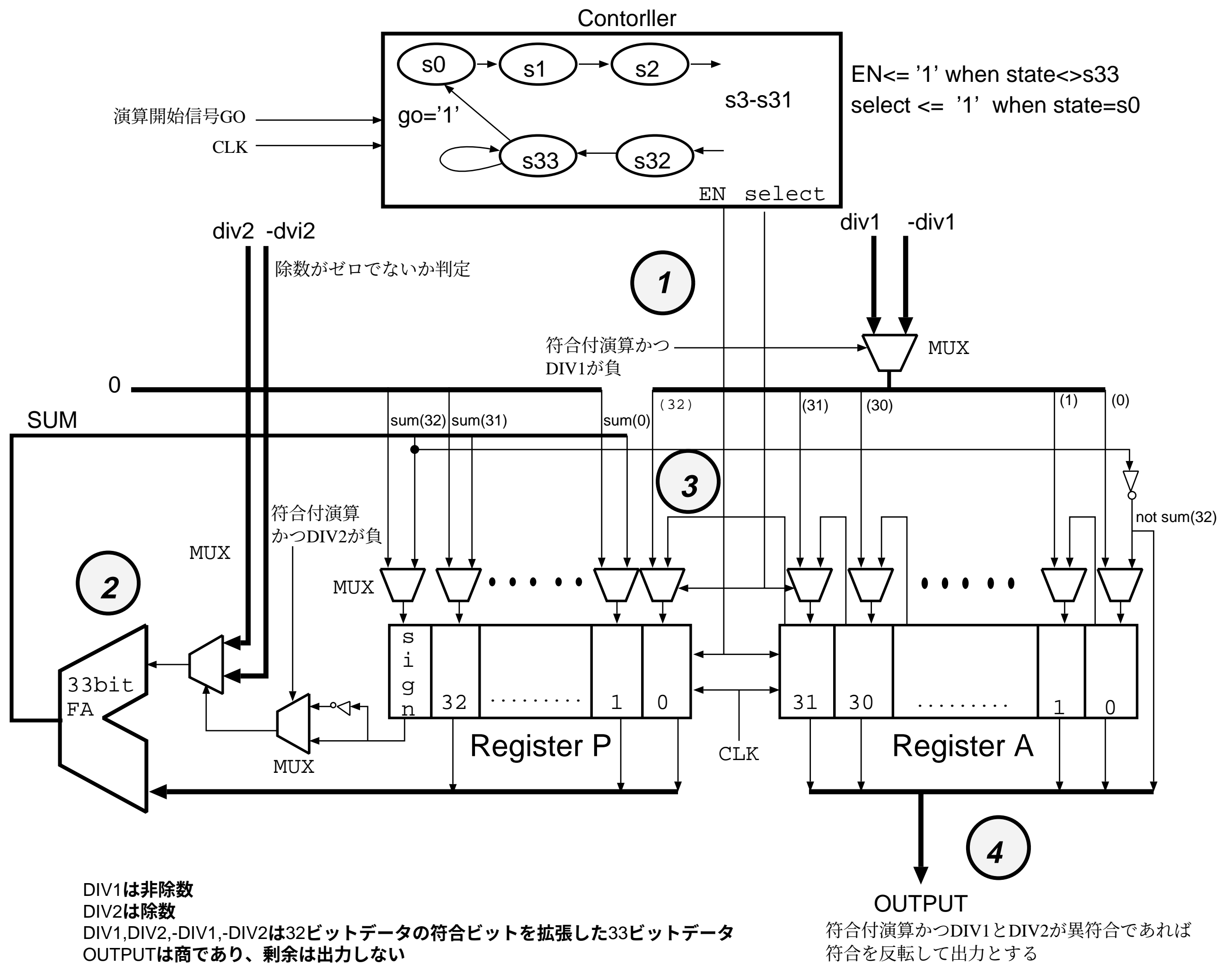

図 14: 除算回路の構成と演算実行過程

を拡張して 33 ビットデータにされる。符号指定信号により符号付除算が指定されており、かつ被除数が負数の時は、符号拡張された被除数は 2 の補数をとってレジスタ A およびレジスタ P の第 0 ビットの入力用のマルチプレクサの入力となる。符号なし演算、また符号つき演算でも符号が正の時は、拡張された被除数そのものがレジスタ A およびレジスタ P の第 0 ビットへの入力用のマルチプレクサへの入力となる。つまり、符号つきの演算では負数は正数に直してから演算を行ない、演算終了後に符号を訂正するのである。

除数がゼロであるかどうかの判定を行なう。

除算開始信号が 1 であり、かつクロックが立ち上がる時、除算制御回路の内部状態は S33 から S0 に遷移する。内部状態 S0 のとき制御回路はレジスタのライトイネーブル信号を 1 として有効にし、レジスタ入力選択信号を 1 として、レジスタ sign およびレジスタ P の第 31 ビットから第 0 ビットまでの入力にはゼロを選択し、レジスタ P の第 0 ビットおよびレジスタ A の入力には、拡張された被除数またはそれを符号反転したものを選択するようにする。クロックの立ち下がりでレジスタに値をセットする。

2 　次のクロックの立ち上がりで制御回路の内部状態は遷移して S1 になる。これ以後演算が終了するまでクロックの立ち上がりで内部状態は S2,S3, . . . S33まで遷移する。

レジスタ P と符号拡張された除数に対して演算を行なう。

- レジスタ P に拡張された除数の 2 の補数を加算する
  - レジスタ signが 1 であり、かつ符号付除算で除数が負数の時。
  - レジスタ signが 0 であり、かつ符号付除算で除数が正数の時。
  - レジスタ signが 0 であり、かつ符号付除算でない時。
- レジスタ P に拡張された除数を加算する
  - レジスタ signが 0 であり、かつ符号付除算で除数が負数の時。
  - レジスタ signが 1 であり、かつ符号付除算で除数が正数の時。
  - レジスタ signが 1 であり、かつ符号付除算でない時。

内部状態が S33 であれば第四段階へ。

3 　レジスタ sign,レジスタ P の書き換えおよびレジスタ A の左 1 ビットシフトを行なう。S2 から S33 までの状態ではレジスタ入力選択信号は 0 となっており、以下のように入力信号が選択されている。またレジスタのライトイネーブル信号は有効のままである。

