マイクロプロセッサの開発

[第 1 週]

1 はじめに

この実験では、

1. ハードウエア記述言語を用いてマイクロプロセッサの設計を行なう。
2. シミュレーションにより動作を確認する。
3. 論理合成により回路に変換する。
4. 書き換え可能なゲートアレイ FPGA(Field Programmable Gate Array) を用いて合成した回路を実現し、実際に信号を与えて動作を確認する。

という一連の操作を通して、マイクロプロセッサの開発を行ないます。

最初の 2 週間は、ハードウェア記述言語を使って回路を設計する練習を行ないます。この部分は、班には分かれずに、ひとりずつ実験します。

次の 2 週間で、班毎にプロセッサを設計し、ハードウェア記述言語による記述とシミュレーションを行ないます。

5 週目に実際に FPGA を用いて動作を確認します。これがうまく動けば、6 週目にディスカッションをしてレポートをまとめておしまいです。

2 LSI 設計の流れ

一口に LSI(Large Scale Integrated circuit) と言ってもいろいろなものがあります。

- フルカスタム方式
  LSI のレイアウト設計全てを基本的に人手を介して行なう方式。性能と集積度は最も大きいが、開発時間、費用がかかるため、数が出るものでなければならない。
- セミカスタム方式
  要素等人手を介してレイアウトしたり、全体の構成を人手で行なうが、要素間の配線等は自動配線用 CAD を用いる方式。フルカスタムほど開発費用は要せず、まずまずの性能、集積度が得られる。
- ゲートアレイ方式
  論理設計より下の部分は、全て CAD により自動的に行なわれる方式。既に用意されているゲートのライブラリを用い、配置配線も自動的に行なわれる。最近 SOG(Sea Of Gates) 方式の登場により柔軟性も増している。ASIC(Application Specific IC) のほとんどはこの方式を取る。

最近は、CAD による自動配置配線が飛躍的に発達したため、開発が行なわれるチップの種類で考えると、ゲートアレイが圧倒的に多くなっています。ただ、どの方式をとるにせよ、以下の手順は変わりません。

1. 仕様設計レベルの設計。
2. レジスタトランスファレベルの設計。
   場合によってはここでシミュレーションして動作の検証。
3. 論理レベルの設計。
   仮配線遅延を用いた論理シミュレーションにより動作の検証。テストベクトルの作成。故障シミュレーション。
4. 配置配線。
   ゲートアレイの場合自動的に行なう。セミカスタムは一部自動。フルカスタムは人手で行なう。
   終了後、実配線遅延を用いた論理シミュレーションによりテストベクトルを用いて動作を検証。
5. マスク生成。
6. 製造。
7. 製品検査。

今回対象となるのはゲートアレイです。この場合、論理レベルの設計を行なって、論理シミュレーションを行なえば、一応後のステップは自動的に CAD が全てを行なってくれます。

さて、今回みなさんは HDL の一種である SFL で設計を行ないます。これは、レベルでいうとレジスタトランスファレベル (本当は少し論理レベル寄りか) の設計を行なうことになります。このレベルで論理シミュレーションを行なって、機能を確認すれば、論理回路の生成は自動的に行なわれるので、より人手を介さずに設計を行なうことができます。

まとめると、今回の実験で人手で行なうのは、1,2,7 のステップです。したがって、1,2 の部分、つまり設計、記述、シミュレーションが主な仕事となります。きちんとシミュレーションで動作を確認すれば、実装に失敗することは、まず、ありません。最後に、実際にできあがったチップの動作を確認して、ああできたなあと感動して、あとは性能を調べて終りです。

本当は、CAD による自動配置配線が終った後、実配線遅延によるシミュレーションを行なわなければならないのですが、FPGA の性質上これは大変難しいので、今回はこの部分は省略します。

3 こて調べ

3.1 ハードウェア記述言語とは

図面上でディジタル回路を設計し入力する方法は、もちろん今でもよく用いられていますが、最近はディジタル回路をちょうど計算機のプログラミングをするように言語で記述し、その記述から回路を自動合成する方法が一般的になりつつあります。この言語のことをハードウェア記述言語 (HDL:Hardware Description Language) といいます。この方法は 2 つの利点があります。

- 高いレベルで記述するため、バグが発見しやすく、他人が見てもわかり易い。
- 論理の簡単化等、細かいレベルの設計は論理合成時に自動的に行なわれるので、設計のための時間が短縮でき、バグも減る。

しかし、一方でこのような方法は以下の弱点を持っています。

- 合成可能な回路の形式が制限される。特に非同期回路は合成できない。
- 最大性能は人手による設計に及ばない。

回路図レベルで設計するか、HDL を使うかは、計算機のプログラムをアセンブラで書くか、高級言語で書くかという関係に似ています。現在アセンブラで書く機会は極端に減っており、将来回路図レベルでの設計がこれほど減るとは思いませんが、HDL を用いた記述、設計は将来増加することは間違いないと言われています。

現在一般的に用いられている HDL には次のようなものがあります。

- VHDL
- UHDL
- Velilog HDL
- Parthenon SFL

これは、それぞれ特徴がありますが、簡単に言って VHDL,UHDL は主にドキュメンテーションの標準化を狙っており、実際に論理合成できる範囲が狭いです。これに対し Velilog HDL はもっとも現場で用いられている HDL ですが、記述レベルが低く、読みにくいです。今回用いる SFL は NTT によって開発された HDL で、Velilog HDL に比べるとエレガントに書け、実際に PARTHENON という論理合成システムにより、論理合成することができるので、評判が高い言語です。

3.2 SFL の概観

SFL は、他の HDL 同様、回路の基本単位をモジュールと呼び、モジュール同士の階層構造で回路を表現します。モジュールは入力 (input)、出力 (output)、制御入力 (instrin) から成り、内部に仮端子 (tmp)、レジスタ (reg)、他のモジュール等を持つことができます。各モジュールの機能は C 言語風に記述します。基本的に同期回路のみの設計が可能で、暗黙のうちにシステムクロックが存在し、各制御文は各システムクロック毎に実行されます。最初に文法をやるよりも、簡単なサイコロカウンタの例を見ながら概観をつかみましょう。

3.3 サイコロカウンタ

サイコロカウンタは、1 から 6 まで数えて 1 に戻るカウンタで、状態を保持するフリップフロップに、次の状態を決定する組合せ回路がついている構成を持ちます。さらに、カウンタを 1 にセットする rst(1 でリセット) と、スタートを示す start(1 でサイコロが動き 0 で止まる) 端子を設けます。

さて、これを SFL で記述した例を下に示します。

```
1 /*************************/
2 /*   Saikoro counter      */
3 /*************************/
4  module dice {
5
6  circuit_type        add3 {
7         input         ina <3>, inb <3> ;
8         output        sum <3> ;
9         instrin       add ;
10        instruct_arg  add (ina, inb) ;
11        instruct      add sum = ina+inb ;
12 }
13
14 output dice_o <3> ;
15 instrin start, rst ;
16 reg   dice_reg <3> ;
17 add3 a3 ;
18 instruct start par {
19   alt {
20    ^rst: alt {
21     ( dice_reg<2> & dice_reg<1> & ^dice_reg<0> ) :
22           dice_reg := 0b001;
23     else :
24           dice_reg := a3.add(dice_reg, 0b001).sum;
25      }
26     else : dice_reg := 0b001;
27     }
28     dice_o = dice_reg;
29 }
30 }
```

最も簡単な SFL の記述は以下の構造をとります。

- モジュール名の宣言:module
- 機能回路またはサブモジュールの宣言:circuit_type (submod_type)
- 入出力、制御入力の宣言: input output instrin
- レジスタの宣言: reg
- 機能モジュール、サブモジュールの生成
- 動作記述: instruct

module 文

まず、4 行目の module 文により、これから記述するモジュール名を定義します。今回は dice がこれに当たります。

circuit_type 文

カウンタを作るのに一番楽なのは加算器を用意することです。ここでは、6-12 の circuit_type 文により 3bit の加算器を定義しています。circuit_type は任意の組合せ回路を定義することができます。circuit_type でも通常のモジュール同様、入力 (input)、出力 (output)、制御入力 (instrin) を定義します。今回入力は加算する 3bit の数 2 つ ($ina < 3 >, inb < 3 >$) で、出力は加算結果 ($sum < 3 >$) です。<>でくくると、ちょうどプログラミング言語の配列のように、バスの記述が可能になります。

9 行目の instrin は少しわかりにくいかもしれませんが、この場合、加算器を動かすかどうか決定する入力と考えて下さい。SFL ではどのモジュールについてもこの制御入力を必要とします。10 行目の instr_arg 文はあるモジュールの制御入力が制御する入力の範囲を示します。今回は両方の入力がこれに当たります。11 行目の instruct add は制御入力である add が 1 の時、この circuit_type で行なわれる機能を示します。この場合加算を行ないます。実はこのような形で加算が書けるのは circuit_type の中だけです。SFL で用いられる演算子を下表に示します。ここで自動合成が否となっている演算子は circuit_type の中でしか使えません。

優先度	演算子	機能	自動合成の可否
1	*op1*<開始桁:終了桁>	ビット切りだし	可
2	^*op1*	論理否定	可
	/&*op1*	桁ビットの AND	可
	/|*op1*	桁ビットの OR	可
	/@*op1*	桁ビットの EOR	可
	/*op1*	デコード	否
	op1	エンコード	否
	ビット幅#*op1*	ビット幅符号拡張	否
3	*op1*||*op2*	連結	可
	op1&*op2*	AND	可
	op1|*op2*	OR	可
	op1@*op2*	EOR	可
	op1+*op2*	加算	否
	op1>>*op2*	ビット右シフト	否
	op1<<*op2*	ビット左シフト	否

この circuit_type は大変便利な記法ですが、現在の所 PARTHENON はこの部分の自動合成ができないので、後でモジュールに入れかえてやります。取りあえず、シミュレーションをする場合、これで問題ありません。

モジュールの端子の定義

14,15 行目はモジュールの端子の定義です。ここでは制御入力として start と rst を定義していますが、実は rst は入力として定義してもかまいません。

レジスタの定義

16 行目は 3bit のレジスタ dice_reg を定義しています。このレジスタは図 1 に示すように、さいころの目を記憶し、現在の状態を示します。バス同様<>の記号をつけてまとめて扱うことが可能です。

機能回路の実体の定義

C 言語に慣れない方は 17 行目の記述がわかりにくいかと思われます。これは、add3 という circuit_type からタイプから、a3 という実体を作りだすことを示します。ちょうど、型紙から型をうち出して実体を作るように、同じ型を持つ機能回路をいくつも使うことができます。この例では 1 つしか使いません。

動作制御文: **par, alt, any** 構造

18-28 行目は 11 行目同様の instruct 文で、このモジュールでの実際の動作を定義します。ここでは、

```
par {

        .......... ;
        .......... ;
}

alt  {
condition1: .....
condition2: .....
else: ....
}
```

