目次

図目次

表目次

内容梗概

本論文ではマルチ ALU プロセッサにおけるシミュレータの設計と試作を行った。本研究では、MAP シミュレータを利用し、アセンブリプログラムのデバッグと並列・連鎖演算、単一実行判別の有効性を示す事を目的とする。Booth の乗算プログラムについて正しく動作し、有用性を確認した。また同時に動的に並列・連鎖演算判定を行い、それぞれのプログラムでどれくらいこれらの演算が使われているのか評価した。その結果、1 次 2 次 3 次 Booth のアルゴリズムでは2ALU で単一が平均 28%動作し、並列演算、連鎖演算は合計 72%動作した。4ALUMAP では並列演算、連鎖演算で 95%動作していた。このことより4ALU プロセッサの有用性を確認できた。

1　はじめに

半導体技術の進歩により、LSI の小型化、軽量化と高速化、低消費電力化が可能となった。近年の発展が見られるスマートフォンなどに挙げられる組み込み機器は、ハードウェアとソフトウェアから構成されている。この普及に伴い半導体は高性能、低消費電力化が加速している。そして要求される仕様は大規模かつ複雑・専用化され、多様性が必要とされている。このようにハードとソフト両方の知識に加え、プロセッサにおける命令セットとコンピュータアーキテクチャの知識が必要不可欠である。

本研究室では、ハード/ソフト協調学習システムを考案し、開発を進めてきた。ハード/ソフト協調学習システムとは、プロセッサを通してハードとソフトの両方の学習を進めていくことを目的としたシステムである。

マルチ ALU プロセッサ MAP とは複数の ALU による並列処理が可能なプロセッサである。演算の種類は、依存関係のない並列演算、依存関係のある連鎖演算などがある。ALU で演算した結果を 32 個の共有レジスタファイルに格納する。1 つの ALU を 32 ビットで制御しており、並列処理する命令数に応じて ALU 数を変更することができる。

本研究では MAP アセンブラを試作し、MAP アセンブラはテキスト形式の MAP アセンブリプログラムを入力として、1 命令ごとに変換を行い、最終的には機械語プログラムを出力する。それに加え、4ALUMAP シミュレータの試作についての考察をすることで現段階完成している2ALUMAP シミュレータでは処理完了までに時間を要する大きなプログラムの更なる速度向上について検証を行った。

本研究では、Booth の乗算プログラムを3種類アセンブルし、出力された機械語プログラムが正しいかどうかを確認しアセンブラの検証、評価を行う。Booth 乗算のプログラムを ALU の数を変更し、MAP の ALU 数の増加による有用性の検証を行う。

第2章では、MAP の特徴、命令セットアーキテクチャについて述べ、以降に使う記述の説明をしている。第3章では Booth のアルゴリズムについて、概要の説明、例題を用いて1次2次3次と Booth の乗算アルゴリズムを説明する。Booth のアルゴリズムは乗算の速度を向上させるアルゴリズムである。第4章では、シミュレーションによる、並列化の評価を記述した。Booth のアルゴリズムをテストデータとして、2ALU と4ALU のシミュレーションを用いてハンドシミュレーションの結果を比較し、2ALU、4 ALUMAP の並列性の評価を連鎖ができる場合とできない場合について行い、考察した。

2　マルチ ALU プロセッサ MAP [2][3][4][5]

2.1　MAP の構成

⑴MAP の特徴

① マルチ ALU プロセッサ MAP とは複数の ALU を有し並列処理が可能である。

② レジスタファイルは全 32 個であり、全 ALU で共有される。

③ ALU で並列演算、連鎖演算を行う。

④ 1ALU を 32 ビットで処理する。

⑵データパス

図 1 に MAP のデータパスを示す。

・PC(プログラムカウンタ):次に実行する命令のアドレスを保持するレジスタ。

・IM(命令メモリ):実行する命令を保持するメモリ。

・RF(レジスタファイル):演算結果を保持し、オペランドに用いるレジスタ。

・ALU(演算機):算術論理演算を行う回路。

・DM(データメモリ):プロセッサで演算するための入力データを保持するメモリ。

・PPU(Parallel Processing Unit):並列実行を判断する並列連鎖単一処理判定部。

・CU(コントロールユニット):命令ごとによって違うデータパスを管理し、各種レジスタ、メモリの動作をコントロールする制御部。

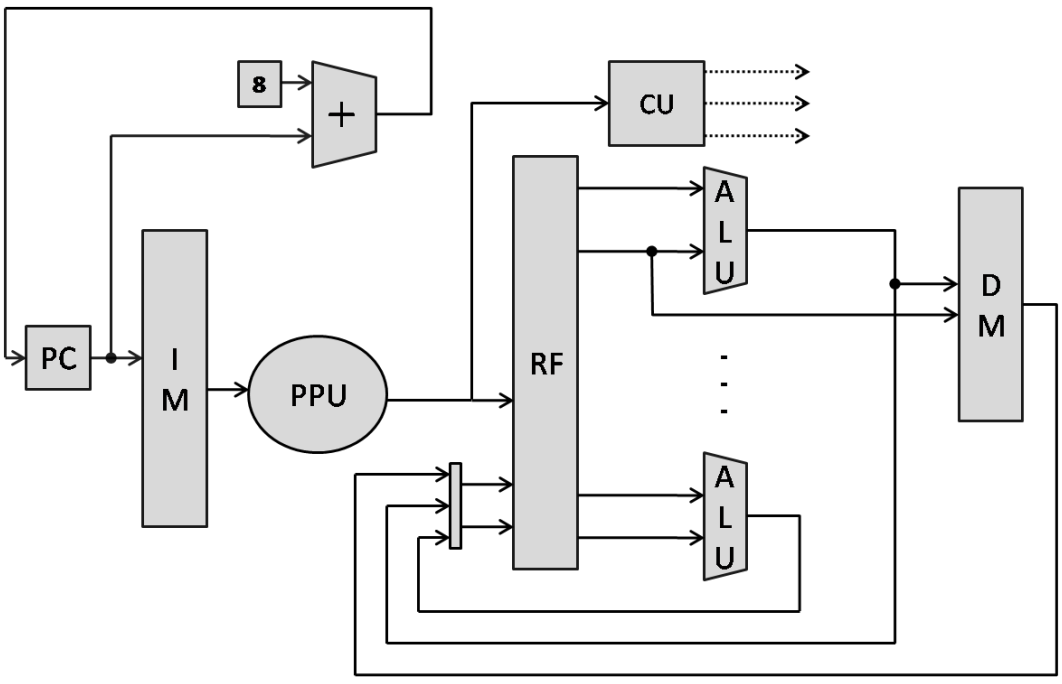

図 1:MAP のデータパス

並列処理する命令数に応じて ALU の数を変更する。

ハーバードアーキテクチャ(命令メモリとデータメモリの分類)、バイトアドレス方式(バイトごとのアドレッシング方式)などの特徴がある。また並列演算とは命令同士に依存関係のない場合に行われる並列処理であり、連鎖演算とは命令同士に依存関係のある場合に行われる演算処理である。

2.2 MAP の命令セットアーキテクチャ

(1)命令形式

MAP には、Jump 形式(J 形式)、Register 形式(R 形式)、Immediate 形式(I 形式)、Long 形式(L 形式)の 4 つの命令形式がある。

J 形式には無条件分岐命令の JUMP、プログラム終了命令となる HALT を定義している。R 形式にはレジスタ間の演算を行う命令を定義している。I 形式にはレジスタ値と即値演算を行う命令を定義している。L 形式には条件分岐命令とメモリ・レジスタへのデータ転送命令を定義している。表1に命令セットアーキテクチャを示す。

表1:命令セットアーキテクチャ

命令語長		32						命令種類
		6	5	5	5	5	6	
命令形式	R形式	Op	Rs	Rt	Rd	Shamt	Fn	レジスタ同士による演算
	I形式	Op	Rs	Rd	imm			レジスタと即値による演算
	L形式	Op	Rs	Rd	address/immediate			メモリアクセスや分岐命令
	J形式	Op	address					JUMP、HALT命令

