特別研究報告書

ARMアーキテクチャにおける命令レベル並列処理の評価

指導教員　　富田 眞治 教授

京都大学工学部情報学科

木村 英雄

平成17年2月10日

ARM アーキテクチャにおける
命令レベル並列処理の評価

木村 英雄

内容梗概

本稿では，高速化と省電力化を目的としたマルチコアプロセッサにおける効率のよいコアの提案をする．ここ数年，プロセッサの周波数向上は鈍化してきており，トランジスタ集積度の増大により消費電力が増大し続けている．また発熱量の増加も問題となっている．単一コアのまま性能向上を図る従来の方法では，高速化や省電力化に限界がみえてきており，効率のよいマルチコア構成のための研究開発が進んでいる．マルチコアとは，１つのパッケージに２個以上のプロセッサコアを実装する技術である．各コアを他のコアの影響を受けることなく動作できる構成とし，処理を分担することにより性能向上が見込まれる．面積を２倍にしても処理能力は約 1.4 倍にとどまると言われており，面積が一定の場合には，より小さなコアを多数配置するほうが，全体を１コアとして構成するよりも性能面で優れていると考えられる．

本研究では，組み込み用途向けの代表的な小型プロセッサである ARM アーキテクチャに基づき，様々なマイクロアーキテクチャの比較検討を行った．一般的なコンパイラは，配列要素のロードに対して，シフト/加算/ロード/ポストインクリメントの４命令を対応付けるなど，テンプレートに従って命令列を生成することが多い．すなわち，より単純な命令に分解して改めてスーパスカラ実行しても，途中のレジスタ依存関係の存在により，さほど高速化できず，むしろテンプレートのようにある程度まとまった命令を単位としてスーパスカラ実行しても効率が落ちないのではないかと考えた．ARM アーキテクチャは，このような複数の機能を１命令により表現できる命令セットを持っている．命令の先頭には４ビットの実行条件が付いており，全命令を条件付き実行することができる．さらにプロセッサ構造の特徴として，利用頻度の比較的少ない浮動小数点演算器を搭載していないため，固定小数点演算からなる演算ライブラリを使用して，浮動小数点演算を実行している．そのため他のプロセッサと比較しても圧倒的にコア面積が小さくなっており，シンプルなコアといえる．ベンチマークプログラムを用いて実行命令数を SPARC と比較した結果，浮動小数点演算のハンディがあるにもかかわらず，平均するとほぼ同じ実行命令数となっ

た．一部のプログラムに関しては，圧倒的に ARM の命令数が多くなったが，これは数パーセントの浮動小数点演算命令が原因であった．

本研究では 4 つのモデルを検討した．

- モデル 1 は，最近のハイエンドプロセッサにおいて一般的に採用されているスーパスカラ方式，すなわち論理レジスタを約 2 倍の本数を有する物理レジスタにマッピングして，命令ウィンドウから発行可能な複数命令を同時に発行する方式により実行するモデルである．SFT/MUL ユニット 1 個，ALU2 個，LD/ST ユニット 1 個を備えている (以下 AP2 と略す)．
- モデル 2 は，単一命令発行方式で，一般的なパイプラインにより実行するモデルである．SFT/MUL ユニット 1 個，ALU1 個，LD/ST ユニット 1 個を備えている (以下 CS1 と略す)．
- モデル 3 は物理レジスタは用いず，実行可能な命令は，隣接する命令からのみ選択することで，大規模な命令ウィンドウを設けないスーパスカラ方式で並列実行するモデルであり，SFT/MUL ユニット 2 個，ALU2 個，LD/ST ユニット 1 個を備えている (以下 CS2a と略す)．
- モデル 4 は CS2a を改良したもので，命令を分解することなく，スーパスカラ方式で並列実行するモデルであり，SFT/MUL ユニット 2 個，ALU2 個，LD/ST ユニット 2 個を備えている (以下 CS2b と略す)．

AP2 は CS1，CS2a，CS2b より各実行ユニットの使用効率が向上することが期待できる．一方で，CS1，CS2a，CS2b は，AP2 と比較してレジスタの面積が約 1/7 倍となるため，演算ユニットが増えたことを考慮しても，総トランジスタ数は飛躍的に減少する．

4 つのモデルの性能を比較した結果，CS1 では AP2 より 14%IPC が低下したものの，CS2a，CS2b ではそれぞれ 10%，11% 向上した．AP2 は，CS1，CS2a，CS2b と比較してレジスタの総トランジスタ数が 7 倍以上多くなることを考慮すると，CS2a もしくは CS2b を採用した方がよりシンプルなコアを実現できることが分かった．CS2a と CS2b の性能を比較した結果，CS2b は CS2a に対して，約 5.1% の性能向上しか得られなかった．性能向上とシンプルなコア設計を考慮すると，CS2a がより要求を満たすコアを実現できるといえる．

Evaluation of Instruction-Level Parallel Processing in ARM Architecture

Hideo Kimura

Abstract

This paper proposes an effective core for high performance and low-power multi-core processors. Recently, the speedup rate in frequency of processors has been saturated, and the increasing transistor density leads to growing power dissipation. Also, increasing heating has been serious problem. Under these situations, it has been said that traditional processors which pursuit the performance with single core reached to upper bound in frequency and power consumption, and researches about effective multi-core architecture are in progress in the world. Multi-core is a new technology that implements two or more cores in single package. If each core is designed so as to work independently, effective parallelizing of processes can boost the total performance. It is said that the performance is limited to about 1.4 times even if the amount of hardware doubles its area. So, under the condition that the area is constant, it is thought that multi-core processor is superior to single-core processor.

In this research, I make a study and a comparison between several micro-architecture based on ARM architecture which is a de facto standard embedded micro-processor. By the way, general compilers tend to generate instruction stream based on some templates such as combination of 4 instructions (shift, add, load, post-increment) that corresponds to loads from array structure. Namely, I suppose the superscalar technique that divide a CISC-type instruction into some RISC-type instructions and execute in parallel dynamically can not gain performance because of the intermediate data dependence. In contrast to such traditional superscalar, I expect another superscalar model that execute several templates in parallel without dividing into RISC-type instruction can sustain the performance with simple hardware. The ARM architecture define the instruction set that is very suitable for such templates which can encode several RISC-type functions in one ARM instruction. The first 4-bits of every instruction specify a condition and control the conditional-execution. Furthermore, as a characteristic of processor structure, floating-point arithmetic

units are not equipped for the reason that such units are rarely used in embedded software, and the floating-point operations are implemented by software libraries written with fixed-point arithmetic operations. In the reason , ARM's core size is extremely smaller than other processors . It is said that ARM's core is simple. We compared the number of ARM's instructions with the number of SPARC's ones ,using bench mark program. On average, ARM's ones is almost as same as SPRAC's ones. About one of the program, ARM's ones grow much larger than SPARC's ones , because floating-point operations account for some percent of all instructions.

In this research, we examined 4models. Model1 is superscalar method generally adopted in high-end-processor. This model is that logical register is associate with physical register and some instructions which can be issued from instruction window is issued at the same time. This model is equipped with 1 SFT/MUL unit, 2 ALU, 1 LD/ST unit. We call it 'AP2'. Model2 is that only 1 instruction is issued. It is executed by general pipeline. This model is equipped with 1 SFT/MUL unit, 1 ALU, 1 LD/ST unit. We call it 'CS1'. Model3 is that instructions are executed in parallel. This model doesn't have large instruction window , for it doesn't have physical register and instructions which can be executed are selected from adjacent instruction. This model is equipped with 2 SFT/MUL unit, 2 ALU, 1 LD/ST unit.We call it 'CS2a'. Model4 is the model improving CS2a , that is superscalar architecture without dividing instructions. This model is equipped with 2 SFT/MUL unit, 2 ALU, 2 LD/ST unit. We call it 'CS2b'. In other hand, AP2's register size is 7 times as large as CS1,CS2a,CS2b's register size. So, in consideration of execution unit's increase, the number of CS1,CS2a,CS2b transistor decreases extremely. The result shows CS1 degrades the performance in 14% against AP2, and CS2a and CS2b can gain the performance in 10% and 11% respectively. Considering that AP2 costs the amount of hardware for registers in 7 times than CS, it is discovered that CS2a or CS2b is superior for simple-cores. The comparison between CS2a and CS2b shows that CS2b can gain only 5.1% against CS2a. I conclude that CS2a is most suitable core model for ARM based multi-core processors which pursuit high performance and small size.

