課題研究報告書

命令セットによるマイクロアーキテクチャへの影響に関する研究

北陸先端科学技術大学院大学
情報科学研究科

桑田　正明

2016 年 3 月

課題研究報告書

命令セットによるマイクロアーキテクチャへの影響に関する研究

1310704　桑田　正明

主指導教員　田中　清史
審査委員主査　田中　清史
審査委員　金子　峰雄
審査委員　井口　寧

北陸先端科学技術大学院大学
情報科学研究科

平成 28 年 2 月

概 要

フォン・ノイマン型プロセッサが実行する命令セットには様々なものがあるが、それらは汎用性を考慮すると類似点が多いため、優劣が議論されることは稀である。しかし、対象アプリケーションによって使用される命令群は異なり、命令効率の観点から命令セットアーキテクチャ（Instruction Set Architecture, ISA）の差異によって性能差が出ることが予想される。

本研究では、ISA がマイクロアーキテクチャにどのように影響しているかを明らかにすることを目的として、FPGA（Field Programmable Gate Array）をターゲットとし、複数の ISA のマイクロアーキテクチャを開発した。具体的には、ISA それぞれについてプロセッサを設計し、その実装した結果を比較することにより、ISA の差異が速度／ハードウェアサイズ等にどのように影響を与えるかを明らかにした。また、マイクロアーキテクチャにおける ISA に非依存な共通コンポーネントと、類似コンポーネント、特殊コンポーネントを整理し、コンポーネントの選定と接続変更により異なる ISA のプロセッサが構成可能であることを示した。

なお、本研究では対象 ISA として MIPS、ARM 及び SPARC の命令セットを採用した。SPARC は近年では使用例が減少しているため実用性は高くないが、MIPS、ARM との比較のために対象にした。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

プロセッサ業界では歴史的に様々な命令セットアーキテクチャ（Instruction Set Architecture, ISA）が開発されてきた。CISC（Complex Instruction Set Computer）対 RISC（Reduced Instruction Set Computer）論争が活発な時代もあったが、近年は ISA の相違による影響を議論する機会はほとんどない。これは、使用する ISA とは無関係に、プロセッサ内部ではある程度共通化したマイクロアーキテクチャが採用されているためである。ここでマイクロアーキテクチャとは ISA より下位の電子回路の設計を示す。従って、現在の研究・開発現場では ISA の性質に着目することなく、自身にとって使用実績のある ISA を採用する傾向がある。しかし、対象アプリケーションによって使用される命令群は異なり、命令効率の観点から ISA の差異によって性能差が出ることが予想される。

また、専用用途の ASIC（Application Specific Integrated Circuit）開発では ISA を固定する必要があるが、FPGA（Field Programmable Gate Array）をターゲットとする場合は ISA の変更がほぼコスト無しで実現できる。したがって FPGA において異なる ISA を使用し、その実行結果に有意な差異が確認できれば、ISA の選択のための方法論を確立する意義を示すことになり、今後のプロセッサアーキテクチャ分野における学術的な道標となりうる。本研究におけるプロセッサ設計は広く設計現場で普及しているハードウェア記述言語 (Hardware Description Language）を使用するため可搬性が高く、さらに複数の ISA に適用可能であることから、幅広い研究開発現場における有用性が期待できる。

1.2　目的

本研究では FPGA をターゲットとし、複数の ISA のマイクロアーキテクチャを開発する。これにより、ISA がマイクロアーキテクチャにどのように影響しているかを明らかにすることを目的とする。

なお、本研究では対象 ISA として MIPS、ARM 及び SPARC の命令セットを採用する。SPARC は近年では使用例が減少しているため実用性は高くないが、MIPS、ARM との比較のために対象にする。

1.3　研究方法

本研究では、MIPS、ARM、SPARC の ISA でプロセッサを設計する。これらは RISC 型命令セットに分類される。プロセッサの動作記述には、ASIC や FPGA の開発で回路記述方法として標準となっているハードウェア記述言語の Verilog HDL を使用する。

次に、実装した各 ISA のプロセッサを比較することにより、ISA の差異が速度／ハードウェアサイズ等にどのように影響を与えるかを明らかにする。

また、上記で設計したプロセッサに基づき、マイクロアーキテクチャにおける ISA に非依存な共通コンポーネントと類似コンポーネント、特殊コンポーネントを整理し、コンポーネントの選定と接続変更により異なる ISA のプロセッサが構成可能であることを示す。

1.4　本報告書の構成

第 2 章では各 ISA の歴史、背景、命令形式について述べ、各 ISA に基づくプロセッサの基本設計とシミュレーションの内容について述べる。

第 3 章ではソースファイルの評価、実装した回路の評価、ISA とモジュールの関係について述べる。

第2章　CPUの設計

2.1　MIPS

2.1.1　MIPSの歴史

MIPSは”Microprocessor without Interlocked Pipeline Stages”（パイプラインステージがインターロックされないマイクロプロセッサ）に由来している [1]。

1981年、スタンフォード大学のJohn L. Hennessy率いるチームは最初のMIPSプロセッサを開発するプロジェクトを開始した [2]。

1984年、Hennessyはミップス・コンピュータシステムズを設立した [2]。その後、1985年に最初の製品であるR2000を完成させ、1988年にそれを進化させたR3000を完成させた [3]。

1991年、ミップス･コンピュータシステムズは最初の64ビットマイクロプロセッサR4000を発売した [3]。

1992年、米国SGI(Silicon Graphics Inc.)はミップス・コンピュータシステムズを買い取った。ミップス・コンピュータシステムズはSGIの子会社となり、社名もミップス・テクノロジーズに変更された [3]。その後MIPSアーキテクチャは機器組み込み用プロセッサ分野で広く使われ、1998年にミップス・テクノロジーズはSGIから独立した [3]。

2013年、ミップス･テクノロジーズは英国イマジネーションテクノロジーズ(Imagination Technologies Group plc.)に買い取られた [3]。

最近ではMIPSアーキテクチャはIP(Intellectual Property)コアとして、機器組み込み用プロセッサの設計に使われることが多い [4, 8]。

2.1.2　MIPSの背景

MIPSアーキテクチャはスーパーコンピュータにも採用された。SGIはスーパーコンピュータ市場への足がかりとして、Onyxシリーズと POWER CHALLENGEシリーズを発売した。CPUはMIPSのR4400SCを使用している [5]。さらにR14000を構成できるOrigin 3000を開発した [6]。しかし、その後SGIはMIPSベースのスーパーコンピュータの開発をやめた [7]。また、最近のデスクトップ市場では、MIPSプロセッサはほぼ完全に消えている。

最近ではMIPSアーキテクチャはIPコアとして、様々な機器組み込み用プロセッサに利用されている [8]。例えば、シスコシステムズのルーター [9]、レーザープリンター [10]、セットトップボックス [11]、家庭用ゲーム機のPlayStation、PlayStation2、Nintendo64などで使われている [12]。

2.1.3 MIPSの命令形式

MIPSの命令は以下のR、I、Jの3種類に分類される。

1. R形式

   R形式の命令は図2.1の形式となる。

フィールド	op	rs	rt	rd	shamt	funct
ビット位置	31-26	25-21	20-16	15-11	10-6	5-0

図 2.1: R形式

   各フィールドの意味は以下の通り。

   - op:命令の基本操作。命令操作コード (opcode, オペコード) と呼ばれている。
     - R形式のopは0
   - rs:第1のソース・オペランドのレジスタ。
   - rt:第2のソース・オペランドのレジスタ。
   - rd:デスティネーション・オペランドのレジスタ。結果を収める先。
   - shamt:シフト量。
   - funct:機能。命令操作コードのバリエーションを表す。機能コード (function code) と呼ばれている。

2. I形式

   I形式の命令は図2.2の形式となる。

フィールド	op	rs	rt	address
ビット位置	31-26	25-21	20-16	15-0

図 2.2: I形式

   各フィールドの意味は以下の通り。

- op:命令の基本操作。命令操作コード (opcode, オペコード) と呼ばれている。
  - op が 35 のとき lw 命令
  - op が 43 のとき sw 命令
  - op が 4 のとき beq 命令
  - op が 5 のとき bne 命令
  - op が 8 のとき addi 命令
- rs:addi 命令、sw 命令、lw 命令ではベース・オペランドのレジスタ。
  beq 命令、bne 命令ではソース・オペランドのレジスタ。
- rt:addi 命令、lw 命令ではデスティネーション・オペランドのレジスタ。結果を収める先。
  sw 命令、beq 命令、bne 命令ではソース・オペランドのレジスタ。
- address:定数。

3. J 形式

J 形式の命令は図 2.3 の形式となる。

フィールド	op	address
ビット位置	31-26	25-0

図 2.3: J 形式

各フィールドの意味は以下の通り。

- op:命令の基本操作。命令操作コード (opcode, オペコード) と呼ばれている。
  - op が 2 のとき j(ジャンプ) 命令
- address:定数。

2.1.4 MIPS プロセッサの基本設計

MIPS プロセッサの基本設計として、1 命令を 1 クロックサイクルで実行するシングルサイクル方式とし、キャッシュメモリは使用せずに、表 2.1 の命令を対象とする。

表 2.1: MIPS で設計した命令

	命令	命令内容	命令形式
1	add	add	R
2	sub	subtract	R
3	and	and	R
4	or	or	R
5	slt	set on less than	R
6	addi	add immediate	I
7	sw	store word	I
8	lw	load word	I
9	beq	branch on equal	I
10	bne	branch on not equal	I

本研究では図 2.4 のようなモジュールを設計し接続した。その際は David A. Patterson,John L. Hennessy のコンピュータの構成と設計第 4 版上 [13] の図 4.21 を参考にした。