の構造が使われています。par 文は構造中の各文が同時に実行されます。これは、つまり回路の別々の部分が独立に動作することに対応します。この場合、20-25 行目の制御 (これは後で解説しますが、要するにさいころの次の状態を決めてそれをレジスタに代入する操作) と、28 行目のレジスタの内容を出力することを同時に行なっています。

これに対し、alt 文は構造中の条件の成立した文がただひとつ実行されます。条件は、上から順に優先順位がついており、つまり最初に成立したものがひとつ実行されるわけです。SFL にはあともうひとつ、条件が成立したものは全て並列に実行される any 文があり、とにかく、動作制御の文が複数行出てくる時は、par,alt,any のどれかの構造の中に入っている必要があります。

さて、20-25 行目は alt 文の入れ子で、まず、rst が 0 であることを判別し (20 行目)、その場合はカウンタを動作させ、そうでない場合 (26 行目の else) 状態つまり dice_reg の値を 1 にします。alt の条件文は演算結果が 1 か 0 になるように書かなければなりません。演算子のうち、モジュールの定義で使えるものは制限が厳しいので注意して下さい。ここで、26 行目の

```
dice_reg := 0b001 ;
```

に注目して下さい。SFL では単につなぐ時は=、記憶素子にデータを格納する時は:=を用います。つまり、28 行目の

```
dice_o = dice_reg ;
```

はレジスタの出力をモジュールの出力につなぐことを示すわけです。記憶素子への格納は、その動作制御文が動作しうる条件が成立すれば、クロックに同期して各サイクルで行なわれます。また代入する数 $0b001$ は、3bit の 2 進数表現を示します。この他 SFL では,$0x$ で始まると 16 進数、$0o$ で始まると 8 進数表現をそれぞれ表します。

さて、カウンタが動作した場合の制御が 21-25 行目にあります。21 行目は長いですが、これは、条件を示す部分で現在のレジスタの状態が 6(つまりは 110) であるかどうかをチェックします。6 ならば、次の状態は 1(22 行目)、そうでなければ、状態に 1 を加えるという操作を行ないます (24 行目)。この 24 行目に SFL の特色が出ています。

```
dice_reg := a3.add(dice_reg, 0b001).sum;
```

この文で、最初に定義した機能回路 add3 の実体である a3 を制御入力 add をアクティブにして起動します。引数は 1 足すわけなので、現在の状態を持つレジスタ dice_reg と 1 を取り、出力の sum の 3bit がレジスタ 3bit に格納されます。

ここまでできたら、module を終了します。これでこの回路のシミュレーションは可能になります。

3.4 論理シミュレータ seconds

seconds は会話型の論理シミュレータです。値は 3 値 (0,1,Undefined), 0 遅延 (つまりゲートの遅延は 0 とする) で、階層構造の回路を効率よくデバッグするのに適しています。

3.4.1 立ち上げ

Parthenon SFL, seconds が使える計算機は以下の 20 台です。所属する班毎に指定の WS に rlogin して下さい。

班	使用 WS
A	ew4307
B	ew4308
C	ew4314
D	ew4315
E	ew4401
F	ew4402
G	ew4409
H	ew4410
I	ew4411
J	ew4412
K	ew4503
L	ew4504
M	ew4505
N	ew4513
O	ew4514
P	ew4515
	ew4606
	ew4607
	ew4608
	ew4616

セッションを開始したあと、まず以下の操作を行ないます。

```
rlogin ewXXXX
```

ここから、この ewXXXX の部分は、上の表にしたがって例えば G 班の人は ew4608 になります。

まず、emacs を用いて、例えば dice.sfl という名前の sfl 記述のファイルを作成します。

ここでは、sfl で記述したこの dice をシミュレーションしながら、seconds の機能を紹介していきます。seconds は会話型のシミュレータなので、まず、とりあえず立ち上げて下さい。単に

```
seconds
```

と打ち込めばシミュレータが立ち上がり

```
*****************************************
*    Welcome to SFL world!                *
*     version 2.1.7   04 Feb '93          *
*                                         *
* Confidential property of N.T.T          *
* (C)Copyright by N.T.T 1992(unpublished) *
*   All rights are reserved.              *
*****************************************
```

```
SFL>
```

というメッセージが表示されます。ちなみに、$SFL>$ は seconds のプロンプトです。立ち上がらない人はちゃんと指定のマシンに rlogin しているかどうかもう一度確認して下さい。チェックをしてみて、それでも駄目なら T.A. に相談して下さい。

さて、まずファイルを読み込む必要があります。ここでは、先ほどのプログラムが dice.sfl というファイルに入っているとします。

```
sflread dice.sfl
```

と、やるとプログラムを読み込むと共にシンタックスチェックをやってくれます。ここで、エラーが出なかった人は次の節まで飛んで下さい。もし、プログラムにエラーがあると、ここで、

```
Shall I open the edit pad ? Type <CR> if ready.
```

という表示があり、CR を押すと、emacs が起動します。カーソルは自動的にエラーが発見されたところに移動しているので、エラーを直して下さい。エラーメッセージはさほど親切ではないので、わからない場合 T.A. に相談して下さい。さて、エラーを直してファイルに書き込んでエディタを閉じると、

```
Error exist just before the cursor position. Please correct sfl source file and
save it and type ‘‘r.’’
TRNS:
```

というメッセージが出ますので、これに従って、

```
r.
```

と打ち込んで下さい。もう一度おなじこと (つまり dice.sfl の読み込み) が実行されます。ここで、またエラーが出るとふたたび emacs が開くのでがんばってデバッグを続けて下さい。

3.4.2 回路のインストール

さて、めでたくシンタックスエラーもなくなり、いよいよ実行することになります。ここで、最初に読み込んだモジュー ル (dice) を階層構造に当てはめる作業が必要になります。dice は階層構造を持ってないので、この話は後回しにします。ここでは、深く考えずに

```
install dice /
```

と打ち込んで下さい。

3.4.3 入力の設定

ここで、回路の入力値を設定します。dice の入力はさいころの目を 1 に戻す rst と、さいころを止めたり動かしたりする start の 2 つしかありません。これを設定してみましょう。設定には、

```
set 入力端子名 入力値 (2 進数)
```

とやります。この場合、最初、さいころは目を 1 にし、(rst をアクティブ) 動く状態 (start をアクティブ) にセットしましょう。

```
set rst 1
set start 1
```

さて、seconds では、設定した値はそのクロックのみで有効で、次のクロックではまた設定し直さなければなりません。rst は最初の 1 クロックだけアクティブにすればいいので、まあいいですが、start は当分の間アクティブになっていて欲しい (最初はさいころが動いている様子が見たい) ので、これではちょっと不便です。そこで、

```
hold start
```

とやると、今 set した値 (この場合 1) がその後のクロックでも自動的に設定されます。

3.4.4 出力値の表示

シミュレーションの結果を観察するためには出力あるいは、内部のレジスタの値、内部端子の値を表示する必要があります。seconds では C 言語の printf 文風の指定でプリントを行ないます。

```
print フォーマット指定 端子 端子...
```

このフォーマットが C 言語風で慣れてないと、ちょっと親しみにくいかもしれません。下表に示すように、どのような形で出力するかを指定します。printf 文と違って、端子名を , ではなくスペースで区切ることに注意して下さい。

変換指示文字	出力
b	2 進数
o	8 進数
d	10 進数
x	16 進数
s	ステージ、セグメントの状態名を表示
t	現在時刻を 10 進数で表示
その他	16 進数

出力 dice_o を 16 進表示するには、

```
print "%x\n" dice_o
```

と打ち込みます。今回は 1 から 6 までの値なので、10 進表示でも変わりません。また、2 進表示も気分が出ていいかもしれません。好きな形を使って下さい。ちなみにまだシミュレーションを始めてないので、今これを打ち込むと、単に u と表示されます。

さて、この表示も set 同様、各クロックで打ち直さないと表示されないので、めんどうです。そこで、クロックが進む度に自動的に表示しなおしてくれるレポート機能が用意されています。

```
rpt_add キー名 フォーマット指定 端子 端子....
```

と打てばいいです。ここで key は、この指定自体につける名前で、適当に設定して下さい。例えば、

```
rpt_add P "CLOCK %t: dice=%x\n" dice_o
```

と打つと毎回シミュレーションが進む度にその時刻と dice_o が表示されます。key を変えて様々な出力をレポートします。レポートの表示は指定した順番です。さて、毎回出てくるのがうっとうしくて、この指定を解除したい時はさっき指定した key を用いて

```
rpt_rmv キー名
```

とやります。

さて、ここまで来るとちょっとしつこいのですが、このレポート機能は全体を有効 / 無効にする機能があります。

```
rpt_on キー名 キー名 ......
```

で有効、

```
rpt_off キー名 キー名 ......
```

で無効です。ちなみに、rpt_add で登録しただけでは、表示してくれないので、忘れずに rpt_on を実行して下さい。

3.4.5 実行

いよいよ実行です。実行には

```
forward 時刻 (または、+相対時間)
```

とやります。time_value に整数値を直接書くと、その時間 (クロック数) まで実行が進み、 + 整数値と書くと、現在の時間を指定した数だけ進めてくれます。なにも書かないと、システムが設定したデフォルト値 (Halmagedon) まで処理を続けます。ここでは、平和に

```
forward +1
```

と打ちましょう。シミュレーションが 1 ステップ実行されて、先ほど指定した dice_o の値が表示されます。もし設計が間違って無ければ、

```
CLOCK 1: dice=1
```

が表示され、さいころの目は 1 になっているはずです。

さて、sfl の記述の文法上のエラーは、動かしてみるまではわからないものもあります。その手のエラーがあると、ここで現れます。 (ものによってはシミュレーションが進むまでわからないものもあります。) この場合はエラーメッセージを読んで、一度 seconds を終了して、エディットし直すのが簡単です。ではがんばってください。

さて、うまく行った場合、どんどんシミュレーションを続けてみましょう。ここで、rst は 0 に固定します。

```
set rst 0
hold rst
```

さて、どんどん forward して、時間を進めさいころがちゃんと動くことを確認して下さい。ひととおり、確認したら、今度は start を 0 にしてみて下さい。そうすると、出力が出なくなります。これは今回の記述では start=0 の時は定義自体が行なわれていないためです。

3.4.6 スクリプトファイル

今までの操作を seconds を立ち上げる度にいちいち行なうのは大変です。そこで、いままで打ち込んだコマンドを (エディタを使って) ファイルに用意しておけば、

seconds を立ち上げ、

`listen` ファイル名（ファイル名だけ書いても `OK` です）

と、やると自動的にファイルに書いたコマンドが実行されます。

例えば、以下のようなスクリプトファイルを用意しておけばあとは、forward から先を手で打ちながら確認していけば良いことになります。

```
sflread dice.sfl
install dice /
set rst 1
set start 1
hold start
rpt_add P "CLOCK %t: dice=%x\n" dice_o
rpt_on P
```

