

計算機科学実験及演習 3 SIMPLE 設計資料

ver 2.3: 2008.4.2

1 はじめに

本資料は、計算機科学実験及演習 3 の課題の一つである、コンピュータのハードウェア設計に関するものである。設計するコンピュータ SIMPLE (SIxteen-bit MicroProcessor for Laboratory Experiment) は、1 語が 16 ビットで命令セットも簡単なものだが、コンピュータが持つべき基本機能は一通り備えている。従って SIMPLE の設計を正しく行なうことが、より高度なコンピュータの設計へ進む大きな第一歩となるだろう。

本資料では、まず SIMPLE の基本的なアーキテクチャ、即ちどのようなレジスタやメモリがあって、どのような形式と機能を持つ命令が実行されるかを示す。次に、このアーキテクチャに基づき、最も単純に設計する方法のごく一部を示す。

また、単に与えられた仕様と方針に沿って設計するだけではなく、独自の改良を施すことも実験課題とされている。そこで参考となるように、命令セット・アーキテクチャやマイクロアーキテクチャの改良の例を最後に示す。

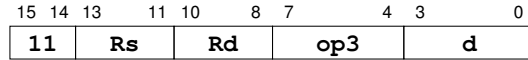
2 SIMPLE のアーキテクチャ

レジスタやメモリなどコンピュータが持つ様々な資源と、それを操作する命令の機能を定義したものを、アーキテクチャと呼ぶ。アーキテクチャは原則として論理的な構造や機能を定めるもので、それをどのように実現するかを定義するものではない。従って一つのアーキテクチャに基づいて、色々なハードウェアが設計されるのが普通である。但しどのような設計を行なっても、プログラムの動作が同じであることが重要なポイントである。なお、ハードウェアの設計の詳細に関するアーキテクチャに対しては、マイクロアーキテクチャという名前が使われる。

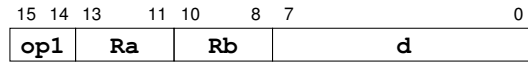
2.1 主記憶とレジスタ

SIMPLE は 1 語 16 ビットのコンピュータであり、主記憶やレジスタはいずれも 16 ビット幅である。

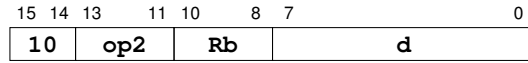
- 主記憶 (main memory)
主記憶のアドレスは 16 ビットであり、語単位にアドレスが付けられる。従ってアドレス空間の大きさは 64KW である。以下、アドレスが a の語のアクセスは、(C 言語風に) $*(a)$ と表記する。
- 汎用レジスタ (register)
8 個の汎用レジスタが備えられ、 $r[0], r[1], \dots, r[7]$ と表記される。演算のソース/デスティネーションや主記憶のアドレス計算に用いられる。
- プログラム・カウンタ (program counter)
実行中の命令のアドレスを保持する。PC と表記する。
- 条件コード (condition codes)
演算命令の結果に基づく分岐条件を保持する 3 個のフラグ S(sign), Z(zero), C(carry), V(overflow) からなり、以下のように設定される。



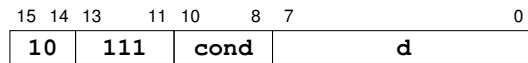
(a) 演算／入出力命令形式



(b) ロード／ストア命令形式



(c) 即値ロード／無条件分岐命令形式



(d) 条件分岐命令形式

図1 SIMPLE の命令形式

S: 負ならば1, そうでなければ0

Z: ゼロならば1, そうでなければ0

C: 桁上げがあれば1, そうでなければ0

V: 演算結果が符号付き16ビットで表せる範囲を越えた場合1, そうでなければ0

なお, 加減算と比較命令以外でのCの値は後に述べる.

2.2 命令セット・アーキテクチャ (instruction set architecture)

■命令形式

SIMPLE の命令は全て1語(16ビット)の固定長であり, 図2.1に示すように4種類の命令形式がある. 各命令形式とフィールドの意味は以下の通りである.

(a) 演算／入出力命令形式

- $I_{15:14}$ (op1) 操作コード(11) (operation code, opcode)
- $I_{13:11}$ (Rs) ソース・レジスタ番号
- $I_{10:8}$ (Rd) デスティネーション・レジスタ番号
- $I_{7:4}$ (op3) 操作コード(0000~1111)
- $I_{3:0}$ (d) シフト桁数

(b) ロード／ストア命令形式

- $I_{15:14}$ (op1) 操作コード(00/01)
- $I_{13:11}$ (Ra) ソース／デスティネーションのレジスタ番号
- $I_{10:8}$ (Rb) ベース・レジスタ番号
- $I_{7:0}$ (d) 変位(displacement)

(c) 即値ロード／無条件分岐命令形式

- $I_{15:14}$ (op1) 操作コード (10)
- $I_{13:11}$ (op2) 操作コード (000 ~ 110)
- $I_{10:8}$ (Rb) ソース／デスティネーション／ベースのレジスタ番号
- $I_{7:0}$ (d) 即値または変位

(d) 条件分岐命令形式

- $I_{15:14}$ (op1) 操作コード (10)
- $I_{13:11}$ (op2) 操作コード (111)
- $I_{10:8}$ (cond) 分岐条件
- $I_{7:0}$ (d) 変位

■演算／入出力命令

SIMPLE の演算／入出力命令を表 1 に示す。演算命令では結果に基づく条件コードが設定される。

1. 算術演算

レジスタ Rd と Rs の加算 (ADD: add) または減算 (SUB: subtract) の結果を Rd に格納し、条件コードを設定する。条件コード C には最上位ビットからの桁上げが設定される。