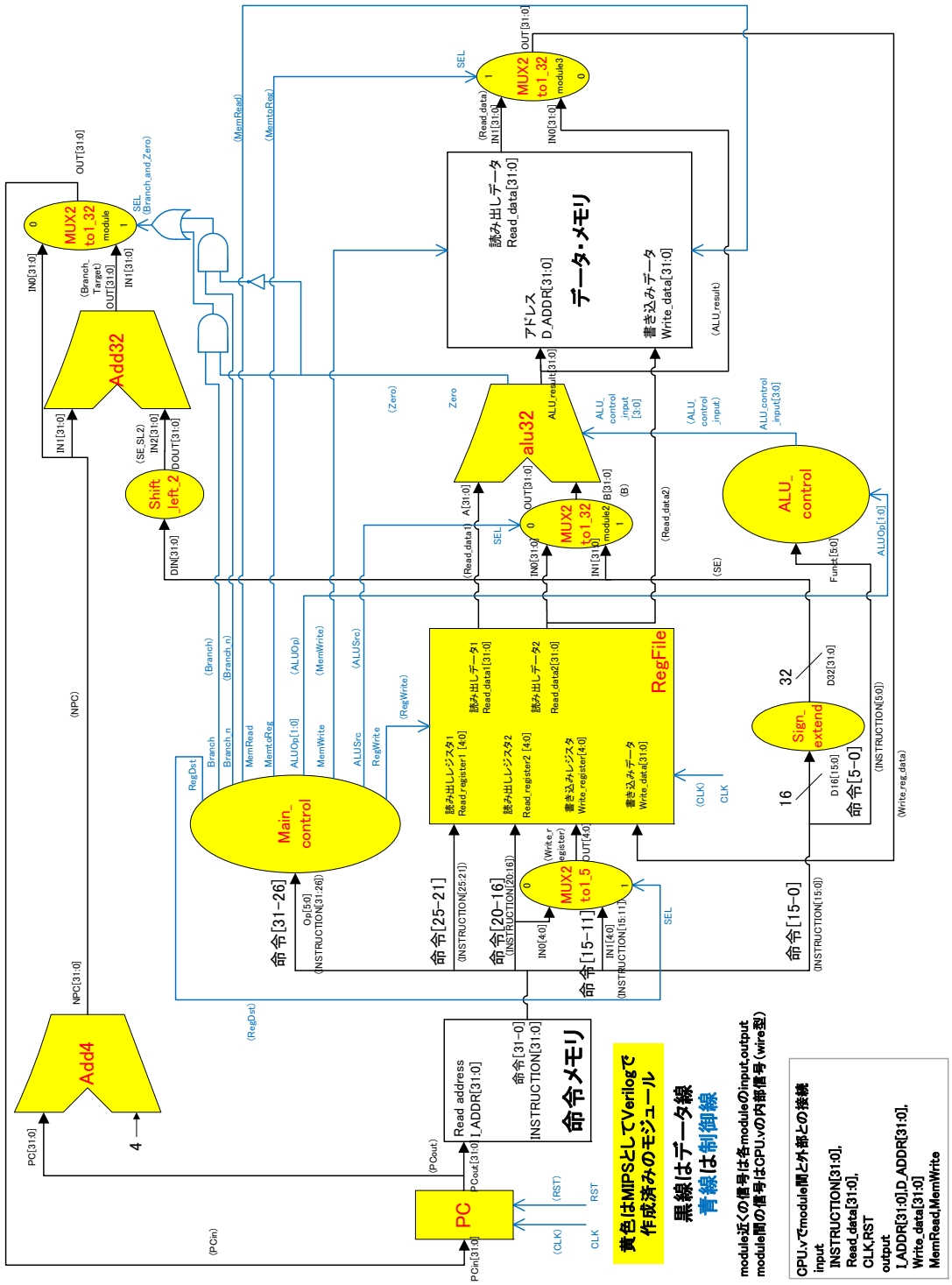

図 2.4: MIPS プロセッサのデータパスの図

図 2.4 のモジュールは以下の通りである。

1. CPU

   モジュール間、モジュールと外部との接続をする。分岐先制御信号の生成。

2. PC

   プログラムカウンタの役割をする。

3. Add4

   次の命令アドレスのために、現在の命令アドレスに 4 を加算する。

4. Add32

   分岐先アドレス生成のために、「現在の命令アドレス+4」に「符号拡張後に 2 ビット左シフト」を加算する。

5. MUX2to1_32(module)

   分岐先アドレスと次の命令アドレスのいずれかを選択するマルチプレクサ。

6. Shift_left_2

   2 ビット左シフト。

7. Sign_extend

   16 ビットを 32 ビットに符号拡張する。

8. alu32

   add、sub、and、or の演算を実行する。演算結果の Zero 信号の生成。

9. MUX2to1_32(module2)

   alu32 の入力を選択するマルチプレクサ。レジスタの値と符号拡張後の値のいずれかを選択する。

10. RegFile

    レジスタの読み出しと書き込み。

11. MUX2to1_5

    書き込みレジスタの選択をするマルチプレクサ。R 形式命令の場合は命令 [15-11](rd)、ロード命令の場合は命令 [20-16](rt) を選択する。

12. MUX2to1_32(module3)

    レジスタに書き込む値を選択するマルチプレクサ。データメモリから読み出した値と ALU 演算結果のいずれかを選択する。

13. Main_control

    命令から制御信号を生成する。

14. ALU_control

    Main_control の制御信号と命令 [5-0](funct) から alu32 の制御信号を生成する。

2.1.5 シミュレーション

本研究でシミュレーションしたプログラムは表 2.2 の通りである。このプログラムでは 1+2+3+4+5+6+7+8+9=45 の計算を加算命令、ストア命令、分岐命令、ロード命令を用いて実行する。

表 2.2: MIPS でシミュレーションしたプログラム

命令メモリ アドレス	命令	説明
0:	add $1,$0,$0	$1 に 0 をセット
4:	addi $2,$0,10	$2 に 10 をセット（ループ回数）
8:	addi $3,$0,$0	$3 に 0 をセット
12:	addi $4,$0,$0	$4 に 0 をセット
16:	sw $1, 0($4)	$4 番地に$1 を格納
20:	addi $4,$4,4	$4 内の番地の値を 4 増やす
24:	addi $1,$1,1	$1 の値を 1 増やす
28:	bne $1,$2,-4	ループ条件; 等しくないなら命令メモリアドレス 16 へ分岐
32:	add $1,$0,$0	$1 に 0 をセット
36:	add $4,$0,$0	$4 に 0 をセット
40:	lw $5,0($4)	$4 番地から$5 に値の読み出し
44:	add $3,$3,$5	読み出した値 ($5) を総和値 ($3) に加算
48:	addi $4,$4,4	$4 内の番地の値を 4 増やす
52:	addi $1,$1,1	$1 の値を 1 増やす
56:	bne $1,$2,-5	ループ条件; 等しくないなら命令メモリアドレス 40 へ分岐
60:	sw $3, 0($0)	総和値 ($3) を 0 番地に格納
64:	beq $0,$0,-1	無限自己ループで終了

表 2.2 のプログラムを Xilinx, Inc. の ISim [14] でシミュレーションした結果の波形画像を図 2.5 に示す。命令メモリアドレス (I_ADDR) が 60 のとき、データ・メモリアドレス (D_ADDR)0 に 45 を書き込んでいる (Write_data=45)。(Write 45 to address 0)

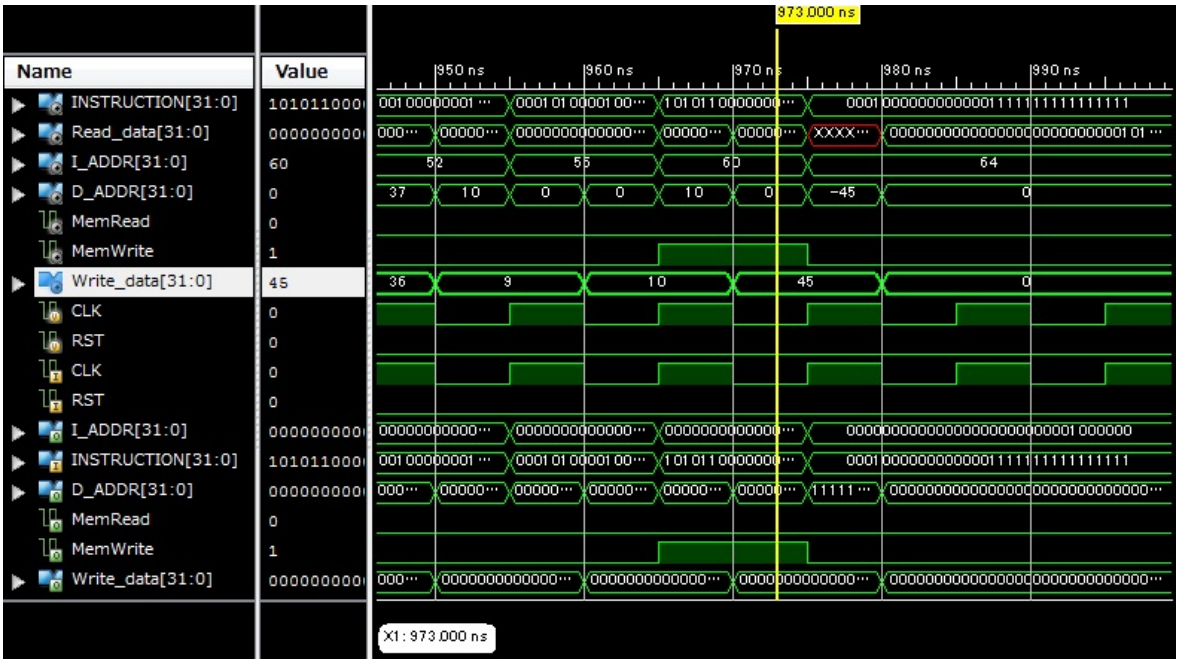

図 2.5: MIPS プロセッサでのシミュレーション結果

2.2 ARM

2.2.1 ARM の歴史

ARM は ”Advanced RISC Machines”の略である [15]。

1983 年、英国 Acorn Computers Ltd. は ARM の開発を開始した [16]。

1985 年、Acorn Computers Ltd. は 32 ビット RISC マイクロプロセッサ ARM1 を開発し、1986 年に ARM2 を開発、その後 ARM3 を開発した [16, 17]。

1990 年、Acorn Computers Ltd. と Apple Computer の共同事業から、新しいマイクロプロセッサ標準を確立する目的で、Advanced RISC Machines Ltd. が独立した [18]。

1991 年、Advanced RISC Machines Ltd. は自社の初めてのエンベデッド可能 RISC コアである ARM6 を発表し、1993 年に ARM7 を発表した [18]。ARM7 は携帯電話で需要が高かった [16]。

1998 年、社名を Advanced RISC Machines Ltd. から ARM Ltd. に変更し、持ち株会社 ARM Holdings Plc の傘下になった。日本法人のアームは、ARM Ltd. の子会社である [17, 19]。

その後、ARM Ltd. は機能を向上しながら ARM8、ARM9、ARM10、ARM11 と製品を展開した [16]。ARM11 ファミリの次のコアは Cortex ファミリとした。ARM Ltd. は携帯電話市場で急成長したが、携帯電話以外の幅広い用途（高性能なデジタル家電から産業機器まで）に使われるようになり、幅広い性能と機能をカバーする必要が出てきた。そこで、性能や機能に関係なく Cortex というファミリで統一し、用途別にアーキテクチャを定義

した。具体的には Cortex-A(ARM Application Processors)、Cortex-R(ARM Embedded Real-time Processors)、Cortex-M(ARM Embedded Processors) となる [20, 21]。

2012 年、ARM Ltd. は同社製品として初となる 64 ビットプロセッサ「Cortex-A50」シリーズを発表した。2011 年に発表した 64 ビット処理向けのアーキテクチャ「ARMv8」に対応し、スマートフォンやタブレット端末、サーバーなどの用途に向けた [22]。