ARM アーキテクチャにおける
命令レベル並列処理の評価

目次

第 1 章　はじめに　1

第 2 章　ARM アーキテクチャ　2

2.1　命令セットの特徴 2

2.2　実行命令流の特徴 3

第 3 章　RISC 型命令への分解に基づくスーパスカラ方式　5

3.1　命令の分解 5

3.2　パイプライン構成とバイパス回路 7

3.3　分岐予測 8

第 4 章　CISC 型パイプラインを並置するスーパスカラ方式　9

4.1　パイプライン構成 9

4.2　予備評価とバイパス回路 10

4.3　分岐予測 13

4.4　測定項目と測定方法 14

第 5 章　ハードウェア量に関する比較　16

第 6 章　評価　18

6.1　評価方法 18

6.2　測定結果 19

6.2.1　分解型モデルとカスケード型モデルの比較 19

6.2.2　各カスケード型モデルの比較 21

6.3　考察 22

第 7 章　おわりに　23

参考文献　24

第１章　はじめに

近年，パソコンや携帯電話，携帯端末の高機能化により，プロセッサに対してより一層の高速化が求められている．また，携帯端末のバッテリ駆動時間を長くするための省電力化も求められている．しかし，ここ数年，プロセッサの周波数向上は鈍化してきており，トランジスタ集積度の増大により消費電力が増大し続けている．また発熱量の増加も問題となっている．単一コアのまま性能向上を図る従来の方法では，高速化や省電力化に限界がみえてきており，効率のよいマルチコア構成のための研究開発が進んでいる．

マルチコアとは，１つのパッケージに２個以上のプロセッサコアを実装する技術である．各コアを他のコアの影響を受けることなく動作できる構成とし，処理を分担することにより性能向上が見込まれる．面積を２倍にしても処理能力は約 1.4 倍にとどまると言われており，面積が一定の場合には，より小さなコアを多数配置するほうが，全体を１コアとして構成するよりも性能面で優れていると考えられる．さらに，コアごとに電圧やクロックの制御を行う機能を付加し，消費電力の増大を抑えることも可能である．また，1 種類のコアを設計し，それを複数個配置することにより，設計効率が上がる利点もある．以上のことから，今後，マルチコア向けの，より小型かつ性能の良いコアが求められていくと言える．ただ，マルチコアの性能を十分に引き出すプログラミングは難しく，一般には，サーバーのように高いスループット性能が要求される場合にのみ有効である．マルチコア構成の代表的な汎用プロセッサには，ホモジニアス構成である Intel 社の Pentium D や，1 個の汎用的なコアと 8 個の小規模なコアを組み合わせたヘテロジニアス構成である SONY 社の Cell があげられる．

さて，一般的なコンパイラは，配列要素のロードに対して，シフト/加算/ロード/ポストインクリメントの４命令を対応付けるなど，テンプレートに従って命令列を生成することが多い．このようなテンプレートをそのまま 1 命令として実行しても，より単機能の RISC 型命令に分解して他のテンプレートに属する命令と混在させて実行しても，演算器間の依存関係により一定の制約を受けることから，抽出可能な命令レベル並列性に大差はないと考えられる．このような複数の機能を 1 命令により表現できる命令セットを持つアーキテクチャとして ARM がある．ARM アーキテクチャは，あまり使用頻度が高くないという理由で浮動小数点演算器を備えておらず，代わりに固定小数点演算からなる演算

ライブラリを使用して，浮動小数点演算を実行している．このため他のプロセッサと比較しても圧倒的にコア面積が小さい．例えば，ARM 社の MPcore は 4 個のコアを搭載しているにも関わらず，ダイ面積が $35mm^2$ と小さく, 消費電力も 3.3mW/MHz と低消費電力である．さらに不要なプロセッサは動的にシャットダウンしたり，必要になったら起動するという仕組みをソフトウェアが利用できる．

本稿では，組み込み用途向けの代表的な小型プロセッサである ARM をとりあげ，命令レベル並列処理により高速化を図るための効率的なマイクロアーキテクチャについて検討し評価する．以下，第２章では，ARM アーキテクチャについて概観する．第３章では，ARM 命令セットを RISC 型命令に分解した後にスーパスカラ実行する方式について詳述する．第４章では，CISC 型パイプラインを並置するスーパスカラ方式について詳述する．第５章では，この２つのスーパスカラ方式のハードウェア量に関する比較を行う．第６章では，この２方式を定量的に比較し，考察する．

第２章　ARM アーキテクチャ

本章では ARM アーキテクチャの主な特徴を述べる．

2.1　命令セットの特徴

図 2 に ARM の命令セットを示す．ARM 命令セットには，全命令の先頭に 4 ビットの実行条件が付いている．条件コードが，指定した実行条件を満たすとき命令を実行し，満たさない場合は命令を実行しない．条件コードとは，図 2 の NZCV フラグを指し，命令実行中に更新されていく．実行条件には，ALWAYS,NEVER，$=$，$\neq, \leq, \geq, <, >$,符号なし $>$，符号なし $\leq$，キャリー，キャリーなし，負，正または 0，オーバフロー，オーバフローなし，の 16 種類があり，4 ビットによりいずれか 1 つを指定する．一方，汎用アーキテクチャレジスタは 16 本と，一般的な RISC プロセッサの 32 本よりも少ないため，3 オペランド形式でもレジスタ指定には計 12 ビットあればよく，実行条件フィールドの存在により命令語のエンコードが大きな制約を受けることはない．実行条件を命令中に取り込むことで，分岐予測ミスペナルティを伴う条件分岐命令を使わずに済ませる利点は大きい．

(a) ソースプログラム

```
if (R0 != 7)
  { R1 = R1 + R2 - R0; }
```

(b) 条件分岐命令を用いる場合

```
    cmp   R0,#7       /*  R0と7を比較                  */
    beq   test        /*  = の場合  testへジャンプ     */
    add   R1,R1,R2    /*  R1 = R1 + R2                 */
    sub   R1,R1,R0    /*  R1 = R1 - R0                 */
test  ...
```

(c) 実行条件を指定する場合

```
    cmp   R0,#7       /* R0と7を比較                   */
    addne R1,R1,R2    /* != の場合  R1 = R1 + R2       */
    subne R1,R1,R0    /* != の場合  R1 = R1 - R0       */
```

図 1: 簡単な if 文の例

C 言語による if 文の例を図 1(a) に，また，条件分岐命令を用いたコンパイル結果を図 1(b) に，さらに実行条件付き命令を用いた結果を図 1(c) に各々示す．実行条件付き命令を用いることにより，命令数が削減され，分岐予測ミスペナルティによる性能低下も回避できる．

次に，シフト演算を算術演算，論理演算，ロード命令中に組み込むことができる．例えば，「$i = i + (j << 2)$」のように，一般的な他のプロセッサではシフト演算と加算の 2 命令に展開される文は，左にシフトするアドレッシングモードを指定し，Operand2 に j を 2 ビットシフトした値を入れ，i に加算する 1 命令 $(addi, i, j, asl2)$ として表すことができる．

さらに特徴的な命令に，Load/Store Multiple (以下，LDM と略す) がある．複数レジスタへメモリから読み込んだり，メモリへ格納できる．1 命令で最大 16 レジスタのロードまたはストアが可能である．

また，ロード/ストア命令には，PC 相対アドレッシングやプレ-/ポスト-インクリメント・アドレッシングモードなど様々なアドレッシングモードがある．

2.2 実行命令流の特徴

ARM と SPARC の実行命令数を MiBench と SPEC CPU2000 を用いて測定した．図 3 および図 4 に測定結果を示す．SPARC の場合の命令数を 1 とした時の ARM の命令数の比率を示している．この図から，ARM アーキテクチャには浮動小数点演算命令がないという制約があるにもかかわらず，平均して浮動小数点演算器を備える SPARC アーキテクチャとおよそ同等の実行命令数となるこ

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
実行条件	operand code	S	Rn (source)	Rd (destination)	Operand 2
	AND命令		Rd = Rn AND Operand2	S=1 なら	NZC flag 更新
	EOR命令		Rd = Rn EOR Operand2	S=1 なら	NZC flag 更新
	SUB命令		Rd = Rn - Operand2	S=1 なら	NZCV flag 更新
	RSB命令		Rd = Operand2 - Rn	S=1 なら	NZCV flag 更新
	ADD命令		Rd = Rn + Operand2	S=1 なら	NZCV flag 更新
	ADC命令		Rd = Rn + Operand2 + Carry	S=1 なら	NZC flag 更新
	SBC命令		Rd = Rn - Operand2 - Not(Carry)	S=1 なら	NZCV flag 更新
	RSC命令		Rd = Operand2 - Rn - Not(Carry)	S=1 なら	NZCV flag 更新
	TST命令		Al= Rn AND Operand2		NZC flag 更新
	TEQ命令		Al = Rn EOR Operand2		NZCV flag 更新
	CMP命令		Al = Rn - Operand2		NZCV flag 更新
	CMN命令		Al = Rn + Operand2		NZCV flag 更新
	ORR命令		Rd = Rn OR Operand2		NZC flag 更新
	MOV命令		Rd = Operand2	S=1 なら	NZC flag 更新
	BIC命令		Rd = Rn AND Not(Operand2)	S=1 なら	NZC flag 更新
	MVN命令		Rd = Not(Operand2)	S=1 なら	NZC flag 更新