実はスクリプトファイルは UNIX のシェルスクリプト同様に、値を引数として渡すことができ、なかなか便利です。詳しく知りたい人はマニュアルを参考にして下さい。

3.4.7 結果のセーブ

簡単な回路はその都度目で見て確認することができますが、複雑な回路はファイルにとっておいて後でじっくり確認したり、表示用のツールを使って見たりすることができます。このために、

`speak` ファイル名

と、打ち込むと、結果が指定したファイルに吐き出されます。前の結果に新しい結果をくっつけて (append) 行きたい時は、

```
speaka ファイル名
```

とやります。レポートを書くために、まず、今行なったシミュレーションの結果をファイルに落してみましょう。

3.4.8 seconds の終了

seconds の実行を終了するには

```
bye
```

というコマンドを入力します。seconds は実行に WS のメモリをたくさん使用しますので、使わない時には終了しましょう。また、^Z で止めるとメモリを占有したままになりますので、実行を終了したい時には必ず bye を使って下さい。

4 合成できる回路の記述

4.1 サブモジュールの記述

残念なことに、今まで記述した回路は今の所 circuit_type の所、つまり加算器の所の自動合成ができません。即ち、circuit_type を使っていると、seconds でシミュレーションを行なうことはできても、これを実際の回路に合成してくれないのです。circuit_type に許される機能を考えると、合成可能な範囲なので、きっと将来はできるようになると思いますが、今の所はやはり加算器の所を自分で書いてやる必要があります。

どうせ 3 桁なので、簡単なリプルキャリ方式で行きましょう。まず、1 桁分の全加算器を作ります。これは、回路図を見ながら、さっきのサイコロと同様の順番で定義して行きます。

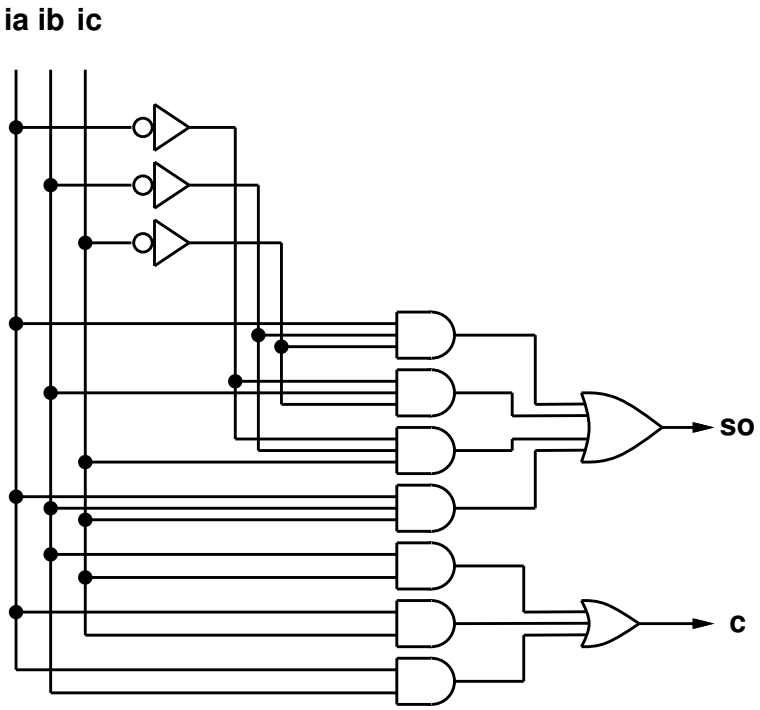

図 1: 全加算器

```
/**************************/
/*       Full adder       */
/**************************/
module f_ader {
    input ia, ib, ic ;
    output so, c ;
    instrin fa ;
    instruct fa par {
      so = (ia & ^ib & ^ic)|(^ia & ib & ^ic)|(^ia & ^ib & ic)|(ia & ib & ic) ;
      c = (ia & ib )|(ib & ic)|(ia & ic) ;
    }
}
```

おわかりと思いますが、ia,ib,ic が加算する入力、so が出力、c が桁上げ (キャリー) です。fa はこの加算器を動かす制御入力です。この記述はもう既に簡単化した結果を書いてしまっています。これではなんのための論理合成かわからんという人は、真理値で 1 の項をそのまま書いてみて下さい。回路合成時にちゃんと自動的に簡単化してくれます。ただ、書くのがめんどくさいので、このようにしました。

さて、この f_ader を 3 つ使って 3 桁のアダーを作ってみます。

```
/*************************/
/*       3bit adder       */
/*************************/
module add3 {
submod_type f_ader {
   input ia, ib, ic ;
   output so, c ;
   instrin fa ;
   instruct_arg fa(ia, ib, ic);
}

input ina<3>, inb<3> ;
output sum<3> ;
instrin add ;
tmp c0,c1,st2,st1,st0 ;
f_ader fa0, fa1, fa2 ;
instruct add par {
   st0 = fa0.fa(ina<0>, inb<0>, 0b0).so;
   c0 = fa0.c;
   st1 = fa1.fa(ina<1>, inb<1>, c0).so;
   c1 = fa1.c;
   st2 = fa2.fa(ina<2>, inb<2>, c1).so;
   sum = st2||st1||st0;
}
}
```

さて、ここで問題なのが、submod_type 文です。これは circuit_type 文同様機能モジュールを定義する所で、ここに、前の全加算器 f_ader を定義しています。回路の動作自体は別に module f_ader で定義されているので、ここでは、入力と出力についてのみ記述すれば OK です。このような記述をサブモジュールといいます。後で、やはり circuit_type 同様、

```
f_ader fa0, fa1, fa2 ;
```

で、全加算器 3 個分の実体を発生させます。後は同様に instruct 文中に、各加算器の接続を関数呼び出しの形で書いてやります。ここで、加算器は so と c の 2 つの出力を持っています。例えば fa0 の出力 so を参照する時に

```
   st0 = fa0.fa(ina<0>, inb<0>, 0b0).so;
```

という文で入力を与えているので、c を参照する時には入力を省略して

```
   c0 = fa0.c;
```

と書くことができます。後者は

```
c0 = fa0.fa(ina<0>, inb<0>, 0b0).c;
```

というように入力をもう一度記述しても構いませんが、入力に矛盾があると実行時にエラーになります。

ちなみにここで使う sum のようにバスで表す出力には、1bit ずつばらばらに=で繋ぐことができないので、一気に代入する必要があります。ここでは仮の端子 (tmp)st0,st1,st2 に順に値を入れておき、最後に

```
sum = st2||st1||st0;
```

で bit を連結し、一気に代入してしまいます。さて、同様にこの add3 をサブモジュールとして、circuit_type の代わりに使うと論理合成可能な dice が出来ます。

即ち、最初のプログラムでは、

```
circuit_type        add3 {
       input         ina <3>, inb <3> ;
       output        sum <3> ;
       instrin       add ;
       instruct_arg  add (ina, inb) ;
       instruct      add sum = ina+inb ;
}
```

となっています。

submod を用いる場合には、この部分を

```
submod_type        add3 {
     input         ina <3>, inb <3> ;
     output        sum <3> ;
     instrin       add ;
     instruct_arg  add (ina, inb) ;
}
```

とします。ここでは add3 というモジュールがどういう入出力を持ち、どのような引数で動作するかを規定しているだけで、実際に内部で何をするかは書かれていません。つまり、この部分は add3 の外側から見た時のインタフェースを書いてあるだけなのです。

4.2 簡単なシミュレーション

さて、出来上がった回路をシミュレーションしてみましょう。同様に seconds を立ち上げ、sflread して下さい。ファイルを別々に分けた人は

```
sflread dice.sfl
sflread add3.sfl
sflread f_ader.sfl
```

と 3 つのファイルを読み込みます。さて、本当は次にこの回路の階層構造を考えて、シミュレーションの環境を作っていきくのですが、ここではモジュール名が階層構造に対応していますから、一気に回路を構成することができます。それには、

autoinstall トップモジュールクラス名 (例:dice)

とやります。もちろん、すべてのモジュールはあらかじめ sflread で読み込んでおく必要があります。

あとは、はじめの circuit_type を用いて記述した例と同様にシミュレーションを行なうことができます。

4.3 階層構造を用いたシミュレーション

この節では、階層構造を用いた、より進んだシミュレーションのやり方を説明します。

4.3.1 シミュレーションイメージの設定

この回路のの階層構造は図 1 のように、根/(モジュール名は dice) になり、a3(モジュール名は add3) という葉を持ち、さらに add3 が fa,fb,fc (モジュール名は f_ader) という葉を持っています。

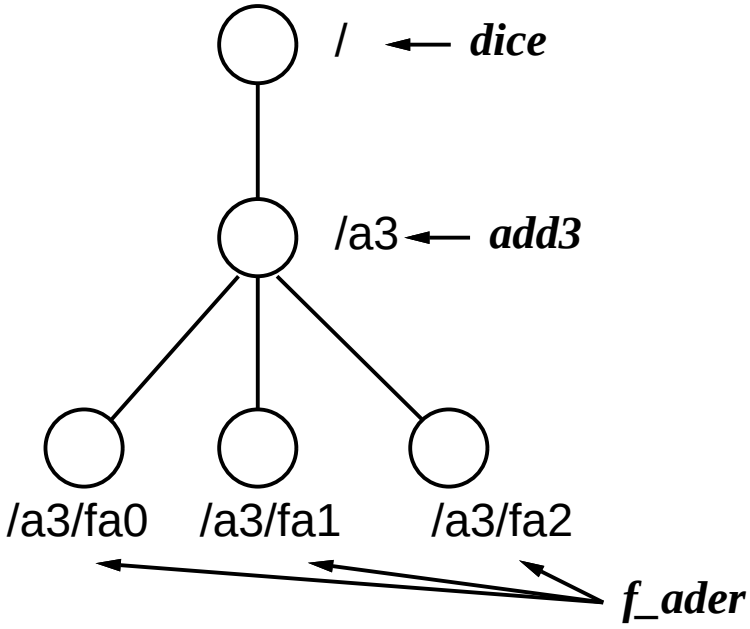

図 2 シミュレータイメージの階層構造

図 2: 階層構造

ファイルを読み込んだ状態では、シミュレーション上では、まだ、a3, fa0, fa1, fa2 という実体がどのモジュールに対応するかは決まってないません。。そこで、各々のモジュールを図 1 に示す構造に埋め込んでいく必要があります。ここで、根は UNIX にならって/とし、階層構造も UNIX 同様/で区切って表現します。

先ほど用いた autoinstall というコマンドは、この作業を名前の一致をとることにより自動的に行なってくれました。

install コマンドを用いると、一つずつ埋め込んでいかなくてはなりませんが、その変わり別の機能モジュールを用意しておいて、シミュレーション中にすり替えることもできます。この機能はモジュール毎にデバックする時などに便利です。先ほどの autoinstall に対応する操作を install を用いて行なうと次のようになります。

```
install dice /
install add3 /a3
install f_ader /a3/fa0
install f_ader /a3/fa1
install f_ader /a3/fa2
```