2.2.2 ARM の背景

ARM Ltd. のビジネスモデルでは、半導体チップの製造と販売ではなく、IP の設計とライセンスに重点を置いている。実際には自社の IP のライセンスを、半導体及びシステムの企業に供与している。これらの企業は、ARM の IP 設計を利用してシステムオンチップ設計を作成及び製造し、ARM Ltd. に元の IP のライセンス料と、製造した各チップまたはウエハーのロイヤリティを支払う [23]。

ARM コアは、コストや低消費電力も重視しながら開発されている為、安価で低消費電力という特徴がある [16]。最近では、携帯電話やデジタルセットトップボックス、自動車のブレーキシステムやネットワークルータなどで使用されており、ARM テクノロジはスマートフォンの 95%、デジタルカメラの 80%、すべての電子デバイスの 35%で使用されている [23]。

2.2.3 ARM の命令形式

ARM の命令形式は主に以下の 5 種類となる。また、本節の説明は ARM アーキテクチャリファレンスマニュアル v6 [24] の A3.1 命令セットのエンコードに基づいている。

1. データ処理イミディエートシフト

   データ処理イミディエートシフトの命令は図 2.6 の形式となる。

フィールド	cond	0,0	I	opcode	S	Rn	Rd	shift_imm	shift	0	Rm
ビット位置	31-28	27,26	25	24-21	20	19-16	15-12	11-7	6,5	4	3-0

図 2.6: データ処理イミディエートシフトの形式

各フィールドの意味は以下の通り。

- cond:条件フィールド。条件コードフラグ (N,Z,C,V) が命令で指定された条件を満たしている場合のみ命令が実行される。条件コードフラグが条件を満たしていない場合、この命令は NOP(No Operation) として動作する。常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行される。

  - cond が 0000 のとき EQ(=) となる。条件コードフラグの状態は Z セット。
  - cond が 0001 のとき NE(≠) となる。条件コードフラグの状態は Z クリア。
  - cond が 1110 のとき AL (常時) となる。条件コードフラグの状態は無条件。
- 27,26 ビット:データ処理命令 (MOV,ADD,SUB,AND,CMP など) では 00 となる。
- I:I ビットと呼ばれ、イミディエートシフタオペランドとレジスタベースのシフタオペランドを区別する。データ処理命令の場合は、イミディエートシフタオペランドのとき I=1, レジスタベースのシフタオペランドのとき I=0 となる。この形式はレジスタベースのシフタオペランドなので I=0 となる。
- opcode:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
  - opcode が 1101 のとき MOV 命令
  - opcode が 0100 のとき ADD 命令
  - opcode が 0010 のとき SUB 命令
  - opcode が 0000 のとき AND 命令
  - opcode が 1010 のとき CMP 命令
- S:S ビットと呼ばれ、命令が条件コードフラグを変更することを示す。この命令で条件コードフラグを変更するとき S=1, この命令で条件コードフラグを変更しないとき S=0 となる。
- Rn:最初のソース・オペランドのレジスタ。
- Rd:デスティネーション・オペランドのレジスタ。結果を収める先。
- shift_imm:シフト量。
- shift:シフトの種類。
  - shift が 00 のとき論理左シフト (LSL)
  - shift が 01 のとき論理右シフト (LSR)
  - shift が 10 のとき算術右シフト (ASR)
  - shift が 11 のとき右ローテート (ROR)
- 4 ビット:イミディエートシフトのとき 0、レジスタシフトのとき 1 となる。この形式はイミディエートシフトなので 0 となる。
- Rm:シフトされる値を保持するレジスタ。イミディエート値による論理左シフトでシフト量が 0(shift_imm=0) のとき、Rm のレジスタの値が直接オペランドとして使用される。

2. イミディエートデータ処理

   イミディエートデータ処理の命令は図 2.7 の形式となる。

フィールド	cond	0,0	I	opcode	S	Rn	Rd	rotate_imm	immed_8
ビット位置	31-28	27,26	25	24-21	20	19-16	15-12	11-8	7-0

図 2.7: イミディエートデータ処理の形式

各フィールドの意味は以下の通り。

- cond:条件フィールド。条件コードフラグ (N,Z,C,V) が命令で指定された条件を満たしている場合のみ命令が実行される。条件コードフラグが条件を満たしていない場合、この命令は NOP(No Operation) として動作する。常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行される。
  - cond が 0000 のとき EQ(=) となる。条件コードフラグの状態は Z セット。
  - cond が 0001 のとき NE($\neq$) となる。条件コードフラグの状態は Z クリア。
  - cond が 1110 のとき AL (常時) となる。条件コードフラグの状態は無条件。
- 27,26 ビット:データ処理命令 (MOV,ADD,SUB,AND,CMP など) では 00 となる。
- I:I ビットと呼ばれ、イミディエートシフタオペランドとレジスタベースのシフタオペランドを区別する。データ処理命令の場合は、イミディエートシフタオペランドのとき I=1, レジスタベースのシフタオペランドのとき I=0 となる。この形式はイミディエートシフタオペランドなので I=1 となる。
- opcode:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
  - opcode が 1101 のとき MOV 命令
  - opcode が 0100 のとき ADD 命令
  - opcode が 0010 のとき SUB 命令
  - opcode が 0000 のとき AND 命令
  - opcode が 1010 のとき CMP 命令
- S:S ビットと呼ばれ、命令が条件コードフラグを変更することを示す。この命令で条件コードフラグを変更するとき S=1, この命令で条件コードフラグを変更しないとき S=0 となる。
- Rn:最初のソース・オペランドのレジスタ。
- Rd:デスティネーション・オペランドのレジスタ。結果を収める先。
- rotate_imm:右ローテート量。immed_8 を 2× rotate_imm ビットだけ右ローテートする。
- immed_8:定数。

3. レジスタオフセットでのロード/ストア

   レジスタオフセットでのロード/ストアの命令は図 2.8 の形式となる。

フィールド	cond	0,1	I	P	U	B	W	L	Rn	Rd	shift_imm	shift	0	Rm
ビット位置	31-28	27,26	25	24	23	22	21	20	19-16	15-12	11-7	6,5	4	3-0

図 2.8: レジスタオフセットでのロード/ストアの形式

各フィールドの意味は以下の通り。

- cond:条件フィールド。条件コードフラグ (N,Z,C,V) が命令で指定された条件を満たしている場合のみ命令が実行される。条件コードフラグが条件を満たしていない場合、この命令は NOP(No Operation) として動作する。常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行される。
  - cond が 0000 のとき EQ(=) となる。条件コードフラグの状態は Z セット。
  - cond が 0001 のとき NE($\neq$) となる。条件コードフラグの状態は Z クリア。
  - cond が 1110 のとき AL (常時) となる。条件コードフラグの状態は無条件。
- 27,26 ビット:ロード/ストア命令 (LDR,STR) では 01 となる。
- I:I ビットと呼ばれ、イミディエートオフセットとレジスタオフセットを区別する。ロード/ストア命令の場合は、イミディエートオフセットのとき I=0, レジスタオフセットのとき I=1 となる。この形式はレジスタオフセットなので I=1 となる。
- P: P ビットと呼ばれ、以下の 2 つの意味がある。
  - P=0
    ポストインデクスアドレシングの使用を指定する。ベースレジスタ値はメモリアドレスに使用された後、オフセットがベースレジスタ値に適用され、ベースアドレスに書き戻される。
  - P=1
    オフセットアドレシングまたはプリインデクスアドレシングの使用を指定する。どちらが使用されるかは W ビットによって決定される。メモリアドレスは、ベースレジスタの値にオフセットを適用して生成される。
- U:U ビットと呼ばれ、オフセットをベースに加算するか（U = 1）ベースから減算するか（U = 0）のいずれかを示す。
- B:B ビットと呼ばれ、符号なしバイトアクセス（B = 1）とワードアクセス（B = 0）を区別する。
- W ビットと呼ばれ、以下の 2 つの意味がある。

- P=0
  W = 0 の場合、通常のメモリアクセスが実行される。W = 1 の場合、非特権の（ユーザモード）メモリアクセスが実行される。
- P=1
  W = 0 の場合、ベースレジスタは更新されない（オフセットアドレシング)。W = 1 の場合、計算されたメモリアドレスがベースレジスタに書き戻される（プリインデクスアドレシング)。

- L:L ビットと呼ばれ、ロード（L = 1）とストア（L = 0）を区別する。
- Rn:ベースアドレスを保持するレジスタを指定する。
- Rd:内容がロードまたはストアされるレジスタを示す。
- shift_imm:シフト量。
- shift:シフトの種類。
  - shift が 00 のとき論理左シフト (LSL)
  - shift が 01 のとき論理右シフト (LSR)
  - shift が 10 のとき算術右シフト (ASR)
  - shift が 11 のとき右ローテート (ROR)
- 4 ビット:この形式では 0 となる。
- Rm:Rn に加算または減算するオフセットを含むレジスタを指定する。

4. イミディエートオフセットでのロード/ストア

   イミディエートオフセットでのロード/ストアの命令は図 2.9 の形式となる。

フィールド	cond	0,1	I	P	U	B	W	L	Rn	Rd	offset_12
ビット位置	31-28	27,26	25	24	23	22	21	20	19-16	15-12	11-0

図 2.9: イミディエートオフセットでのロード/ストアの形式

各フィールドの意味は以下の通り。

- cond:条件フィールド。条件コードフラグ (N,Z,C,V) が命令で指定された条件を満たしている場合のみ命令が実行される。条件コードフラグが条件を満たしていない場合、この命令は NOP(No Operation) として動作する。常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行される。
  - cond が 0000 のとき EQ(=) となる。条件コードフラグの状態は Z セット。
  - cond が 0001 のとき NE($\neq$) となる。条件コードフラグの状態は Z クリア。