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
実行条件	operand code	S	Rd (destination)	Rn (source)	Rs (source)	1 0 0 1	Rm (source)
	ML命令		Rd = (Rm * Rs) [31 : 0]	S=1 なら	NZ flag 更新		
	MLA命令		Rd = (Rm * Rs + Rn) [31 : 0]	S=1 なら	NZ flag 更新		
実行条件	operand code	S	RdHi(destination)	RdLo(destination)	Rs (source)	1 0 0 1	Rm (source)
	UMULL命令		RdHi = (Rm * Rs) [63 : 32]	RdLo = (Rm * Rs) [31 : 0]	S=1 なら	NZ flag 更新	
	UMLAL命令		RdLo = (Rm * Rs) [31 : 0] + RdLo RdHi = (Rm * Rs) [63 : 32] + RdHi + CarryFrom((Rm * Rs) [31 : 0] + RdLo)	S=1 なら	NZ flag 更新		
	SMULL命令 (符号・補正)		RdHi = (Rm * Rs) [63 : 32]	RdLo = (Rm * Rs) [31 : 0]	S=1 なら	NZ flag 更新	

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
実行条件	operand code	L	Rn (source)	Rd (destination)	アドレッシングモード	operand code	アドレッシングモード
	LD/ST UH命令	L=1なら Rd = Memory(address) [15 : 0]		L=0なら Memory(address)[15 : 0]= Rd[15 : 0]			
	LDSH命令	Rd = SignExtend(Memory(address) [15 : 0])					
	LDSB命令	Rd = SignExtend(Memory(address) [7 : 0])					
	LD/ST W命令	L=1 なら Rd = ROR (Memory (address)[31:0] , # (8 * address[1:0])) L=0なら Memory(address)[31 : 0]= Rd[31 : 0]					

実行条件	operand code	L	Rn (source)	register list
	LD/ST M命令	L=1 なら for (i=0 to 14){if(register list[i]=1) Ri = Memory(address) [31 : 0] address +4} L=0 なら for (i=0 to 15){if(register list[i]=1) Memory(address) [31 : 0]= Ri address +4}		

31 30 29 28	27 26 25	24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
実行条件	operand code	L	Operand 1
	B , BL命令		PC = PC + (SignExtend (Operand1) << 2) L=1なら R14=後続命令のアドレス

図 2: ARM の命令セット

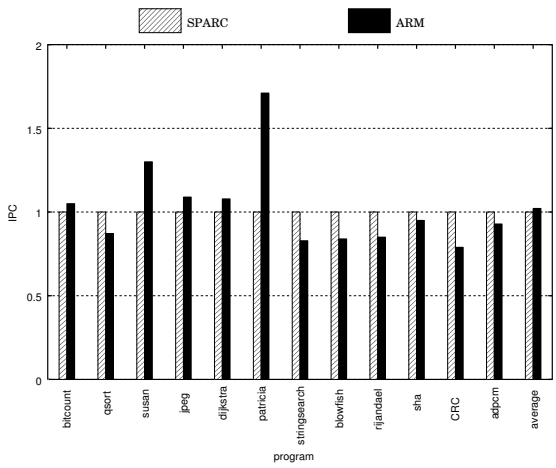


図 3: MiBench

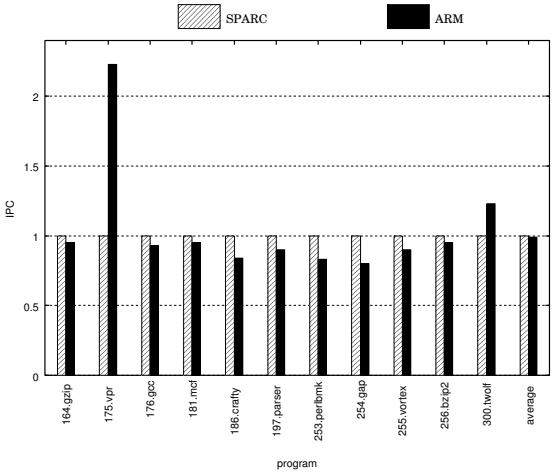


図 4: SPEC CPU2000

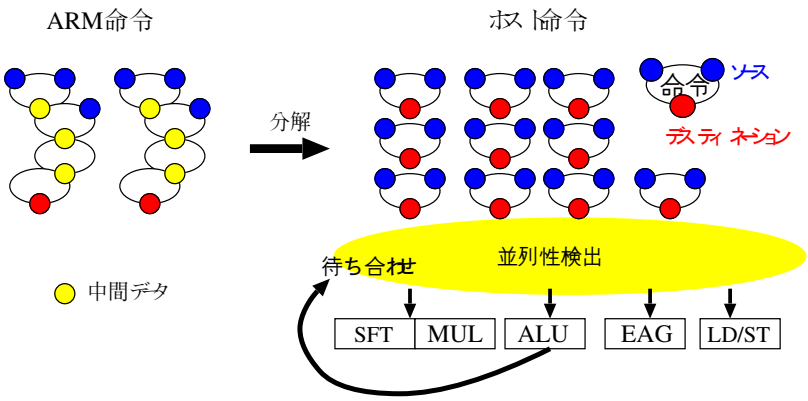


図 5: 分解型モデル

とが分かった．個々のプログラムを比較すると，susan の一部のプログラムに関しては，5.60 と圧倒的に ARM の命令数が多くなった．175.vpr も同様であり，数パーセントの浮動小数点演算命令が原因である．

第 3 章　RISC 型命令への分解に基づくスーパスカラ方式

3.1　命令の分解

ARM 命令セットは 1 命令中にシフト演算 (以下 SFT)，算術論理演算 (以下 ALU)，ロード (以下 LD) を同時に記述できる．このような CISC 型命令を図 5 に示すように SFT，ALU，LD 命令など単機能の RISC 型命令 (以下，ホスト命

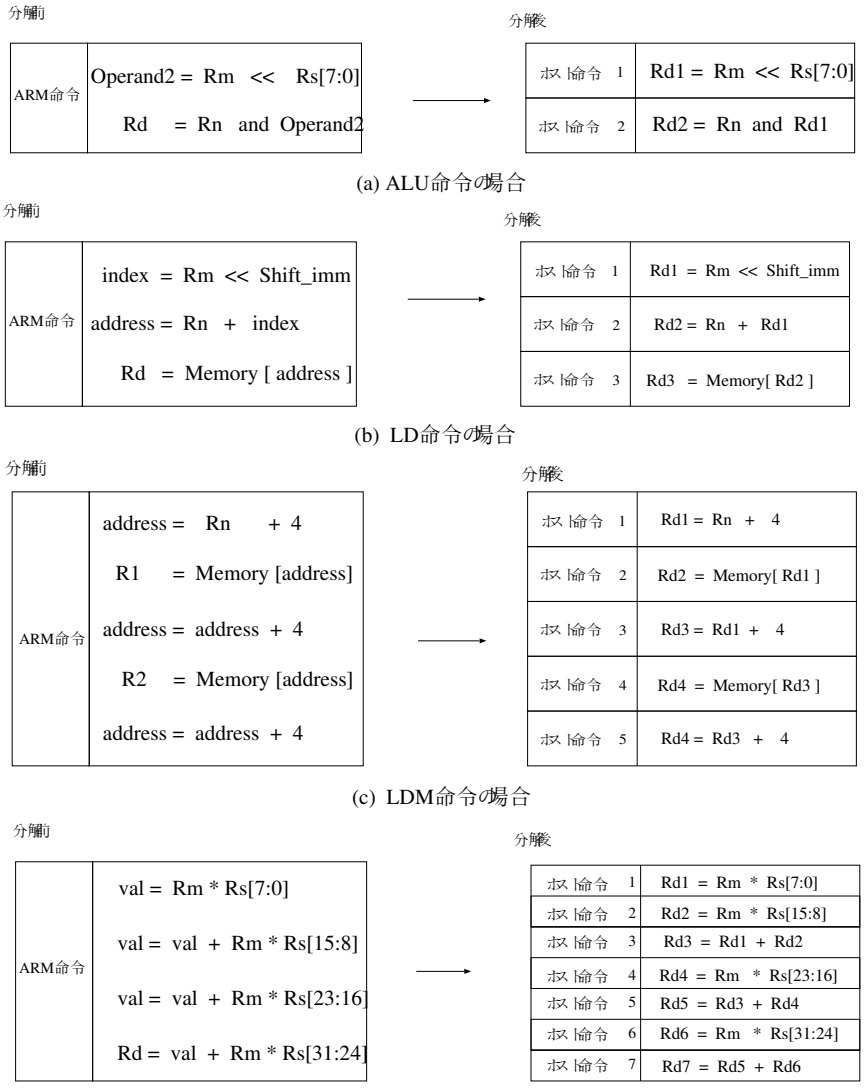


図 6: ARM 命令の分解

令と略す) に分解し，ホスト命令を並列実行することにより高速化を図るスーパスカラ方式が考えられる (以下，分解型モデルと略す)．商用プロセッサでは，Intel 社の Pentium-Pro が採用していた．

ARM の演算命令は，SFT と一体のものとなっており，分解型モデルではSFT 命令と ALU 命令の２つのホスト命令に分解する．図 6(a) に AND 演算の分解方法を示す．

ARM 命令が LDM 命令以外の LD 命令の場合，シフト，アドレス計算を経て，メモリからロードすることになっており，図 6(b) に示すようにSFT，ALU，LD

I A	I F	ADE	HDE	MAP	SEL	E X	W R	RET

図 7: 分解型のパイプライン

命令の３つのホスト命令に分解する．

ARM 命令が LDM 命令の場合，ALU 命令と，(LD，ALU 命令)×(ロード/ストア回数) に分解する．連続 16 回のロード/ストアができるため, 最大で 32 命令＋ ALU 命令 1 個の合計 33 命令に分解される．図 6(c) は 2 回ロードする場合の LDM 命令の分解方法である．

乗算命令は，種類によって分解方法が変わってくる．ML 命令，MLA 命令の場合，４個の乗算命令に分解する．UMULL 命令，UMLAL 命令の場合，８個の乗算命令と８個の ALU 命令に分解する．SMULL 命令の場合，８個の乗算命令と 14 個の ALU 命令の計 22 個のホスト命令に分解する．図 6(d) に ML 命令の分解方法を示す.