これで、全部のモジュールが穴に埋まります。

4.3.2 階層間の移動

さて、それでは階層の下の方からシミュレーションしてその機能を確認していきます。階層を渡り歩くコマンド (UNIX の cd) は

```
move 位置 (モジュール名)
```

です。確認のため、構造内のファシリティを表示するコマンド lc を使ってみましょう。このコマンドはある回路の入出力、モジュール、レジスタ、制御構造等 (ファシリティ) を表示するコマンドです。

```
move a3
lc
move fa0
lc
```

と順に打ち込んで下さい。それぞれ階層に移動したことが、表示されるファシリティによってわかります。この場合、一番下の階層まで一気に行きたければ、

```
move /a3/fa0
```

とやって下さい。また、UNIX 風に

```
move ..
```

とやると、親の位置に行くことが出来ます。単に move と打ち込むと現在の位置が表示されます。さて、ここで、制御入力 fa に 1, ia, ib, ic にそれぞれ適当な値を set で設定し、print を使ってフルアダーの動作を確認してみましょう。

うまく行ったら上の階層の 3bit フルアダーに行き同様に動作を確認します。ここで、下の階層のフルアダー fa0 に値を既に設定しており、これを一度クリアしたいので、一度 forward +1 を実行し、次のクロックに進みます。 (設定されてないとのエラーが出ますが気にしない。) フルアダーの確認を行なうためには 3bit の ina と inb に値をセットする必要がありますが、これは、

```
set ina 010
set inb 011
```

で行ないます。ここは 16 進の設定は使えないので注意して下さい。

ここまで確認したら、一度シミュレータを出て、先ほどのスクリプトファイルに f_ader と add3 の読み込みとシミュレーションイメージの部分を付け加えこれを利用して最終的にサイコロが動くことを確認して下さい。

5 回路図への変換

回路図に変換するには、全てのプログラムを一つのファイルにまとめておきます。(そうしなくても変換できますが、このほうが簡単です。)

例えば、

```
cat add3.sfl f_ader.sfl >> dice.sfl
```

などとして、一つのファイルにまとめます。この時、ファイルの `.sfl` の前の部分が、中で使用している top module(この例では dice) の名前と一致するようにしておきます。そして、

```
auto dice ps DEMO demo
```

とすると、自動的に論理合成が行なわれ、dice.ps という名前の回路図のファイルができます。これを

```
lpr dice.ps
```

でプリントアウトし、回路図を見ることができます。

auto コマンドを使うには、

- 全てのプログラムが一つのファイルにはいっていて
- トップモジュール名がファイル名 (の `.sfl` の前の部分) と一致している

必要があります。注意して下さい。モジュール名には-(マイナス) は使えません。

6 例題

例題 1-1
UP/DOWN 端子を持つサイコロを記述せよ。
例題 1-2
IKA 端子をアクティブにすると、偶数の目しかでないいかさまさいころを記述せよ。
例題 1-3
偶数の目が奇数の目の倍出易いいかさまサイコロを記述せよ。
例題 1-4
作成したプログラムを論理合成し、回路図をプリントアウトして回路を検討せよ。

[第 2 週]

7　簡単な演算器

7.1　アキュムレータと ALU

CPU の内部で演算を行なう装置を ALU(Arithmatic Logic Unit) といいます。演算を実行するためには、変数を仮に蓄えておくレジスタが必要であるので、ALU とレジスタがセットになって、演算装置を形成します。

ここでは、図 1 の構造で入力された 2 つの数を足す装置を記述します。本来、入力される値はメモリから取ってくるべきですが、メモリとの配線や初期値の格納がめんどうなので、ここでは、スイッチから取ってくることにします。ユーザは入力値を設定するスイッチ (val) と、cancel スイッチを操作します。

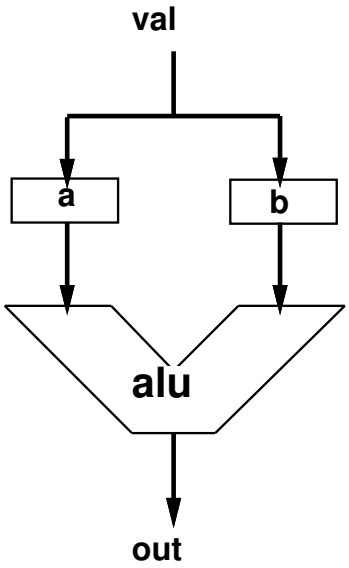

図 3: 演算装置

- read1 のステートで val に値をセットする。
- 1 クロック進める
- read2 のステートで val にもうひとつの値をセットする。このとき、cancel スイッチを 1 にすると、次のクロックで、read1 に戻り、最初の値のセットをやり直すことができる。
- cancel をセットしていない状態でもう 1 クロック進めると read1 で読み込んだ値と、read2 で読み込んだ値の合計が out に出力される。

以上の動作を実現する状態遷移を図 4に示します。

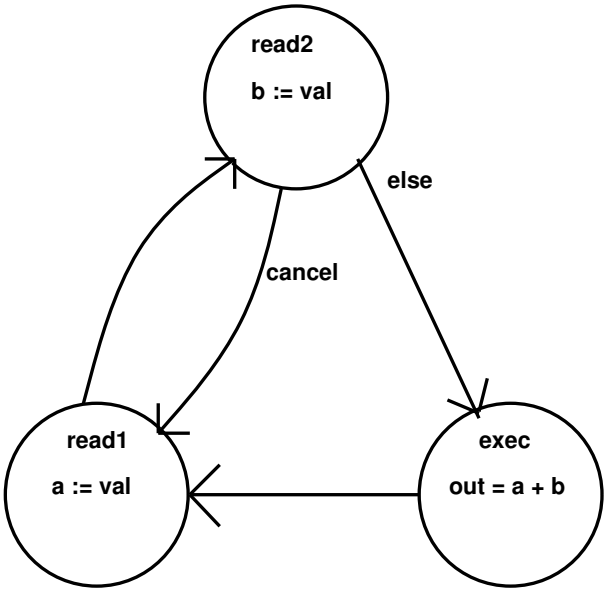

図 4: 状態遷移

7.2 stage 文と state 文

この動作を記述するには状態遷移の記述を行なう必要があります。このために、SFL ではステートマシンを動かす場所として stage を用意しています。SFL では、この stage 上で、タスクを動かしそれらの間の通信を記述することにより、パイプライン動作等がスマートに記述できるようになっていて、これが大きな特徴になっています。しかし、stage 上でのタスク制御は、かなり複雑なシステムでなければ出番がありません。ここでは、単一の stage 上で、タスクが一度発生されたら消滅しないように記述します。以下、4bit 構成の回路の記述例を示します。加算器は、前回用いたフルアダーを用いています。

```
 1      /* simmple calculator */
 2      module calc{
 3        /* 3 bit adder */
 4        submod_type  add3 {
 5          input ina<3>, inb<3>;
 6          output sum<3>;
 7          instrin add;
 8          instruct_arg add(ina, inb);
 9        }
10
11        input    val<3>;
12        input    cancel;
13        output   out<3>;
14        instrin  start;
```

```
15
16        add3 adder;
17        reg a<3>, b<3>;
18
19        stage_name stage_a {
20          task run();
21        }
22
23        instruct start generate stage_a.run();
24
25        stage stage_a {
26          state_name    read1;
27          state_name    read2;
28          state_name    exec;
29          first_state read1;
30
31          /* data read 1 */
32          state read1 par {
33            a := val;
34            goto read2;
35          }
36
37           /* data read 2 */
38          state read2 alt {
39            cancel: goto read1;
40            else: par {
41              b := val;
42              goto exec;
43            }
44          }
45
46          /* execute operation */
47          state exec par {
48            out = adder.add(a, b).sum;
49            goto read1;
50           }
51        }
52      }
```

ここで、1 行目から 17 行目までは前回の知識で理解できるはずです。

7.2.1 stage の定義

19-21 にステージ名とタスク名が定義されています。ここでは、stage_a というステージ上で run というタスクが走ることになります。これは、23 行目のタスクの発生と対応しますが、今回はほとんど意味がありません。

7.2.2 タスクの発生

23 行目でタスクを発生させます。instruct 文自体は前回やったように、ある制御入力がアクティブになった時の処理を記述します。start は、例によってこの回路のイネーブル入力であり、この入力がアクティブになると、stage_a 上で run が生成されます。

7.2.3 ステートの記述

25-52 に図 4のステートマシンが記述されています。文法的には

```
stage ステージ名 {
   state_name ステート名 ;
   state_name ステート名 ;
   .....
   first_state 初期ステート名 ;

   共通した処理の動作記述

   state ステート名 動作記述 ;
   state ステート名 動作記述 ;
   .....
}
```

という構造を持ちます。ステート間の遷移は goto 文で行ないます。記述がない場合、自分から自分への遷移が起こり、状態遷移は行なわれません。動作記述は前回述べた通り、alt, any, par のいずれかの制御文できちんとかこまなければなりません。
例題 **2-1** 制御入力を追加し、加算と減算を行なうことのできるシステムを記述せよ。
例題 **2-1** さらに、AND, OR, NOT や桁移動などの機能を追加せよ。

7.3 シミュレーション

シミュレーションに関しては前回と特に変わる点はありませんが、ひとつだけ、状態の表示を知っておくと便利です。rpt_add または print の記述で、

```
rpt_add P ‘‘Time: %t stage\_a %s out %x\n’’ stage_a out
```

と記述すると、指定したステージの現在のステート名を表示してくれます。例題 2 の各問題を記述し、シミュレーションにより動作を確認して下さい。

8 PARTHENON 使用上のヒント

レジスタの内容が不定のまま読み出す操作をすると、実行時エラーになります。実行 (seconds によるシミュレーション) の時には、まず全てのレジスタに何らかの値がセットされるようにして下さい。例えば、rst が 1 であればレジスタがリセットされるようにプログラムを作ってあるのであれば、まず rst を 1 にして、その部分が実行されるように動かして下さい。

[第 3,4 週]

9 PICO-16

いよいよ今回は、簡単な CPU を SFL によって記述します。ここからは、班毎に実験を行ないます。

本実験では、CPU の命令セットは、基本的には皆さんの設計におまかせします。しかし、いきなり「作れ」と言っても、SFL もまだちょっと使っただけで、時間も足りないし無理な点が多いでしょう。そこで、ここでは簡単なマイクロプロセッサ PICO を紹介しますので、これを参考にして皆さん独自の命令や構成を検討して下さい。

ここでは簡単なマイクロプロセッサ PICO-16 を紹介します。PICO-16 は、慶応大と東京工科大で共同で開発した実験教育用の CPU です。実験教育用ですから、命令セットは様々なバラエティを許しますが、基本方針は以下の通りです。