- cond が 1110 のとき AL (常時) となる。条件コードフラグの状態は無条件。

- 27,26 ビット:ロード/ストア命令 (LDR,STR) では 01 となる。
- I:I ビットと呼ばれ、イミディエートオフセットとレジスタオフセットを区別する。ロード/ストア命令の場合は、イミディエートオフセットのとき I=0, レジスタオフセットのとき I=1 となる。この形式はイミディエートオフセットなので I=0 となる。
- P: P ビットと呼ばれ、以下の 2 つの意味がある。
  - P=0
    ポストインデクスアドレシングの使用を指定する。ベースレジスタ値はメモリアドレスに使用された後、オフセットがベースレジスタ値に適用され、ベースアドレスに書き戻される。
  - P=1
    オフセットアドレシングまたはプリインデクスアドレシングの使用を指定する。どちらが使用されるかは W ビットによって決定される。メモリアドレスは、ベースレジスタの値にオフセットを適用して生成される。
- U:U ビットと呼ばれ、オフセットをベースに加算するか（U = 1）ベースから減算するか（U = 0）のいずれかを示す。
- B:B ビットと呼ばれ、符号なしバイトアクセス（B = 1）とワードアクセス（B = 0）を区別する。
- W ビットと呼ばれ、以下の 2 つの意味がある。
  - P=0
    W = 0 の場合、通常のメモリアクセスが実行される。W = 1 の場合、非特権の（ユーザモード）メモリアクセスが実行される。
  - P=1
    W = 0 の場合、ベースレジスタは更新されない（オフセットアドレシング)。W = 1 の場合、計算されたメモリアドレスがベースレジスタに書き戻される（プリインデクスアドレシング)。
- L:L ビットと呼ばれ、ロード（L = 1）とストア（L = 0）を区別する。
- Rn:ベースアドレスを保持するレジスタを指定する。
- Rd:内容がロードまたはストアされるレジスタを示す。
- offset_12:Rn の値と共にアドレスを構成するイミディエートオフセットを指定する。

5. 分岐およびリンク付き分岐

   分岐およびリンク付き分岐の命令は図 2.10 の形式となる。

フィールド	cond	1,0,1	L	signed_immed_24
ビット位置	31-28	27,26,25	24	23-0

図 2.10: 分岐およびリンク付き分岐の形式

各フィールドの意味は以下の通り。

- cond:条件フィールド。条件コードフラグ (N,Z,C,V) が命令で指定された条件を満たしている場合のみ命令が実行される。条件コードフラグが条件を満たしていない場合、この命令は NOP(No Operation) として動作する。常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値にかかわらず実行される。
  - cond が 0000 のとき EQ(=) となる。条件コードフラグの状態は Z セット。
  - cond が 0001 のとき NE($\neq$) となる。条件コードフラグの状態は Z クリア。
  - cond が 1110 のとき AL (常時) となる。条件コードフラグの状態は無条件。
- 27,26,25 ビット: 分岐およびリンク付き分岐 (B,BL) では 101 となる。
- L:L ビットと呼ばれる。L=1 のとき、復帰アドレスがリンクレジスタ（R14）に格納される。L=0 のとき、命令は復帰アドレスを格納せず、単に分岐を発生させる。
- signed_immed_24:定数。

2.2.4 ARM プロセッサの基本設計

ARM プロセッサの基本設計として、1 命令を 1 クロックサイクルで実行するシングルサイクル方式とし、キャッシュメモリは使用せずに、表 2.3 の命令を対象とする。なお、ARM プロセッサは Thumb 命令と呼ばれるコード効率の向上を意図した 16 ビットの命令を持っているが、Thumb 命令は対象外とした。

表 2.3: ARM で設計した命令

	命令	命令内容	命令の種類
1	MOV	move	データ処理命令
2	ADD	add	データ処理命令
3	SUB	subtract	データ処理命令
4	AND	and	データ処理命令
5	CMP	compare	データ処理命令
6	STR	store register	ロード/ストア命令
7	LDR	load register	ロード/ストア命令
8	BNE	branch on not equal	分岐命令
9	BEQ	branch on equal	分岐命令
10	BAL	branch always	分岐命令

また、図 2.6 - 2.10 の命令形式と表 2.3 の命令の種類との関係を表 2.4 に示す。

表 2.4: ARM での命令形式と命令の種類との関係

	命令形式	命令の種類
1	データ処理イミディエートシフト	データ処理命令
2	イミディエートデータ処理	データ処理命令
3	レジスタオフセットでのロード/ストア	ロード/ストア命令
4	イミディエートオフセットでのロード/ストア	ロード/ストア命令
5	分岐およびリンク付き分岐	分岐命令

本研究では図 2.11 のようなモジュールを設計し接続した。その際は ARM Ltd. の ARM アーキテクチャリファレンスマニュアル v6 [24] の ARM 命令を参考にした。

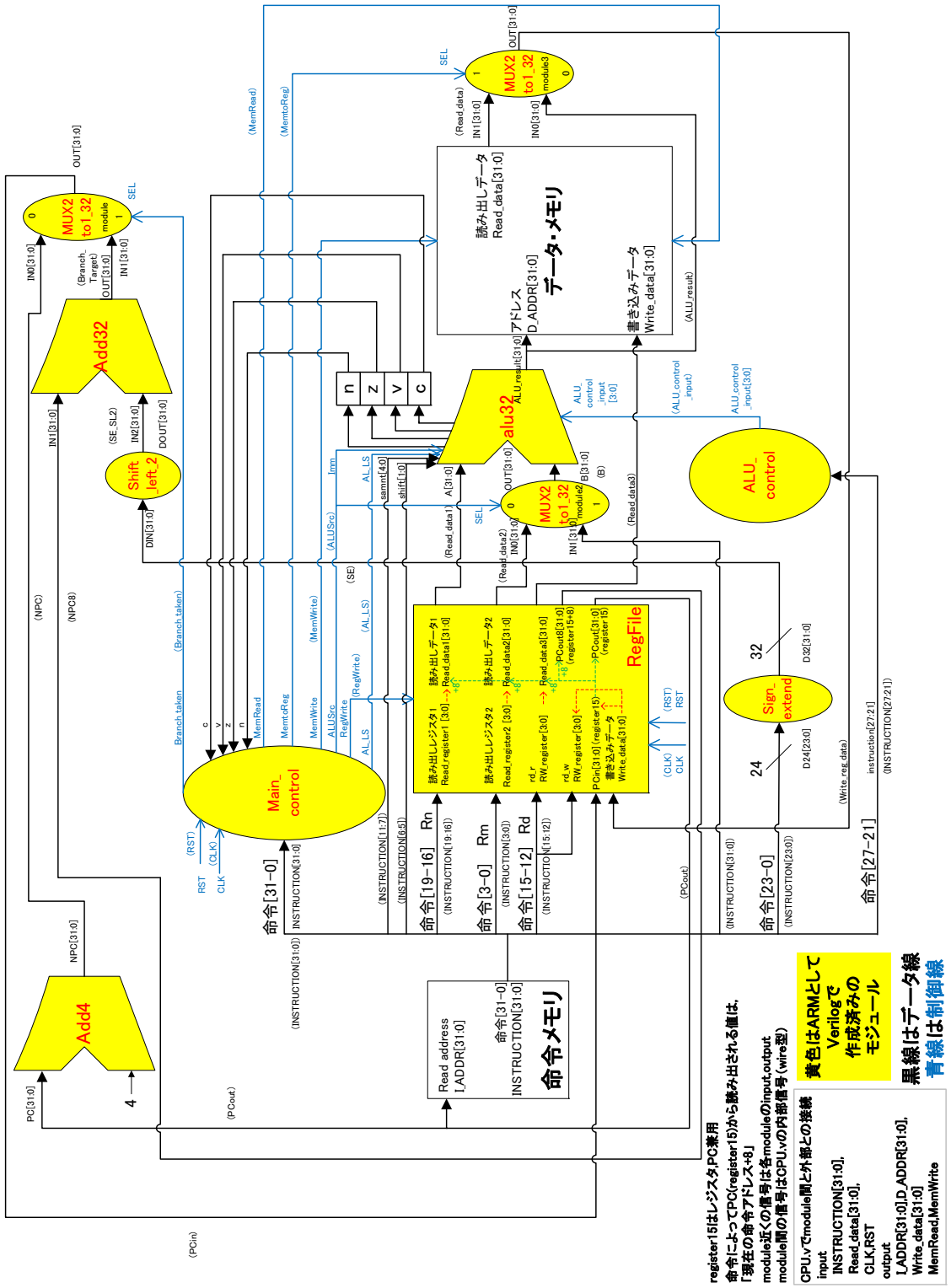

図 2.11: ARM プロセッサのデータパスの図

図 2.11 のモジュールは以下の通りである。

1. CPU

   モジュール間、モジュールと外部との接続をする。

2. Add4

   次の命令アドレスのために、現在の命令アドレスに 4 を加算する。

3. Add32

   分岐先アドレス生成のために、「現在の命令アドレス+8」に「符号拡張後に 2 ビット左シフト」を加算する。

4. MUX2to1_32(module)

   分岐先アドレスと次の命令アドレスのいずれかを選択するマルチプレクサ。

5. Shift_left_2

   2 ビット左シフト。

6. Sign_extend

   24 ビットを 32 ビットに符号拡張する。

7. alu32

   add、sub、and、論理左シフト、論理右シフト、算術右シフト、右ローテートの演算を実行する。演算結果から条件コードフラグ (N,Z,C,V) を生成する。

8. MUX2to1_32(module2)

   alu32 の入力を選択するマルチプレクサ。レジスタの値と命令の値のいずれかを選択する。

9. RegFile

   レジスタの読み出しと書き込み。プログラムカウンタの役割もする。register15 はレジスタとプログラムカウンタを兼用している。命令によってプログラムカウンタ (register15) から読み出される値は、「現在の命令アドレス+8」。

10. MUX2to1_32(module3)

    レジスタに書き込む値を選択するマルチプレクサ。データメモリから読み出した値と ALU 演算結果のいずれかを選択する。

11. Main_control

    命令と条件コードフラグ (N,Z,C,V) から制御信号を生成する。

12. ALU_control

命令 [27-21] から alu32 の制御信号を生成する。

2.2.5 シミュレーション

本研究でシミュレーションしたプログラムは表 2.5 の通りである。このプログラムでは 1+2+3+4+5+6+7+8+9=45 の計算を移動命令、ストア命令、加算命令、比較命令、分岐命令、ロード命令を用いて実行する。

表 2.5: ARM でシミュレーションしたプログラム

命令メモリアドレス	命令	説明
0:	MOV R1, #0	R1 に 0 をセット
4:	MOV R3, #0	R3 に 0 をセット
8:	MOV R4, #0	R4 に 0 をセット
12:	STR R1,[R4,R1,LSL #2]	R4 番地に R1 を格納, メモリアドレスの値を 4 増やす
16:	ADD R1,R1,#1	R1 の値を 1 増やす
20:	CMP R1,#10	R1 と 10 を比較する
24:	BNE -5	ループ条件;R1≠10 なら命令メモリアドレス 12 へ分岐
28:	MOV R1,#0	R1 に 0 をセット
32:	LDR R5,[R4,R1,LSL #2]	R4 番地から R5 にロード, メモリアドレスの値を 4 増やす
36:	ADD R3,R3,R5	ロードした値 (R5) を総和値 (R3) に加算
40:	ADD R1,R1,#1	R1 の値を 1 増やす
44:	CMP R1,#10	R1 と 10 を比較する
48:	BNE -6	ループ条件;R1≠10 なら命令メモリアドレス 32 へ分岐
52:	STR R3,[R4]	総和値 (R3) を 0 (R4) 番地に格納
56:	BAL -2	無限自己ループで修了