3.2　パイプライン構成とバイパス回路

図 7 に，分解型モデルのパイプライン構成を示す．IA はアドレス生成，IF は命令フェッチ，ADE は ARM 命令のデコード，HDE はホスト命令のデコード，MAP は物理レジスタへのマップ，SEL は実行可能な命令の選択と，レジスタからの読みだし，EX は演算の実行，WR はレジスタへの書き込み，RET は命令のリタイアを意味している．

まず，命令アドレスを生成し，連続する 2 命令をフェッチする．ARM 命令をホスト命令に分解し，元の ARM 命令のソース，デスティネーションの情報を分解したホスト命令に付加する．1 つの ARM 命令は，一度に最大 4 命令のホスト命令に分解するものとする．多くのホスト命令に分解する際には，複数サイクルにより分解する．最終的にはソースレジスタ 2 個と，デスティネーションレジスタ 1 個の単純な RISC 型命令となる．分解したホスト命令は，次の MAP ステージにおいて論理レジスタが物理レジスタに対応付けられる．1 サイクルにつき，最大 4 命令までマップテーブルに登録できるものとする．次の SELECT ステージでは，マップテーブルに登録されたホスト命令の中から実行可能な命令を選択し, 実行に必要な値をレジスタやバイパス回路から読み出す用意をし，次の実行ステージに移る．実行ステージには，LD ステージ，SFT/MUL ステー

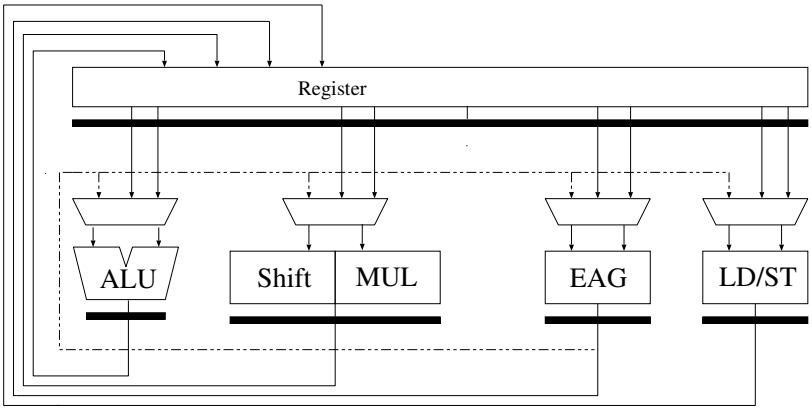


図 8: 分解型のブロック図

ジ,ALU ステージ,EAG ステージがある．分解型モデルでは ALU ユニットとは別に，アドレス計算用の EAG ユニットを配置しており，ARM の LD 命令を SFT，ALU，LD 命令に分解する時の ALU 命令に EAG ユニットを割り当てている．これを EAG ステージと呼ぶ．次の WRITE ステージでは，演算結果を物理レジスタに書き込む．そして RETIRE ステージでは，マップテーブルに基づき論理レジスタへの書き込みを行う．

図 8 に分解型のブロック図を示す．ブロック図中の太線は，ラッチを表している．ALU，SFT/MUL ユニット，EAG ユニットには点線で示すバイパスがある．これによって後続命令が次のサイクルで直ちに結果を利用でき，レジスタに値が書き込まれるのを待つ必要がなくなる．分解型モデルは，RISC 型命令に分解することで各ユニットの使用効率を上げることを狙っている．しかし，各ホスト命令間でのデータ依存関係が多いと，命令レベル並列実行のためにより大きな命令ウィンドウが必要となるため，１つのコアの面積が大きくなってしまう．以上のモデルを AP2 と呼ぶことにする．

3.3 分岐予測

分岐予測には taken 予測方式を仮定した．taken 予測とは条件分岐命令の実行条件が常に成立するものとして予測する静的分岐予測である．分岐予測ミスは，分岐命令以前の命令が終了するのを待ってから，投機状態の命令実行を中止し，図 9 に示すように命令フェッチからやり直す．

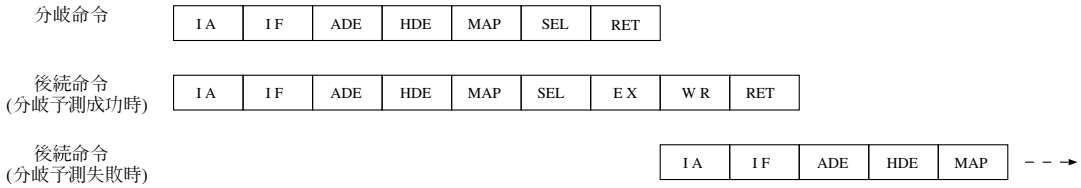


図 9: 分岐命令時のパイプライン

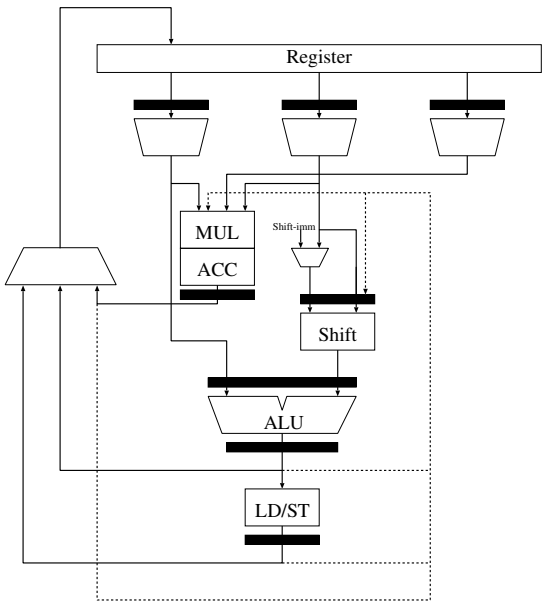


図 10: カスケード型のブロック図

第 4 章　CISC 型パイプラインを並置するスーパスカラ方式

4.1　パイプライン構成

CISC 型の ARM 命令をそのまま並列実行するスーパスカラ方式が考えられる (以下，カスケード型モデルと略す)．1 命令語中の演算の組合せに合うよう，図 10 に示すように各演算ユニットを配置する．ブロック図の太線はラッチを表している．カスケード型モデルは命令を分解せず物理レジスタも設けないため，コアの面積を分解型モデルより小さくできる．

カスケード型モデルのパイプライン構成について述べる．命令の種類は大きく，ALU 命令，LD/ST 命令，MUL 命令の 3 種類に分類できる．

ALU 命令とは加算，減算など乗算を除く算術演算と論理演算のための命令である．依存が無い場合，ALU 命令は IF/DE/SFT/ALU/WR の計 5 段のステー

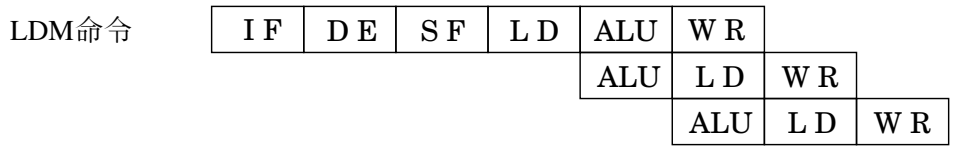


図 11: LDM 命令

ジにより実行される．ただし，IF は命令フェッチ，DE はデコード，SFT はシフト，ALU は演算，LD はメモリからの読み込み，WR はレジスタへの書き込みを意味している．

LDM 命令を除く LD 命令は，依存が無い場合，IF/DE/SFT/ALU/LD/WR の計 6 段のステージにより実行される．LDM 命令についてはメモリアクセスの回数によってサイクル数が変化し，N 回のメモリアクセスでは (5+N) 段が必要となる．例えば 3 回ロードを繰り返す LDM 命令の場合，図 11 に示すように，8 段が必要となる．