- RISC である。すなわち単一命令長、Load/Store アーキテクチャである。
- 基本的に命令長を半分にして、半分はイミーディエイトにする命令群とそれ以外から構成される。
- レジスタ数は可能な限り多く取る。
- 基本的には 2 オペランド。
- 命令長の半分はイミーディエイト領域に用いる。

というわけです。以下、PICO-16 の命令セットを解説します。

9.1 PICO-16 の命令セット

すべての命令は 16 ビット長で、5 ビットの命令操作コード (opcode) があります。命令形式は I 形式, R 形式, J 形式の 3 種類があり、命令長の制限によりレジスタ数は 8 としています。

- I 型命令 (即値命令) は語、半語に対するロードとストア、即値演算などに使用する。
- R 型命令 (レジスター レジスタ命令) は算術論理演算、分岐命令などに使用する。命令操作コードの他にファンクションコードを有している。
- J 型命令 (ジャンプ命令) はジャンプ命令、ノーオペレーション、ホールト命令などに使用する。

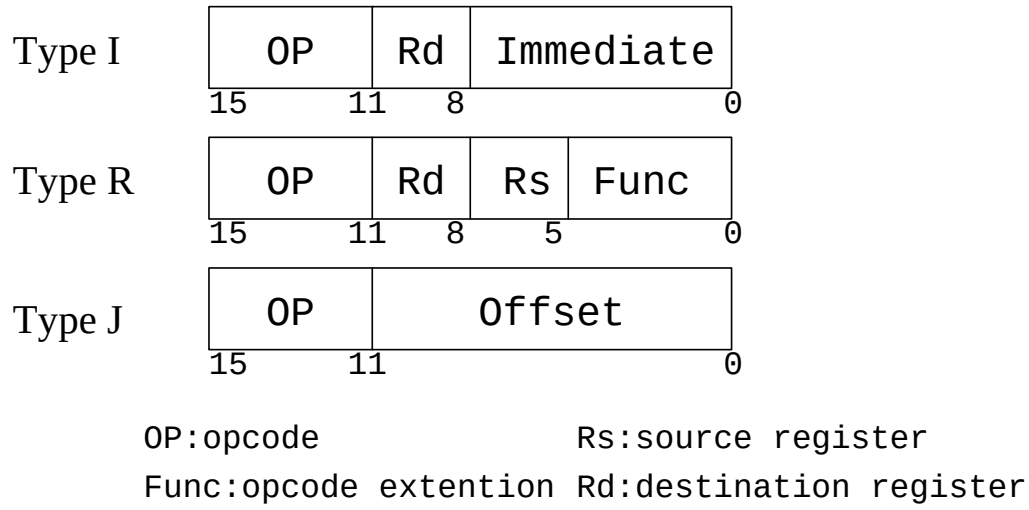

図 5: 命令形式

9.2 PICO-16 の命令の例

ここでは、PICO-16 の命令セットの中で基本的な命令のみを示します。PICO-16 は R0 ~ R7 の 8 つのレジスタを持っています。命令にはロード/ストア、算術論理演算、ジャンプなどの種別があります。順に代表的な命令セットの作り方を紹介します。

最初はロード / ストア命令です。LD 命令と ST 命令は、フィールドが足りないため、レジスタ間接指定のみを用いています。一般に RISC ではパイプラインの高速化のため凝ったアドレッシングモードは用いないのが普通です。LDL(*Load Low Immediate*) 命令はレジスタの下位半分に値をロードし、上位半分に 0 を設定する命令です。LDH と組み合わせて 16 ビット定数の設定は 2 命令で行えます。

命令形式		意味
LDL	Rd, #imm	Rd $\leftarrow 0^8$ ## $imm_{7..0}$;
LDH	Rd, #imm	Rd $\leftarrow imm_{7..0}$ ## $Rd_{7..0}$;
LD	Rd, (Rs)	Rd $\leftarrow$ M[Rs];
ST	(Rd), Rs	M[Rd] $\leftarrow$ Rs;
MOV	Rd, Rs	Rd $\leftarrow$ Rs;

表 1: ロード/ストア命令

算術論理演算はレジスター レジスタ間及びレジスター 即値間でのみ行なえます (ロード/ストア ・アーキテクチャ)。即値形式では 8 ビットの 0 拡張 (上位 8 ビットは 0 にして 16 ビットとする) された即値が利用できます。ここでは加算、減算のみを示しましたが、同様に AND、OR、XOR、シフトなど単純な算術論理演算を備えることができます。

命令形式		意味
ADDI	Rd, #imm	Rd ← Rd + $(0^8$ ## $\text{imm}_{7..0})$;
SUBI	Rd, #imm	Rd ← Rd - $(0^8$ ## $\text{imm}_{7..0})$;
ADD	Rd, Rs	Rd ← Rd + Rs;
SUB	Rd, Rs	Rd ← Rd - Rs;

表 2: 算術論理演算

制御はジャンプ命令で行ないます。ジャンプ命令は条件分岐と無条件分岐がありますが、条件分岐はステータスレジスタを持たずに、レジスタの値をチェックして 0 かどうか判定して飛ぶ方法をとっています。これだけだと大小比較が不可能なので、他に大小比較をやって結果をレジスタに 0 か 1 かで保存する比較命令を設ける必要があります。

命令形式		意味
J	disp	PC ← PC + 2 + $((\text{disp}_{10})^5$ ## $\text{disp}_{10..0})$;
BEQZ	Rd, disp	if (Rd == 0)
		PC ← PC + 2 + $((\text{disp}_7)^8$ ## $\text{disp}_{7..0})$;
BNEZ	Rd, disp	if (Rd != 0)
		PC ← PC + 2 + $((\text{disp}_7)^8$ ## $\text{disp}_{7..0})$;

表 3: 制御命令

9.3 PICO-16 のデータパス

図 6に PICO-16 のデータパスを示します。加減算、論理演算能力を持つ ALU に汎用レジスタ (R0-R7) が 8 つ、プログラムカウンタ (PC)、インストラクションレジスタ (IR)、メモリアドレスレジスタ (MAR) とメモリデータレジスタ (MDR) から成る 3 バス方式の典型的なデータパスです。レジスタには入力と出力にそれぞれデータを仮に保持しておくバッファレジスタ (rbuf1, rbuf2, rbuf3) をつけてあります。 (ちなみにこれは、実際の商用 CPU でもレジスタファイルを使う場合、同様な構成です。) 全てのレジスタは 16bit です。

外部メモリは、実際に外付け RAM を使います。この RAM はアドレス線は mar に直結され、データは先に紹介したように、双方向で datain, dataout に接続されます。mar で示した番地に対して、oe=0 で読みだし、we=0 で書き込みを行ないます。この oe と we は安定して L, H のレベルが出力されていないといけないので、CPU 内部に moe, mwe というフリップフロップを設けて、これに値を設定して読み書きを行ないます。わかり易くするため、moe, mwe はアクティブ 1 にして出力を反転して oe, we に繋いでいます。

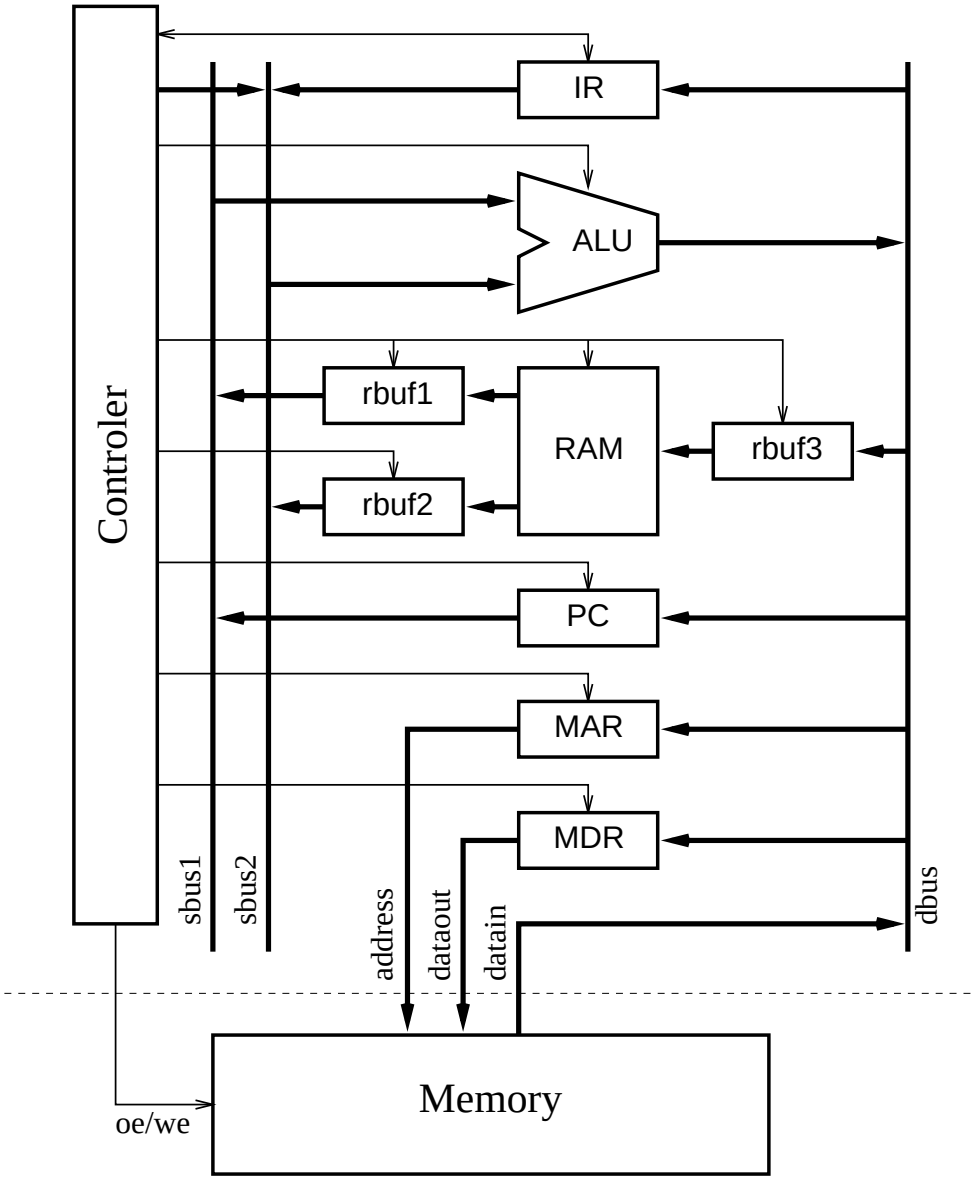

図 6: PICO のデータパス

ちょっと変わっている所は、データバスが入力と出力に分離されていることです。このため、外部にメモリを繋ぐ場合少し工夫が必要です。

9.4 実行の基本ステップ

PICO-16 の命令の実行は、以下の 6 ステップからなります。これらの 6 ステップを繰り返すことにより実行が進んでいきます。ここでは、LD, ST, ADD, LDL, J 命令のみを示します。←は、次のステップに移る際に値が左辺のレジスタへ値が格納されることを示します。