表 2.5 のプログラムを Xilinx, Inc. の ISim [14] でシミュレーションした結果の波形画像を図 2.12 に示す。命令メモリアドレス (I_ADDR) が 52 のとき、データ・メモリアドレス

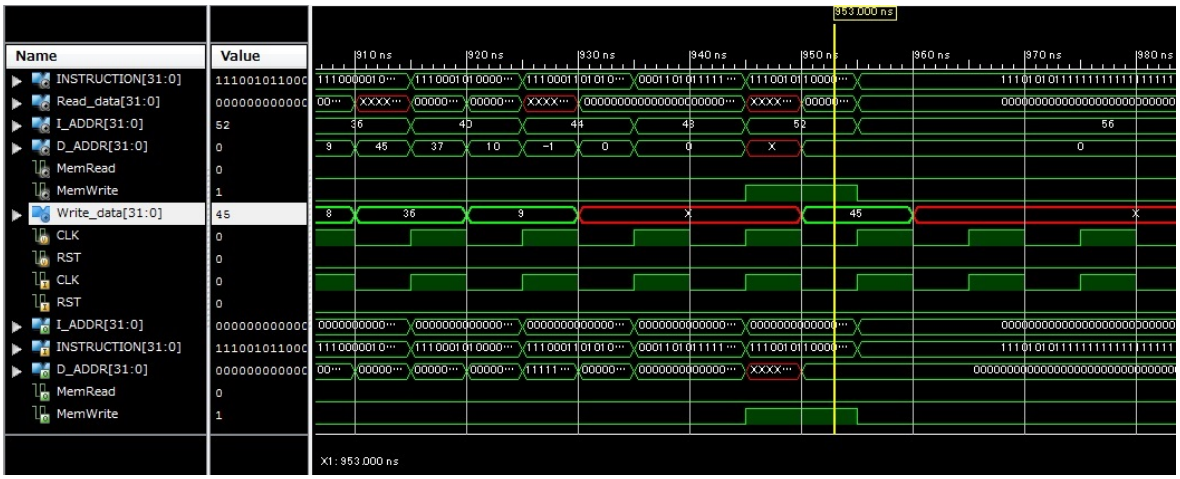

図 2.12: ARM プロセッサでのシミュレーション結果

(D_ADDR)0 に 45 を書き込んでいる (Write_data=45)。(Write 45 to address 0)

2.3 SPARC

2.3.1 SPARC の歴史

SPARC は”Scalable Processor Architecture”の略である [25]。

1984 年、米国カリフォルニア州立大学バークレー校の Dabid Patterson と米国 Sun Mycrosystems が SPARC アーキテクチャの開発に着手した [26]。

1986 年、SPARC アーキテクチャを採用した最初のプロセッサを Sun Mycrosystems と富士通が協同で開発した [27]。

1987 年、Sun Mycrosystems から SPARC プロセッサを搭載したワークステーションが出荷された [27]。

1989 年、Sun Mycrosystems が SPARC アーキテクチャ仕様の所有権を独立非営利組織の SPARC International, Inc. に譲渡し、SPARC International, Inc. が SPARC 技術の管理やライセンス供与を行うようになった [27]。

1990 年、SPARC V8 アーキテクチャ仕様が公開された [26]。1993 年、SPARC V9 アーキテクチャ仕様が公開された [26]。

1994 年、SPARC の命令セットは IEEE 標準 1754-1994 として公開された [26, 27]。

2012 年、理化学研究所と富士通が共同で開発したスーパーコンピュータ「京」が完成した。「京」は SPARC64VIIIfx プロセッサを使用している [28]。

2.3.2 SPARCの背景

2010年、米国OracleはSun Microsystemsを買収した [29]。創業以来、ソフトウェア専業を貫いてきたOracleがSun Microsystemsの買収によって、UNIXサーバーやストレージ装置といったハードウエア事業も手に入れた [30]。

最近では、SPARCプロセッサはOracleや富士通のサーバなどに搭載されている [31, 32]。

2.3.3 SPARCの命令形式

SPARCの命令形式は主に以下の3種類となる。また、本節の説明はThe SPARC Architecture Manual Version 8 [25] のFigure5-1 Summary of Instruction Formatsに基づいている。

1. レジスタオペランド形式

   レジスタオペランド形式の命令は図2.13の形式となる。

フィールド	op	rd	op3	rs1	i=0	asi	rs2
ビット位置	31-30	29-25	24-19	18-14	13	12-5	4-0

図 2.13: レジスタオペランド形式

   各フィールドの意味は以下の通り。

   - op:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
     - opが10のとき arithmetic,logical,save,restore命令
     - opが11のとき memory instructions命令
   - rd: arithmetic,logical,LD,save,restore命令ではデスティネーション・オペランドのレジスタ。結果を収める先。ST命令ではソース・オペランドのレジスタ。
   - op3:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
     - op3が000000のとき ADD命令
     - op3が000100のとき SUB命令
     - op3が010100のとき SUBcc命令
     - op3が000001のとき AND命令
     - op3が000010のとき OR命令
     - op3が000100のとき ST命令
     - op3が000000のとき LD命令

- op3 が 111100 のとき SAVE 命令
- op3 が 111101 のとき RESTORE 命令

- rs1:第 1 のソース・オペランドのレジスタ。
- i=0: arithmetic,logical,save,restore 命令では rs1 と rs2 を対象とする。memory instructions 命令ではメモリアドレス rs1+rs2。
- asi: arithmetic,logical,save,restore 命令では unused(zero)、memory instructions 命令では asi(Address Space Identifiers) を示す。
- rs2: 第 2 のソース・オペランドのレジスタ。

2. 即値オペランド形式

即値オペランド形式の命令は図 2.14 の形式となる。

フィールド	op	rd	op3	rs1	i=1	simm13
ビット位置	31-30	29-25	24-19	18-14	13	12-0

図 2.14: 即値オペランド形式

各フィールドの意味は以下の通り。

- op:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
  - op が 10 のとき arithmetic,logical,save,restore 命令
  - op が 11 のとき memory instructions 命令
- rd: arithmetic,logical,LD,save,restore 命令ではデスティネーション・オペランドのレジスタ。結果を収める先。ST 命令ではソース・オペランドのレジスタ。
- op3:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
  - op3 が 000000 のとき ADD 命令
  - op3 が 000100 のとき SUB 命令
  - op3 が 010100 のとき SUBcc 命令
  - op3 が 000001 のとき AND 命令
  - op3 が 000010 のとき OR 命令
  - op3 が 000100 のとき ST 命令
  - op3 が 000000 のとき LD 命令
  - op3 が 111100 のとき SAVE 命令
  - op3 が 111101 のとき RESTORE 命令
- rs1:第 1 のソース・オペランドのレジスタ。

- i=1: arithmetic,logical,save,restore 命令では rs1 と simm13 を対象とする。memory instructions 命令ではメモリアドレス rs1+simm13。
- simm13: 定数。

3. branch 形式

   branch 形式の命令は図 2.15 の形式となる。

フィールド	op	a	cond	op2	disp22
ビット位置	31-30	29	28-25	24-22	21-0

図 2.15: branch 形式

各フィールドの意味は以下の通り。

- op:命令の基本操作。命令操作コード (オペコード) と呼ばれている。
  - op が 00 のとき branch 形式
- a:annul ビットと呼ばれ、a=1 のとき分岐条件不成立時にはディレイスロットを実行しない。
- cond: condition code と呼ばれている。
  - cond が 0001 のとき BE 命令
  - cond が 1001 のとき BNE 命令
  - cond が 0011 のとき BL 命令
- op2:Bicc(Branch on integer condition codes) のとき 010。
- disp22:定数。

2.3.4 SPARC プロセッサの基本設計

SPARC プロセッサの基本設計として、1 命令を 1 クロックサイクルで実行するシングルサイクル方式とし、キャッシュメモリは使用せずに、表 2.6 の命令を対象とする。icc(Integer Condition Codes) を構成する n,z,v,c は SUBcc 命令で保存され、次の branch 命令で使用される。

表 2.6: SPARC で設計した命令

	命令	命令内容	命令の種類
1	ADD	add	arithmetic
2	SUB	subtract	arithmetic
3	SUBcc	subtract and modify icc	arithmetic
4	AND	and	logical
5	OR	inclusive or	logical
6	ST	store word	memory instructions
7	LD	load word	memory instructions
8	BE	branch on equal	branch
9	BNE	branch on not equal	branch
10	BL	branch on less	branch
11	SAVE	save caller ' s window	save
12	RESTORE	restore caller ' s window	restore

また、レジスタウィンドウを含む場合 (レジスタウィンドウあり) と、含まない場合 (レジスタウィンドウなし) の 2 通りを設計した。以下にレジスタウィンドウの設計内容を示す。

レジスタウィンドウは 8 つのウィンドウを持ち、CWP(Current Window Pointer)=0∼7 で選択される。各ウィンドウは、8 個の global registers、8 個の out registers、8 個の local registers、8 個の in registers の計 32 個のレジスタで構成されている。

- global registers は CWP の状態に関わらずアクセスできる。
- out registers は CWP が 1 減少したとき、次のウィンドウの in registers になる。
- local registers は当該ウィンドウのみアクセスできる。
- in registers は CWP が 1 増加したとき、次のウィンドウの out registers になる。

RESTORE 命令は CWP を 1 増加させ、SAVE 命令は CWP を 1 減少させる。CWP=7 のときに CWP が 1 増加した場合は CWP=0 となる。CWP=0 のときに CWP が 1 減少した場合は CWP=7 となる。レジスタの総数は 16×8+8=136 個となる。

なお、レジスタウィンドウなしの設計では、32 個のレジスタのみを持ち、SAVE 命令と RESTORE 命令の使用は不可とする。

本研究では図 2.16, 図 2.17 のようなモジュールを設計し接続した。その際は SPARC International Inc. の The SPARC Architecture Manual Version 8 [25] を参考にした。