- レジスタ signへの入力には全加算器の出力の最上位ビット（第 32 ビット）が選択される。
- レジスタ P への入力には、第 32 ビットから第 1 ビットまでは全加算器の出力の第 31 ビットから第 0 ビットまでが選択される。第 0 ビットにはレジスタ A の最上位ビット（第 31 ビット）が選択される。
- レジスタ A への入力には、第 31 ビットから第 1 ビットまではそれぞれレジスタ A の第 30 ビットから第 0 ビットが選択される。第 0 ビットには全加算器の出力の符号ビットの否定が選択される。

クロックの立ち下がりでレジスタに値がセットされる。

第二段階へ戻る。

4 　内部状態 S33 ではレジスタのライトイネーブル信号は 0 となるのでレジスタ書き込みは無効となり、クロックの立ち下がりでもレジスタ A に値はセットされない。従って加算は計 33 回行なわれ、レジスタ A のシフトは計 32 回行なわれる。このときレジスタ A の値を上位 32 ビットとし、全加算器の出力の符号ビットの否定を下位 1 ビットとしたデータは 33 ビットの除算の商を表している。しかし最初に負数を正数に変換して演算したので最後に符号の修正を行わなければならない。すなわち、符号付除算を行なった時、除数と被除数の符号ビットの排他的論理和が 1 のとき商は負となるので、33 ビットの商に対して 2 の補数をとり、その下位 32 ビットを商として出力する。それ以外の時は 33 ビットの商の下位 32 ビットを商として出力する。

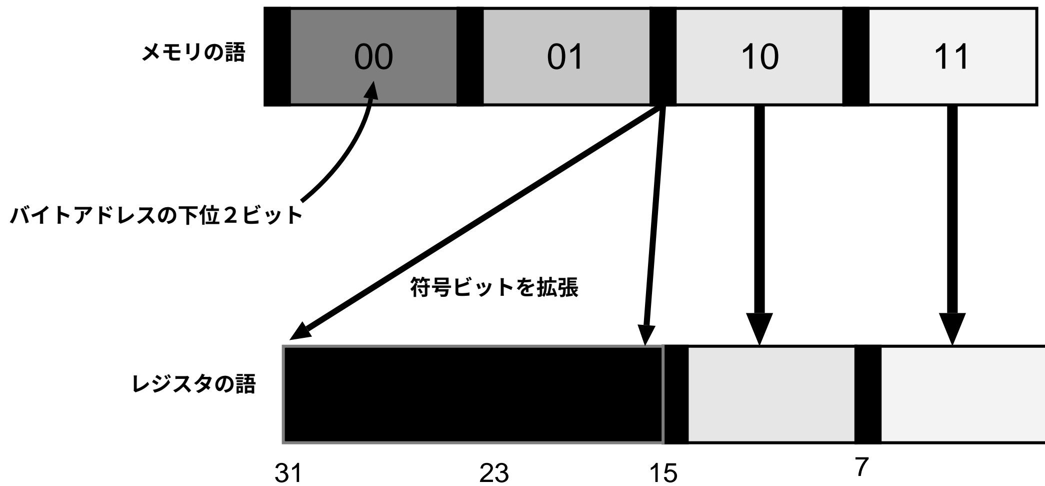

図 15: ロード時のデータの整列化（ LH 命令の時）

オーバーフローは 33 ビットの商の第 32 ビットと第 31 ビットの排他的論理和と符号指定信号の論理和をとって得られる。

内部状態は次に演算開始信号が 1 となるまで S33 に保たれる。

内部状態 S33 になった時演算は終了してライトイネーブル信号は 0 であるから演算終了信号にはレジスタのライトイネーブル信号を利用する。制御回路はこの信号の値が 0 になると次の状態に遷移する。

5.6 整列化ネットワーク

RS32 はメモリへのアクセスを全て語単位で行う。このとき、語の境界を越える長さのデータが存在すると、メモリへのアクセスは 2 回必要となる。しかし、語、半語のデータをロードするのにメモリアクセスを 2 回行なうのは、動作速度を低下させる原因となる。このため RS32 ではこのようなデータの存在は許しておらず、バイトより長いオブジェクトはメモリ内で整列化されていなければならない。この整列化されたメモリ内のデータと 32 ビットのレジスタとのデータ交換を正確に行なうために、整列化ネットワークが必要となる。

整列化ネットワークはメモリユニットと MDR の間に接続されており、次のような動作をする。

ロード命令の実行時は、命令コードより得られるロードするデータのサイズと、アドレスの下位 2 ビットより得られるバイトアドレスに従って、メモリからのデータバス上の 32 ビットデータから、必要とするバイト、半語データを抜き出して出力データの下位に入れる。命令コードより符号つきロードであることがわかると、バイト、半語データの最上ビット（符号ビット）を出力データの上位に拡張して 32 ビットデータとする。符号なしロードの場合は、出力データの上位にゼロを拡張して 32 ビットデータとする。（図 15）

バイト、半語のストアの時には、データをメモリに格納する前に、レジスタの下位のバイト、半語データを、4 つ、または 2 つ並んだ形の 32 ビットデータに変換する（図 16)。また、メモリユニットはストア命令の命令コードとアドレスにより、書き換えの必要なアドレスのみデータの書き換えを行う。これらの動作を正しく行なうためには、メモリアドレスを正確に指定することが必要である。すなわち、語のアクセスの時には、アドレスの下位 2 ビットは 00 でなければならず、半語の時には 00 または 10 、バイトの時は 00 、10 、10 、11 でなければならない。

5．7 制御回路

5.7.1 制御回路の構成

制御回路の構成を図 17に示す。

命令レジスタ (IR) の上位 6 ビットは命令コードであるのでデコーダに送られる。また、命令語に含まれるオペランドの情報、ALU の機能コードはそれぞれ制御信号生成部に送られる。命令語内の即値は 26 ビットまたは