2. 論理演算

レジスタ Rs と Rs の、ビットごとの論理積 (AND: and)、論理和 (OR: or)、または排他的論理和 (XOR: exclusive-or) の結果を Rd に格納し、条件コードを設定する。但し条件コード C は演算結果に関わらず 0 となる。

3. 比較演算 (CMP: compare)

レジスタ Rd から Rs を減算し、結果に基づく条件コード設定のみを行なう。条件コード C には最上位ビットからの桁上げが設定される。

4. 移動演算 (MOV: move)

レジスタ Rd に Rs の値を単に格納し、Rd の値に基づき条件コードを設定する。但し条件コード C は Rs の値に関わらず 0 となる。

5. シフト演算

レジスタ Rd の値を、以下のようにシフトした値を Rd に格納し、条件コードを設定する。

- SLL (shift left logical) 左論理シフト。左シフト後、空いた部分に 0 を入れる。
- SLR (shift left rotate) 左循環シフト。左シフト後、空いた部分にシフト・アウトされたビット列を入れる。
- SRL (shift right logical) 右論理シフト。右シフト後、空いた部分に 0 を入れる。
- SRA (shift right arithmetic) ... 右算術シフト。右シフト後、空いた部分に符号ビットの値を入れる。

シフト桁数は即値 d (0 ~ 15) である。また条件コード C には、シフト桁数が 0 の時または SLR では 0 が、それ以外では最後にシフト・アウトされたビットの値が設定される。条件コード V は常に 0 が設定される。

6. 入出力命令

- IN (input) スイッチなどの機器から入力した値をレジスタ Rd に格納する。

- OUT (output) ... レジスタ Rs の値を 7SEG LED などの機器に出力する.
- HLT (halt) SIMPLE を停止させる.

なお“(reserved)”と記された命令は何の動作もせず、単に次の命令に移行する.*1

表 1 SIMPLE の演算/入出力命令

		15	14	13	11	10	8	7	4	3	0
		11	Rs	Rd	op3			d			
mnemonic		op3		function							
ADD	Rd, Rs	0000		r[Rd] = r[Rd] + r[Rs]							
SUB	Rd, Rs	0001		r[Rd] = r[Rd] - r[Rs]							
AND	Rd, Rs	0010		r[Rd] = r[Rd] & r[Rs]							
OR	Rd, Rs	0011		r[Rd] = r[Rd] r[Rs]							
XOR	Rd, Rs	0100		r[Rd] = r[Rd] ^ r[Rs]							
CMP	Rd, Rs	0101		r[Rd] - r[Rs]							
MOV	Rd, Rs	0110		r[Rd] = r[Rs]							
	(reserved)	0111									
SLL	Rd, d	1000		r[Rd] = shift_left_logical(r[Rd], d)							
SLR	Rd, d	1001		r[Rd] = shift_left_rotate(r[Rd], d)							
SRL	Rd, d	1010		r[Rd] = shift_right_logical(r[Rd], d)							
SRA	Rd, d	1011		r[Rd] = shift_right_arithmetic(r[Rd], d)							
IN	Rd	1100		r[Rd] = input							
OUT	Rs	1101		output = r[Rs]							
	(reserved)	1110									
HLT		1111		halt()							

■ロード/ストア命令

SIMPLE のロード命令 (LD: load) とストア命令 (ST: store) の機能を表 2 に示す. ソース/デスティネーションは, フィールド Ra で指定されたレジスタ Ra である. また実効アドレスはベース・レジスタ・アドレス指定により, フィールド Rb で指定されたレジスタ Rb と, フィールド d を符号拡張した `sign_ext(d)` を加算して求める.

■即値ロード/無条件分岐命令

SIMPLE の即値ロード命令 (LI: load immediate) と, 無条件分岐命令 (B: branch) の機能を表 3 に示す.

- LI 即値 `sign_ext(d)` をレジスタ Rb に格納する.
- B d を符号拡張した値を変位として, PC 相対アドレス指定による分岐を行なう.

■条件分岐命令

*1 「未定義命令なので動作は保証しない」というポリシーのもとに, ドント・ケアにしてもかまわない.

表 2 SIMPLE のロード/ストア命令

15 14 13 11 10 8 7 0			
op1	Ra	Rb	d

mnemonic	op1	function
LD Ra,d(Rb)	00	$r[Ra] = *(r[Rb] + \text{sign_ext}(d))$
ST Ra,d(Rb)	01	$*(r[Rb] + \text{sign_ext}(d)) = r[Ra]$

表 3 SIMPLE の即値ロード/無条件分岐命令

15 14 13 11 10 8 7 0			
10	op2	Rb	d

mnemonic	op2	function
LI Rb,d	000	$r[Rb] = \text{sign_ext}(d)$
(reserved)	001	
(reserved)	010	
(reserved)	011	
B Rb,d	100	$PC = PC + 1 + \text{sign_ext}(d)$
(reserved)	101	
(reserved)	110	
(条件分岐命令)	111	表 4 参照

SIMPLE の条件分岐命令は表 4 に示すように、フィールド `cond` で定められる分岐条件が成り立てば PC 相対アドレスによる分岐を行ない、成り立たなければ単に次の命令に移行する。各命令の分岐条件は以下の通り。

- BE (branch on equal-to) 条件コード Z が 1
- BLT (branch on less-than) 条件コード S と V の XOR(S ^ V) が 1
- BLE (branch on less-than or equal-to) Z または (S ^ V) が 1
- BNE (branch on not-equal-to) 条件コード Z が 0