MUL 命令とは，２つの 32bit オペランドの積を出力する乗算命令であり，全部で 5 種類の乗算命令がある．ML 命令は２つの 32bit オペランドの乗算をし，最下位 32bit の値をレジスタにストアする命令で,MLA 命令はその最下位 32bit の値に別の 32bit オペランドを加算する積和命令である．この 2 つの命令は積値，もしくは積和値を出力するまでに 4 段が必要となり，IF，DE，WR を含めると計 7 段が必要となる．UMULL 命令は 2 つの符号なし 32bit オペランドの乗算をし，出力された 64bit の値を最上位 32bit，最下位 32bit に分け，2 つの独立したレジスタにストアする命令であり，UMLAL 命令は各 32bit の乗算結果に別の 32bit オペランドを加算する積和命令である．この 2 つの命令は積値，もしくは積和値を出力するまでに 8 段が必要であり，IF，DE，WR を含めると計 11 段が必要となる．SMULL 命令は 2 つの符号付き 32bit オペランドの乗算をし，出力された 64bit の値を最上位 32bit，最下位 32bit に分け，2 つの独立したレジスタにストアする命令である．この命令は積値を出力するまでに 12 段が必要であり，IF，DE，WR を含めると計 15 段が必要となる．表 1 に，依存がない時の各命令処理に必要な段数をまとめる．

4.2 予備評価とバイパス回路

カスケード型モデルを詳細化するために，各ユニット間にどのようなバイパスが必要であるかについて検討した．

命令	段数
ALU	5
MUL MLA	7
UMULL UMLAL	11
SMULL	15
LD	6
LDM	5+N

表 1: ARM 命令処理に必要な段数

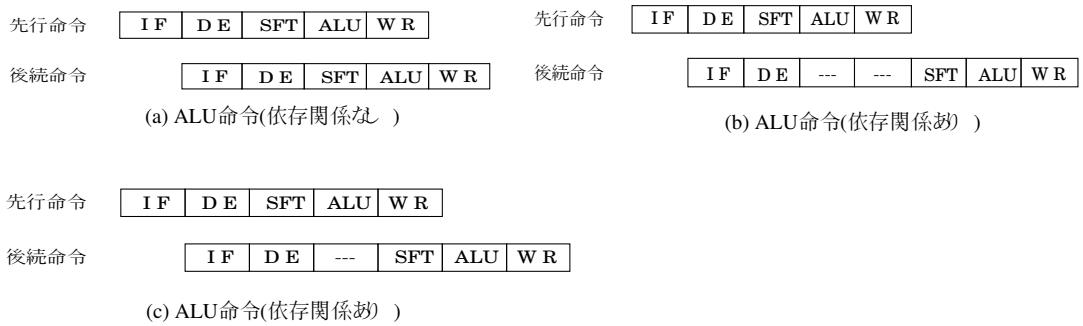


図 12: パイプラインのインタロック

先行/後続命令間にデータ依存がある時，そのデータ依存に関係するユニット間にバイパスがない場合は，後続命令は先行命令の出力した演算値がレジスタにストアされるのを待つ必要がある．バイパスがある場合，後続命令はレジスタへのストアを待つ必要なく，バイパスを経由して先行命令の値が出力された次のサイクルでその値を処理でき，結果として全体のサイクル数も短縮でき，IPC の向上につながる．しかし，全てのユニット間にバイパスを繋ぐと，利用頻度の低いバイパスにより回路が複雑になったり，無駄な消費電力が増加するため，利用頻度の高いバイパスがどこであるかを確認する必要がある．

まず，図 12 に，先行命令に対し，バイパスの有無によって後続命令の処理に影響を与えるような先行/後続命令の組合せを示す．
例えば，図 12(a) は，先行/後続命令ともに ALU 命令で，前後の命令間にデータ依存がない場合のパイプラインの流れをしている．

データ依存がある場合，後続命令は WR ステージでレジスタに値が書き込まれてから SFT ステージを開始する必要があるので，図 12(b) で示すパイプライン

先行命令	後続命令	依存のあるユニット
ALU	ALU	ALU→SFT
	LD	ALU→SFT
	MUL	ALU→MUL
LD	ALU	LD→SFT
	LD	LD→SFT
	MUL	LD→MUL
MUL	ALU	MUL→SFT
	LD	MUL→SFT
	MUL	MUL→MUL

表 2: 先行，後続命令による依存関係

の流れになる．この時にシフタ，ALU ユニット間にバイパスがあると，図 12(c) で示すようなパイプラインの流れになり，バイパスがない場合より，1 サイクル分短縮できる．この例では，依存が ALU→SFT 間にあるといえる．先行命令，後続命令の種類によって，依存関係が生じるユニットの種類も変わる．一覧を表 2 に示す．表 2 は，後続命令が ALU 命令でも LD 命令でも SFT ユニットに依存することが分かる．バイパスの候補は，図 10 中の点線に示すように 6 箇所ある．

次に，表 2 に示した関係に基づき，MiBench を用いて SFT,MUL,ALU,LD/ST の各ユニット間の依存回数を計測した．表 3 に，MiBench の計測結果を示す．表 3 に示した命令数は MiBench の各プログラムにおいて依存関係にある命令数の総和であり，内訳を比率により示している．この結果から，ALU → SFT の依存割合が過半数を占め，次に LD/ST → SFT，LD/ST → MUL の順になっていることが分かる．その他の依存関係は 1 % 以下となっている．これは，全命令に対する MUL 命令の割合が低いため，相対的に依存する割合が低いことを意味する．

使用頻度の低いバイパスを設けるのは無駄であるため，依存割合が 5%以上のユニット間に限りバイパスを設けることにした．すなわち，ALU → SFT，LD/ST → SFT，LD/ST → MUL 間である．以上の検討結果をふまえたブロック図を図 13 に示す．

依存関係	依存している命令数	依存している割合 (%)
ALU → SFT	221,029,228	65%
ALU → MUL	5,320,562	1%
MUL → SFT	1,666,316	1%未満
MUL → ALU	782	1%未満
MUL → MUL	5,646	1%未満
LD → SFT	81,299,587	19%
LD → ALU	3,288,433	1%未満
LD → MUL	49,629,183	11%

表 3: MiBench における依存割合

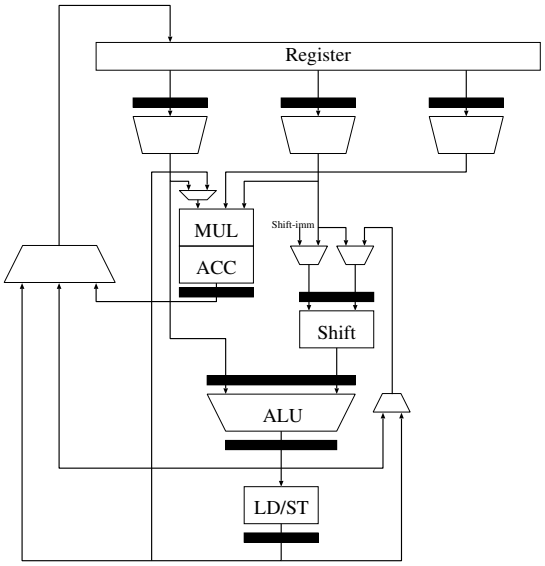


図 13: バイパスを加えたブロック図

4.3 分岐予測

分岐予測には taken 予測方式と gshare 予測方式を用いた．gshare 予測とは 2 レベル適応型の分岐予測を拡張したもので，グローバル分岐履歴レジスタと分岐アドレスとの排他的論理和によりパターン履歴表へのインデックスを作成する．パターン履歴表は 2 ビット飽和型カウンタの配列であり，選択された 2 ビットカウンタの値により分岐方向を予測する方式である．

さて，分岐予測が失敗した場合，次に実行すべき命令フェッチし直す必要がある．分岐命令の ALU ステージの次のサイクルに後続命令を再フェッチする．

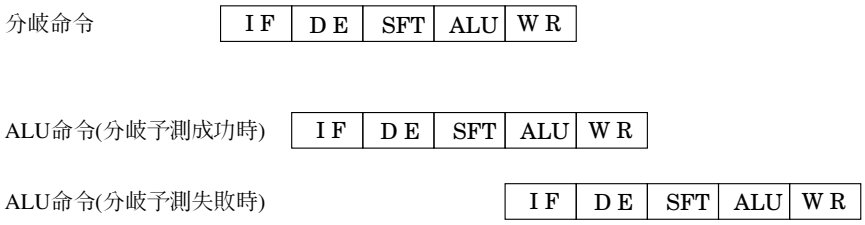
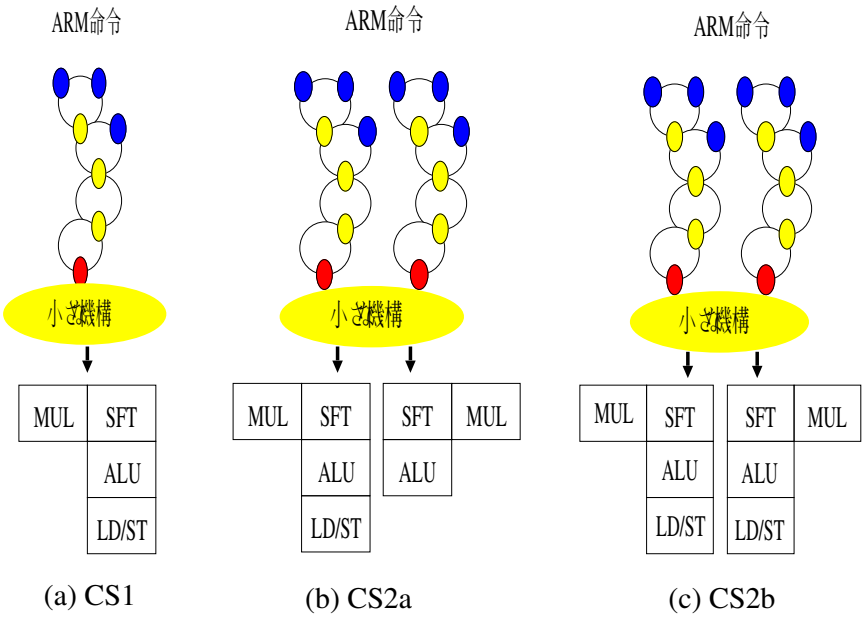