以下に各命令フィールドの意味を示す。

- Op(Opecode) ・・・ オペコード、命令を識別
- Fn(Function) ・・・ ファンクション、R 形式命令を詳細に識別
- Rs ・・・ ソースレジスタ
- Rt ・・・ ソースレジスタ
- Rd ・・・ 演算結果を格納するレジスタ
- Shamt ・・・ シフト量
- Imm(Immediate) ・・・ 即値
- Address ・・・ アドレス

⑵アドレス指定方式と命令セット

R 形式ではレジスタ直接、I 形式ではレジスタ即値、L 形式でメモリアクセスはベース相対アドレス、分岐命令は絶対アドレス、J 形式は絶対アドレスである。

表 2 に MAP 命令セットの一覧と動作の内容を示す。

表2:MAP 命令セット一覧

	命令	Op	Rs/Rt/Rd			Shamt	Fn	動作
R	NOP	0	XXXX				0	No operetion
	ADD	1	Rs	Rt	Rd	X	0	Rd = Rs+Rt
	SUB	1	Rs	Rt	Rd	X	1	Rd = Rs-Rt
	AND	1	Rs	Rt	Rd	X	10	Rd = Rs&Rt
	OR	1	Rs	Rt	Rd	X	11	Rd = Rs|Rt
	XOR	1	Rs	Rt	Rd	X	100	Rd = Rs^Rt
	NOT	1	Rs	X	Rd	X	101	Rd = !(Rs)
	SLT	10	Rs	Rt	Rd	X	0	(if Rs < Rt) Rd = 1
	SGT	10	Rs	Rt	Rd	X	1	(if Rs > Rt) Rd = 1
	SLE	10	Rs	Rt	Rd	X	10	(if Rs ≦ Rt) Rd = 1
	SGE	10	Rs	Rt	Rd	X	11	(if Rs ≧ Rt) Rd = 1
	SEQ	10	Rs	Rt	Rd	X	100	(if Rs = Rt) Rd = 1
	SNE	10	Rs	Rt	Rd	X	101	(if Rs ≠ Rt) Rd = 1
	SLL	11	Rs	X	Rd	Shamt	0	Rd = Rs << Shamt
	SRL	11	Rs	X	Rd	Shamt	1	Rd = Rs >> Shamt
	SRA	11	Rs	X	Rd	Shamt	10	Rd = Rs >>> Shamt
	JR	100	Rs	X			0	PC = Rs
I	ADDI	1000	Rs	Rd	Imm(16bit)			Rd = Rs+imm
	SUBI	1001	Rs	Rd	imm			Rd = Rs-imm
	ANDI	1010	Rs	Rd	imm			Rd = Rs&imm
	ORI	1011	Rs	Rd	imm			Rd = Rs|imm
	XORI	1100	Rs	Rd	imm			Rd = Rs^imm
	SLTI	1101	Rs	Rd	imm			(if Rs < imm) Rd = 1
	SGTI	1110	Rs	Rd	imm			(if Rs > imm) Rd = 1
	SLEI	1111	Rs	Rd	imm			(if Rs ≦ imm) Rd = 1
	SGEI	10000	Rs	Rd	imm			(if Rs ≧ imm) Rd = 1
	SEQI	10001	Rs	Rd	imm			(if Rs = imm) Rd = 1
	SNEI	10010	Rs	Rd	imm			(if Rs ≠ imm) Rd = 1
	LDHI	10011	Rs	Rd	imm			Rd = {imm,Rs[15:0]}
	LDLI	10100	Rs	Rd	imm			Rd = {Rs[31:16],imm}
L	BEQZ	100000	Rs	X	imm			(if Rs = 0) PC = imm
	BNEZ	100001	Rs	X	imm			(if Rs ≠ 0) PC = imm
	LD	100010	Rs	Rd	imm			Rd = DM[Rs+imm]
	ST	100011	Rs	Rd	imm			DM[Rs+imm] = Rd
	JAL	100100	X	Rd	imm			PC = imm:Rd = PC+4
J	JUMP	111110	Imm(26bit)					PC = imm
	HALT	111111	XXXX					exit

＊X は Don't care を意味しており使用しないビット分だけ 0 が入る。

⑶疑似命令

MAP には疑似命令として COPY と CLEAR の 2 種類が用意されている。

・ COPY Rs Rd ・・・ Rs に Rd の内容をコピー ADDI Rs Rd 0

・ CLEAR Rs ・・・ Rs の内容を初期化 SUB Rs Rs Rs (Rs に 0 を設定)

2.3　並列・連鎖・単一実行判定のアルゴリズム

2.3.1　並列・連鎖演算と単一実行について

MAP(マルチ ALU プロセッサ)は、並列処理が可能であり、2命令を判定して並列・連鎖演算、単一実行など命令に合わせて処理を変えていく。この三つの命令について説明する。

(1)並列演算

並列演算は、2命令に関して依存関係がないときに行う。以下に例を記述する。

例

上位命令　　ADD $1 $2 $3

下位命令　　SUB $3 $2 $4

上のような2命令を判定すると、上位命令の Rd にあたる、$1 と下位命令の Rs/Rt にあたる$2/$4を比較した場合、特に依存関係はなく、並列演算することが可能と判断される。

図 2 に並列演算のデータパスを示す。

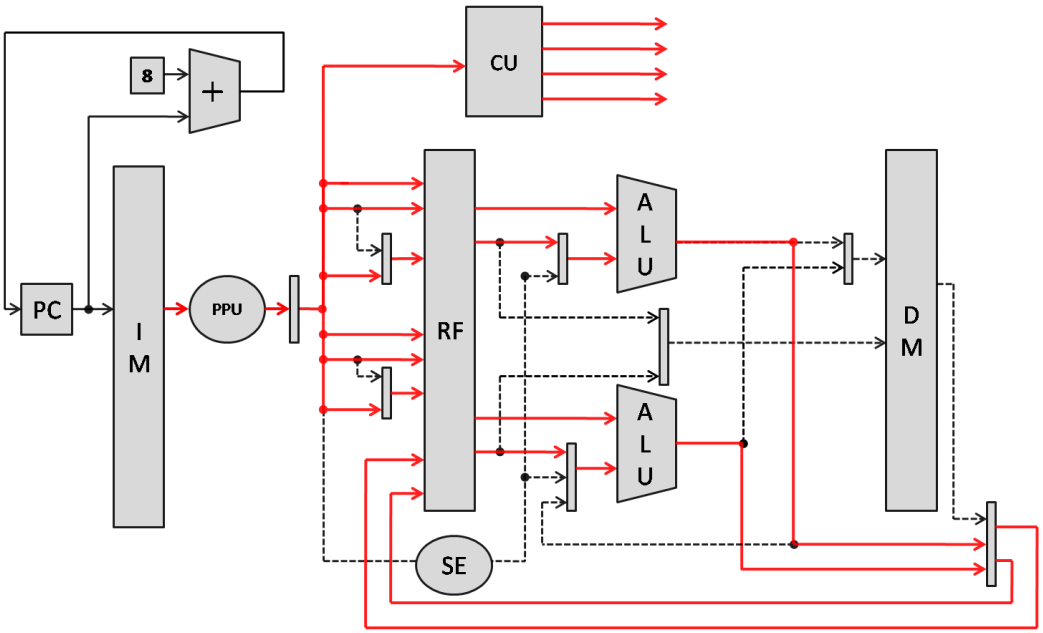

図2：並列演算のデータパス

(2)連鎖演算

連鎖演算は、2命令に関して依存関係があるときに行う。以下に例を記述する。

例

上位命令:　　ADD $1 $2 $3

下位命令:　　SUB $3 $1 $4

上のような2命令を判定する。上位命令の Rd に当たる、$1(汎用レジスタ:1)と下位命令の Rs/Rt に当たる、$1 と$4(汎用レジスタ:4)を比較する。すると、上位命令で使われた$1 が下位命令で扱われているため、この2命令は依存関係があり、連鎖演算が必要となる。