3 基本的な設計

前述のように、一つのアーキテクチャに基づいて様々なハードウェアを設計できる。本節では SIMPLE の CPU を 5 つのフェーズに分割したハードウェア SIMPLE/B を、基本的な設計例として示す。なお、後に示すような改良の余地がいくつもあり、SIMPLE/B に因われることなく、十分に学んだ後で各自の創意により設計を改良することを強く期待する。

SIMPLE/B は図 2 に示すように、順次活性化する p1 ~ p5 の 5 つのフェーズ、各フェーズに制御信号を供給する制御回路、主記憶、及びスイッチ/LED / 7SEG LED からなる入出力機器から構成される。なお図中の太線はバスまたはセレクタを表す。

表 4 SIMPLE の条件分岐命令

		15 14 13	11 10	8 7	0
		10	111	cond	d
mnemonic	d	cond	function		
BE	d	000	if (Z) PC = PC + 1 + sign_ext(d)		
BLT	d	001	if (S ^ V) PC = PC + 1 + sign_ext(d)		
BLE	d	010	if (Z (S ^ V)) PC = PC + 1 + sign_ext(d)		
BNE	d	011	if (!Z) PC = PC + 1 + sign_ext(d)		
(reserved)		100			
(reserved)		101			
(reserved)		110			
(reserved)		111			

3.1 制御回路

SIMPLE/B には以下の信号が外部から供給される。

1. clock

動作クロック。適切な発振回路を用いて、Hi/Lo の期間ができるだけ 1:1 に近いクロックを供給する。制御回路は図 3 に示すように、クロックの立ち上がりエッジに同期して各フェーズを一つずつ順番に活性化する。従って、クロックの周期は遅延が最大であるようなフェーズの遅延時間により定まり、多くの場合は主記憶のアクセス時間に周辺回路の遅延時間などを加えた値となる。

2. reset

リセット信号。プッシュ・スイッチを用いて、スイッチを押すと 1 が、離すと 0 が供給されるようにする*2。また電源投入時にも 1 になるようにするのが望ましい。reset が 1 になると SIMPLE/B は PC 等をクリアし、適切な初期状態に移行する。

3. exec

起動/停止信号。プッシュ・スイッチを用いて、スイッチを押すと 1 が、離すと 0 が供給されるようにする*2。SIMPLE/B が停止状態にある時に exec が 0 から 1 に変化すると、SIMPLE/B は命令の実行を開始する。SIMPLE/B が実行状態にある時に exec が 0 から 1 に変化すると、その時点で実行中の命令を完了してから停止する。

3.2 フェーズ p1 ~ p5

SIMPLE/B は一つの命令を、p1 ~ p5 の 5 つのフェーズを順番に活性化しながら実行する。

1. p1 (命令フェッチ)

*2 [reset] や [exec] は負論理にしてもかまわない。

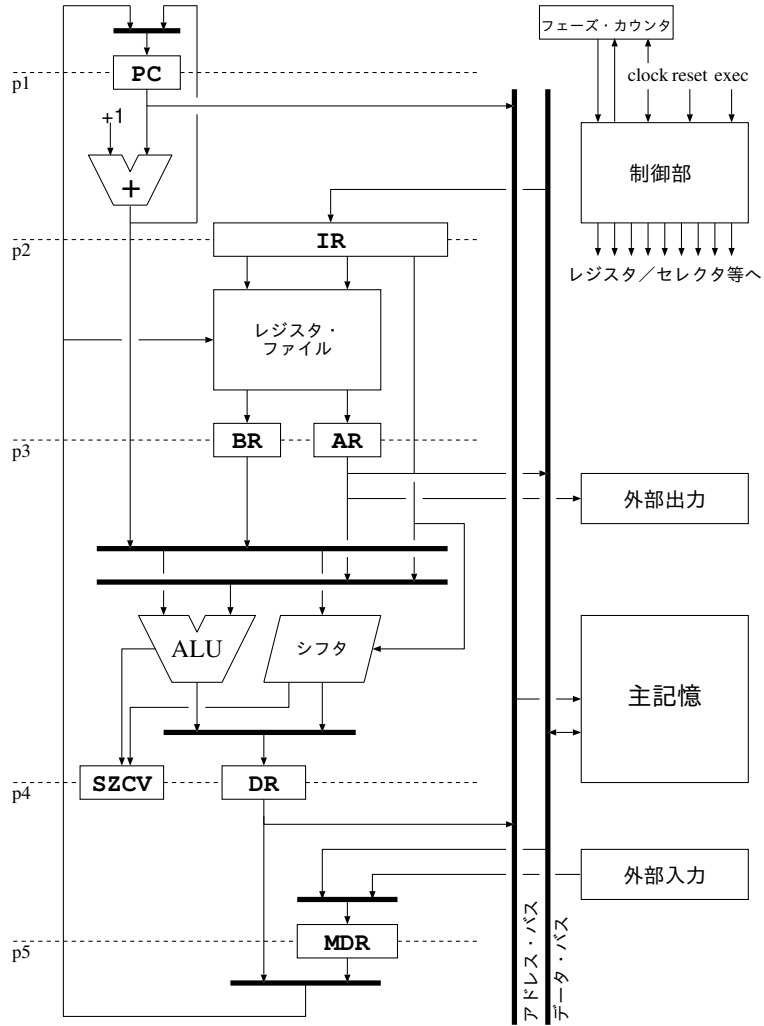


図2 SIMPLE/Bのブロック図

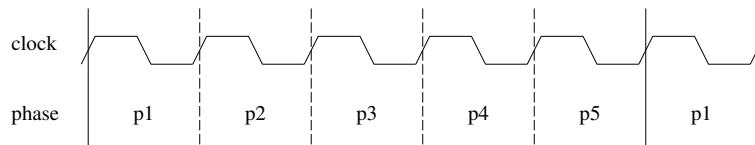


図3 SIMPLE/Bのフェーズ

PC(Program Counter) が保持するアドレスの命令を主記憶からフェッチし、IR(Instruction Register) に格納するとともに、PC に 1 を加える。