図 14: 分岐予測時のパイプライン



(a) CS1 (b) CS2a (c) CS2b

図 15: カスケード型モデル

例えば，先行命令が分岐命令，後続命令が ALU 命令の場合，分岐予測を行った時のパイプラインの流れは図 14 に示すようになり，分岐予測失敗によるペナルティサイクル数は 1 命令発行の場合，３サイクルとなる．後続命令が MUL 命令，LD/ST 命令であっても ALU 命令と同様，３サイクルのペナルティサイクルとなる．

4.4 測定項目と測定方法

分解型モデルでは EAG ユニットを含めて ALU ユニットは 2 個ある．そこでカスケード型モデルでも図 15(a) で示すような 1 命令発行のモデルの他に，2 命令発行で ALU ユニットを 2 個搭載した図 15(b) に示すようなモデルも加える．さらに，LD/ST ユニットを 2 個搭載した図 15(c) に示すようなモデルを加えた．

評価に用いるパイプラインモデルは以下の３つである．

先行命令＼後続命令	ALU	LD/ST	MUL MLA	UMULL UMLAL	SMULL
ALU	1 2	2 3	3 5	7 9	11 13
LD/ST	1 2	1 3	2 4	6 8	10 12
MUL MLA	1 3	1 4	1 5	5 9	9 13
UMULL UMLAL	1 3	1 4	1 5	1 9	5 13
SMULL	1 3	1 4	1 5	1 9	1 13

表 4: 依存有無時の命令のサイクル数

CS1：1 命令発行カスケード型モデル．SFT/MUL ユニットを 1 個，ALU を 1 個，LD/ST ユニットを 1 個搭載している．

CS2a：2 命令発行カスケード型モデル，SFT/MUL ユニットを 2 個，ALU を 2 個，MUL ユニットを 2 個，LD/ST ユニットを 1 個搭載している．

CS2b：2 命令発行カスケード型モデル．SFT/MUL ユニットを 2 個，ALU を 2 個，LD/ST ユニットを 2 個搭載している．

CS2a は 2 個の ALU に対して 1 個の LD/ST ユニットを搭載しているが，どちらの ALU からでも LD/ST ユニットが使用できると仮定した．

分岐予測方式は taken 予測方式と gshare 予測方式の 2 方式で測定し，それぞれ，-t，-g のサフィックスを付加して区別する．

先行命令の WR ステージから後続命令の WR ステージまでの所要サイクル数をデータ依存がある場合とない場合とに分けて，表 4 に示す．上段が依存なしの場合，下段が依存ありの場合のサイクル数である．

例えば，先行命令が ALU 命令で後続命令が LD 命令の場合，命令間にデータ依存が無ければ 図 16(a) に示すように，先行命令が終了して 2 サイクル経過後に後続命令が終了する．これに対し，データ依存がある場合，図 16(b) で示すようになり，ALU→SFT ユニット間にバイパスがあるため，先行命令の ALU ステージの次のサイクルで，後続命令が SFT ステージになる．すなわち，先行命令が終了してから 3 サイクル経過した後に，後続命令が終了する．

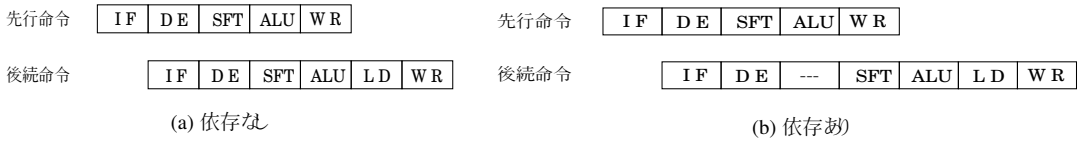


図 16: 依存有無時の命令のパイプライン

第5章　ハードウェア量に関する比較

本章では，カスケード型モデルと分解型モデルの性能を比較した際に予想される測定結果について述べ，次にハードウェア量の比較をする．AP2 と CS1 を比較した場合，AP2 は ALU 相当の EAG ユニットが 1 個余分にあるため，AP2 の性能が若干上回ると予想される．一方，AP2 と CS2a を比較した場合，CS2a は SFT/MUL ユニットが 1 個余分にあるため，CS2a の性能が若干上回ると予想される．AP2 と CS2b を比較した場合，CS2b は SFT/MUL ユニット，LD/ST ユニットそれぞれ 1 個余分にあるため，CS2b の性能がかなり上回ると予想される．

次に，各カスケード型モデルの性能を比較した際に予想される結果について述べる．CS1 と CS2a を比較した場合，CS2a は SFT/MUL ユニット，ALU がそれぞれ 1 個余分にあるため，命令出現頻度から考えて，CS2b の性能が 1.5 倍程度上回ると予想される．CS1 と CS2b を比較した場合，CS2b は各演算器が 2 倍あるため，CS2b の性能が 2 倍近く上回ると予想される．CS2a と CS2b を比較した場合，CS2b は LD/ST ユニットが 1 個余分にあるため，命令出現頻度から考えて，CS2b の性能がかなり上回ると予想される．マルチコアでは高性能で，かつ，シンプルなコアが求められており，性能が若干良くなる程度なら，LD/ST ユニットを余分に 1 個搭載し，わずかな性能向上を求めるより，よりシンプルなコアとする方が本稿の目的に沿っている．もちろん，大幅に性能向上するのであれば，LD/ST ユニットを追加搭載するのが良いといえる．

さらに，各モデルにおけるレジスタと SFT/MUL ユニットのハードウェア量について詳述する．AP2 の物理レジスタは 32 エントリである．各エントリには，少なくともレジスタの値用に 32 ビット，プログラムカウンタ用に 32 ビット，即値用に 32 ビットの合計 96 ビットの幅が必要となる．また，AP2 の論理レジスタは 32 ビット幅のものが 16 本ある．AP2 のホスト命令はソース 2 個，デスティネーション 1 個でかつ，最大 4 命令発行可能なので，物理レジスタと論理レ

ジスタのリード線は各 8 本，ライト線は各 4 本必要である．一方，CS1，CS2a，CS2b の論理レジスタは 16 本である．各エントリにはレジスタの値用に 32 ビットの幅が必要となる．ARM 命令は最大ソース 4 個，デスティネーション 1 個でかつ，最大 2 命令発行可能なので，論理レジスタのリード線は 8 本，ライト線は 2 本となる．

ここで，より詳細に各モデルにおいて必要なレジスタのトランジスタ数を比較する．1 ビットのレジスタには記憶用にトランジスタが 6 個，ライト線用にポート 1 個あたり 4 個，リード線用のポート 1 個につき 8 個が必要となる．従って，AP2 の 1 ビット幅の 1 エントリ分に相当するレジスタ素子には，8 個の読み出しポートが必要となるので，計 48 個，4 個の書き込みポートのために計 16 個，さらに記憶用の 6 個で合計 70 個のトランジスタが必要となる．AP2 の物理レジスタは 32 エントリ，96 ビット幅であるため，$70 \times 96 \times 32$ で約 21.5 万個のトランジスタが必要となる．AP2 の論理レジスタは 16 エントリ，32 ビット幅であるため，$70 \times 32 \times 16$ で約 3.5 万個のトランジスタが必要となる．論理レジスタ，物理レジスタをあわせた総トランジスタ数は約 25 万個となる．CS の 1 ビット幅で 1 エントリ分に相当するレジスタ素子には，8 個の読み出しポートが必要となるので，計 48 個，2 個の書き込みポートのために計 8 個，さらに記憶用の 6 個で合計 62 個のトランジスタが必要となる．CS の論理レジスタは 16 エントリ，32 ビット幅であるため，$62 \times 32 \times 16$ で約 3.1 万個のトランジスタが必要となる．AP2 と CS のトランジスタ数を比較した場合，AP2 の方が約 22 万個多くなる．

さて，AP2 と CS2b を比較した場合，CS2b の方が SFT/MUL ユニットが 1 個多く搭載されている．そこで，SFT/MUL ユニットのトランジスタ数を検討する．MUL ユニットでは 32 ビット ×8 ビットの乗算ができ，全加算器を横に 32 個結合し，それを 8 段組むことで構成されている．1 個の全加算器は，28 個のトランジスタで構成されており，MUL ユニットでは $28 \times 32 \times 8$ で 7168 個のトランジスタが必要となる (文献 [5, P.273])．SFT ユニットは 32 ビット幅のデータを 1 ビット，2 ビット，4 ビット，8 ビット，16 ビットシフトする 5 個の回路を 5 段にして組むことで構成できる．各段は，2 入力セレクタを横に 32 個並べることで構成できる．2 入力 NAND3 個から構成でき，NAND 回路は，1 個につき 4 個のトランジスタを必要とする．すなわち，全体では $4 \times 3 \times 32 \times 5$ の約 2000 個のトランジスタが必要となる．あわせて，SFT/MUL ユニットの総トラ