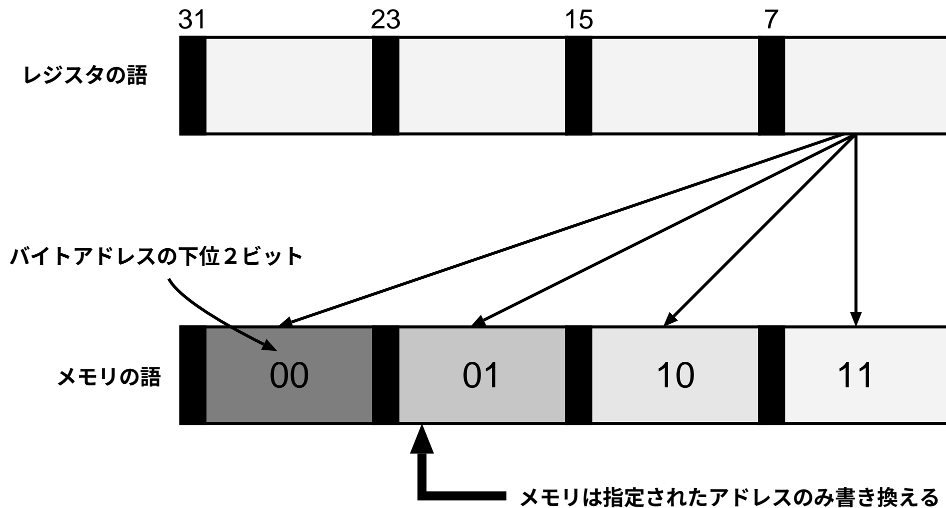

図 16: ストア時のデータの整列化（ SB 命令でメモリのアドレスの下位 2 ビットが 01 のとき）

16 ビットの値であり、それぞれ符号拡張されて、3 ステートバッファを通してソースバスに接続される。

デコーダでは命令コードをデコードして命令タイプを判別する。命令タイプの判別信号はステートマシンおよび制御信号生成部に送られる。ステートマシンは命令タイプの判別信号、キャッシュメモリのアクノリッジ信号、コンディション信号、および ALU 、除算器、乗算器から得られるオーバーフロー信号に従って状態遷移を行ない、状態信号を制御信号生成部に送る。制御信号生成部はマシン状態、ALU 機能コード、命令タイプおよびオペランド情報より各バスの制御信号を出力する。

VHDL 記述においては、デコーダにより生成される命令タイプ信号と、ステートマシン部により生成される状態信号は、列挙型信号として取り扱っている。これにより面倒な状態信号の割当をすることなく設計することが出来た。

5.7.2 ステートマシン

ステートマシンの内部は、ムーア順序回路によって構成される。図 18にステートマシンの状態遷移を示す。

VHDL 記述においては各状態は列挙型の信号「state」として表される。信号「state」は論理合成時に遷移に最適なコード化が行なわれ、状態遷移時に 1 ビットのみが変化するようになる。これにより一般に時間のかかる作業である制御部の設計を簡単に行なうことができた。

状態の遷移はクロックの立ち上がりで行なわれる。デコーダからの命令タイプ信号、データパスからの ALU 、乗算器、除算器より得られるオーバーフロー信号、ALU から得られるコンディション信号、キャッシュメモリからのアクノリッジ信号により次に遷移する状態が決定される。

また、リセット信号がゼロの時はどの状態からでも、クロックと非同期で S0 の状態に遷移する。

このようなステートマシンの VHDL 記述は以下のようになる。クロックの立ち上がり、およびリセットが 0 になったときに、状態遷移が行なわれることを、mainseq_clkd プロセスで記述する。次状態の決定は、mainseq_ste_trans プロセスで case文、if文などを用いて記述される。各状態の状態信号は列挙型信号として制御信号生成部に受け渡される。

```
-- 列挙型の状態信号の定義
type mainstates is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15);
signal main_ste,main_nxt_ste : mainstates :=s0;

-- 遷移条件と状態遷移の記述
```