2. p2 (レジスタ読み出し)

IR が保持する命令の、Ra/Rs フィールドと Rb/Rd フィールドで指定される汎用レジスタの値を読み出し、それぞれレジスタ AR と BR に格納する。

3. p3 (演算)

ALU またはシフト回路により、命令が定める演算 (アドレス計算や単なるデータ移動を含む) を行ない、その結果をレジスタ DR(Data Register) に格納する。また演算命令では条件コード S, Z, C をセットする。演算のソースには AR, BR の他、PC や命令の即値が用いられることがある。

4. p4 (主記憶アクセス)

ロード/ストア命令では、DR の値をアドレスとして主記憶をアクセスする。ロード命令では読み出した値をレジスタ MDR(Memory Data Register) に格納し、ストア命令では AR が保持する値を書き込む。また入出力命令では、入力スイッチの値の MDR への格納や、AR の値の 7SEG LED などへの表示を行う。

5. p5 (レジスタ書き込み)

汎用レジスタの書き込みを伴う命令では、DR または MDR の値を、命令の Ra,Rb, または Rs フィールドで指定される汎用レジスタに書き込む。また分岐命令では DR の値を分岐先アドレスとして PC に書き込む。

3.3 主記憶と入出力機器

主記憶の容量は SIMPLE のアドレス空間の大きさと同じ 64KW とする。またアドレス変換などは行なわず、命令で定められるアドレスをそのまま主記憶のアドレスとする。この主記憶は、FPGA 上に搭載されている RAM にて構成する。ただし、SIMPLE の実装に使う FPGA によっては、FPGA 上に搭載されている RAM のサイズの制限により、主記憶の容量を 64KW 取れないことがありうる。(PowerMedusa EC6S ボード上の EP1C6Q240C8 では、4KW が最大値となる。)

なお、主記憶の読み出し/書き込みアドレスは、PC または DR の値を選択したものをアドレス・バスに出力することで指定する。主記憶の読み出しデータは、主記憶からデータ・バスに出力される。これを、必要に応じて、IR もしくは MDR に取り込む。主記憶への書き込みデータは AR よりデータ・バスに出力され、主記憶へ書き込み許可 (write enable) 信号とクロック信号を 1 にすることで主記憶に書き込まれる。

PowerMedusa EC6S ボードには以下の入出力機器が存在する。以下の入力機器から各自が使いやすいものを選択し、IN 命令および OUT 命令の入出力対象とする。

- 入力
 - テンキー：20 個
 - ロータリースイッチ：2 個
- 出力
 - LED：8 個
 - 7SEG LED：8 個
 - ブザー

なお、ボード上にダウンロードしたプロセッサをデバッグをする場合、プロセッサ内部の信号線の状態を観察したり、信号線に任意の値を入れたい場合がある。その場合、IN 命令や OUT 命令のための入出力もデバッグ用の入出力に使うなど、柔軟な運用をすること。

4 命令セット・アーキテクチャの改良

2 節で述べた SIMPLE の基本アーキテクチャは、とりあえず動作するプロセッサを作りやすいように、非常に単純なものとなっている。本節と次節では、課題とされているアーキテクチャの改良のため、拡張のヒントを記す。

本節は、以下の命令セット・アーキテクチャの拡張に関するヒントを記す。

1. 即値オペランドの強化
2. 入出力命令の強化
3. Branch Register と Branch And Link 命令の追加
4. 条件分岐の 1 命令化
5. ボードからの入力による割り込みのサポート
6. 複合演算命令の追加
7. 条件付き演算命令の追加

なお、ここで示すアーキテクチャはあくまで一例であるので、独自の改良を行なって構わないし、またそれを期待している。また、基本の命令セットに追加する形だと、拡張時に使用するビットが不足するような大規模な拡張を行う場合、基本の命令セットからいくつかの命令を削除したうえで、その拡張を行うのもありとする。

4.1 即値オペランドの強化

基本アーキテクチャでは、演算命令のオペランドはいずれも汎用レジスタであり、例えばレジスタ $r[0]$ に 1 を加えるには、

```
LI    R1, 1
ADD   R0, R1
```

のように 2 命令を要する。そこで、表 1 の演算命令のうち、 d フィールドを使わないシフト以外の演算命令に対し、レジスタ Rd と d フィールドで指定した即値を演算させるという拡張が考えられる。

以下、ヒントを列記する。

- $r[Rd] + r[Rs]$ と $r[Rd] + \text{sign_ext}(d)$ の切り替えを考慮して命令セットを拡張しなくてはならない。 $(d$ 中の 1 ビットをフラグに用いるとか)
- もちろん、演算器を追加した上で、 $r[Rd] + r[Rs] + \text{sign_ext}(d)$ という演算を行わせるという拡張方法もありうる

4.2 入出力命令の強化

基本アーキテクチャでは、ボードからの入力として IN 命令、ボードへの出力として OUT 命令がある。基本アーキテクチャでは、これらの入出力命令の入出力対象は特定の 16 ビットの入力／出力に固定されている。しかし、PowerMedusa EC6S ボードには入力／出力対象ともに 16 ビットを越える数があり、入力／出力ともそれらの中から選択して使える方がプログラムを作成する側にとってより嬉しい。そこで、IN 命令ではフィールド Rs と d, OUT 命令ではフィールド Rd と d が未使用なことを利用し、そのフィールドを利用して入出力先を変更する拡張が考えられる。