1. 命令フェッチステップ 1(IF1)

$$MAR \leftarrow PC$$

MAR に PC を格納して、PC の値を address に出力します。実際には $PC \rightarrow sbus1 \rightarrow ALU$(素通し) $\rightarrow dbus \rightarrow MAR$ の経路を通って値が伝えられます。

$$moe \leftarrow 0b1$$

moe を 1 にすることにより、メモリへの読みだし要求 (oe) をアクティブにします。

2. 命令フェッチステップ 2(IF2)

$$IR \leftarrow datain$$

メモリから読みだした値 (命令) を IR に格納します。

$$moe \leftarrow 0b0$$

moe を 0 にして、メモリ読みだし要求 (oe) をインアクティブ (アクティブでない状態) にします。

3. レジスタリードステップ 1(RR1)

$$rbuf2 \leftarrow R(IR_{7..5})$$

IR の 7 ビット目から 5 ビット目で指定される番号のレジスタ (Rs に相当) から値を読みだして rbuf2 に格納します。

$$PC \leftarrow PC + 2$$

次の命令の実行に備えて PC の値を 2 増やします。実際には sbus2 に 2 という定数が用意され、$PC \rightarrow sbus1 \rightarrow ALU(sbus2$ と $sbus1$ を加算$) \rightarrow dbus \rightarrow PC$ の経路を通って値が伝えられます。

4. レジスタリードステップ 2(RR2)

$$rbuf1 \leftarrow R(IR_{10..8})$$

IR の 10 ビット目から 8 ビット目で指定される番号のレジスタ (Rd に相当) から値を読みだして rbuf1 に格納します。

R type:

LD:

$$MAR \leftarrow rbuf2$$

$$moe \leftarrow 0b1$$

ST:

$$MDR \leftarrow rbuf2$$

5. 実行ステップ 1(EX1)

R type:

LD:

$$rbuf3 \leftarrow datain$$

ST:

$$MAR \leftarrow rbuf1$$

$$mwe \leftarrow 0b1$$

ADD:

$$rbuf3 \leftarrow rbuf1 + rbuf2$$

LDL:

$$rbuf3 \leftarrow 0b00000000\#\#IR_{7..0}$$

IR の 7 ビット目から 0 ビット目までの 8 ビットの値を 16 ビットに 0 拡張して rbuf2 に格納します。

J:

$$PC \leftarrow PC + (IR_{10})^5\#\#IR_{10..0}$$

IR の 10 ビット目から 0 ビット目までの 11 ビットの値を 16 ビットに符号拡張して PC に加算します。。

6. 実行ステップ 2(EX2)

J, ST 以外

$$R(IR_{10..8}) \leftarrow rbuf3$$

全命令

$$moe \leftarrow 0b0$$

$$mwe \leftarrow 0b0$$

例題 3-1
1 から 10 までの合計を求めるプログラムを作れ。
例題 3-2
$0018_{(16)}$番地からの連続するアドレスに格納されている 8 つの 16 ビット整数の合計を求め、$0016_{(16)}$番地に格納するプログラムを作れ。
例題 3-3
今、仮に各命令が下表のようにコーディングされているとする。例題 3-1,3-2 のプログラムを機械語に変換せよ。

命令	OP	FUNC
LDL	00000	—
LDH	00001	—
ADDI	00010	—
SUBI	00011	—
LD	11111	00000
ST	11111	00001
MOV	11111	00010
ADD	11111	00011
SUB	11111	00100
J	10000	—
BEQZ	10001	—
BNEZ	10010	—

例題 3-4
この他の命令は、各ステップでどのような操作をすればよいか。それぞれ考えよ。
例題 3-5
このままだと Jump 命令は全てのアドレス空間を飛び切れない。うまい方法を考えよ。
例題 3-6
大小比較をする命令はどうすればよいか。いくつか案を出して検討せよ。
例題 3-7
サブルーティンコールの方法を考えよ。

10 SFL による PICO-16 の記述

PICO-16 を SFL で記述した場合の記述例を別紙に示します。ただし、命令自体の動作を記述しているのは LD,ST,LDL,ADD 命令だけです。この SFL の記述は合成されたハードウェアを減らすために、いろいろ工夫をしています。順に説明します。

- サブモジュール
  まず、module alu16 は 16bit の加減算器ですが、制御入力の切替えにより A 入力 B 入力それぞれをす通し (tha, thb)、加算 (add) の機能を切替えます。皆さんはこれに減算 (sub) その他必要

な機能を付け加えて下さい。本体は前回やった Adder を 16bit に拡張した簡単なものです。前回あまり意味をなさなかった制御入力と instr_arg の使い方がわかると思います。

dec5to32 は 5bit 入力のどの値が来たかを判別するためのデコーダで、命令のオペコード部 (OP)、ファンクション部 (FUNC) を判別するため、2 個の実体を定義します。

- 外部メモリのアクセス
  外部メモリは、実際に外付け RAM を使います。この RAM はアドレス線は mar に直結され、データは先に紹介したように、双方向で datain, dataout に接続されます。mar で示した番地に対して、oe=0 で読みだし、we=0 で書き込みを行ないます。この oe と we は安定して L, H のレベルが出力されていないといけないので、CPU 内部に moe, mwe というフリップフロップを設けて、これに値を設定して読み書きを行ないます。わかり易くするため、moe, mwe はアクティブ 1 にして出力を反転して oe, we に繋いでいます。

- RAM の使用
  レジスタは、FPGA で用意されている 16words × 1bits の RAM を用い、8words× 16bits のレジスタファイルを構成しています。ram8x16 は sfl 上では`circuit_type` で定義されていますが、FPGA に RAM を実現する機能があるので、これで大丈夫です。(本当は、16×16words の RAM が作れるのですが、レジスタ数は 7 つしかないので、こうなっています。) 但し、RAM は一度に一つのアドレスからしか値を読み出せないので、レジスタの内容を sbus1 と sbus2 にそれぞれ出力するには 2 つの state が必要になってしまいます。さらに、書き込みデータを確実に書き込むために、state が余計に 1 つ必要になります。レジスタ rwe を 1 にすることにより書き込みを行ないます。

- 入出力、レジスタ
  アドレスバス (address), データバス (datain, dataout) の他にモニタ用の `o_sbus1`, `o_sbus2`, `o_dbus`, `o_pc` が用意され、動作をモニタできるようにします。`r_sbus1`, `r_sbus2`, `r_dbus` はデータパスの状態をモニタするためのレジスタです。

  仮端子 sbus1, sbus2, dbus はそれぞれソースバス 1,2、ディスティネーションバスに相当します。データの転送は面倒でも全てこの仮端子を経由して行ないます。このようにしないと論理合成により専用線と入力を選択するマルチプレクサが各所に生成されて、収拾がつかなくなります。

stage_name 以降は、状態遷移を記述するための構文です。SFL はパイプライン動作を記述できる点が特徴で、stage と task の定義と起動はパイプラインをうまく書くための構文です。今回はパイプライン動作は行なわないので、

```
stage_name stage_a {
task run();
}
instruct start generate stage_a.run();
```

まではおまじないと思っても結構です。run という task が stage_a という名前のステージ (つまり状態遷移の起きる場所) で起動されます。以降の文は PICO-16 の状態遷移の記述です。

SFL の構文ではまず、ステージの宣言の後に使用する状態の名前を記述します。ここで使う状態は次の 4 つです。

- init: 初期設定、PC を 0 にする。
- if1: 命令フェッチ 1、PC の値を mar に格納する。
- if2: 命令フェッチ 2、命令を 1 ワード (16bit) 取ってきて IR に入れる。同時に PC の値を 2 増やす。
- rr1: RAM は同時に一つの値しか読めないので、まず一つ目 (Rs) の値を読み出す (reg2)。
- rr2: ST, LD 命令では mar, mdr をセットする。 その他の R タイプの命令では、もう一方 (Rd) のレジスタを読み出す (reg1)。また、他の命令でも所定の処理を行なう。
- ex1: 命令の実行。
- ex2: レジスタへの書き込み。

PICO-16 は最初に init を一回通る他は、if1, if2, rr1, rr2, ex1, ex2 を順番に回って命令を実行して行きます。

SFL の記述では初期状態を first_state 文で定義できます。ここでは init を初期状態として定義しています。その後の par 文で全ての状態で共通に有効である操作を記述します。この場合、モニタ用の出力の接続を記述しています。後は順に各状態での操作を記述していきます。状態が終了し、条件が成り立ったら goto 文で次の状態に遷移します。以下の二点に注意してこの SFL の記述を読みとってみましょう。

- =文は、その状態で有効になる。:=文はその状態から次の状態への移り変わりのクロックで有効になる。
- 全てのレジスタに対する代入は一度ディスティネーションバスを介して行なう。

例題 **3-8**
例に示した pico.sfl で各命令はどのようにコーディングされているか考えよ。

11 seconds を用いたシミュレーション

プロセッサの外側のメモリと一緒に seconds でシミュレーションをします。

このシミュレーションのために、`test.sfl` を用意しました。ここでは、プロセッサ本体である pico16 が submodule として使われており、外付けのメモリが circuit_type として定義されています。

さて、この`test.sfl` を用いてシミュレーションを行なうには、次のコマンドファイルを用います。このファイルは test という名前になっています。ここでは、`pico16.sfl` というファイルにプロセッサ本体が記述されているという前提になっています。プロセッサのトップモジュール名は pico16、入出力端子名は配布したプリントと同じにしておいて下さい。

```
1   sflread test.sfl
2   sflread pico16.sfl
3   autoinstall test /
4
```

```
 5  rpt_add CLK "####  clock:%t  state:%s  ####\n"\
 6  /PICO/stage_a
 7  rpt_add P1  "    ir:%04x    pc:%04x   rwe:%b      moe:%b      mwe:%b\n"\
 8  /PICO/ir /PICO/pc /PICO/rwe /PICO/moe /PICO/mwe
 9  rpt_add P2  "  reg1:%04x  reg2:%04x  reg3:%04x   mar:%04x   mdr:%04x\n"\
10  /PICO/reg1 /PICO/reg2 /PICO/reg3 /PICO/mar /PICO/mdr
11  rpt_add BUS " sbus1:%04x subs2:%04x  dbus:%04x\n"\
12  /PICO/sbus1 /PICO/sbus2 /PICO/dbus
13  rpt_add IO  "  adrs:%04x  dout:%04x   din:%04x    oe:%b       we:%b\n"\
14  /PICO/address /PICO/dataout /PICO/datain /PICO/oe /PICO/we
15
16  test.p16
17
18  set /PICO/moe 0
19  set /PICO/mwe 0
20
21  set init 1
22  hold init
23  forward +1
24  set init 0
25  set run 1
26  hold run
```