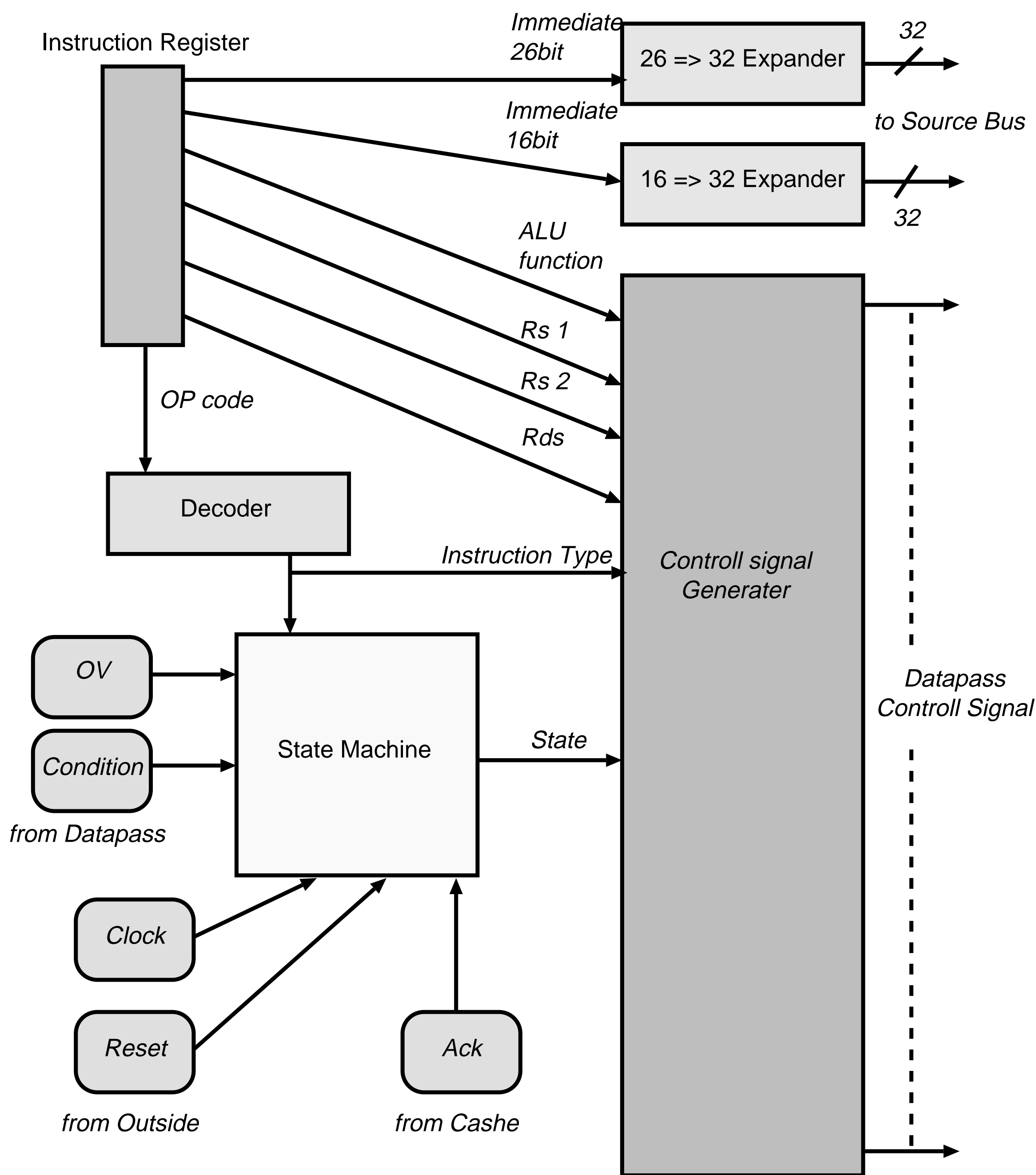

図 17: 制御回路の構成

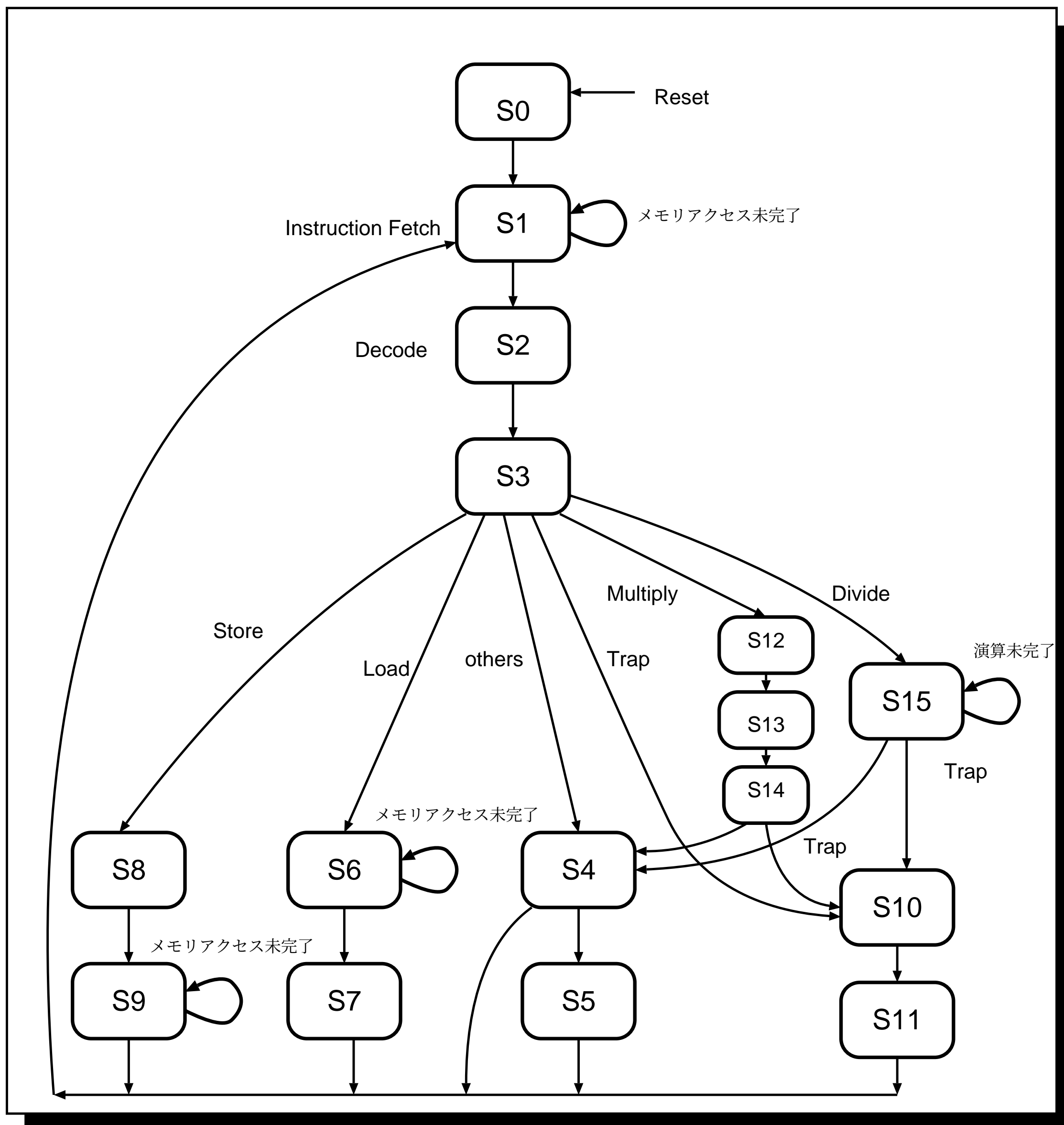

図 18: ステートマシンの状態遷移

```
mainseq_clkd:process(clk,rst)
begin
  if rst='0' then
    main_ste <= s0;
  elsif clk'event and clk='1' and clk'last_value='0' then
    main_ste <= main_nxt_ste;
  end if;
end process mainseq_clkd;

-- 次状態の決定
mainseq_ste_trans:process(main_ste,msop1,msop2,ack,DivOV,DivEN,MultiOV)
begin
  case main_ste is
    when s0 =>  main_nxt_ste <= s1;
    when s1 =>  if ack='1' then
                  main_nxt_ste <= s2;
                else
                  main_nxt_ste <= s1;
                end if;
    when s2 =>  main_nxt_ste <= s3;
    when s3 =>  case msop1 is
                  when op10 => main_nxt_ste <= s1;
                  when op11 => main_nxt_ste <= s4;
.
.
```

5.7.3 データパス制御信号

制御信号生成部でデータバスの制御信号が生成される。RS32 のデータバスを制御する信号には次のようなものがある。これらのうちクロックとリセットはプロセッサ外部より供給されるが、その他の信号は制御回路で命令のタイプ、実行フェーズおよびデータパスより得られるコンディション信号、オーバーフロー信号により生成される。

- 外部入力信号

  クロック すべての動作に対し基本信号となる。マシンの状態遷移はすべて 1 クロックで遷移する。1 クロックは信号の立ち上がりから次の立ち上がりまでである。各レジスタ内データのソースバスへの読み込みはクロックの立ち上がりで行なわれ、デスティデーションバスからレジスタへの書き込みはクロックの立ち下がりを基準として行なわれる。

  リセット マシンの通常の稼働状態では 1 に保たれなければならない。非同期リセットを採用しているので、マシン状態をリセットする時には値を 1 から 0 にすればよい。リセットが 0 のときステートマシンはその状態を S0 にし、データパスのレジスタファイル、プログラムカウンタほかすべてのレジスタ類は値をゼロに設定される。

- 制御回路が供給する信号
  必要となるフェーズで 1 クロックの間アクティブとなる。

  各レジスタのライト イネーブル信号 各レジスタが書き込み可能であることをしめす。レジスタはライト イネーブル信号が 1 の時、クロックの立ち下がりでデータを読み込む。(図 19)