ンジスタ数は約 9000 個となる．また，他の演算ユニットも同程度となる．

2 つのモデルのハードウェア量を比較すると，演算ユニットによるトランジスタ数の増加量よりも，レジスタによる増加量が上回るため，分解型モデルのコア面積は増大する．例えば，AP2 と CS2b のトランジスタ数を比較した場合，CS2b は AP2 より SFT/MUL ユニットが 1 個多く搭載されており，トランジスタ数は約 9000 個増加するが，AP2 は物理レジスタによりトランジスタが約 22 万個増加するため，全体的に見ても，AP2 の方が約 21 万個増加する．

第 6 章　評価

3 章と 4 章に提案した 2 つのパイプラインモデルをシミュレーションにより，評価する．

6.1　評価方法

ARM アーキテクチャの命令セットをベースにした分解型パイプラインモデルに対して，3 つのカスケード型パイプラインモデルを実装し，ベンチマーク MiBench を用いて IPC 向上の効果を測定した．また，3 種類のカスケード型パイプラインモデルの性能比較を SPEC CPU2000 を用いて行った．コンパイラは gcc を用いた．最適化オプションは -O2 である．

MiBench では 17 個のプログラムを使用した．

- **basicmath** :3 次関数や平方根などの簡単な算術計算を行うプログラム
- **bitcount**： 整数列の bit 数を数えることで bit 操作能力をテストするプログラム
- **qsort**： クイックソートで単語列を昇順に整列させるプログラム
- **susan** :核磁気共鳴現象を利用して脳の内部構造を画像認識するプログラム
- **jpeg** :カラー画像を jpeg という静止画像データの圧縮方式を用いて圧縮するプログラム
- **typeset**： ウェブページの HTML データを活字に組むプログラム
- **dijkstra** :隣接行列表現を構成し，ダイクストラのアルゴリズムを繰り返し使って， 2 点間の最短経路を計算するプログラム
- **patricia**： サーバの 2 時間の IP トラフィックを patricia というデータ構造

にするプログラム

- **ghostscript** :PostScript 言語を解釈するプログラム
- **ispell** :文書のスペルチェックを行うプログラム
- **stringsearch** :文字列を探すプログラム
- **blowfish** :ある記事を 32 ビットから 448 ビットまでの可変長の鍵を使い，ブロックごとに暗号化を行うプログラム
- **rijndael** :ある記事を鍵の長さを 128 ビット，192 ビット，256 ビットの中から指定してブロックとして分割して暗号化するプログラム
- **sha** :ある記事を sha というハッシュアルゴリズムを用いて 160 ビットのハッシュ値を生成するプログラム.
- **CRC32** :サウンドファイルに 32 ビットの巡回冗長検査を行うプログラム
- **FFT** :疑似ランダム振幅の多項式等を高速フーリエ変換するプログラム
- **adpcm** : スピーチを適応的差分パルス符号変調という方式を用いてデジタルデータに変換するプログラム

SPEC CPU2000 では，以下の 11 個のプログラムを使用した.

- **164.gzip** :データを gzip 方式で圧縮，解凍するプログラム
- **175.vpr** :FPGA の回路配置と配線を行うプログラム
- **176.gcc** :C 言語をコンパイルするプログラム
- **181.mcf** :最小コストフロー法により最適化を行うプログラム
- **186.crafty** :64 ビット設計のチェスプログラム
- **197.parser** :文書を自然言語処理するプログラム
- **253.perlbmk** :Perl 言語を解釈するプログラム
- **254.gap** :言語，ライブラリを実装するプログラム
- **255.vortex** :オブジェクト指向のデータベース処理を行うプログラム
- **256.bzip2** :データを bzip2 方式で圧縮，解凍するプログラム
- **300.twolf** :シミュレーテッドアニ　リングにより配置配線を行うプログラム

6.2　測定結果

6.2.1　分解型モデルとカスケード型モデルの比較

図 17 に taken 分岐予測方式の場合の MiBench プログラムを用いた各カスケード型モデルと分解型モデルの IPC の値を示す．図には MiBench の各プログラムに対応した 17 組のバーがある．各組は 4 個のバーからなり，左から AP2-t,CS1-

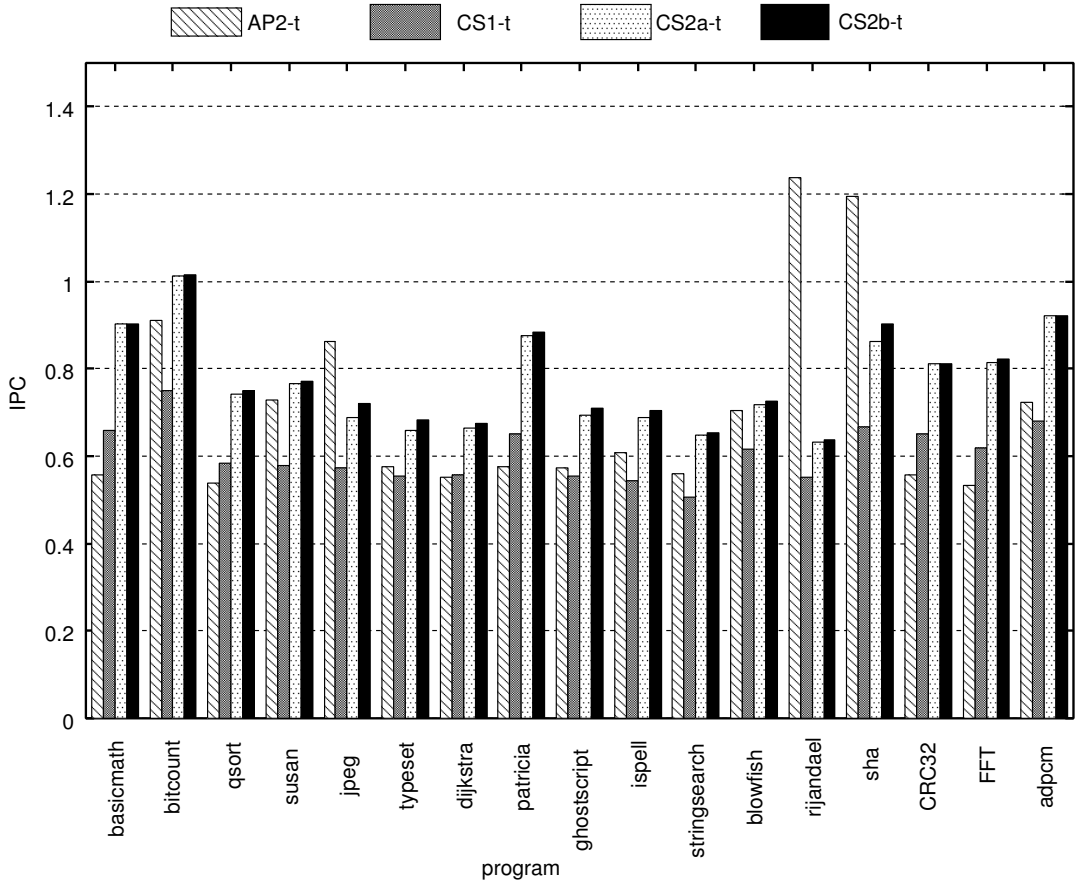


図 17: MiBench における各モデルの IPC

t,CS2a-t,CS2b-t の順に並んでいる．

個々のプログラムを比較した結果，AP2-t の IPC が顕著に向上したプログラムを挙げる．

- jpeg での AP2-t の IPC が CS1-t より約 50%，CS2a-t より約 20%，CS2b-t より約 25% 向上した．
- rijandael での AP2-t の IPC が CS1-t より約 124%，CS2a-t より約 96%，CS2b-t より約 94% 向上した．
- sha での AP2-t の IPC が CS1-t より約 78%，CS2a-t より約 38%，CS2b-t より約 32% 向上した．