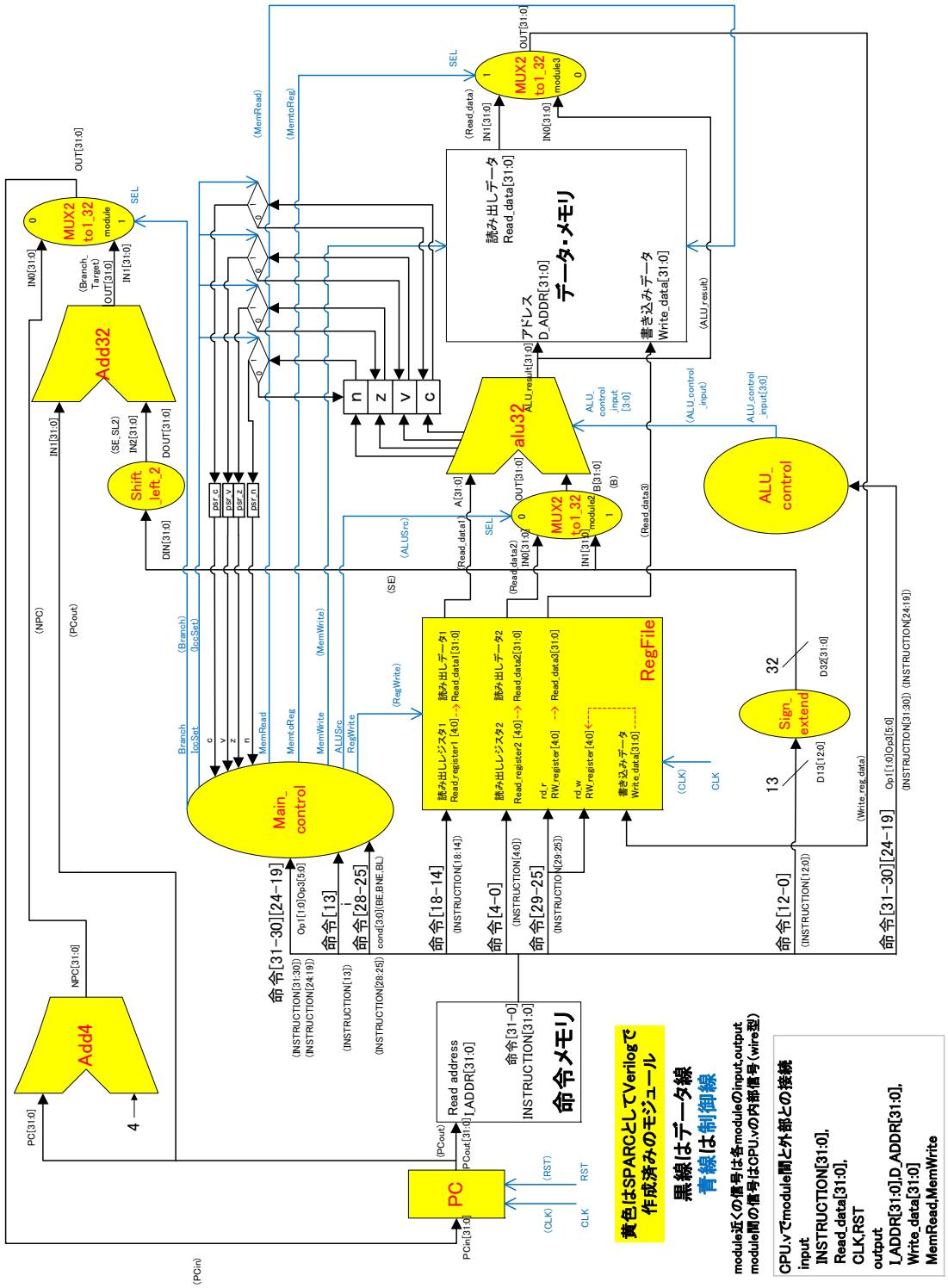

図 2.16: SPARC プロセッサのデータパスの図 (レジスタウィンドウなし)

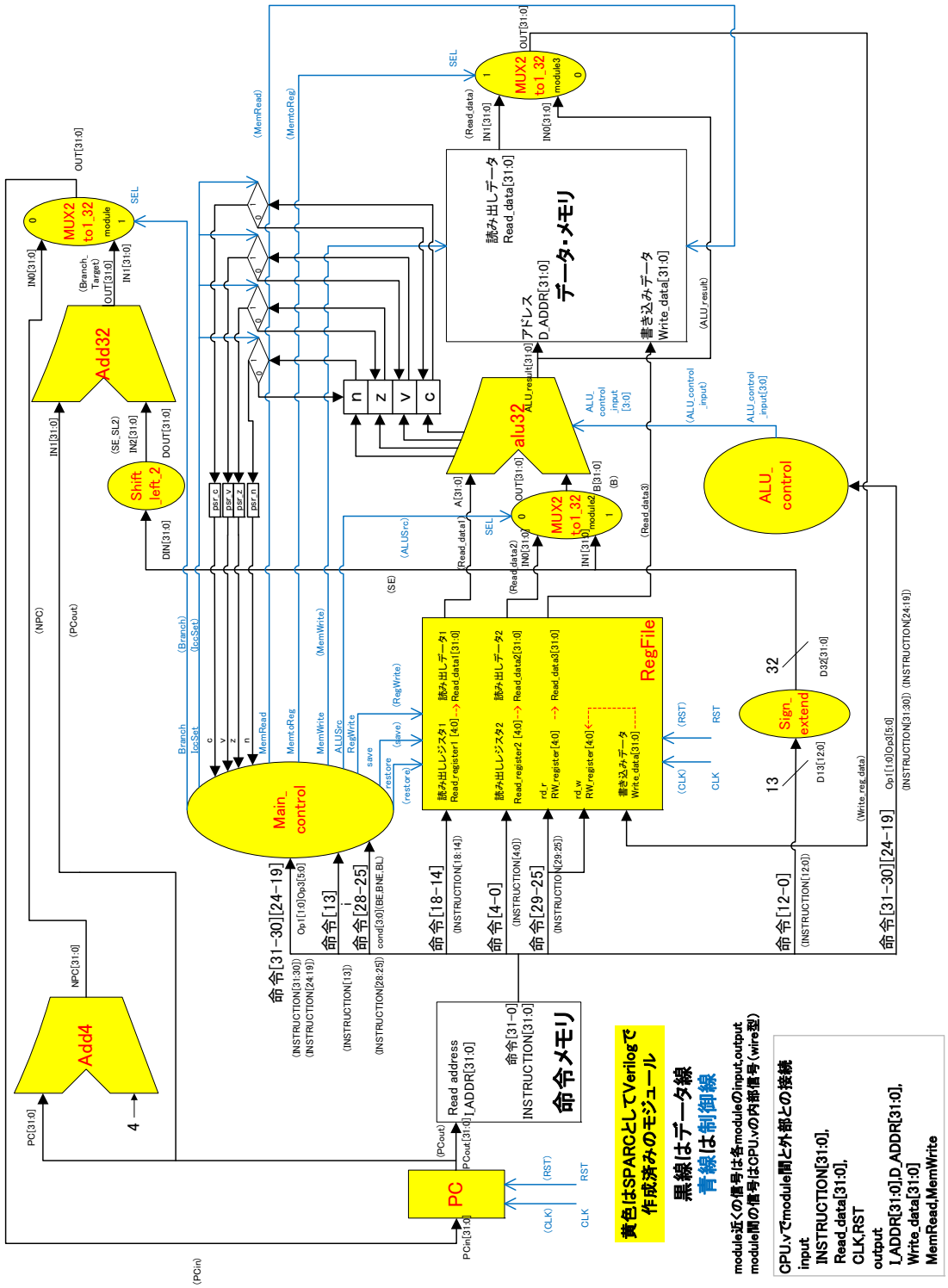

図 2.17: SPARC プロセッサのデータパスの図 (レジスタウィンドウあり)

図 2.16, 図 2.17 のモジュールは以下の通りである。

1. CPU

   モジュール間、モジュールと外部との接続をする。SUBcc 命令のとき psr_n, psr_z, psr_v, psr_c へ alu32 の出力 icc(n,z,v,c) を入力する。

2. PC

   プログラムカウンタの役割をする。

3. Add4

   次の命令アドレスのために、現在の命令アドレスに 4 を加算する。

4. Add32

   分岐先アドレス生成のために、「現在の命令アドレス」に「符号拡張後に 2 ビット左シフト」を加算する。

5. MUX2to1_32(module)

   分岐先アドレスと次の命令アドレスのいずれかを選択するマルチプレクサ。

6. Shift_left_2

   2 ビット左シフト。

7. Sign_extend

   13 ビットを 32 ビットに符号拡張する。

8. alu32

   add、sub、and、or の演算を実行する。演算結果から icc(n,z,v,c) を生成する。

9. MUX2to1_32(module2)

   alu32 の入力を選択するマルチプレクサ。レジスタの値と符号拡張後の値のいずれかを選択する。

10. RegFile

    レジスタの読み出しと書き込み。レジスタウィンドウの設定。

11. MUX2to1_32(module3)

    レジスタに書き込む値を選択するマルチプレクサ。データメモリから読み出した値と ALU 演算結果のいずれかを選択する。

12. Main_control

    命令と icc(n,z,v,c) から制御信号を生成する。

13. ALU_control

    命令 [31-30][24-19] から alu32 の制御信号を生成する。

2.3.5 シミュレーション

本研究でシミュレーションしたプログラムは表 2.7 の通りである。このプログラムでは 1+2+3+4+5+6+7+8+9=45 の計算を加算命令、ストア命令、減算命令、分岐命令、ロード命令を用いて実行する。

表 2.7: SPARC でシミュレーションしたプログラム

命令メモリアドレス	命令	説明
0:	add %r0, %r0, %r1	r1 に 0 をセット
4:	add %r0, 10, %r2	r2 に 10 をセット（ループ回数）
8:	add %r0, %r0, %r3	r3 に 0 をセット
12:	add %r0, %r0, %r4	r4 に 0 をセット
16:	st %r1, [%r4+0]	r4 番地に r1 を格納
20:	add %r4, 4, %r4	r4 内の番地の値を 4 増やす
24:	add %r1, 1, %r1	r1 の値を 1 増やす
28:	subcc %r1, %r2,%r0	r1-r2 の結果を icc(n,z,v,c) に保存する
32:	bne,a -4	ループ条件; 等しくないなら命令メモリアドレス 16 へ分岐
36:	add %r0, %r0, %r1	r1 に 0 をセット
40:	add %r0, %r0, %r4	r4 に 0 をセット
44:	ld [%r4+0], %r5	r4 番地から r5 に値の読み出し
48:	add %r3, %r5, %r3	読み出した値 (r5) を総和値 (r3) に加算
52:	add %r4, 4, %r4	r4 内の番地の値を 4 増やす
56:	add %r1, 1, %r1	r1 の値を 1 増やす
60:	subcc %r1, %r2, %r0	r1-r2 の結果を icc(n,z,v,c) に保存する
64:	bne,a -5	ループ条件; 等しくないなら命令メモリアドレス 44 へ分岐
68:	st %r3, [%r0+0]	総和値 (r3) を 0 番地に格納
72:	subcc %r0, %r0, %r0	r0-r0=0 の結果を icc(n,z,v,c) に保存する
76:	be,a 0	無限自己ループで修了

表 2.7 のプログラムを Xilinx, Inc. の ISim [14] でシミュレーションした結果の波形画像を図 2.18, 図 2.19 に示す。命令メモリアドレス (I_ADDR) が 68 のとき、データ・メモリアドレス (D_ADDR)0 に 45 を書き込んでいる (Write_data=45)。(Write 45 to address 0)