以下、ヒントを列記する。

- 8 桁 7SEG LED は 32 ビット分のデータを表示できる。フィールドのうちの 1 ビットを、8 桁の 7SEG LED の上位／下位の切り替えに使うのが良いだろう。
- 7SEG ボードの全ての出力を利用する場合、8 つの 8 桁 7SEG LED と 64 個の LED を選択するのに、最低でも 5 ビットのフィールドが必要となる。そのため、Rd フィールドと d フィールドの両方を使う必要がある。
- レジスタ・ファイルは 2 つのレジスタ値を同時に読み出せるので、1 つの OUT 命令で 2 つのレジスタ値を出力するように拡張するのも面白いだろう。

4.3 Branch Register と Branch And Link 命令の追加

一般的なコンパイラでは、関数呼び出しに対して、以下のような動作をするコードを出力する。

1. 関数を呼び出した命令の次の PC をレジスタに格納し、関数の先頭にジャンプする。
2. 関数を実行する。
3. 1. でレジスタに格納された値を PC に書き込み、関数を呼び出した命令の次の命令からの実行を開始する。

上記のように、わざわざレジスタに戻り先のアドレスを格納するのは、関数を呼び出す場所が 1 箇所とは限らないからである。

通常のプロセッサでは、関数呼び出しをサポートするため、上記の 1. の動作を行う Branch And Link(BAL) 命令と、3. の動作を行う Branch Register(BR) 命令が存在する。これらの実装という拡張を行えば、任意の場所から呼び出せる関数を実装できるようになる。プロセッサの上で実用的なプログラミングを行うためには、ぜひ実装しておきたい命令である。

以下、ヒントを列記する。

- PC をレジスタに退避するために新たなデータ・パスが必要となる。
- BAL で PC を退避するレジスタや、BR で読み出すレジスタを自由に指定するという実装もあるが、これを固定してしまうという実装もある。ちなみに、MIPS アーキテクチャでは、BAL に相当する命令が PC を退避する先はレジスタ 31 番に固定されている。このように、退避するレジスタを固定することの得失について考察するのも面白いだろう。
- 呼び出された関数からさらに別の関数を呼び出すプログラムを書く場合、別の関数の呼び出しを行う前

に、現在の関数からの戻り先を格納したレジスタの内容を主記憶等に退避する必要がある。また、それ以外のレジスタ値に対しても、必要に応じて主記憶等に退避する。(詳細はコンパイラの実験を参照)

- 上記のような階層化された関数呼び出しのうち、階層がそれほど深くないものの呼び出し／復帰を高速化するために、戻り先のアドレスやレジスタを退避しておくスタックをプロセッサに追加し、主記憶への退避を減らすのも面白いかもしれない。

4.4 条件分岐の1命令化

基本アーキテクチャでは、条件分岐は、演算の結果によって設定される条件コードを利用している。この構成では、表4に示される命令のビット列の中のdフィールドを大きくすることにより、分岐先をより広い範囲に設定することが可能となる。しかし、この構成では基本的に、条件分岐を行う前に比較命令を実行しなくてはならない。そこで、比較と分岐を1命令で行わせることを考える。当然、命令のビット列の中に、比較する2つのレジスタを指定する部分を準備しなくてはならないため、dフィールドに使えるビット数は少なくなる。そのため、分岐先の範囲が狭くなるが、基本アーキテクチャの条件分岐命令も残しておけば問題ないだろう。以下、ヒントを列記する。

- 表4のフォーマットではレジスタを2つ指定できるほどビットに余裕がないので、表3の”(reserved)”の部分を使うと良いだろう。
- 基本アーキテクチャもちゃんと考えられて作られたアーキテクチャであり、プログラムの作り方によっては、比較命令を排除して条件分岐を実現できる。そのあたりを考察するのも面白いだろう。
- 表4にも”(reserved)”の部分がある。これを使って、条件分岐命令の分岐条件を増やすのも良いだろう。

4.5 ボードからの入力による割り込みのサポート

基本アーキテクチャでは、ボードからの入力による割り込み処理のサポートをしていない。「ボード上のあるボタンが押されたら特定ルーチンを実行する」ような処理ができない*3。そこで、ボードからの入力によって割り込みを発生させ、ボードからの入力等をトリガとして特定の処理を開始することができるようにするという拡張を行うことが考えられる。

基本的に、割り込みが発生した場合、以下の流れで「現在の処理の中断→割り込み処理の実行→元の処理の再開」を行う。

1. 現在の実行中の命令を完了し、レジスタ値と条件コードを更新し、次に実行する命令のPCを確定する。
2. レジスタ・ファイル内の全ての値、条件コード、PCの値をメモリに退避する。
3. PCに割り込み処理の命令列の最初のメモリ・アドレスを渡す。
4. PCに従って命令を読み出し、割り込み処理を行う。
5. 割り込み処理が終了したら、2.で退避したレジスタ値、条件コード、PCの値を書き戻す。

*3 IN命令のみで実現できなくもないが、IN命令は、IN命令が実行された瞬間の値しか読めないで、ボタンが押されている間にIN命令が実行されないといけない。よって、IN命令のみでは、ステップ実行を行ったりループで常に入力を監視する形にする必要があるため、使い勝手が非常に悪い。