図 3 に連鎖演算のデータパスを示す。

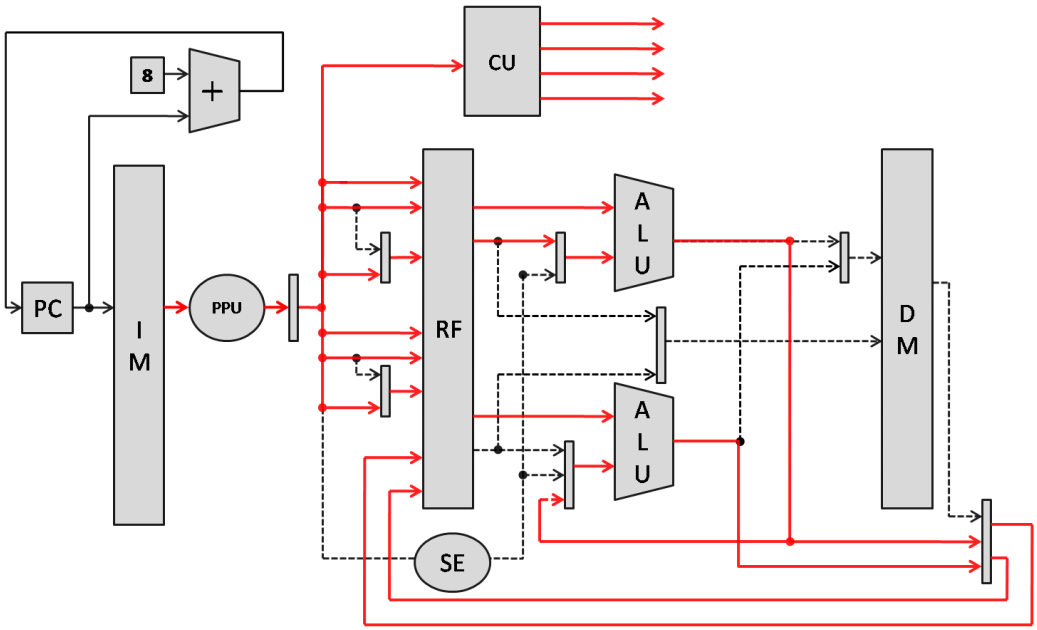

図3：連鎖演算のデータパス

(3)単一演算

単一演算が必要となる場面は 2 つだけである

① J 形式(JUMP・HALT)の命令が実行されるとき

② 条件分岐命令(BEQ・BENZ 等)が実行されるとき

3 Booth 乗算のアルゴリズム [1]

3.1 1次 Booth 乗算アルゴリズム

(1) 原理

乗数(X)、被乗数(Y)と定義された計算式は以下のようになる。

$$X \cdot Y = X \cdot (-Y_n 2^n + Y_{n-1}2^{n-1} + Y_{n-2}2^{n-2} + \cdots + Y_1 2^1 + Y_0 2^0)$$

$$= X \cdot \{(-Y_n + Y_{n-1})2^n + (-Y_{n-1} + Y_{n-2})2^{n-1} + \cdots + (-Y_0 + Y_{-1})2^0\}$$

ただし、$Y_{-1}=0$

手順

① 乗数 X(4bit)と被乗数 Y(4bit)を決める。

② 下位 2bit 毎に表3の 1 次 booth デコード表に従って部分積 PP_iを求める。

③ PP_iをそれぞれ 1bit ずつ左算術シフトさせる。

④ PP_iを加算して積を求める。

表3:1 次 Booth デコード表

Y_n	Y_{n-1}	部分積(PP_i)
0	0	0
0	1	+X
1	0	-X
1	1	0

(2) 例題

X=2(0010)，Y=−3(1101)の場合

① X、Y の値を2進4bit で表す。

② Y_{-1} = 0を追加し、Y_{-1}=0 Y_0=1 Y_1=0 Y_2=1 Y_3=1 となる。

③ Y を下位2bit 毎に表3の 1 次 booth デコード表と比較して対応する部分積PP_iを求める。PP_0=−X=1110、PP_1=+X=0010, PP_2=−X=1110, PP_4=0=0000 となる。

④ PP_iを i bit 分左算術シフトし、符号 bit 拡張を行う。

PP_0=−X=1111110、PP_1=+X=0000010, PP_3=−X=1111000, PP_4=0=0000000 となる。

⑤ $PP_0 + PP_1 + PP_2 + PP_3$ を行い、最終的な積を求める。

今回の X=2(0010)，Y=−3(1101)の場合は 1111101 となり X・Y＝-6 となる

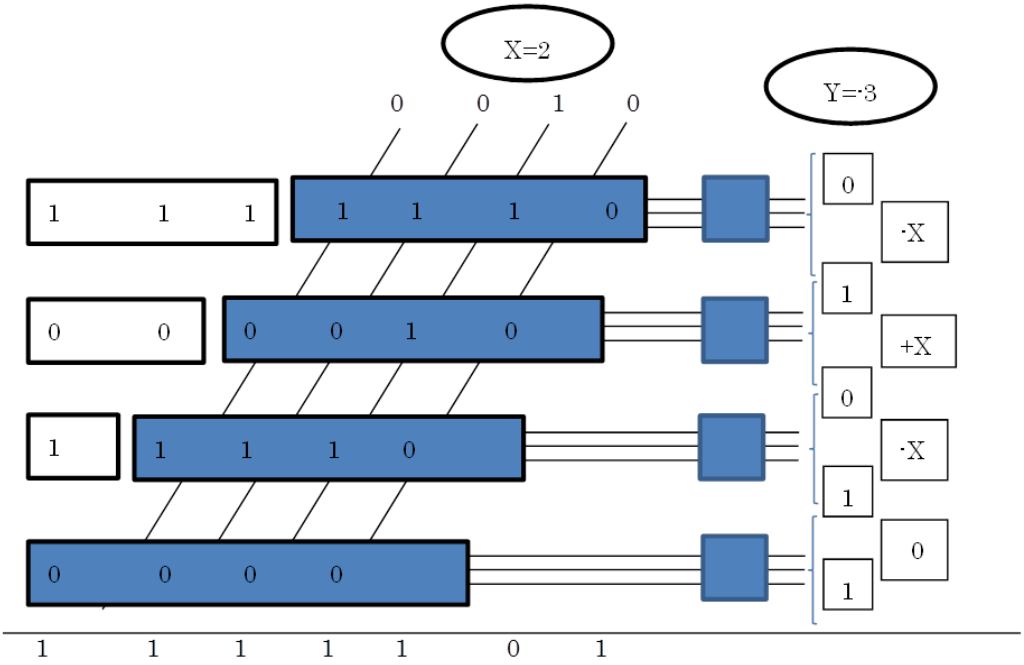

図 4:1 次 booth 乗算器(X=2、Y=-3)

3.2　2 次 Booth 乗算アルゴリズム

(1) 原理

乗数(X)、被乗数(Y)と定義された計算式は以下のようになる。

$$X \cdot Y = X \cdot (-Y_n 2^n + Y_{n-1} 2^{n-1} + Y_{n-2} 2^{n-2} + \cdots + Y_1 2^1 + Y_0 2^0)$$

$$= X \cdot \{ (-2Y_n + -Y_{n-1} + -Y_{n-2}) 2^{n-1} + (-2Y_{n-2} + -Y_{n-3} + -Y_{n-4}) 2^{n-3} + \cdots + (-2Y_1 + -Y_0 + -Y_{-1}) 2^0 \}$$

ただし、$Y_{-1}=0$

手順

① 乗数 X(4bit)と被乗数 Y(4bit)を決める。

② 下位 3bit 毎に表4の 2 次 booth デコード表に従って部分積 PP_iを求める。

③ PP_iをそれぞれ 2bit ずつ左算術シフトさせる。

④ PP_iを加算して積を求める。

表4:2 次 booth デコード表

Y_{i+1}	Y_i	Y_{i-1}	部分積
0	0	0	0
0	0	1	X
0	1	0	X
0	1	1	2X
1	0	0	-2X
1	0	1	-X
1	1	0	-X
1	1	1	0

⑵ 例題

X=2(0010)，Y=3(0011)の場合

手順

① X、Y の値を2進4bit で表す。

② Y_{-1} = 0を追加し、Y_{-1}=0 Y_0=0 Y_1=0 Y_2=1 Y_3=1 となる。

③ Y を下位 3bit 毎に表 4 の 1 次 booth デコード表と比較して対応する部分積PP_iを求める。PP_0=+X=0010、PP_1=-X=1110, PP_3=+X=0010, PP_4=0=0000 となる。