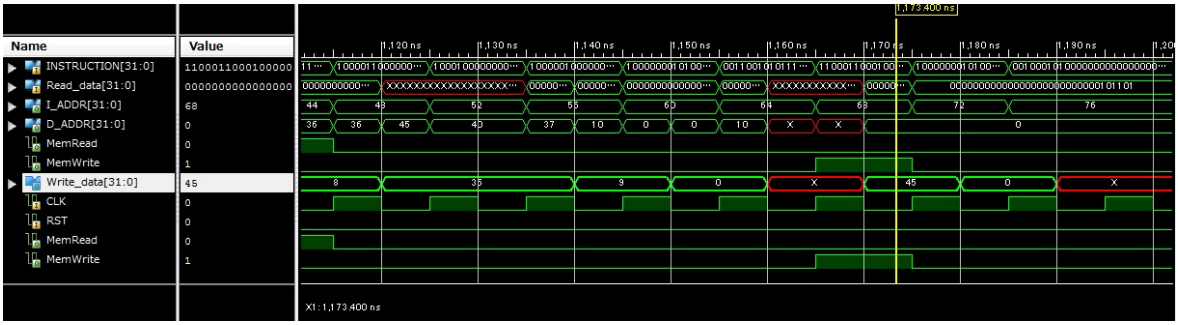

図 2.18: SPARC プロセッサでのシミュレーション結果 (レジスタウィンドウなし)

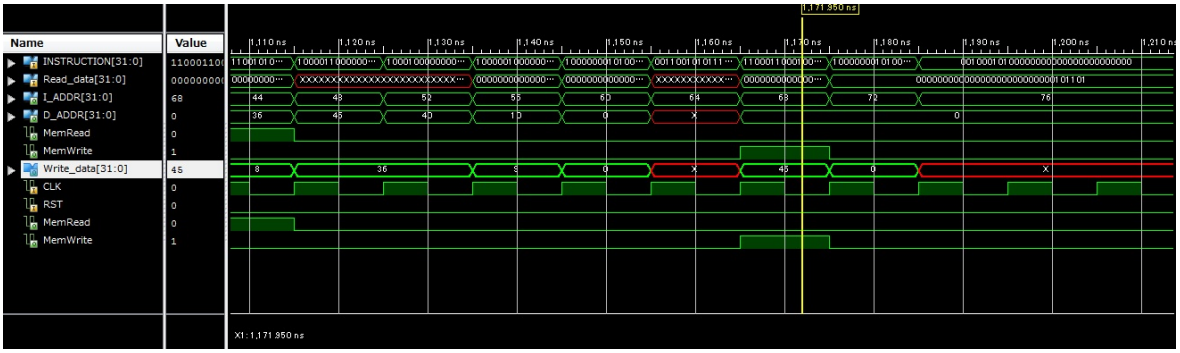

図 2.19: SPARC プロセッサでのシミュレーション結果 (レジスタウィンドウあり)

第3章　評価

3.1　ソースファイルの評価

各 ISA の図 2.4, 図 2.11, 図 2.16, 図 2.17 の各モジュールのソースファイルの行数とサイズを表 3.1 に示す。ここでソースファイルとは Xilinx, Inc. の ISE Design Suite 14.7 [33] で設計した Verilog モジュールファイル [34] で、拡張子は.v である。

表 3.1: Verilog モジュールファイルの行数とサイズ

		MIPS		ARM		SPARC(レジスタウィンドウなし)		SPARC(レジスタウィンドウあり)	
	モジュール名	行数	[バイト]	行数	[バイト]	行数	[バイト]	行数	[バイト]
1	CPU	142	2977	170	3664	154	3177	159	3290
2	PC	38	727	-	-	38	727	38	727
3	Add4	29	600	29	600	29	600	29	600
4	Add32	30	620	30	620	30	620	30	620
5	MUX2to1_32 (module)	36	730	36	730	36	730	36	730
6	Shift_left_2	29	624	29	624	29	624	29	624
7	Sign_extend	30	743	30	679	30	767	30	767
8	alu32	70	1509	126	3522	143	3249	143	3249
9	MUX2to1_32 (module2)	36	730	36	730	36	730	36	730
10	RegFile	55	1246	82	2035	63	1439	195	5541
11	MUX2to1_5	38	730	-	-	-	-	-	-
12	MUX2to1_32 (module3)	36	730	36	730	36	730	36	730
13	Main_control	111	2066	220	4365	105	2325	124	2763
14	ALU_control	59	1605	58	1357	70	1829	73	2014
	計	739	15637	882	19656	799	17547	958	22385

表 3.1 より以下のことが言える。

- より少ないソースコードで設計できるのは、MIPS、SPARC(レジスタウィンドウなし)、 ARM、SPARC(レジスタウィンドウあり) の順になる。

3.2 回路の評価

MIPS では表 2.2、ARM では表 2.5、SPARC では表 2.7 のプログラムに対応した回路を Xilinx, Inc. の ISE Design Suite 14.7 [33] で Spartan6 FPGA XC6SLX45 を対象として実装した。その回路について表 3.2 のような結果を得た。

表 3.2: 実装した回路の評価結果

	MIPS	ARM	SPARC (レジスタウィンドウなし)	SPARC (レジスタウィンドウあり)
Number of Slice Registers	30	158	97	3064
Number of Slice LUTs	292	888	297	2751
Number of RAMB8BWERs	2	0	1	0
Number of RAMB16BWERs	0	0	0	0
Critical Path[ns]	14.195	21.792	14.791	14.904
Maximum Frequency[MHz]	70	46	68	67

表 3.2 より以下のことが言える。

- 当該実装で、より少ない資源で実行できるのは、MIPS、SPARC(レジスタウィンドウなし)、 ARM、SPARC(レジスタウィンドウあり) の順になる。
- 当該実装で、より高い周波数で実行できるのは、MIPS、SPARC(レジスタウィンドウなし)、SPARC(レジスタウィンドウあり)、 ARM の順になる。
- 当該実装では MIPS が一番少ない資源で、一番高い周波数で実行できる。

また、ISE Design Suite 14.7 [33] の Verilog モジュールファイル [34] で設計したプロセッサの汎用レジスタ数を表 3.3 に示す。

表 3.3: 汎用レジスタ数

	MIPS	ARM	SPARC (レジスタウィンドウなし)	SPARC (レジスタウィンドウあり)
汎用レジスタ数	32	16	32	136

表 3.2、表 3.3 より SPARC(レジスタウィンドウあり) では以下のことが言える。

- レジスタウィンドウの影響により、表 3.3 のように SPARC(レジスタウィンドウあり) で汎用レジスタが増え、表 3.2 のように SPARC(レジスタウィンドウなし) よりも周波数が低くなった。

これは、レジスタウィンドウがあると、レジスタウィンドウがない場合に比べて、汎用レジスタが増え、ハードウェアサイズが大きくなることからクロック周期が長くなり、クロック周波数が低くなったためと考えられる。

3.3 ISA とモジュールの関係

各 ISA の図 2.4, 図 2.11, 図 2.16, 図 2.17 の各モジュールを共通コンポーネント、類似コンポーネント、特殊コンポーネントに分類する。ISA に非依存な共通コンポーネントは、複数の ISA でソースコードが同じモジュールで、異なる ISA のプロセッサを構成可能である。類似コンポーネントはソースコードの一部が違うモジュールで、一部を変更すれば、異なる ISA のプロセッサを構成可能である。特殊コンポーネントはソースコードが違う、または 1 つの ISA にしかないモジュールで、ISA に依存しており ISA の特殊性を示す。

表 3.4 に共通コンポーネント、類似コンポーネント、特殊コンポーネントを示す。

表 3.4: ISA とコンポーネントの関係

コンポーネントの種類		MIPS	ARM	SPARC(レジスタウィンドウなし、あり)
共通コンポーネント (複数の ISA でソースコードが同じモジュール)	1	PC	-	PC
	2	Add4	Add4	Add4
	3	Add32	Add32	Add32
	4	Shift_left_2	Shift_left_2	Shift_left_2
	5	MUX2to1_32 (module)	MUX2to1_32 (module)	MUX2to1_32 (module)
	6	MUX2to1_32 (module2)	MUX2to1_32 (module2)	MUX2to1_32 (module2)
	7	MUX2to1_32 (module3)	MUX2to1_32 (module3)	MUX2to1_32 (module3)
類似コンポーネント (ソースコードの一部が違うモジュール)	8	Sign_extend	Sign_extend	Sign_extend
特殊コンポーネント (ソースコードが違う、または 1 つの ISA にしかないモジュール)	9	alu32	alu32	alu32
	10	RegFile	RegFile	RegFile
	11	Main_control	Main_control	Main_control
	12	ALU_control	ALU_control	ALU_control
	13	CPU	CPU	CPU
	14	MUX2to1_5	-	-

また、類似コンポーネント、特殊コンポーネントのISA間での違いを各モジュール毎に以下に示す。

- 類似コンポーネント
  - Sign_extend; 符号拡張の量
- 特殊コンポーネント
  - alu32; シフトの有無、フラグ値（n, z, v, c）生成の有無
  - RegFile; プログラムカウンタの有無、読み出しデータ数の違い、レジスタウィンドウの有無
  - Main_control; フラグ値（n, z, v, c）入力の有無、SAVE命令・RESTORE命令の有無
  - ALU_control; 入出力の違い、SAVE命令・RESTORE命令の有無
  - CPU; 分岐先制御信号・フラグ値（n, z, v, c）の比較による制御の有無、SAVE命令・RESTORE命令の有無
  - MUX2to1_5;MIPSのみ使用、書き込みレジスタの選択をするマルチプレクサ。

類似コンポーネント、特殊コンポーネントのISA間での違いによるISAの特殊性は以下の通り。

- MIPS
  - Sign_extend; 符号拡張量16bit
  - alu32; シフト無し、フラグ値（n, v, c）の生成無し、フラグ値（z）生成有り
  - RegFile; プログラムカウンタ無し、読み出しデータ数2、書き込みデータ数1
  - Main_control; フラグ値（n, z, v, c）入力無し
  - ALU_control; Main_controlからの制御信号と命令[5-0](funct)からalu32の制御信号を生成。
  - CPU; モジュール間、モジュールと外部との接続をする。分岐先制御信号の生成。
  - MUX2to1_5;MIPSのみ使用、書き込みレジスタの選択をするマルチプレクサ。
- ARM
  - Sign_extend; 符号拡張の量8bit
  - alu32; シフト有り、フラグ値（n, z, c, v）の生成有り
  - RegFile; プログラムカウンタ有り、読み出しデータ数3、書き込みデータ数1

- Main_control; フラグ値（n, z, c, v）の入力有り
- ALU_control; 命令 [27-21] から alu32 の制御信号を生成。
- CPU; モジュール間、モジュールと外部との接続をする。