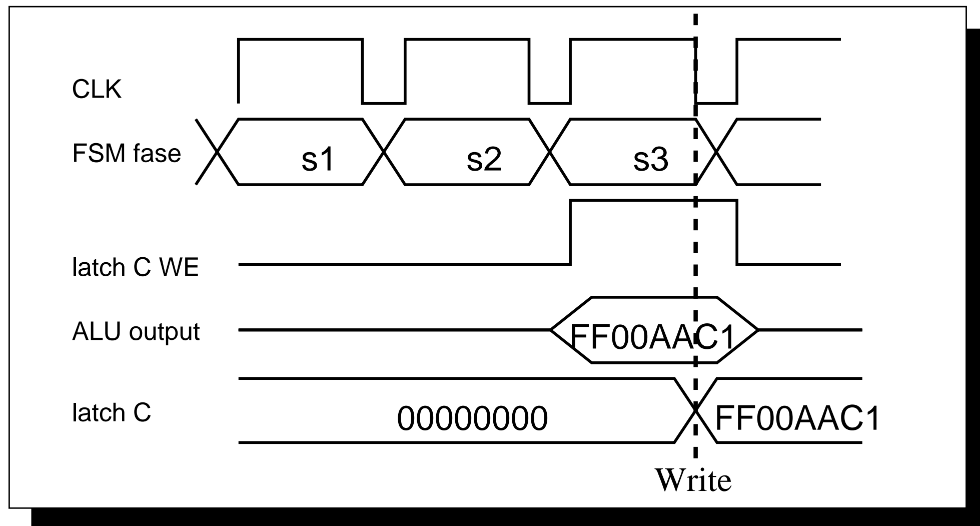

図 19: レジスタ読み込みサイクル

レジスタファイルのアドレス信号 レジスタファイルは 1 入力 2 出力であるため、それぞれのレジスタアドレスを別々に指定する。従って、各 5 ビット × 3 で合計 15 ビットの信号になる。

ソースバス入力選択信号 各レジスタからソースバスへは 3 ステートバッファを通して接続される。この信号は各レジスタに付随する 3 ステートバッファを制御してバッファ出力をハイインピーダンスにするかレジスタの出力にするかを指定する。ソースバス 1 本に対し 1 つのレジスタが選択されアクティブにされる。

Dst バス入力選択信号 ALU 、乗算器、除算器はそれぞれ 3 ステートバッファを通して Dst バスに接続されている。この信号は 3 ステートバッファを制御して Dst バス入力を一つだけアクティブにし、その他のものをハイインピーダンス状態にする。

PC 入力選択信号 PC への入力が Dst バスと PC インクリメンタのどちらであるかを指定する。

PC インクリメンタのカウントアップ信号 PC インクリメンタにインクリメント出力をアクティブにさせる。

MDR 入力選択信号 MDR 入力が Dst バスとメモリ出力のどちらであるかを指定する。

メモリアドレス選択信号 メモリアドレス入力が MAR と PC のどちらであるかを指定する。

ALU 制御信号 ALU 演算時の ALU 操作コードや、実効アドレス生成時の加算、データの転送などの ALU への演算指定を行なう。(5.3参照)

乗算器、除算器制御信号 乗算器、除算器の符号つき演算を指定する信号と除算器の演算開始信号である。乗算器、除算器の符号つき演算信号は命令符号の機能部では同じビットであるからこれらは共用されている。

5.8 メモリシステム

5.8.1 概要

RS32ではメモリアクセスによるオーバーヘッドを軽減するためキャッシュヒット時のデータの書き込み、読み出しが 1 クロックで実行可能なキャッシュメモリを備える。また、RS32のロードストア命令にはそれぞれ、バイト、半語、語にたいしてメモリアクセスするものが用意されているため、メモリシステムにはそれぞれの単位で読み書きが可能な機構が設けられている。