6. PC の値に従って命令を読み出し、1. で実行していた処理の続きを行う。

以下、ヒントを列記する。

- 基本的に、割り込み処理の呼出／復帰の処理は、関数呼出／復帰の処理の上位集合と考えることができるので、まず、BAL, BR 命令を先に実装すると良いだろう。
- レジスタ、条件コード、PC の退避／復帰処理をハードウェアで実装するのも良いが、退避／復帰処理も割り込み処理の一部として実装し、ハードウェアを簡略化するのも良いだろう。
- 割り込み処理の命令列の最初のメモリ・アドレスは固定でも良いが、レジスタで指定できるようにすると、いくつかの割り込み処理を切り替えて使うことができる。
 - 指定するレジスタは汎用レジスタの一部でも良いし、専用のレジスタを準備しても良い。
 - 専用のレジスタを準備したならば、それを操作する専用命令も必要となる。
- ハードウェア・コストは増大するが、割り込み処理用の PC、レジスタ・ファイル、条件コードを持ち、割り込み処理の開始／終了とともに切り替えるようにすれば、実装は簡単になるだろう。
- 割り込み信号を 1 個のみとするのではなく、複数の割り込み信号をサポートするのも良いだろう。もちろん、その場合、複数の割り込み処理を準備し、信号によってその中の 1 つを選択する形となる。
- 割り込み処理のテストに使うことも兼ねて、割り込みを発生する命令も実装するのも良いだろう。
- 通常のプロセッサの設計では、割り込み処理の命令は入出力命令と同様に、通常のプログラムでは実行できない特権命令にすることが多い。しかし、SIMPLE の規模では無理にそのように実装する必要はないだろう。
- タイマ割り込みを実装すると、リアルタイム処理のプログラムが書けて面白くなるだろう。ただし、PowerMedusa EC6S ボードにはタイマはないため、一定のクロック周波数を仮定し、クロック数からタイマ割り込みを発生させる形するのが良いだろう。

4.6 複合命令の追加

音声／画像を専門に処理するプロセッサ (Digital Signal Processor 等) では、音声／画像処理を効率的に実行するために、1 つの命令で複数の演算を行う命令を実装することが多い。このような命令の代表的なものに、以下のような積和演算命令 (MAC: Multiply ACcumulate) がある。

$$r[c] = r[c] + r[a] * r[b]$$

アセンブリでプログラムを書いた時、以下のような命令列がよく出現する場合、 $r[c] = r[c] \text{ op2 } (r[a] \text{ op1 } r[b])$ という動作を行う命令を追加する拡張も面白いと思う (op1, op2 は演算を示す)。

```
op1  r[a], r[b]
op2  r[c], r[a]
```

以下、ヒントを列記する。

- 実行時に読み出さなくてはならないレジスタ値が増えるので、レジスタ・ファイルの読み出しポート数を増やすか、マルチ・サイクルで読み出しを行うようにしなくてはならない。
- 演算器も複数実装する構成とマルチ・サイクル実行で 1 つの演算器を使いまわす構成が考えられる。

- 複雑な複合命令を実装すると、回路の複雑化によって動作クロック周波数が大きく下がることもありうる。また、拡張によるハードウェアの増大に見合った性能向上が得られないことも多い。そのあたりをちゃんと考察すると面白いだろう。
- 余裕があれば、演算器などを複数実装する構成とマルチ・サイクル実行を行う構成の両方を実装し、比較すると面白いだろう。

4.7 条件付き演算命令の追加

世の中には、条件コードの状態によって実行されたり実行されなかったりする、条件付き実行命令を持つプロセッサが存在する。このような命令は、パイプライン化されたプロセッサでは、分岐によるパイプライン・バブルを削減するのに有用である（詳細な説明は割愛する）。SIMPLE/B はパイプライン化されていないので、条件付き実行命令を実装する意義はない。しかし、同様の考え方により、条件コードの状態によって行う演算が変化する命令を実装するのはありだと考える。

以下、例を挙げて説明する。プログラム中に、if-else の構文の if 節で変数 A,B の加算を、else 節では変数 A,B の減算を行う構文があるとする。ここで、条件コードの状態が if 節の条件と同じ状態ならば加算、それ意外なら減算を行う命令を実装されているとすれば、if-else 節を表現するのに必要な命令数が大幅に削減される。上記のようなプログラムがあるのなら、このような条件付き演算命令の追加という拡張は非常に面白いだろう。

以下、ヒントを列記する。

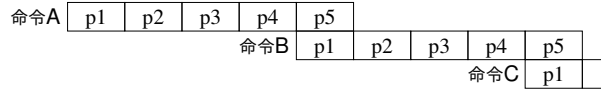
- 4.6 節の複合命令よりもさらに使える状況が限定された命令となる。1つのプログラムにしか使い道がないの命令とならないよう、それが有用なプログラムが複数あるかちゃんと検討して実装するべきだろう。
- レポートの中で複数のプログラムへの応用例を示し、その有用性をアピールするのが望ましいだろう。
- パイプライン化されたプロセッサにおいて、条件付き実行によってどのぐらいの利得が得られるかをレポートの中で考察するのも面白いだろう。

5 マイクロアーキテクチャの改良

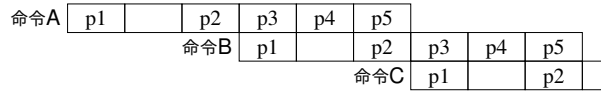
5.1 フェーズの並列実行

SIMPLE/B の改良法の例として、フェーズの並列実行による命令サイクルの短縮をあげる。

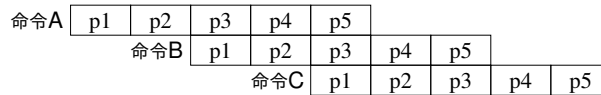
1. p1/p5 の並列実行 (図 4(a))
命令サイクルは 4 サイクルになる。分岐先アドレスを PC ヘットする操作を少し工夫すれば、簡単に実現できる。
2. p1/p3 と p2/p5 の並列実行 (図 4(b))
命令サイクルは 3 サイクルになる。
p1/p3 を常に並列実行するためには、分岐命令の実行をかなり工夫する必要がある (ヒント：分岐アドレス計算に p3 を使わない)。別の方法として分岐命令だけは並列実行しないというのも考えられる。
p2/p5 の並列実行については、汎用レジスタを更新する命令の直後に、同じレジスタを参照する命令が