- SPARC(レジスタウィンドウなし)
  - Sign_extend; 符号拡張の量 19bit
  - alu32; シフト無し、フラグ値（n, z, v, c）の生成有り
  - RegFile; プログラムカウンタ無し、読み出しデータ数 3、書き込みデータ数 1
  - Main_control; フラグ値（n, z, v, c）の入力有り
  - ALU_control; 命令 [31-30][24-19] から alu32 の制御信号を生成する
  - CPU; モジュール間、モジュールと外部との接続をする。SUBcc 命令のとき psr_n,psr_z,psr_v,psr_c へ alu32 の出力フラグ値（n, z, v, c）を入力する。

- SPARC(レジスタウィンドウあり)
  - Sign_extend; 符号拡張の量 19bit
  - alu32; シフト無し、フラグ値（n, z, v, c）の生成有り
  - RegFile; プログラムカウンタ無し、読み出しデータ数 3、書き込みデータ数 1、レジスタウィンドウ有り
  - Main_control; フラグ値（n, z, v, c）の入力有り、SAVE 命令・RESTORE 命令有り
  - ALU_control; 命令 [31-30][24-19] から alu32 の制御信号を生成する。SAVE 命令・RESTORE 命令有り
  - CPU; モジュール間、モジュールと外部との接続をする。SUBcc 命令のとき psr_n,psr_z,psr_v,psr_c へ alu32 の出力フラグ値（n, z, v, c）を入力する。SAVE 命令・RESTORE 命令有り

共通コンポーネント、類似コンポーネント、特殊コンポーネントについて例を以下に示す。

- 共通コンポーネント
  - モジュール名; Add32
  - 機能; 分岐先アドレス生成のために、「現在の命令アドレス」に「符号拡張後に 2 ビット左シフト」を加算する。

- ソースコード

```
module Add32(
  input [31:0] IN1,
  input [31:0] IN2,
  output [31:0] OUT
  );
  assign OUT = IN1 + IN2;
endmodule
```

- 類似コンポーネント
  - モジュール名; Sign_extend
  - 機能; 符号拡張
  - 違い; MIPS; 符号拡張の量 16bit
    ARM; 符号拡張の量 8bit
    SPARC; 符号拡張の量 19bit
  - MIPS のソースコード

```
module Sign_extend(
  input [15:0] D16,
  output [31:0] D32
  );
  assign D32 =
   {D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],
    D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],D16[15],D16[15:0]};
endmodule
```

  - ARM のソースコード

```
module Sign_extend(
  input [23:0] D24,
  output [31:0] D32
  );
  assign D32 =
   {D24[23],D24[23],D24[23],D24[23],
    D24[23],D24[23],D24[23],D24[23],D24[23:0]};
endmodule
```

– SPARC のソースコード

```
module Sign_extend(
  input [12:0] D13,
  output [31:0] D32
  );
  assign D32 =
   {D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],
    D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],D13[12],
    D13[12],D13[12],D13[12],D13[12:0]};
endmodule
```

- 特殊コンポーネント
  - モジュール名; alu32
  - 機能; 演算の実行
  - 特徴; MIPS; シフト無し、フラグ値（n, v, c）の生成無し、
    フラグ値（z）生成有り
    ARM; シフト有り、フラグ値（n, z, c, v）の生成有り
    SPARC; シフト無し、フラグ値（n, z, v, c）の生成有り
  - MIPS での alu32 の機能; add、sub、and、or の演算を実行する。演算結果の Zero 信号の生成。
  - ARM での alu32 の機能; add、sub、and、論理左シフト、論理右シフト、算術右シフト、右ローテートの演算を実行する。演算結果からフラグ値 (n,z,c,v) を生成する。
  - SPARC での alu32 の機能; add、sub、and、or の演算を実行する。演算結果からフラグ値 (n,z,v,c) を生成する。

以上の共通コンポーネント、類似コンポーネント、特殊コンポーネントの選定と接続変更により、図 2.4, 図 2.11, 図 2.16, 図 2.17 の各 ISA のプロセッサが構成可能である。

第4章　まとめ

本研究では、ISA がマイクロアーキテクチャにどのように影響しているかを明らかにすることを目的として、FPGA をターゲットとし、複数の ISA のマイクロアーキテクチャを開発した。対象 ISA として MIPS、ARM 及び SPARC の命令セットを採用した。

MIPS、ARM、SPARC のそれぞれについてプロセッサを設計し、その実装した結果を比較した。その結果、本研究での実装では、より高い周波数で実行できたのは、MIPS、SPARC、ARM の順だった。また、MIPS が一番少ないソースコード及び資源で、一番高い周波数で実行できた。

また、マイクロアーキテクチャにおける ISA に非依存な共通コンポーネントと、類似コンポーネント、特殊コンポーネントを整理し、コンポーネントの選定と接続変更により異なる ISA のプロセッサが構成可能であることを示した。

参考文献

[1] MIPS Rx000 information:

http://www.cpu-collection.de/?tn=1&l0=cl&l1=MIPS%20Rx000 (2015 年 10 月 9 日閲覧)

[2] Biography:https://president.stanford.edu/biography/ (2015 年 10 月 9 日閲覧)

[3] MIPS:A Brief History:

http://alanclements.org/mips_history.html (2015 年 10 月 9 日閲覧)

[4] CPU コアベンダからの脱却:

http://news.mynavi.jp/articles/2009/03/05/mips_technologies/002.html (2015 年 12 月 6 日閲覧)

[5] HPC の歩み 50 年:http://www.hpcwire.jp/archives/8062 (2015 年 12 月 24 日閲覧)

[6] プロダクトガイド SGI Origin 3000 シリーズ:

https://www.sgi.co.jp/products/pdf/Origin3000_product_guide.pdf (2015 年 12 月 24 日閲覧)

[7] 米 SGI、MIPS/IRIX 搭載マシンの販売打ち切りへ:

http://japan.cnet.com/news/ent/20228827/ (2015 年 12 月 24 日閲覧)

[8] MIPS、新プロセッサコア「Aptiv」ファミリーを発表:

http://pc.watch.impress.co.jp/docs/news/20120525_535532.html (2015 年 12 月 24 日閲覧)

[9] データシート Cisco10000 シリーズ Performance Routing Engine:

http://www.cisco.com/web/JP/product/hs/routers/c10000/prodlit/prezo_ds.html (2015 年 12 月 24 日閲覧)

[10] HP 社、PMC-Sierra 社の高性能低消費電力 64 ビット MIPS-based プロセッサ RM7000C を新しいカラープリンタ 9500 シリーズに採用:

http://media.corporate-ir.net/media_files/nsd/pmcs/press/japan/printer_uses_RM7000C_PR_Japanese.pdf (2015 年 12 月 24 日閲覧)

[11] ミップス・テクノロジーズ MIPS アーキテクチャ向け Android のソースコードを公開:

http://japan.zdnet.com/article/20397947/ (2015 年 12 月 24 日閲覧)

[12] ポスト PC 時代のキーワード「エンベデッド」のすべて:

http://www.kumikomi.net/archives/2001/05/03postpc.php?page=16 (2015 年 12 月 24 日閲覧)

[13] David A. Patterson,John L. Hennessy:コンピュータの構成と設計 第 4 版 上, 下 日経 BP 社,2011,ISBN978-4-8222-8478-7, ISBN978-4-8222-8479-4

[14] ISE Simulator (ISim):

http://japan.xilinx.com/products/design-tools/isim.html (2016 年 1 月 26 日閲覧)

[15] An Introduction to the ARM System Architecture:

https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/ (2016 年 1 月 4 日閲覧)

[16] ARM の歴史:

http://www.aps-web.jp/academy/cortex-m/01/a.html (2016 年 1 月 4 日閲覧)

[17] Interface 編集部:ARM プロセッサ入門 CQ 出版社,2003,ISBN978-4-7898-3329-5

[18] ARM の沿革:

https://www.arm.com/ja/about/company-profile/milestones.php (2016 年 1 月 4 日閲覧)

[19] ARM Locations Around The World:

http://www.arm.com/ja/about/company-profile/locations-around-the-world.php (2016 年 1 月 4 日閲覧)

[20] ARM のロードマップ:

http://www.aps-web.jp/academy/cortex-m/01/b.html (2016 年 1 月 4 日閲覧)

[21] プロセッサ:

https://www.arm.com/ja/products/processors/index.php (2016 年 1 月 4 日閲覧)

[22] ARM が初の 64 ビット CPU「Cortex-A50 シリーズ」発表、サーバー向けに 16 コア以上に対応:

http://itpro.nikkeibp.co.jp/article/NEWS/20121101/434442/?rt=nocnt (2016 年 1 月 4 日閲覧)

[23] 会社概要:

https://www.arm.com/ja/about/company-profile/index.php (2016 年 1 月 4 日閲覧)

[24] ARM アーキテクチャリファレンスマニュアル v6 (ARM DDI 0100HJ-00), ARM Ltd. 1996-1998, 2000, 2004, 2005

[25] The SPARC Architecture Manual Version 8 Revision SAV080SI9308,SPARC International Inc.1991,1992

[26] SPARC の歴史:

http://sparc.org/sparc%e3%81%ae%e6%ad%b4%e5%8f%b2/?lang=ja (2016 年 1 月 12 日閲覧)

[27] FAQ:http://sparc.org/1859-2/?lang=ja (2016 年 1 月 12 日閲覧)

[28] スーパーコンピュータ「京」:

http://www.fujitsu.com/jp/about/businesspolicy/tech/k/ (2016 年 1 月 12 日閲覧)

[29] Oracle の Sun 買収、ついに決着:

http://itpro.nikkeibp.co.jp/article/COLUMN/20100128/343903/ (2016 年 1 月 12 日閲覧)

[30] オラクルのサン買収で、オープンシステム時代は終わる？:

http://itpro.nikkeibp.co.jp/article/COLUMN/20090423/329032/?ST=ittrend (2016 年 1 月 12 日閲覧)

[31] SPARC サーバー:

http://www.oracle.com/jp/servers/sparc/overview/index.html (2016 年 1 月 12 日閲覧)

[32] UNIX サーバ SPARC M10:

http://www.fujitsu.com/jp/products/computing/servers/unix/sparc/ (2016 年 1 月 12 日閲覧)

[33] ISE Design Suite 14 : リリースノート、インストールおよびライセンス:

http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx14_7/irn.pdf (2016 年 1 月 26 日閲覧)

[34] ソースファイルのタイプ:

http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx12_1/ise_r_source_types.htm (2016 年 1 月 26 日閲覧)

謝辞

本研究を進めるにあたり、丁寧なご指導をいただきました、田中清史准教授に心より感謝致します。また、課題研究計画提案発表で適切なご指導をしていただきました、金子峰雄教授、井口寧教授、副テーマのご指導をいただきました前園涼准教授に感謝いたします。