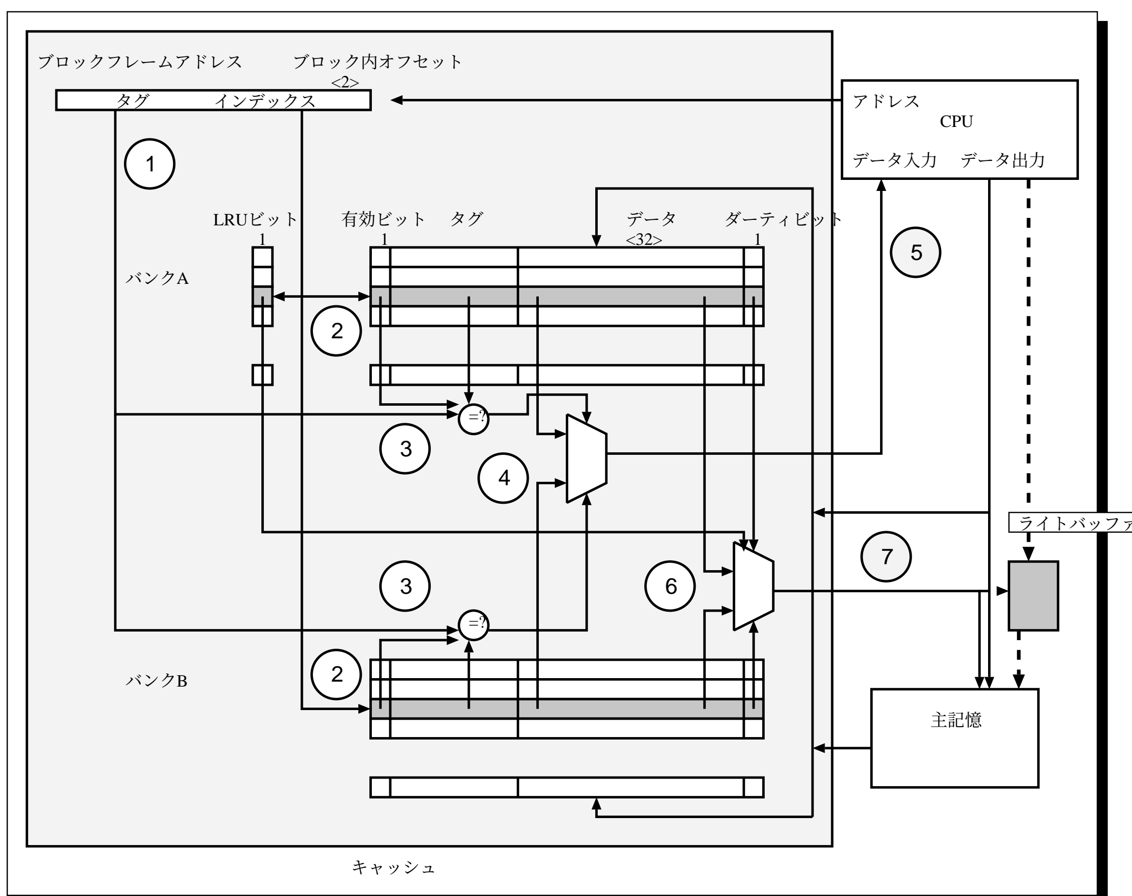

図 20: キャッシュの構成
置き換えアルゴリズムがランダム法の場合には、LRU ビットは不要。
ライトスルーモデルには、ダーティービットが不要。

5.8.2 キャッシュメモリ

図 20にキャッシュの構成を示す。1 ブロックあたり 4 バイト、2 ウェイセットアソシエイティブ方式の構成である。RS32 ではキャッシュにブロック置き換え方式と、書き込み時の動作の違う 4 つのモデルを試作した。それらのモデル間の違いは、ブロック置き換え方式が LRU 法であるか、またはランダム法であるか、書き込時の動作がライトスルー方式であるか、またはライトバック方式であるか、ライトバッファを備えているか否か、の三点である。また、それぞれのモデルで書き込み時のキャッシュミスの際の動作アルゴリズムにはノーライトアローケート方式を採用しており、書き込時のキャッシュミスが起きた時はキャッシュへの書き込みを行なわない。

図にしたがって、キャッシュの動作について説明する。ブロック書き換え方式が LRU 法でライトバック方式のものは、回路構成においてその他のモデルを含んだモデルであるので、これについての説明を行なう。丸数字は説明の番号に一致している。

まず、メモリ読みだし時にキャッシュがヒットしている場合を説明する。

1 キャッシュに送られてきたアドレスは図のように 2 つのフィールドに分けられる。すなわち、それらは 2 ビットのブロック内アドレスと 30 ビットのブロックフレームアドレスである。ブロックフレームアドレスはさらに 2 つに分けられ、タグ部とインデックス部に分けられる。

2 インデックスによりキャッシュのブロックセットを選択する。この動作はブロック A 、ブロック B ともにおこなう。

3 ブロック内のデータは 4 つのフィールドに分けられている。すなわち、1 ビットの有効ビット、1 ビットのダーティビット、アドレスタグ、32 ビットのデータである。

各ブロックで選択されたブロックセットから有効ビットとアドレスタグを読み出す。このとき有効ビットが 1 であり、かつブロックから読み出したアドレスタグがブロックフレームアドレスのタグ部と一致していたら、そのブロックはヒットしていることになる。

4 ヒットしている方のブロックのデータをマルチプレクサで選択する。このとき、第三段階でブロック内の有効ビット、アドレスタグを読み出している時点で、ブロック内のデータはすでにマルチプレクサの入力まで読み出されている。

ブロック置き換えアルゴリズムに LRU(Least-Recently Used)法を用いたモデルでは、同じキャッシュインデックスで、ブロック A 、B のどちらが新しく参照されたかを知るために、LRU ビットという 1 ビットのフィールドが用意されている。あるキャッシュインデックスでブロック A が参照された時は 1 をセットし、ブロック B が参照された時は 0 をセットする。こうしておくと、LRU ビットにより同一キャッシュインデックスでどちらのブロックのデータが長時間使用されなかったかがわかる。

置き換えアルゴリズムにランダム法を用いたモデルでは、LRU ビットは備えられていない。

5 マルチプレクサの出力を CPU に送る。

以上がキャッシュヒット時の動作である。これらの動作は 1 クロックサイクルで行なわれる。

読み込時にキャッスミスが起こった場合は次のようになる。第 3 段階でキャッシュミスが起きたかどうかが判別される。

6 LRU ビットを参照して置き換えるべきブロックを決定する。つまり、LRU ビットが 1 の時はブロック B のデータが置き換えられ、0 のときはブロック A のデータが置き換えられる。

置き換えアルゴリズムにランダム法を用いているモデルでは、置き換えブロック決定に、1 クロックごとに値が 1 と 0 の間で切り替わるようなラッチの値を用いる。置き換え動作が必要な時はこのラッチの値を参照して、A 、B どちらかのブロックを置き換える。

また置き換えるべきブロックのダーティビットを参照して、データがダーティである（キャッシュと主記憶の内容が異なっている）かどうかを調べる。

ライトスルー方式のモデルではキャッシュ内のデータは必ず主記憶のデータと一致するので、ダーティービットは必要なく。データの書き出しも行なわれない。

7 データがダーティであれば、メモリへの書き出しを行なう。この時、主記憶への書き出しを行なうブロックのアドレスタグを主記憶へのブロックフレームアドレスのタグ部とし、CPU からのアドレスのブロックフレームアドレスのインデックス部を主記憶へのブロックフレームアドレスのインデックス部とする。

データがクリーンであるとき、またデータがダーティでもすでに主記憶にキャッシュの内容を書き出した時には、主記憶からデータを読み出してキャッシュにデータを書き込む。この時、ダーティビットを 0 とし、有効ビットを 1 にし、LRU ビットを反転する。LRU ビットを反転することにより、現在データが読み込まれたブロックが最も最近使用されたブロックであるということが示されることになる。

これらの動作をすべて終了すると、通常のヒット時の動作に移る。また、これらの動作の間はプロセッサ制御部に送るアクノリッジ信号を 0 にしてプロセッサの動作をストールさせる。

書き込みの時キャッシュがヒットしている時は上記の第 4 段階までは同じである。違うのは第 5 段階でキャッシュにデータを書き込むところである。第 4 段階でヒットしたブロックが確定すると、CPU からのアドレスのブロック内アドレスと CPU から送られてくる書き込み単位制御信号にしたがって書き込むべきブロック内の更新部分ににデータを書き込む。ライトスルー方式の場合はキャッシュがヒットしている場合でも主記憶にデータを書き込む。

書き込時のキャッシュミスの際はノーライトアロケート方式を採用しているのでキャッシュの内容は変更せず、主記憶に直接データを書き込む。

表 6: キャッシュのミスペナルティ

ライトバック方式、LRU 法	ミスペナルティ（クロックサイクル）
読みだしでミス キャッシュ内データはクリーン	6
読みだしでミス キャッシュ内データはダーティ	12
書き込みでミス	6
ライトバック方式、ランダム法	ミスペナルティ
読みだしでミス キャッシュ内データはクリーン	6
読みだしでミス キャッシュ内データはダーティ	12
書き込みでミス	6
ライトスルー方式、ランダム法	ミスペナルティ
読みだしでミス	6
書き込時	必ず 6 クロックサイクルのストール
ライトスルー方式、ランダム法、ライトバッファつき	ミスペナルティ
読みだしでミス	6
書き込時	0
	しかしバッファにデータが存在する時のメモリアクセスでストールが起きる可能性がある