(a)p1/p5 の並列実行



(b)p1/p3 と p2/p5 の並列実行



(c)p1/p3/p5 と p2/p4 の並列実行



(d)p1/p2/p3/p4/p5 の並列実行

図4 フェーズの並列実行

現れた時の処置を工夫しなければならない。

3. p1/p3/p5 と p2/p4 の並列実行 (図 4(c))

命令サイクルは 2 サイクルになる。2. のデータ依存の解決を更に工夫しなければならない。

4. p1/p2/p3/p4/p5 の並列実行 (図 4(d))

いわゆるパイプライン化であり、命令サイクルは 1 サイクルになる。実現は飛躍的に困難になるが、現在の多くのプロセッサと同じ構成となるため、チャレンジする価値は高い。

5.2 命令の並列実行 (スーパスカラ実行)

現在の PC 用プロセッサは、プログラム中の命令の中から並列に実行できる組を検出し、それらを並列に実行している。このような処理をスーパスカラ実行と呼び、これを行うプロセッサをスーパスカラ・プロセッサと呼ぶ。現在の PC 用プロセッサでは、スーパスカラ実行とアウトオブオーダー実行 (命令をプログラム順とは異なる順で実行可能とする実行) を組み合わせているが、この構成を実現するとハードウェアが大規模になりすぎると考えられるため、実現するならば、インオーダー実行 (命令はプログラム順のみで実行) との組み合わせが妥当だと考える。このスーパスカラ実行とインオーダー実行の組み合わせは、Intel 社の初代 Pentium でも採用されている。

インオーダー実行とスーパスカラ実行の組み合わせの場合、プロセッサはプログラム上で連続する命令を一度に主記憶から読み出し、それらの命令の間にデータ依存関係 (ある命令の実行結果を別の命令が使う関係) がない場合、それらの命令の並列実行を行う。プログラム上で連続する 3 命令以上がデータ依存関係にない確率

は非常に低いため、インオーダー実行とスーパースカラ実行を組み合わせるプロセッサの多くは、並列に実行する命令数を2命令にとどめている。以下、2命令の並列実行に対するヒントを述べる。

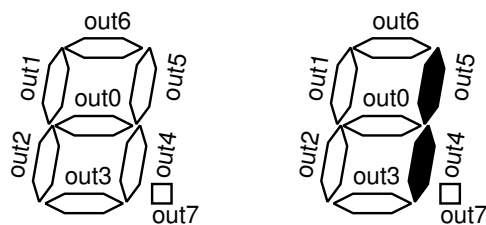
- 複数命令の主記憶の読み出しは、一般には、主記憶の読み出し単位を命令のビット長より大きくし、読み出した中から個々の命令を切り出す形で行う。ただし、後述するロード命令の並列実行のサポートのためにマルチポート RAM を使うのならば、RAM に複数の命令アドレスを送って複数の命令を読み出す形にするのもありだろう。
- 複数の命令が並列実行可能かどうかは、p2 に相当する部分で行う。具体的には、1 つ目の命令の Rd(Ra,Rb) と、2 つ目の命令 Rb,Rs,Rb,Ra が一致しない場合、並列実行が可能となる。ここで、並列実行可能でないと判断された場合、1 つ目の命令のみを p3 に送ることになる。2 つ目の命令については、廃棄した後に主記憶より再度読み出す方法や、レジスタに残しておいて次の命令の読み出しを1命令のみにするという方法などが考えれる。
- 2命令の並列実行を行うため、レジスタ・ファイルは4リード/2ライトの構成になる。
- 2命令並列実行時に、p3 で条件コードを更新するのは、2 つ目の命令側になる。
- p4 の主記憶アクセスを並列に行うために、主記憶はマルチポート RAM で構成すること。ただし、実験で使う FPGA では2リード/1ライトの RAM までしか準備されていないので、ストアの並列実行はできない。この並列実行しようとする命令がストア同士かの判断は、p2 での並列実行の判断に含めると良いだろう。

6 レポート提出/デモンストレーション

実験3ハードウェアにおける SIMPLE の実装の中で、初回レポート、中間報告、最終報告の3回のレポート提出/デモンストレーションを行ってもらう。

■初回レポート

個人単位で、以下の順による7SEG LEDの駆動回路の作成過程をレポートで報告せよ。なお、入力は4bitの2進数(1桁の16進数)であり、出力は下の図左のように7SEG LEDの各発光部につながる信号線である。



1. SEG LED を用いて1桁の16進数0~Fを表現するため、その表現方法を図示せよ。例えば、"1"の表現は上の図右のようになる。
2. 1桁の16進数を表すための4ビットの信号線 in[3..0] に対する8つの出力 out[7..0] の出力の真理値表を作成せよ。
3. 2. で作った真理値表を基に、out[7..0] の各出力の論理関数とその導出過程を示せ。なお、論理関数はカルノー図、クワイン-マクラスキー法等を使って最小化し、積和標準形で表すこと。
4. 3. を CAD のブロック・ダイアグラム・エディタ上で実装すること。また、ボード上の適当なスイッチ