④ PP_iを 2i bit 分左算術シフトし、符号 bit 拡張を行う。

PP_0=+X=0000010、PP_1=+2X=0010000 となる。

⑤ PP_0 + PP_1 を行い、最終的な積を求める。

今回の X=2(0010)，Y=3(0011)の場合は 0010010 となり X・Y＝6 となる。

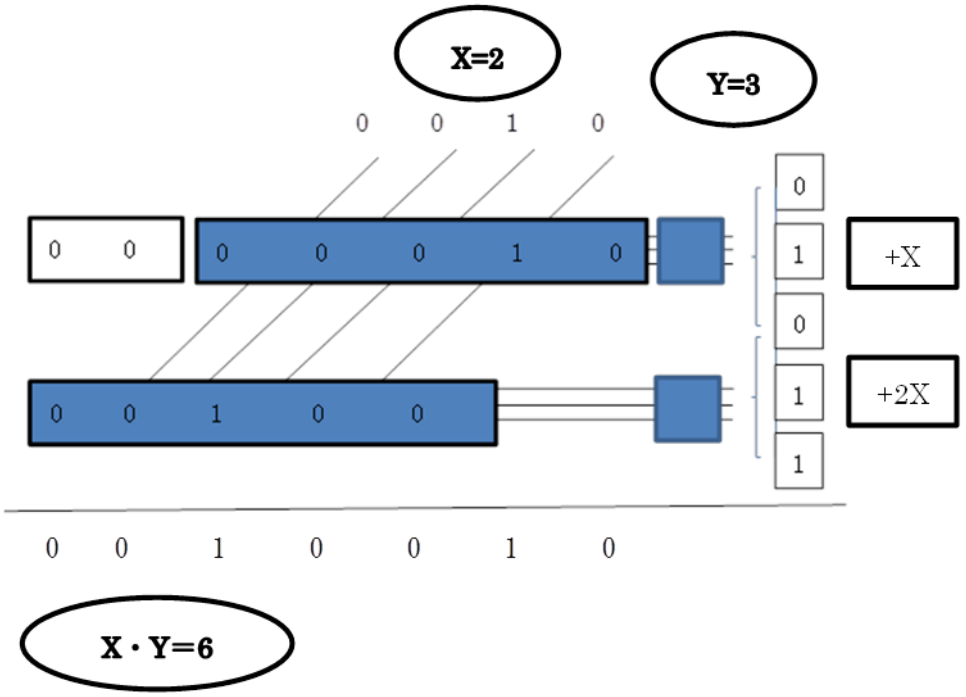

図 5:2次 booth 乗算器(X=2、Y=3)

3.3　3 次 Booth 乗算アルゴリズム

(1) 原理

乗数(X)、被乗数(Y)と定義された計算式は以下のようになる。

$$X \cdot Y = X \cdot (-Y_n 2^n + Y_{n-1} 2^{n-1} + Y_{n-2} 2^{n-2} + \cdots + Y_1 2^1 + Y_0 2^0)$$

$$= X \cdot \{ (-2Y_n + -Y_{n-1} + -Y_{n-2}) 2^{n-1} + (-2Y_{n-2} + -Y_{n-3} + -Y_{n-4}) 2^{n-3} + \cdots + (-2Y_1 + -Y_0 + -Y_{-1}) 2^0 \}$$

ただし、$Y_{-1}=0$

手順

① 乗数 X(6bit)と被乗数 Y(6bit)を決める。

② 下位 4bit 毎に表5の 3 次 booth デコード表に従って部分積 PP_iを求める。

③ PP_iをそれぞれ 3ibit ずつ左算術シフトさせる。

④ PP_iを加算して積を求める。

表5:3 次 booth デコード表

Y_{i+1}	Y_i	Y_{i-1}	Y_{i-2}	PP_i
0	0	0	0	0
0	0	0	1	X
0	0	1	0	X
0	0	1	1	2X
0	1	0	0	2X
0	1	0	1	3X
0	1	1	0	3X
0	1	1	1	4X
1	0	0	0	−4X
1	0	0	1	−3X
1	0	1	0	−3X
1	0	1	1	−2X
1	1	0	0	−2X
1	1	0	1	−X
1	1	1	0	−X
1	1	1	1	0

⑵ 例題

X=21(010101), Y=10()の場合

手順

① X、Y の値を2進6bit で表す。

② Y_{-1} = 0を追加し、Y_{-1}=0 Y_0=0 Y_1=0 Y_2=1 Y_3=0 Y_4=0 Y_5=0 となる。

③ Y を下位 4bit 毎に表5の 3 次 booth デコード表と比較して対応する部分積PP_iを求める。PP_0=−4X=111000、PP_1=+X=000010 となる。

④ PP_iを 3i bit 分左算術シフトし、符号 bit 拡張を行う。

PP_0=−4X=11111111000、PP_1=+X=00000010000 となる。

⑤ PP_0 + PP_1 を行い、最終的な積を求める。

今回の X=2(000010)，Y=4(000100)の場合は 000000001000 となり X・Y＝8 となる。

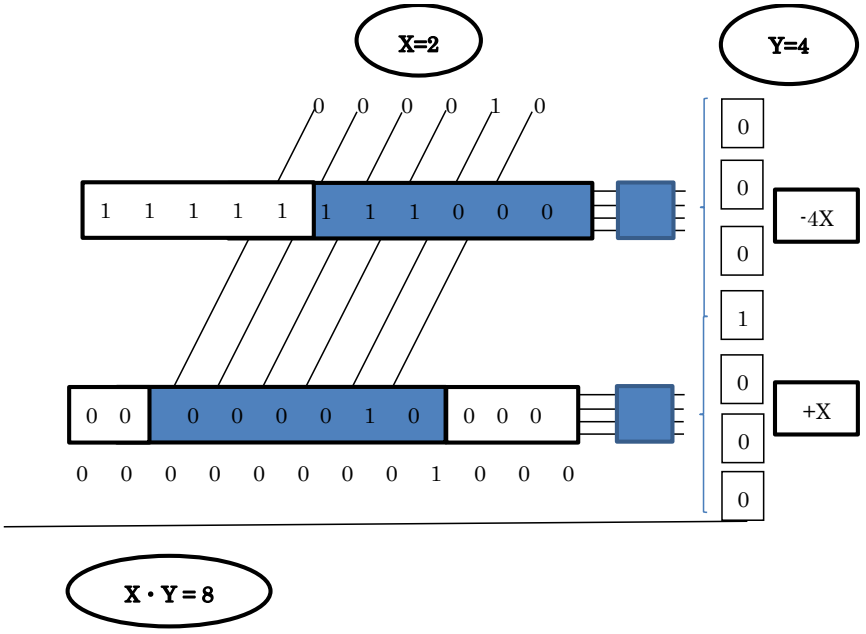

図 6：3次 booth 乗算器（X=2、Y=4）

4. シミュレーションによる並列化の評価

4. 1 Booth 乗算のアセンブリプログラムと並列性の評価

(1) 1 次 Booth 乗算アセンブリプログラム

1		LD	$1	0[$0]	
2		LD	$2	4[$0]	
3		ANDI	$2	$2	15
4		SLL	$1	$1	5
5		SLL	$2	$2	1
6	LOOP:	ADDI	$15	$15	1
7		SEQI	$16	$15	5
8		BNEZ	$16	LAST	
9		ANDI	$3	$2	3
10		SEQI	$4	$3	2
11		BNEZ	$4	SKIP1	
12		SEQI	$5	$3	1
13		BNEZ	$5	SKIP2	
14		SRL	$2	$2	1
15		JUMP	LOOP		
16	SKIP1:	SUB	$2	$2	$1
17		SRL	$2	$2	1
18		JUMP	LOOP		
19	SKIP2:	ADD	$2	$2	$1
20		SRL	$2	$2	1
21		JUMP	LOOP		
22	LAST:	SRL	$2	$2	1
23		ST	$2	12[$0]	
24		HALT			