ここで、test.p16 はあらかじめメモリの中に用意しておくデータで、以下の内容です。また、最後に\がある行は、プリントアウトの都合で改行されていて、実際には次の行とつながっている (改行がはいらない) ことを示します。

```
 1  #
 2  #       check program for PICO
 3  #
 4
 5  memset /MEMORY/memory  0 X0118 XFA20 X011A XFB20 XFA63 X011C XF941 X0000
 6  memset /MEMORY/memory  8 X0000 X0000 X0000 X0000 X1234 X1010 X0000 X0000
 7  memset /MEMORY/memory 16 X0000 X0000 X0000 X0000 X0000 X0000 X0000 X0000
 8  memset /MEMORY/memory 24 X0000 X0000 X0000 X0000 X0000 X0000 X0000 X0000
 9
10  #       LDL     r1, 0x18        | 0000:0118   0000 0001 0001 1000
11  #       LD      r2, r1          | 0002:FA20   1111 1010 0010 0000
12  #       LDL     r1, 0x1A        | 0004:011A   0000 0001 0001 1010
13  #       LD      r3, r1          | 0006:FB20   1111 1011 0010 0000
14  #       ADD     r2, r3          | 0008:FA63   1111 1010 0110 0011
15  #       LDL     r1, 0x1C        | 000A:011C   0000 0001 0001 1100
```

```
  16   #       ST      r1, r2          | 000C:F941   1111 1001 0100 0001
```

この部分を適宜変更して、シミュレーションを行なって下さい。

ここの書式は

```
memset メモリ名 スタートアドレス データ データ ...
```

というもので、スタートアドレス以降のアドレスに順にデータが格納されます。データの最初の大文字の X は、16 進数を表します。X 以降の部分は数字と a ~ f の英小文字で構成されます。

test.sfl で定義したメモリは 16bit 幅なので、メモリ上のアドレスを 2 倍したものが pico16 から見えるアドレスの値になります。つまり最初の X0118 は pico16 から見た 0 番地と 1 番地に、次の XFA20 は 2 番地と 3 番地に格納されることになります。ここでは、11 行目以降にコメントで示されているプログラムの機械語がまず格納され、それに引き続いて $0018_{(16)}$番地以降に $1234_{(16)}$、$1010_{(16)}$、$0000_{(16)}$ というデータを格納しています。

seconds でシミュレーションを実行すると、エラーメッセージがたくさん出ますが、(W) で始まっているものはワーニング (警告) で、ほとんどの場合無視して大丈夫です。とくに、値が定まっていない端子を参照した場合、`.... is not driven` というメッセージが出ますが、これは問題ありません。

(E) で始まるものはエラーなので、メッセージを見て対処して下さい。

また、メモリの内容を確認するためのコマンドファイル mem、レジスタの内容を確認するためのコマンドファイル reg も用意しました。利用して動作を確認して下さい。メモリのアドレス空間は広いので、mem は最初の 32 バイトの内容を見ることができます。必要に応じて書き換えて下さい。

12 FPGA の配線情報への変換

seconds 上でのシミュレーションで、正しく動作することが確認できたら FPGA の配線情報へ変換します。この過程は大きく、

1. PARTHENON による FPGA 用回路情報の生成
2. XACT とよばれる FPGA 用コンパイラによる配線情報の生成

に分けられます。いずれも大変時間がかかりますので、seconds 上でのシミュレーションをきっちりやって、確実に動作する sfl の記述をつくってから変換作業をするようにします。

PARTHENON による回路情報の生成は、以下のコマンドで行ないます。

```
% auto pico16 xnf XILINX lca_lib
```

これが、エラーなく終ったら TA に確認してもらい、問題なければ配線情報の生成を依頼して、4 週目の実験は終りです。うまくいかない時には、一度

```
% auto pico16 clean
```

を実行して、やり直してみて下さい。

FPGA 用コンパイラは 1 組しかなく、1 班分の配線情報の生成に 1 時間程度かかりますので TA が次の週までにコンパイルしておいてくれます。

[第 5 週]

13 FPGA の構造

通常、IC は、マスクパターンによって、内部のトランジスタや配線が決定されます。しかし、書換え可能なゲートアレイ FPGA は、内部の回路接続 (コンフィグレーションといいます) の情報を外部からディジタル信号の形で与え、内部構造を変えることのできる集積回路です。基本となる回路の構成法の違いにより、様々な種類があるが、この実験では、Xilinx 社の LCA(Logic Cell Array) XC4000 シリーズを使います。この形式の LCA は、現在最も一般的です。

XC4000 シリーズでは、IC 内に CLB(Configurable Logic Blocks) と呼ばれるプログラマブルな論理素子が一定の間隔 (配線領域) で並んでいます。今回の実験で使う XC4010 では 400 個の CLB を持っています。CLB は小規模な PLD(Programmable Logic Device) で、フリップフロップ 2 個と、任意の組合せ回路を構成できる組合せ論理部、周辺マルチプレクサから構成されます。これらの CLB 同士は、配線領域中の配線マトリックスと結線を行なうトランスファーゲート (スイッチのようなものです) により、相互に接続することができます。チップ外部とのインターフェースは IOB (I/O Block) が受け持ち、同様に配線マトリックス、トランスファーゲートを介して CLB と接続されます。

さて、これらの接続をどう行なうかという情報と CLB 内部の組合せ論理部のルックアップテーブル (つまりは真理値表です) は、IC 内部にアレイ状に分布したコンフィグレーションメモリに蓄えられます。コンフィグレーションメモリはスタティック RAM で、その情報は外部からビットシリアルに送られます。RAM の各 bit の出力は、配線用のトランスファーゲートの ON/OFF を制御すると共に、CLB 内の組合せ論理ブロックについては、そのまま真理値表として用いられ、組合せ論理を決定します。このような構成により、コンフィグレーション RAM の内容を変えることで、様々な構成のハードウェアを実現できるわけです。

LCA は現在、商用レベルで、集積度は 13000 ゲート相当、動作周波数は、内部フリップフロップが基本的な動作を行なうことのできる周波数 (トグル周波数) が 125MHz に達していて、小規模なマイクロプロセッサを組み込むことができます。構成についても、CLB を基本とするもの以外に、記憶素子のブロックを設けるもの、ゲート全面敷き詰め方式等、様々な方法が実現されています。

この実験で使うのは X4010 といい、比較的大規模の LCA です。ゲート数は 10000 ゲートで、動作周波数は 125MHz です。

13.1 SFL からコンフィグレーション用データまで

皆さんの合成した回路は論理合成の結果、PARTHENON により pico16.xnf という名前のネットリストが作成されますこのネットリストから、Xilinx 社の自動配置配線ツールにより、内部の CLB, IOB の配置と配線が決定され、コンフィグレーションデータが作成されます。この部分の操作は、TA が行ないます。最終的に pico16.exo と呼ばれるファイルが作られます。このファイルには Motorola S-format と呼ばれるフォーマットでコンフィグレーションデータが記述されています。

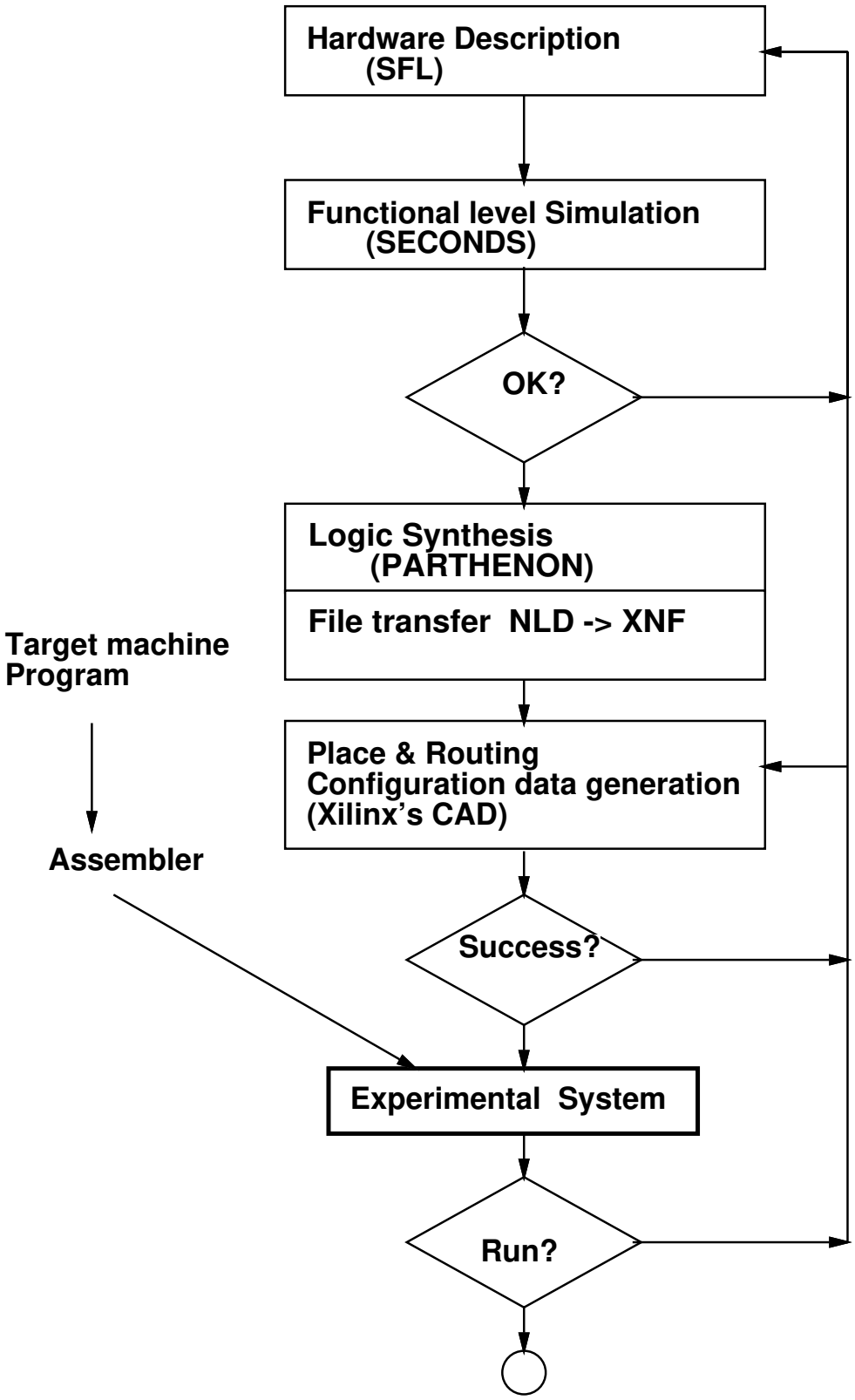
Hardware Description
(SFL)
Functional level Simulation
(SECONDS)
OK?
Logic Synthesis
(PARTHENON)
File transfer NLD -> XNF
Target machine
Program
Place & Routing
Configuration data generation
(Xilinx's CAD)
Assembler
Success?
Experimental System
Run?