による入力を in に、適当な 7SEG LED を out に割り当て、動作を確認すること。

なお、レポートでは、1.,2.,3. は手書きで記述し、4. は CAD のブロック・ダイアグラム・エディタ上の回路図を印刷して添付すること。

■中間報告

班単位で以下のデモンストレーションを行い、その後、レポートを提出すること。

- デモンストレーション

PowerMedusa ボード上に設計したプロセッサをダウンロードした上、何らかの命令が動作していることを示す。示し方は各自で工夫すること。

- レポート

製作する計算機に関して、以下の内容についてまとめ、各班ごとに 1 部提出すること。

- 方式設計と機能設計レベルの仕様書
- 考察など

なお、仕様書についての詳細は実験のウェブ・ページを参照すること。

■最終報告

班単位で以下のデモンストレーションを行い、その後、レポートを提出すること。

- デモンストレーション

製作したコンピュータの特徴などについて発表(セールストーク)を行い、完成したコンピュータ上で応用プログラムを実行する デモンストレーションを行う。

- レポート

完成した計算機に関して、以下の内容について、各班ごとに 1 部にまとめて提出すること。

- 前回仕様書からの変更点
- 実装した基本アーキテクチャからの拡張の説明
 - * どのような拡張を行ったか
 - * その拡張を行うとどのように嬉しいかの説明
 - ・ どのようなことが新たにできるようになるかを、サンプル・コードを使ったりして説明
 - * 拡張の評価 (以下の箇条書きは評価の例であり、必ずしも全て行う必要はない。)
 - ・ ゲート数 (LUT 数) がどう変わったか
 - ・ クロック周波数はどう変わったか (実動作 / CAD の予測値)
 - ・ プログラムの命令数 / 実行命令数 / 実行サイクル数の変化
- 考察および感想

7 関連資料

- 実験ホームページ: <http://www.lab3.kuis.kyoto-u.ac.jp/le3/>
実験の詳細な資料があります。また、様々なアナウンス等もここに載せます。
- QuartusII 関連

Altera 社の QuartusII という論理 CAD を用います。以下、関連ページを示します。

- QuartusII ソフトウェア :

<http://www.altera.co.jp/products/software/products/quartus2/qts-index.html>

QuartusII のホームページです。

- QuartusII オンライン資料 :

<http://www.altera.co.jp/literature/lit-qts.jsp>

QuartusII の日本語によるクイックスタートガイド、および、英語による詳細な資料があります。

- QuartusII Web Edition ソフトウェア :

http://www.altera.co.jp/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp

ウェブ・ライセンスを申請することにより、5 ヶ月間使える Quartus II Web Edition がダウンロードできます。家でも設計を行いたい人へ。

付録 A 基本アーキテクチャの命令セット

15 14 13 11 10 8 7 4 3 0				
11	Rs	Rd	op3	d
mnemonic		op3	function	
ADD	Rd, Rs	0000	$r[Rd] = r[Rd] + r[Rs]$	
SUB	Rd, Rs	0001	$r[Rd] = r[Rd] - r[Rs]$	
AND	Rd, Rs	0010	$r[Rd] = r[Rd] \& r[Rs]$	
OR	Rd, Rs	0011	$r[Rd] = r[Rd] r[Rs]$	
XOR	Rd, Rs	0100	$r[Rd] = r[Rd] \wedge r[Rs]$	
CMP	Rd, Rs	0101	$r[Rd] - r[Rs]$	
MOV	Rd, Rs	0110	$r[Rd] = r[Rs]$	
(reserved)		0111		
SLL	Rd, d	1000	$r[Rd] = \text{shift_left_logical}(r[Rd], d)$	
SLR	Rd, d	1001	$r[Rd] = \text{shift_left_rotate}(r[Rd], d)$	
SRL	Rd, d	1010	$r[Rd] = \text{shift_right_logical}(r[Rd], d)$	
SRA	Rd, d	1011	$r[Rd] = \text{shift_right_arithmetic}(r[Rd], d)$	
IN	Rd, d	1100	$r[Rd] = \text{input}$	
OUT	Rs	1101	output = r[Rs]	
(reserved)		1110		
HLT		1111	halt()	
15 14 13 11 10 8 7 0				
op1	Ra	Rb	d	
mnemonic		op1	function	
LD	Ra, d(Rb)	00	$r[Ra] = *(r[Rb] + \text{sign_ext}(d))$	
ST	Ra, d(Rb)	01	$*(r[Rb] + \text{sign_ext}(d)) = r[Ra]$	
15 14 13 11 10 8 7 0				
10	op2	Rb	d	
mnemonic		op2	function	
LI	Rb, d	000	$r[Rb] = \text{sign_ext}(d)$	
(reserved)		001		
(reserved)		010		
(reserved)		011		
B	Rb, d	100	$PC = PC + 1 + \text{sign_ext}(d)$	
(reserved)		101		
(reserved)		110		
(条件分岐命令)		111		
15 14 13 11 10 8 7 0				
10	111	cond	d	
mnemonic		cond	function	
BE	d	000	if (Z) $PC = PC + 1 + \text{sign_ext}(d)$	
BLT	d	001	if ($S \wedge V$) $PC = PC + 1 + \text{sign_ext}(d)$	
BLE	d	010	if ($Z (S \wedge V)$) $PC = PC + 1 + \text{sign_ext}(d)$	
BNE	d	011	if (!Z) $PC = PC + 1 + \text{sign_ext}(d)$	
(reserved)		100		
(reserved)		101		
(reserved)		110		
(reserved)		111		