図 7:1 次 Booth 乗算アセンブリプログラム

1 次 Booth の乗算プログラム(4bit×4bit)のアルゴリズムは LOOP の記述でデコード表(表3)との比較を行い$3の値が 2(10)なら SKIP1 に分岐し、SKIP1 で SUB $2 $2 $1 を行い$3の値が 1(01)なら SKIP2 で ADD $2 $2 $1 を、行い部分積を求め、それぞれ算術左 1bit シフトを行い、部分積を求める。部分積を加算していき、4 回繰り返して、最終的な積を$2に格納する。

⑵2 次 Booth 乗算アセンブリプログラム

1		LD	$1	0[$0]	
2		LD	$2	4[$0]	
3		ANDI	$2	$2	15
4		SLL	$1	$1	5
5		SLL	$2	$2	1
6	LOOP:	ADDI	$15	$15	1
7		SEQI	$16	$15	3
8		BNEZ	$16	LAST	
9		ANDI	$3	$2	7
10		SEQI	$5	$3	1
11		BNEZ	$5	SKIP1	
12		SEQI	$4	$3	2
13		BNEZ	$4	SKIP1	
14		SEQI	$5	$3	3
15		BNEZ	$5	SKIP3	
16		SEQI	$5	$3	4
17		BNEZ	$5	SKIP4	
18		SEQI	$5	$3	5
19		BNEZ	$5	SKIP2	
20		SEQI	$5	$3	6

21		BNEZ	$5	SKIP2	
22		SRL	$2	$2	2
23		JUMP	LOOP		
24	SKIP1:	ADD	$2	$2	$1
25		SRL	$2	$2	2
26		JUMP	LOOP		
27	SKIP2:	SUB	$2	$2	$1
28		SRL	$2	$2	2
29		JUMP	LOOP		
30	SKIP3:	ADD	$2	$2	$1
31		ADD	$2	$2	$1
32		SRL	$2	$2	2
33		JUMP	LOOP		
34	SKIP4:	SUB	$2	$2	$1
35		SUB	$2	$2	$1
36		SRL	$2	$2	2
37		JUMP	LOOP		
38	LAST:	SRL	$2	$2	1
39		ST	$2	12[$0]	
40		HALT			

図 8：2次 Booth 乗算アセンブリプログラム

2 次 Booth の乗算プログラム(4bit×4bit)のアルゴリズムは LOOP の記述で 2 次 Booth デコード表（表 4）との比較を行い$3の値によって、分岐する場所を比較で選択し、部分積を求め、それぞれ算術左 2bit シフトを行い、部分積を求める。部分積を加算していき、2 回繰り返して、最終的な積を$2に格納する。

⑶3 次 Booth 乗算アセンブリプログラム

```
 1        LD    $1   0[$0]
 2        LD    $2   4[$0]
 3        ANDI  $2   $2   63
 4        SLL   $1   $1   7
 5        SLL   $2   $2   1
 6 LOOP:  ADDI  $15  $15  1
 7        SEQI  $16  $15  3
 8        BNEZ  $16  LAST
 9        ANDI  $3   $2   15
10        SEQI  $5   $3   1
11        BNEZ  $5   SKIP1
12        SEQI  $4   $3   2
13        BNEZ  $4   SKIP1
14        SEQI  $5   $3   3
15        BNEZ  $5   SKIP3
16        SEQI  $5   $3   4
17        BNEZ  $5   SKIP3
18        SEQI  $5   $3   5
19        BNEZ  $5   SKIP5
20        SEQI  $5   $3   6
21        BNEZ  $5   SKIP5
22        SEQI  $5   $3   7
23        BNEZ  $5   SKIP7
24        SEQI  $5   $3   8
25        BNEZ  $5   SKIP8
26        SEQI  $5   $3   9
27        BNEZ  $5   SKIP6
28        SEQI  $5   $3   10
29        BNEZ  $5   SKIP6
30        SEQI  $5   $3   11
31        BNEZ  $5   SKIP4
32        SEQI  $5   $3   12
33        BNEZ  $5   SKIP4
34        SEQI  $5   $3   13
35        BNEZ  $5   SKIP2
36        SEQI  $5   $3   14
37        BNEZ  $5   SKIP2
38        SRL   $2   $2   3
39        JUMP  LOOP
40 SKIP1: ADD   $2   $2   $1
41        SRL   $2   $2   3
42        JUMP  LOOP
43 SKIP2: SUB   $2   $2   $1
44        SRL   $2   $2   3
45        JUMP  LOOP
46 SKIP3: ADD   $2   $2   $1
47        ADD   $2   $2   $1
48        SRL   $2   $2   3
49        JUMP  LOOP
50 SKIP4: SUB   $2   $2   $1
51        SUB   $2   $2   $1
52        SRL   $2   $2   3
53        JUMP  LOOP
54 SKIP5: ADD   $2   $2   $1
55        ADD   $2   $2   $1
56        ADD   $2   $2   $1
57        SRL   $2   $2   3
58        JUMP  LOOP
59 SKIP6: SUB   $2   $2   $1
60        SUB   $2   $2   $1
61        SUB   $2   $2   $1
62        SRL   $2   $2   3
63        JUMP  LOOP
64 SKIP7: ADD   $2   $2   $1
65        ADD   $2   $2   $1
66        ADD   $2   $2   $1
67        ADD   $2   $2   $1
68        SRL   $2   $2   3
69        JUMP  LOOP
70 SKIP8: SUB   $2   $2   $1
71        SUB   $2   $2   $1
72        SUB   $2   $2   $1
73        SUB   $2   $2   $1
74        SRL   $2   $2   3
75        JUMP  LOOP
76 LAST:  SRL   $2   $2   1
77        ST    $2   12[$0]
78        HALT
```

図 9：3次 Booth 乗算アセンブリプログラム

3 次 Booth の乗算プログラム(6bit×6bit)のアルゴリズムは LOOP の記述で 3 次 Booth デコード表（表 5)との比較を行い$3の値によって、分岐する場所を比較で選択し、部分積を求め、それぞれ算術左 3bit シフトを行い、部分積を求める。部分積を加算していき、2 回繰り返して、最終的な積を$2に格納する。

4. 2 2ALU シミュレータによる並列化

4.2.1 ハンドシミュレーションの条件

a. 単一の場合

上位に J 形式(JUMP 命令・HALT 命令)、条件分岐命令(BEQZ 等)が格納されたとき。

b. 2ALU 連鎖の場合

連鎖演算は、2命令に関して依存関係があるときに行う。以下に例に用いて説明を行う。

上位命令 ADDI $15 $15 1 ($15 = $15 + 1)

下位命令 SEQI $16 $15 5 ($15 = 5 → $16 = 1)

この2命令は$15 の値を上位命令で変更し、その変更された$15 を下位命令で使っているので、命令間で依存関係が生じている。これで連鎖演算が可能だと判断される。

c. 2ALU 並列演算の場合

並列演算は、2命令に関して依存関係がないときに行う。以下に例に用いて説明を行う。

上位命令 ANDI $2 $2 15 ($2 = $2 & 15)

下位命令 SLL $1 $1 5 ($1 = $1 << 5)