キャッシュが主記憶にデータを書き出す時には、プロセッサ制御部へのアクノリッジ信号をゼロとしてプロセッサ動作をストールさせる。このとき、ライトバッファを備えるモデルでは主記憶にデータを書き出す際にはライトバッファにデータを格納するのでプロセッサをストールさせる必要はない。しかし、ライトバッファにデータが存在しており、ライトバッファが主記憶に書き込みを行なっている際には、主記憶へのアクセス、およびライトバッファへの書き込みを行なうことは出来ないのでプロセッサ動作はストールされる。

表 6にミスペナルティの一覧を示す。RS32 は現在では、デバイス上の配置、配線を行なったシュミレーションを行なっていない。したがってクロック周波数の測定は行なうことができないため、これらの値は主記憶への書き込みと、読み込みにそれぞれ 6 クロックサイクルを要すると仮定した結果、算出される値である。

5.8.3 メモリの書き込み指定方式

4.3で述べたように、RS32 はメモリバンド幅が 32 ビットでありデータのやり取りはすべて 32 ビット単位で行う。しかし RS32 ではストア命令には 32 ビットの語だけでなく、半語（ 16 ビット）、バイトのストアも用意している。このためメモリシステムには、1 語内でバイト、半語単位で任意のバイトアドレスにデータを書き込む機構を設けている。

図 7にメモリの書き込み指定方式を示す。メモリシステムには 32 ビットアドレスとアウトイネーブル信号、ライトイネーブル信号、および 32 ビットデータのほか、書き込み単位（語、半語、バイト）を指定のための制御信号が入力される。書き込み単位指定信号とバイトアドレスにより、語の中のどのバイトに書き込みを行なうかが決定される。なお、メモリの語の内部のバイトアドレスはビッグエンディアン方式で設定されている。

表 7: メモリの書き込み指定方式

書き込み単位制御	アドレスの下位 2 ビット	書き換えを行うバイトアドレス
11	00	00
11	01	01
11	10	10
11	11	11
10	10	10 and 11
10	00	00 and 01
01	00	00 and 01 and 11 and 11

表 8: 命令セットの CPI

命令	最小クロックサイクル
ロード	6 3
ストア	5
算術論理演算	4
乗算命令	7
除算命令	20
セット	5
ジャンプ	3
ジャンプアンドリンク	5
分岐（成立）	4
分岐（不成立）	3
MOVS2I	4
MOVI2S	3
TRAP	4
RFE	3

6 性能評価

現在、RS32 では素子遅延、配線遅延を含んだ実時間のシュミレーションを行なうことが出来ていないので、性能評価は、ベンチマーク実行時の平均 CPI を測定するという方法で行なった。

RS32は各命令で命令フェッチに必ず一度のメモリアクセスを必要とし、またロード／ストア命令ではさらに一回メモリアクセスを行なう。このときキャッシュミスが起こった時はプロセッサの動作はストールされるが、このストールを無視したときの各命令の CPI は表 8のようになる。

RS32の VHDL モデルのキャッシュのアルゴリズムが異なる 4 つのモデルに対して、キャッシュサイズを変化させてベンチマークを行なった時の平均 CPI を表 9, 図 21に示す。ベンチマークは Livermore Loop の Kernel1 である。今回の測定ではプログラムのメモリ消費がシュミレーション可能な最大限の大きさになるように、ループの回数を決定したが、キャッシュサイズを 1024 バイト以上にすると、プログラムで使用するすべてのデータがキャッシュに収まってしまい、キャッシュの効果を確認することが出来なくなるので、キャッシュサイズは 1024 バイトまでとした。

測定の結果、RS32 が平均 CPI 値 5.6 から 6.9までの間で動作可能であることが確認できた。キャッシュのサイズの増大により CPI が低減していることがわかる。

LRU 法とランダム法との比較では LRU 法の方が低い平均 CPI 値が得られたが、その差はキャッシュサイズの増大とともに少なくなっていることがわかった。キャッシュサイズがある程度大きくできれば、LRU 法よりも回

表 9: RS32 の平均 CPI

キャッシュサイズ [byte]	1024	512	256	128
A ライトバック方式、LRU 法	6.20	6.47	6.51	6.58
B ライトバック方式、ランダム法	6.22	6.55	6.70	6.87
C ライトスルー方式, ランダム法	6.39	6.59	6.72	6.89
D ライトスルー方式、ランダム法、ライトバッファつき	5.67	5.87	6.01	6.17
キャッシュがすべてヒットする時の平均 CPI	4.73			