個々のプログラムを比較した結果，CS1-t，CS2a-t，CS2b-t の IPC が顕著に向上したプログラムを挙げる．

- basicmath での CS1-t の IPC が AP2-t より約 18%，CS2a-t の IPC が AP2-t

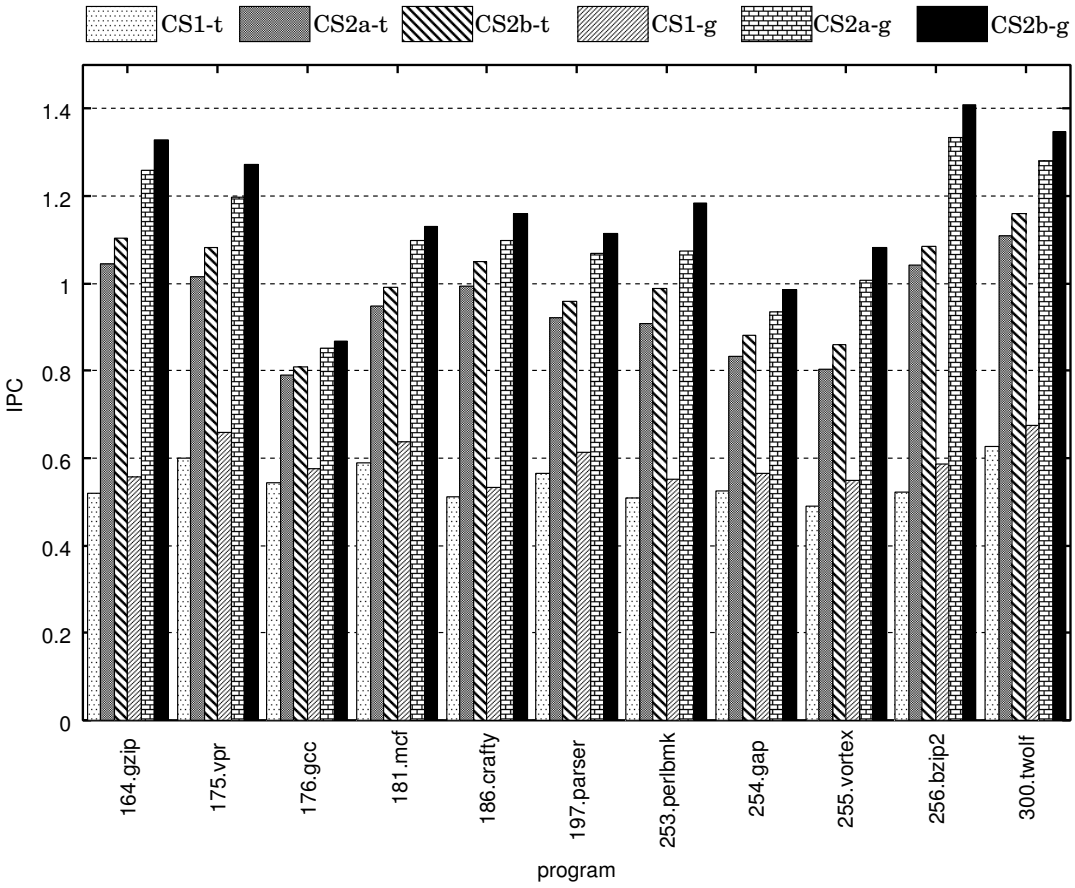


図 18: SPEC CPU2000 を用いた各カスケード型モデルの IPC

より約 61%，CS2b-t の IPC が AP2-t より約 62% 向上した．

- patricia での CS1-t の IPC が AP2-t より約 12%，CS2a-t の IPC が AP2-t より約 51%，CS2b-t の IPC が AP2-t より約 53% 向上した．
- FFT での CS1-t の IPC が AP2-t より約 15%，CS2a-t の IPC が AP2-t より約 52%，CS2b-t の IPC が AP2-t より約 53% 向上した．

全プログラムの幾何平均した IPC は AP2-t は 0.70 となり，CS1-t，CS2a-t，CS2b-t はそれぞれ，0.60，0.77，0.78 となった．

6.2.2 各カスケード型モデルの比較

図 18 に taken 分岐予測方式, gshare 分岐予測方式の場合の SPEC CPU2000 プログラムを用いた各カスケード型モデルの IPC の値を示す．図には SPEC CPU2000 の各プログラムに対応した 11 組のバーがある．各組は 6 個のバーからなり，左から CS1-t,CS2a-t, CS2b-t,CS1-g,CS2a-g,CS2b-g の順に並んでいる．

全プログラムの IPC の幾何平均を求め，比較した．

- CS1-t，CS2a-t，CS2b-t の平均した IPC はそれぞれ 0.54，0.94，0.99 となった．
- CS1-t から CS2a-t にすると平均して IPC が約 87% 向上した．
- CS1-t から CS2b-t にすると平均して IPC が約 98% 向上した．
- CS2b-t から CS2a-t にすると平均して IPC が約 5% 低下した．
- CS1-g，CS2a-g，CS2b-g の平均した IPC はそれぞれ 0.59，1.10，1.17 となった．
- CS1-g から CS2a-g にすると平均して IPC が約 73% 向上した．
- CS1-g から CS2b-g にすると平均して IPC が約 83% 向上した．
- CS2b-g から CS2a-g にすると平均して IPC が約 5% 低下した．

6.3 考察

MiBench を用いて，カスケード型モデルと分解型モデルの性能を比較した結果，CS1-t から AP2-t にすると平均して IPC が約 16% 向上した．AP2-t は，CS1-t と比較して ALU と同等な EAG ユニットを 1 個多く搭載しており，5 章で述べた予想通りの結果となっている．ただし，basicmath などいくつかのプログラムでは，CS1-t の性能が大きく向上した．これは，AP2 のパイプラインは長く，分岐予測失敗によるペナルティが増大したと考えられる．また，rijandael，sha では AP2 の性能が大きく向上した．これは，アウトオブオーダ実行による効果が大きいと考えられる．AP2-t から CS2a-t にすると平均して IPC が約 10% 向上した．CS2a-t は，AP2-t と比較して SFT/MUL ユニットを 1 個多く搭載しており，5 章で述べた予想通りの結果となっている．AP2-t から CS2b-t にすると平均して IPC が約 11% 向上したが，CS2b-t は，AP2-t と比較して SFT/MUL ユニット，LD/ST ユニットを 1 個ずつ多く搭載しており，5 章では大幅に IPC が向上すると予想していたが，CS2a-t とそれほど IPC 向上率は変わらなかった．以上の結果より，AP2 は CS と比較して 5 章で述べたようにレジスタのトランジスタ数が 7 倍以上多くなることを考慮すると，AP2 を採用して性能向上を求めるより，CS を採用した方が性能を維持したまま，よりシンプルなコアを実現できる．

SPEC CPU2000 を用いて，各カスケード型モデルを性能比較した結果，CS1 から CS2b にすると平均して IPC が約 90% 向上したが，CS2b は，CS1 と比較してすべての演算ユニットが 2 倍になっており，5 章で述べた IPC が約 2 倍になる

という予想に近い値となっている．IPCが2倍にならなかったのは，依存によるインタロックのためである．CS1からCS2aにすると平均してIPCが約80%向上したが，CS2aは，CS1と比較してSFT/MULニット，ALUを1個ずつ多く搭載しており，5章で述べた予想通りの結果となっている．CS2bからCS2aにすると平均してIPCが約5.1%低下したが，CS2bは，CS2aと比較してLD/STユニットを1個余分に搭載しているものの，5章で述べたような大きな効果は得られなかった．LD/STユニットを1個多く搭載し，コア面積を大きくするCS2bを採用するよりは，よりシンプルなコアを実現するCS2aを採用する方が本研究の目的に沿っている．

2つの測定結果から，マルチコアにおけるコアの検討をした場合，分解型モデルを採用するより，カスケード型モデルを採用した方が，シンプルなコアを実現できる．さらに，性能向上とシンプルなコア設計を考慮すると，カスケード型モデルの中でもCS2aがより要求を満たすコアを実現できるといえる．

第7章　おわりに

本稿ではマルチコアにおける1つのコアとして，ARMアーキテクチャを用いた2つのモデルを検討した．1つは，3章に述べた分解型モデルであり，ARM命令をRISC型命令へ分解し，並列実行するモデル(AP2)である．もう1つは，4章に述べたカスケード型モデルであり，CISC型パイプラインを並置して，ARM命令を並列実行するモデルである．このモデルには同一のパイプラインを並置したモデル(CS2b)の他に，LD/STユニットを1つだけ搭載するモデル(CS2a)も検討した．

ベンチマークプログラムMiBenchを用いて分解型モデルとカスケード型の性能を比較した結果，CS1ではAP2より14%IPCが低下したものの，CS2a，CS2bではそれぞれ10%，11%向上した．分解型モデルは，カスケード型モデルと比較してレジスタの総トランジスタ数が7倍以上多くなることを考慮すると，カスケード型モデルを採用した方がよりシンプルなコアを実現できることが分かった．SPEC CPU2000を用いて各カスケード型モデルの性能を比較した結果，CS2bはCS2aに対して，約5.1%の性能向上しか得られなかった．CS2aの構成がマルチコアに向いているといえる．

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚なる謝意を表します．また本研究に関して適切なご指導を賜わった中島康彦助教授，森眞一朗助教授，嶋田創特任助手，三輪忍助手に心から感謝いたします．

さらに，日頃からご討論頂いた京都大学情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝いたします．

参考文献

[1] Steve Furber：ARM プロセッサ,CQ 出版 (2001)

[2] 神保進一：マイクロプロセサテクノロジ, 日経 BP 社 (1999)

[3] 安藤秀樹：命令レベル並列処理, コロナ社 (2005)

[4] 中島康彦, 五島正裕, 森眞一郎, 富田眞治：外部連想バッファを備える SpMT モデルの分析, 先進的計算基盤システムシンポジウム SACSIS 2005,pp. 397-406 (2005)

[5] Neil H.E.West,Kamran Eshraghian, 富沢孝, 松山秦男：CMOS VLSI 設計の原理 システムの視点から, 丸善 (1999)