上のような2命令を判定すると、上位命令の$2 が下位命令と特に依存関係はなく、並列演算することが可能と判断される。

4.2.2 で 1 次 Booth 乗算アセンブリプログラムの動作を例に用い、ハンドシミュレーション例を提示する。

4.2.2　1次 Booth 乗算のハンドシミュレーション

Booth の乗算は Y の値によって動作が変わるのでここでは Y=-3 の場合を例にする。

番号	命令						2並列	2連鎖	単一
1		LD	$1	0[$0]			1 2		
2		LD	$2	4[$0]					
3		ANDI	$2	$2	15		3 4		
4		SLL	$1	$1	5				
5		SLL	$2	$2	1		5 6		
6	LOOP:	ADDI	$15	$15	1				
7		SEQI	$16	$15	5			7 8	
8		BNEZ	$16	LAST					
9		ANDI	$3	$2	3			9 10	
10		SEQI	$4	$3	2				
11		BNEZ	$4	SKIP1					11
16	SKIP1:	SUB	$2	$2	$1			16 17	
17		SRL	$2	$2	1				
18		JUMP	LOOP						18
6	LOOP:	ADDI	$15	$15	1			6 7	
7		SEQI	$16	$15	5				
8		BNEZ	$16	LAST					8
9		ANDI	$3	$2	3			9 10	
10		SEQI	$4	$3	2				
11		BNEZ	$4	SKIP1					11
12		SEQI	$5	$3	1			12 13	
13		BNEZ	$5	SKIP2					
19	SKIP2:	ADD	$2	$2	$1			19 20	
20		SRL	$2	$2	1				
21		JUMP	LOOP						21
6	LOOP:	ADDI	$15	$15	1			6 7	
7		SEQI	$16	$15	5				
8		BNEZ	$16	LAST					8
9		ANDI	$3	$2	3			9 10	
10		SEQI	$4	$3	2				
11		BNEZ	$4	SKIP1					11
16	SKIP1:	SUB	$2	$2	$1			16 17	
17		SRL	$2	$2	1				
18		JUMP	LOOP						18
6	LOOP:	ADDI	$15	$15	1			6 7	
7		SEQI	$16	$15	5				
8		BNEZ	$16	LAST					8
9		ANDI	$3	$2	3			9 10	
10		SEQI	$4	$3	2				
11		BNEZ	$4	SKIP1					11
12		SEQI	$5	$3	1			12 13	
13		BNEZ	$5	SKIP2					
14		SRL	$2	$2	1		14 15		
15		JUMP	LOOP						
6	LOOP:	ADDI	$15	$15	1			6 7	
7		SEQI	$16	$15	5				
8		BNEZ	$16	LAST					8
22	LAST:	SRL	$2	$2	1			22 23	
23		ST	$2	12[$0]					
24		HALT							24
						合計	4	15	12

図10：1 次 Booth のハンドシミュレーション

4.2.3　2ALU での並列性評価

⑴演算数の評価

表6：Booth 並列性評価（2ALU）

		連鎖あり			連鎖なし	
		2並列	連鎖	単一	並列	単一
1次	個数	4	15	12	7	36
	割合（%）	13	48	39	16	84
2次	個数	3	14	7	5	31
	割合（%）	13	58	29	14	86
3次	個数	4	30	6	5	64
	割合（%）	10	75	15	7	93

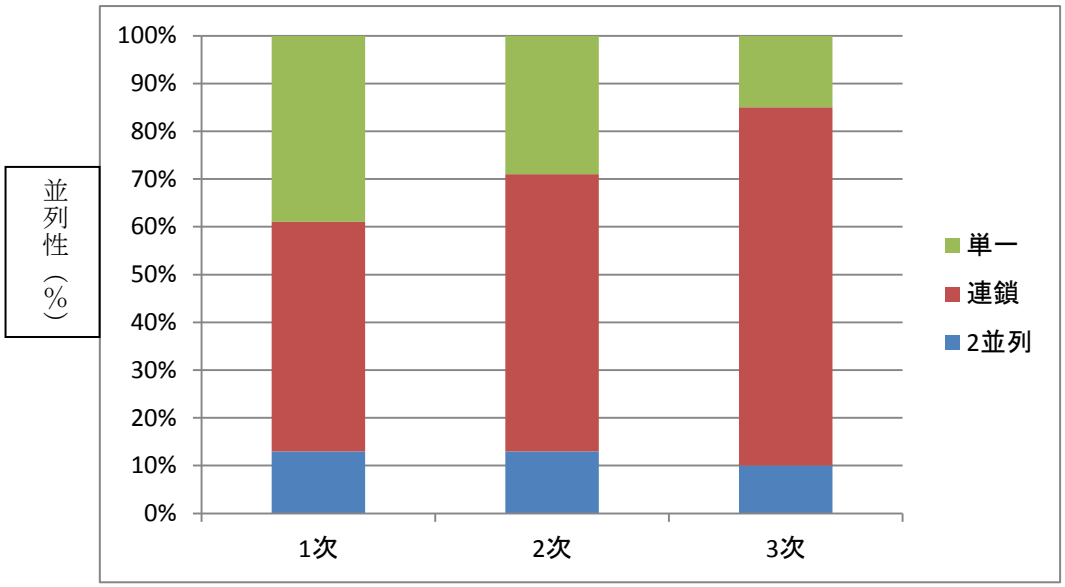

図11：並列性の比率の評価（連鎖あり）

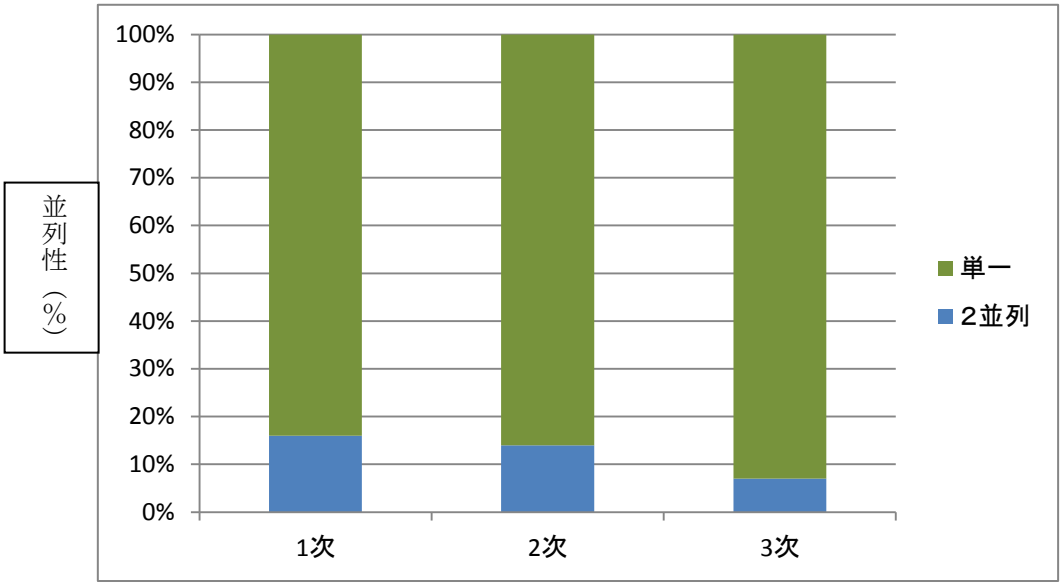

図12：並列性の比率の評価（連鎖なし）

⑵考察

1 次 Booth のアルゴリズムを2ALU 連鎖ありで動的に実行すると、並列・連鎖演算で 60％近く割合を占めており、単一は少ない結果となったが、2ALU 連鎖なしで動的に実行すると単一演算が 84％とほとんどの命令を単一で実行することになった。これより連鎖演算を有効にすることで 1 次 Booth 乗算アルゴリズムは並列性が増加したと言える。2次 Booth のアルゴリズムを2ALU 連鎖ありで実行すると、並列・連鎖演算で 70％近く割合を占めており、単一は 29%少ない結果となった。この結果は 1 次 booth と比べると、単一演算の割合が 10％減少していた。この原因は 1 次 Booth より分岐命令の数が減っていることが原因だと思われる。2ALU 連鎖なしで動的に実行すると単一演算が 83％とほとんどの命令を単一で実行することになった。これより連鎖演算を有効にすることで 2 次 Booth 乗算アルゴリズムは並列性が増加したと言える。3次 Booth のアルゴリズムを2ALU 連鎖ありで動的に実行すると、並列・連鎖演算で 85％を占めており、単一は 15%と Booth の乗算アルゴリズムの中では一番少ない結果となった。この原因は 1 次 Booth と 2 次 Booth を比較した場合と同様に分岐命令の数が減っていることが原因だと思われる。2ALU 連鎖なしで動的に実行すると単一演算が 93％とほとんどの命令を単一で実行することになった。これより連鎖演算を有効にすることで 3 次 Booth 乗算アルゴリズムは並列性が増加したと言える。結果、Booth の乗算アルゴリズムにおいて連鎖演算が多く使われているということが分かる。