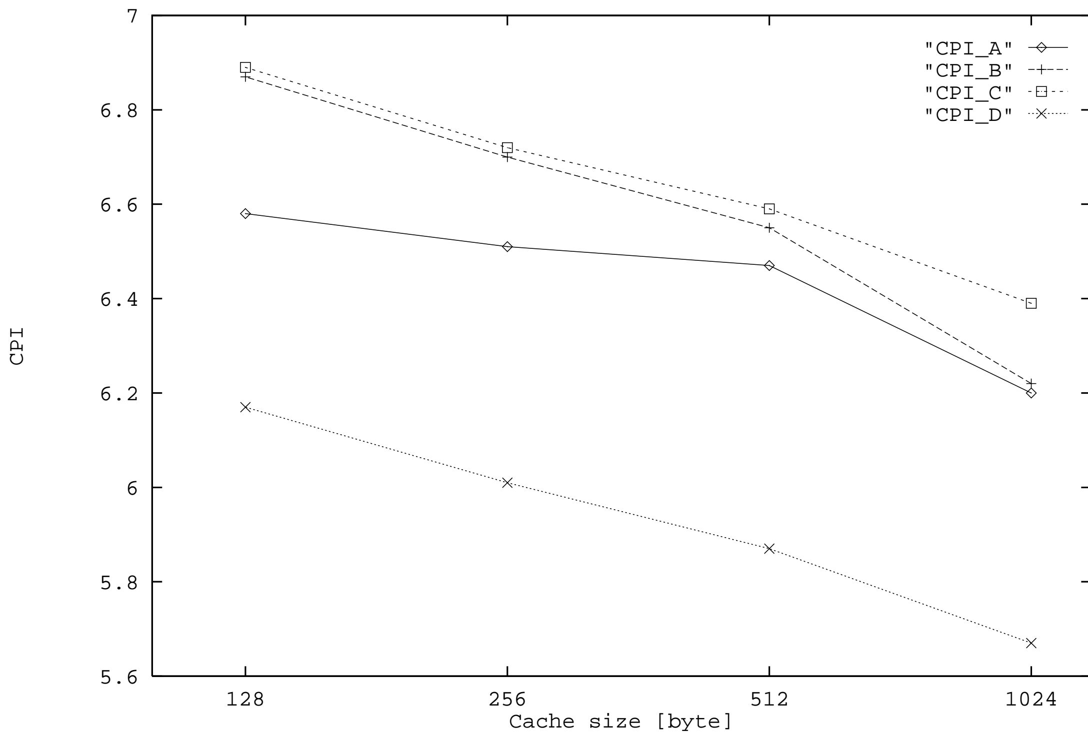

図 21: RS32 の平均 CPI 測定結果

路規模が小さく、実現も容易であるランダム法を用いる方が良いと言える。

ライトバック方式とライトスルー方式との比較では、ライトバック方式の方が低い CPI 値が得られた。その差はキャッシュサイズの増大に伴って大きくなっている。

また、ライトバッファをつけたモデルと、ライトバッファをつけないモデルとの比較では、ライトバッファをつけることにより CPI の大幅な向上が得られることが確認できた。

現在 RS32ではデバイス上での配線遅延、素子遅延を含んだシュミレーションが行なえていないので、キャッシュサイズやキャッシュの方式は一概にどの方式が良いとは決定できないが、クロックサイクル、回路規模などの要素との関係より RS32の性能改善を行なっていく上での一つの指針を得ることが出来た。

7　総括

7.1　RS32 の課題

現在 RS32 は一般的な論理回路を合成するところまでの設計がなされているが、今後は実際にデバイス上に構成するために素子遅延、配線遅延などを考慮した最適化を行なう必要がある。これにともなって、制御部の構成、状態遷移、乗除算器の構成、冗長な記述などは修正する必要があるであろう。

また、RS32はマルチサイクル計算機として設計されたが、RISCプロセッサとして演算速度を向上させるためには、これをパイプライン化する必要があるだろう。この際には VHDL モデルとして設計された回路構成要素のコンポーネントが流用できるため、比較的早く実現が可能だと思われる。さらに、言語設計がパイプラインプロセッサの設計にどれほどの効力を発揮できるかを確認するべきである。

7.2　言語記述設計による利点と問題点

言語記述による設計による利点として、設計変更の容易さ、回路規模変更の自由が挙げられる。今回の設計作業では、配列を使用することにより、レジスタファイル、キャッシュなどでの容量サイズなどのパラメータの変更は、定数の値を数箇所変更するだけで良く、効率的に作業を行なうことができた。

当初 RS32は乗算器、除算器を含まない構成で設計されたが、後にこの構成でプロセッサが完成してから、これらを含む構成に設計を変更した。この変更に伴い制御部の設計変更を行なったが、変更作業は簡単に行なうことができた。手作業によるステートマシンの設計においては各状態符号の割り当てに多くの時間を割く必要があるが、言語記述では各状態は列挙型の信号として指定すればよく、また論理合成により状態の符号割り当ては自動的に指定されるため、設計変更は機能記述を数行訂正すればよかった。この結果、個人作業で 3 カ月弱で設計作業を完了することが出来た。言語設計はとくに制御回路の設計に大きな効力を発揮すると言えよう。

大規模な回路になるほどこれらの利点は効力を増し、生産性の大幅な向上につながるであろう。

問題点は論理合成された回路の信頼性である。大規模回路の設計になるほど言語記述によるハードウェア設計を行なう利点は大きいと言える。また、回路が大規模になればなるほど、設計された回路の性能が、論理合成結果の良否に左右されるところが大きい。しかし現在の段階では、論理合成ツールの性能によっては、必ずしも効率の良い回路設計ができるわけではないということが言える。

さらに、現在発表されている論理合成ツールがすべての VHDL 記述をサポートしているわけではなく、そのサブセットでしか入力言語をサポートしていないという問題がある。除算演算子、while loop文、単純Loop文など本来 VHDL で使用できるはずの記述を行なうことができない。また、ステートマシンなどの論理合成を設計者の意図どうりに行なうためには論理合成ツールによって規定されたフォーマットに従った記述を行なわなければならないという制約がある。

ハードウェア記述言語による設計手法が、今後のプロセッサ設計の主流となるには、論理合成ツールの VHDL フルサポート、論理記述の自由度を増すこと、パイプライン処理の設計のサポートなどを行なっていく必要があるといえよう。

8　謝辞

本論文をまとめるにあたり、研究の指導、適切な助言を与えて下さった名古屋大学工学部電子情報工学科の島田俊夫教授に感謝いたします。

優秀なソフトウエアを貸してくださったメンターグラフィックス社ならびに VHDL に関しての相談にのってくださった江森氏に感謝します。

卒業研究をともに行ない、数々のアドバイスをしてくださった島田研究室の卒研生の皆さんに感謝の意を示します。

研究を影ながらささえ、応援してくれた工藤寿子嬢に感謝します。

参考文献

[1] ヘネシー & パターソン,「コンピュータアーキテクチャ」, 日経ＢＰ社,1992 年 12 月

[2] R. リプセット,C. シェーファー,C ユーザリィ,「ＶＨＤＬ, 言語記述によるハードウェア設計へのアプローチ」, マグロウヒル,1990 年 6 月

[3] 「AutoLogic VHDL シンセシスガイド」, メンターグラフィックスジャパン株式会社,1993 年 9 月

[4] 池田,「情報工学実験」,1993 年 3 月

[5] 末吉, 田中, 小羽田, 志々目, 奥村, 久我,「ＦＰＧＡを利用した教育用マイクロプロセッサ:ＫＩＴＥ」

[6] ＫＩＴＥマイクロプロセッサプロジェクト,「ＫＩＴＥマイクロプロセッサ リファレンスマニュアル」,1993 年

[7] 藤原,「コンピュータの設計とテスト」, 工学図書,1990 年８月

[8] 河原林 他,「Varchsyn(2):VHDL からの合成」情報処理学会講演論文集,1993 年 3 月

[9] 山口 他,「トップダウン方式による LSI の自動論理設計」, 情報処理学会講演論文集,1993 年 3 月