図 7: 実験のながれ

14　実験用ボードと動作確認

実験で用いるボードは、この実験のために特別に作成したもので、以下のような構成になっています。

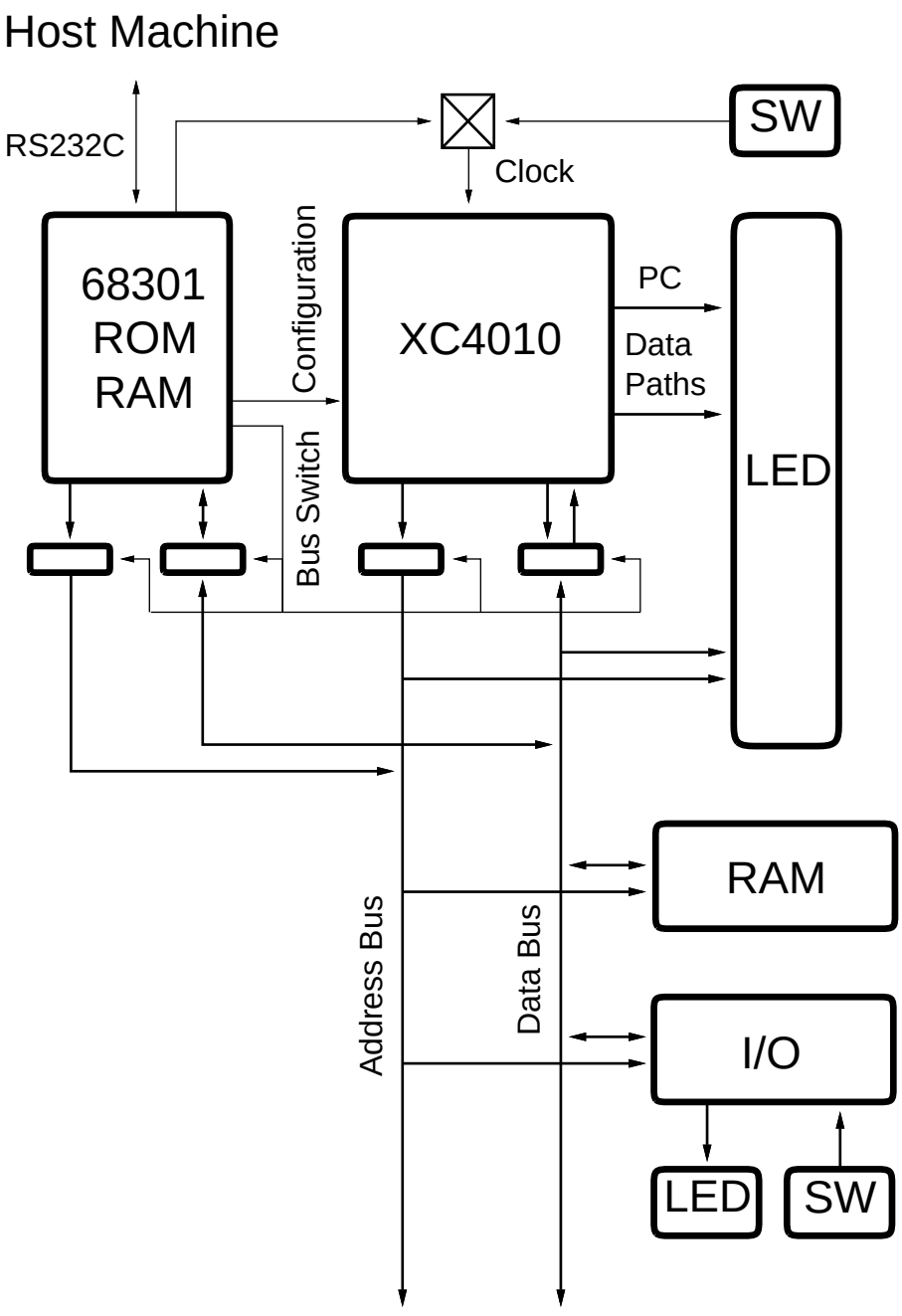

図 8: 実験用ボードの構成

このボードには、TMP68301 という 68000 をコアに持つプロセッサが載っており、コンフィグレー

ションデータの LCA へのロード、pico16 が使用するメモリの内容の設定/表示を行なってくれます。

まず、使用する pico16 が接続されているワークステーションにtelnet コマンドでログインします。(最初から接続されているワークステーションを使っている幸運な人は必要ありません)

telnet *ew4XXX*

次に TA の人から、前回作成を依頼した FPGA のコンフィグレーションデータを受け取ってください。pico16.exo というファイルになっているはずです。確認したら pico16 の電源をいれましょう。LED が明るく光って迎えてくれます。

pico16 とは tip というコマンドを用いて交信します。取り合えず一連の動作を述べて行きます。

- まず、pico16 とワークステーションを交信できるようにします。

```
% tip pico
connected
```

- つぎに、LCA に作製したプログラムを読み込ませます。lcaset と入力しましょう。

```
Pico16(1.0)  > lcaset
S-Format loading for LCA
please type file
```

ここで、~>と入力し、pico16.exo と入力します。

```
~>Local file name? pico.exo
1393 lines transferred in 1 minute 8 seconds
!
start address = 0x10   end address = 0x56f0
lca configuration normally terminated.
End
Pico16(1.0)  >
```

- ここで、実行したいプログラムを入力します。

  実験用ボード上には、FPGA をプロセッサとして動かす際にプログラム等を格納するのに必要なメモリが 32K バイト (0000h ~ 7FFFh) 実装されています。このメモリへの書き込みおよび確認には pmm コマンドを使用します。

  まず、モニタのプロンプトが表示されている状態で

  ```
  pmm
  ```

と入力してください。するとアドレスと現在のメモリの内容が表示されますので、書き込みたい値を 16 進数で 1 ワードずつ入力してください。

このとき Return キーだけを打つと、書き込みは行なわれずに、つまり現在の内容のまま次のアドレスに移ります。また “.”(ピリオド) だけを入力すると、メモリへの書き込みを終了します。この pmm コマンドを使用している様子を図 9に示します。

```
Pico16(1.0)  > pmm            (pmm と打つ)
[pico] 0000: 33c2 ? 6118      (書き込みたい値を打つ)
[pico] 0002: ae8a ? 7100
[pico] 0004: ee89 ? f231
[pico] 0006: dece ? ffb2
[pico] 0008: 5ec9 ? df01
[pico] 000a: 5b7a ? f2cf
[pico] 000c: 7026 ? 3ffa
[pico] 000e: c5ff ? c102
[pico] 0010: 0b96 ? f172
[pico] 0012: 4a96 ? 0ffe
[pico] 0014: 6202 ?           (ここは変更しない)
[pico] 0016: eed0 ?
[pico] 0018: 77db ? 000A
[pico] 001a: 27e4 ? 0000
[pico] 001c: 49b1 ? .         (書き込み終了)
Pico16(1.0)  >
```

図 9: メモリへの書き込み

また次のように、書き込みを行なうアドレスを指定することもできます。

```
pmm 1000
```

この場合、1000h 番地から書き込みを開始します。

- クロックの入力

  FPGA を動作させるためには、外部からクロックを入力する必要があります。このクロックの入力方法には次の二通りの方法があります。

  - 手動で 1 クロックずつ入力する方法

    実験用ボード上の白いボタン (CLOCK) を一回押すと、1 クロック分のパルスを発生します。つまり、seconds 上で “forward +1” を行なったのと同じ動作を、実験用ボード上で行ないます。

  - 連続的に入力する方法

    モニタのプロンプトが表示されている状態で

    ```
    ctime 100
    ```

と入力します。このとき数字は周波数を示します。つまりこの例では 100Hz のクロックを入力していることを示します。また、クロックを止めるには

```
cswitch
```

と入力します。これにより先の手動入力に切り替わります。

```
Pico16(1.0)  > ctime 100
Pico16(1.0)  > cswitch
```

図 10: クロックの入力

- プログラム ・カウンタのリセット

  なんらかの理由でプログラム ・カウンタのリセットを行ないたい場合、モニタのプロンプトが表示されている状態で

  ```
  reset
  ```

  と入力して下さい。これにより、コンフィギュレーション ・データはそのままで FPGA 内部の記憶素子だけを初期化します。

```
Pico16(1.0)  > reset
```

図 11: プログラム ・カウンタのリセット

なお、実験用ボード上の赤いボタン (RESET) を押すと、FPGA のコンフィギュレーション ・データも初期化されてしまいますので注意して下さい。

- 接続の終了

  実験用ボードとの接続を終了するには

  ```
  ~.
  ```

  と入力してください。このとき入力した文字は表示されませんが、気にせず行なってください。正しく終了できた場合、接続する前の状態に戻ります。

15 レポート

レポートは第 6 週に提出です。また、この週にディスカッションを行ないます。
この実験は、

- ハードウェア記述言語がどのようなものか理解すること
- プロセッサの内部構造の基本を理解すること

が目的でした。この目的に沿うように、以下の体裁でレポートをまとめて下さい。

1. 目的
2. 実験結果

   第 1 週: さいころカウンタ

   - 例題 1-1 ~ 1-3 のうち 1 つの SFL ソースコード
   - 例題 1-X:実験中に与える課題の SFL ソースコード
   - 上記ソースコードの概略説明
   - 上記の 2 設計例のうち 1 つの回路図
   - 回路図中のフリップフロップを探し、フリップフロップの機能について考察

   第 2 週: 簡単な演算器

   - 例題 2-1,2-2 の機能を実現した SFL ソースコード
   - 上記ソースコードの概略説明

   第 3,4 週: PICO16

   - PICO16 の SFL ソースコード (各班オリジナル)
   - 班で作成した PICO の仕様と自分で作成した箇所のプログラムの説明。
   - 例題 3-2 のアセンブラコードおよびシミュレーション結果
     - seconds によるシミュレーション結果を本文に添付する。そのままレポートに添付するのではなく、説明に必要な情報のみを編集すること。
   - 例題 3-3:上記例題のアセンブラコードに対応する機械語 (例題 3-8 の命令コーディング表に基づく)
   - 例題 3-4:各班の PICO の SFL ソースコードに基づき，命令語の各ステップにおける動作の説明
   - 例題 3-5 ~ 3-7:PICO16 の拡張方法に関する考察
     これ以外に改良点があれば考察をつける
   - 例題 3-8:各班の命令コーディング表
3. 考察、感想

   - 実験中に与える考察課題

- SFL 記述の感想，実験の感想

レポート作成上の留意点

- レポートには必ず頁をつける。
- プログラムリスト、回路図は付録としてレポートの最後に添付する。レポート本文にはプログラムの説明を記す (概要でよい)。
- auto を実行した際の回路図やプログラム以外の、プリンタが出力する表紙などのレポートに関係のないものを添付しないこと。