次に 4.3 で4ALU シミュレータによる booth 乗算の並列性の評価を行う。

4. 3　4ALU シミュレータによる並列化

4.3.1　命令間の並列性と連鎖性の例

動的にシミュレーションを行うときに 1 ステップを終了する条件を以下に2つ定義する。

① J 形式の命令（JUMP、HALT）が実行されたとき、1 ステップを終了する。

②　条件分岐命令が実行されたとき、1 ステップを終了する。

以上の2つの命令の下位命令が存在するときに、1 ステップを終了する。

この条件により4ALUMAP シミュレータでは1ALU で1ステップが終了する場合、2ALU で終了する場合。同様に3、4ALU を使用する場合がある。以下に命令の例を示す。

⑴1命令実行

step1	命令1	JUMP	LOOP		
step2	命令2	ADD	$1	$2	$3

主に分岐命令で単一命令が実行される。上の例は命令1が JUMP 命令になっているのでステップを終了し、次のステップに移行している。よって命令1は単一演算になる。

⑵2 命令実行

（i）2 並列演算

命令1	ADD	$1	$2	$2
命令2	JUMP	LOOP		

命令1、2に依存関係がなく、JUMP 命令が命令2にあるので、ステップを終了させる。

よって 2 並列演算となる。

（ii）2 連鎖演算

命令1	ADD	$1	$2	$2
命令2	BNEZ	$1	LOOP	

命令1、2には依存関係があるので、2ALU 連鎖演算を行う。命令2で条件分岐命令があるので、ステップを終了させ 2 連鎖演算となる。

⑶3命令実行

（i）3並列演算

命令1	ADD	$3	$2	$2
命令2	SUB	$4	$2	$2
命令3	BNEZ	$1	LOOP	

命令1、2、3に依存関係がなく、条件分岐命令が命令3にあるので、ステップを終了させる。よって3並列演算となる。

（ii）3連鎖演算

命令1	ADD	$3	$2	$2
命令2	SUB	$1	$3	$2
命令3	BNEZ	$1	LOOP	

命令1、2、3には依存関係があるので、3ALU 連鎖演算を行う。命令 3 で条件分岐命令があるので、ステップを終了させ 3 連鎖演算となる。

（iii）2 連鎖/2 並列演算

命令1	ADD	$3	$2	$2
命令2	SUB	$1	$3	$2
命令3	BNEZ	$5	LOOP	

命令1、2には依存関係があるので、2ALU 連鎖演算を行う。命令 3 は依存関係はないので、命令1、2と並列に処理を行うことができるので、2 連鎖・2 並列演算となる。

⑷4命令実行

（i）4並列演算

命令1	ADD	$3	$2	$2
命令2	SUB	$1	$2	$2
命令3	SUB	$4	$2	$2
命令4	BNEZ	$5	LOOP	

命令1、2、3、4に依存関係がなく、命令1、2、3に分岐命令や、終了命令がないので、4ALU を1ステップで使用する。よって上の命令は 4 並列演算である。

（ii）4連鎖演算

命令1	ADD	$2	$1	$1
命令2	SUB	$3	$2	$2
命令3	SRL	$4	$3	$3
命令4	BNEZ	$4	LOOP	

命令1、2、3、4には依存関係があるので、4ALU 連鎖演算を行う。

（iii）2 連鎖・2 連鎖演算

命令１	ADD	$2	$1	$1
命令２	SUB	$3	$2	$2
命令３	SRL	$4	$1	$1
命令４	BNEZ	$4	LOOP	

命令1、2には依存関係があるので、2ALU 連鎖演算を行う。命令 3、4は依存関係があるので、命令1、2と命令3、4を並列に処理を行うことができるので 1 ステップで 2 連鎖演算を2つ並列に実行することができる、2 連鎖・2 連鎖演算となる。

（iv）3 並列/2連鎖

命令１	ADD	$2	$1	$1
命令２	SUB	$3	$2	$2
命令３	SRL	$4	$1	$1
命令４	BNEZ	$5	LOOP	

命令1は命令2と依存関係が生じている。それ以外の命令は依存関係がないので並列に演算を行うことができる。この場合 3 並列・2 連鎖となる。

（v）2 並列/3 連鎖

命令１	ADD	$2	$1	$1
命令２	SUB	$3	$2	$2
命令３	SRL	$4	$3	$3
命令４	BNEZ	$5	LOOP	

命令1、2、3には依存関係があるので、3ALU 連鎖演算を行う。命令4は他の命令と依存関係がないので、3 連鎖演算と並列に処理することが可能である。
4.3.2 で 1 次 Booth の乗算プログラムの動作を例に用い、ハンドシミュレーション例を提示する。

4.3.2　1次 Booth 乗算のハンドシミュレーション

Booth の乗算は Y の値によって動作が変わるのでここでは Y=-3 の場合を例にする

番号	命令						2並列	2連鎖	3連鎖	4連鎖	単一
1		LD	$1	0[$0]				(1,4) (2,3)			
2		LD	$2	4[$0]							
3		ANDI	$2	$2	15						
4		SLL	$1	$1	5						
5		SLL	$2	$2	1		5 6		6 7 8		
6	LOOP:	ADDI	$15	$15	1						
7		SEQI	$16	$15	5						
8		BNEZ	$16	LAST							
9		ANDI	$3	$2	3				9 10 11		
10		SEQI	$4	$3	2						
11		BNEZ	$4	SKIP1							
16	SKIP1:	SUB	$2	$2	$1			16 17			
17		SRL	$2	$2	1						
18		JUMP	LOOP				17 18				
6	LOOP:	ADDI	$15	$15	1				6 7 8		
7		SEQI	$16	$15	5						
8		BNEZ	$16	LAST							
9		ANDI	$3	$2	3				9 10 11		
10		SEQI	$4	$3	2						
11		BNEZ	$4	SKIP1							
12		SEQI	$5	$3	1			12 13			
13		BNEZ	$5	SKIP2							
19	SKIP2:	ADD	$2	$2	$1			19 20			
20		SRL	$2	$2	1						
21		JUMP	LOOP				20 21				
6	LOOP:	ADDI	$15	$15	1				6 7 8		
7		SEQI	$16	$15	5						
8		BNEZ	$16	LAST							
9		ANDI	$3	$2	3				9 10 11		
10		SEQI	$4	$3	2						
11		BNEZ	$4	SKIP1							
16	SKIP1:	SUB	$2	$2	$1			16 17			
17		SRL	$2	$2	1						
18		JUMP	LOOP				17 18				
6	LOOP:	ADDI	$15	$15	1				6 7 8		
7		SEQI	$16	$15	5						
8		BNEZ	$16	LAST							
9		ANDI	$3	$2	3				9 10 11		
10		SEQI	$4	$3	2						
11		BNEZ	$4	SKIP1							
12		SEQI	$5	$3	1			12 13			
13		BNEZ	$5	SKIP2							
14		SRL	$2	$2	1		14 15				
15		JUMP	LOOP								
6	LOOP:	ADDI	$15	$15	1				6 7 8		
7		SEQI	$16	$15	5						
8		BNEZ	$16	LAST							
22	LAST:	SRL	$2	$2	1			22 23			
23		ST	$2	12[$0]							
24		HALT									24
						合計	4	7	10	0	1

図 13:1次 Booth 乗算アルゴリズム4ALU 動的演算(連鎖あり)

4.3.3 4ALU での並列性評価

(1) Booth の演算数比較

表7：Booth 並列性評価（4ALU）

		連鎖あり					連鎖なし	
		2並列	2連鎖	3連鎖	4連鎖	単一	並列	単一
1次	個数	5	8	9	0	1	7	36
	割合（%）	28	44	50	0	6	16	84
2次	個数	3	11	5	0	1	5	31
	割合（%）	19	69	31	0	6	14	86
3次	個数	2	25	5	1	2	5	64
	割合（%）	6	76	15	3	6	7	93

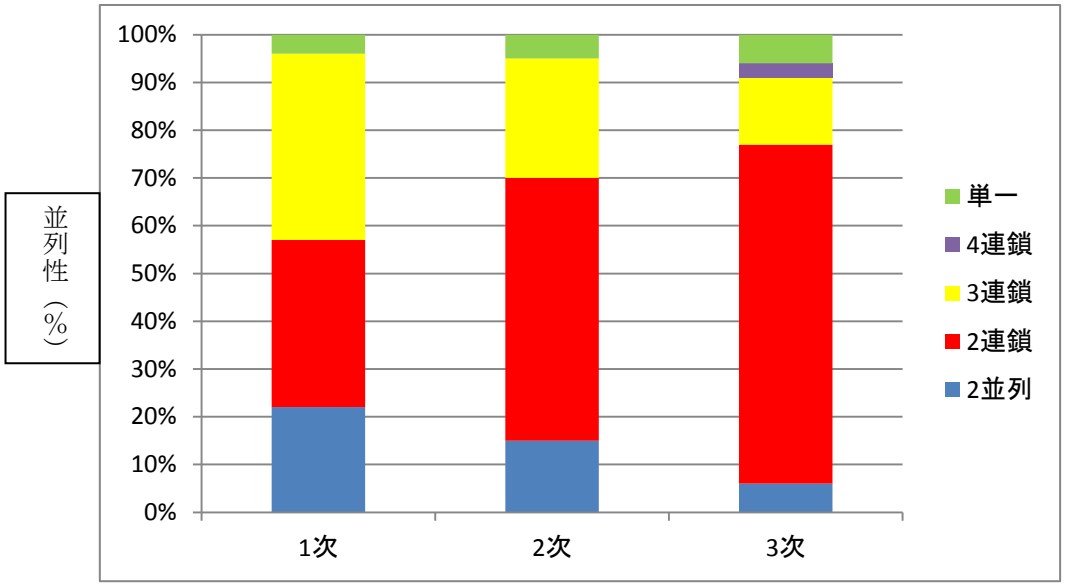

図14：各 Booth アルゴリズムの並列性比較（連鎖あり）

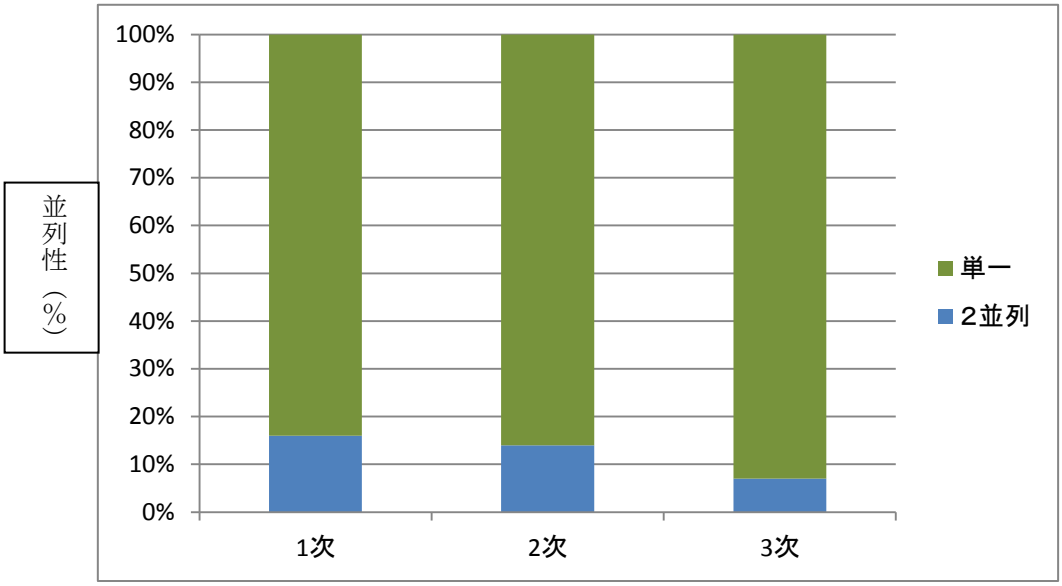

図15：各 Booth アルゴリズムの並列性比較（連鎖なし）

⑵考察

1次 Booth のアルゴリズムを4ALU 連鎖ありで動的に実行すると、並列・連鎖演算で 96％割合を占めており、単一は4%少ない結果となった。この結果は 1 次 booth を2ALU 連鎖ありと比較すると単一実行が大きく減少していることが分かる。この原因は、最上位命令に単一演算で処理される分岐命令がほとんど存在せず、分岐命令が並列や連鎖演算で処理できることが多くなったことである。4ALU 連鎖無しで実行すると、単一実行が大きな割合を占め 84％という結果になった。連鎖ありとなしを比較すると大きく単一演算の数が変化している点から、1 次 Booth のアルゴリズムは連鎖演算を多く使用し、ALU の数を増やすと並列性が増加していることが分かる。2次 Booth のアルゴリズムを4ALU 連鎖ありで動的に実行すると、並列・連鎖演算で 95％割合を占めており、単一は5%少ない結果となった。この結果は2次 booth を2ALU 連鎖ありと比較すると単一実行が 34％減少していることが分かる。この原因は、1 次 Booth 同様に最上位命令に単一演算で処理される分岐命令がほとんど存在せず、分岐命令が並列や連鎖演算で処理できることが多くなったことである。次に4ALU 連鎖無しで実行すると、単一実行が大きな割合を占め 86％という結果になった。連鎖ありとなしを比較すると大きく単一演算の数が変化している点から、2次 Booth のアルゴリズムは連鎖演算を多く使用し、ALU の数を増やすと並列性が増加していることが分かる。3次 Booth のアルゴリズムを4ALU 連鎖ありで動的に実行すると、並列・連鎖演算で 94％割合を占めており、単一は 6%と 1 次 2 次同様に少ない結果となった。この結果は3次 booth を2ALU 連鎖ありと比較すると単一実行が 9％減少していること

が分かる。これは 1 次 2 次 Booth と比較すると一番少ない減少となった、この原因は Y=-3 という値で実行した時に、単一命令が2回実行された事と、2ALU 連鎖ありで実行した時に、十分な並列性があったからである。しかし、4ALU 連鎖ありで実行することで並列性は増加しているので、4ALU の有用性は確かである。次に4ALU 連鎖無しで実行すると、単一実行が大きな割合を占め 93%という結果になった。連鎖ありとなしを比較すると大きく単一演算の数が変化している点から、3次 Booth のアルゴリズムは連鎖演算を多く使用し、ALU の数を増やすと並列性が増加していることが分かる。

5. おわりに

本論文では、本研究室で開発をしているハード/ソフト協調学習システムを利用し設計されたMAP の一部である MAP シミュレータの設計と試作を行った。以前に設計した2ALU シミュレータを参考に4ALU への拡張を目的としてシミュレータの試作を行った。
その時にテストデータとして Booth 乗算の並列性を評価し、Booth の乗算は連鎖演算・並列演算を4ALU で多く動作することを確認することで4ALU プロセッサの有用性も確認できた。本研究を通して、MAP の仕組み、MAP シミュレータの制御と仕組み、連鎖・並列・単一実行回数判別の学習を行えた。今後の課題としては、4ALUMAP シミュレータの作成である。

謝辞

本研究の機会を与えて下さり、貴重な助言、ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、本研究に関して様々な相談に乗って頂き、貴重なご意見を頂きました、孟　林助教授、石川陽章氏、杵川大智氏に深く感謝いたします。

参考文献

[1] 泉知論：マイクロプロセッサデザイン，講義レジュメ,Booth のアルゴリズム,2013

[2] 田中亮佑：マルチ ALU プロセッサにおけるアセンブラの設計と試作（Ⅰ），立命館大学理工学部電子情報デザイン学科卒業論文，2012.

[3] 高松良太：マルチ ALU プロセッサにおけるシミュレータの設計と試作，立命館大学理工学部電子情報デザイン学科卒業論文，2012.

[4] 境直樹：演算レベル並列処理用マルチ ALU プロセッサの設計と実現，立命館大学大学院、理工学研究科創造理工学，専攻修士論文，2013.

[5] 境直樹：MAP 仕様書，2011.

[6] David A.Patterson and John L.Hennessy 著，成田光彰 訳：コンピュータの構成と設計 第四版 (上)(下)，日経 BP 社，2011.

[7]石川陽章，杵川大智，境直樹，孟林，山崎勝弘：演算レベル並列処理マルチ ALU プロセッサの設計と実現， C-005, FIT2013，第 12 回情報科学フォーラム 2013