例題

```
Z= X + Y - Z
```

これは、A+B の結果がアキュムレータに残っているのでそれがそのまま使えるので簡単である。これがアキュムレータの良い点である。

```
LD 0  0001 0000
ADD 1 0110 0001
SUB 2  0111 0010
ST 2 1000 0010
```

しかし、最初に示した演算では、アキュムレータの値を直接利用することはできない。

```
(X<<1)+(Y<<1)
```

この場合 X の 1 ビットシフトを計算し、その結果をどこかにとっておく必要がある。この中間結果をとっておく場所をここでは 2 番地としよう。

```
LD 0  0001 0000
SL    0100 0000
ST 2 1000 0100
LD 1 0001 0001
SL   0100 0000
ADD 2 0110 0010
ST 2 1000 0010
```

2 番地に一度中間結果を入れてから Y を LD し、シフトをした後、加算をする。入れる。これは、ちょうどメモリ付きの電卓を使う際に中間結果をメモリにとっておくのと似ている。

演習

0 番地,1 番地,2 番地にそれぞれ A,B,C が格納されている。(A-B) OR (B+C) の演算を行い、3 番地に答えを格納するプログラムをアセンブラ表記、機械語の両方で示せ。

2.5 Verilog での記述

2.5.1 加算器の記述

Verilog では、一つのまとまりをもった回路をモジュールと呼ぶ。Verilog の記述は、まずモジュールの名前と入出力を定義することから始まる。

```
module 名前 (
  input 入力 1,
  input 入力 2,
  output 出力 1,
  output 出力 2 );
```

この定義の方法は、C 言語の関数定義とちょっと似ているが、最後にセミコロン (;) が付いて文の形になっている点が違っている。まず a, b の 2 入力を持ち、s の出力を持つ 1bit の加算器の記述を示す。これは以下のように非常に簡単である。

```
/* 1 bit adder */

 module adder (
 input a, b, output s );

  assign s = a + b  ;  // add a b

endmodule
```

コメントは C 言語同様 2 種類 (/* */と//) が可能である。さて、加算器は、後の章に解説するようにリプルキャリアダーからプリフィックスアダーまで様々な構成があるが、Verilog でも VHDL でもここは抽象化して単に'+' で表す。どのような加算器を使うかは、後に論理合成の段階で、必要なコストと性能を考えて、自動合成用の CAD(Computer Aided Design) プログラムが選択する。設計者はこの選択時に必要な指示を与えてやる。s=a+b; の前に assign が付いているのに注意されたい。assign 文は、ある回路の出力ある端子に出力する、あるいは接続することを意味する。ここでは加算の結果を s に出力する、あるいは加算結果出力を s と接続することを示している。最後にモジュール文は endmodule で終わるが、ここは; は付かない。

この記述は adder.v というファイルに入れておく。C 言語のプログラムの拡張子が.c であるのと同様、Verilog-HDL 記述の拡張子は.v である。

次にこれをシミュレーションしてみよう。第 1 章に紹介したように、Verilog は元々シミュレーション記述用の言語であり、シミュレーションに対する指示の方法は多様で、理解するのが大変である。ここはまずは動かしてみるのが良いと思う。シミュレーションを行って動作を確認するためには、入力に値を設定する。という操作と、

下の test.v は上記の加算器のモジュールをテストするシミュレーション用の Verilog 記述である。この記述は、ハードウェアの回路を表すのではなく、テストするための入力の与え方、出力の表示を示す。このような記述をテストベンチと呼ぶ。

Verilog を習得する時にやっかいなのは、テストベンチ中には結構難しい文法が必要とされるのにも関わらず、これを書かないとテスト対象のモジュールがいかに簡単なものであっても全くシミュレーションができない点である。これを避けるためにあ

る種の CAD では、パラメータを与えてテストベンチを自動生成する機能を持つものもある。

で、ここでは、テストベンチの解説は一応しておくが、重要な文法は後でまた解説することにして、直感的に理解する程度で深く突っこまないで先に進みたい。実の所、テストベンチの作り方は定型的なので、一定のパターンをマスターしてこれを場合に応じて修正すれば、かなりの範囲で応用が効く。どうか細部にこだわらず、大体の意味を掴んでほしい。

```
/* test bench */
‘timescale 1ns/1ps

module test;
   parameter STEP = 10;
   reg ina, inb;
   wire outs;
   adder adder_1(.a(ina), .b(inb), .s(outs));
   initial begin
      $dumpfile("adder.vcd");
      $dumpvars(0,adder_1);
      ina <= 1’b0;
      inb <= 1’b0;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1’b0;
      inb <= 1’b1;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1’b1;
      inb <= 1’b0;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1’b1;
      inb <= 1’b1;
     #STEP
      $display("a:%b b:%b s:%b", ina, inb, outs);
      ina <= 1’b0;
      inb <= 1’b0;
     $finish;
   end
```

```
endmodule
```

最初に出てくるタイムスケール文は、シミュレーションの基本時間、時刻刻みの最小時間を定義する。

```
‘timescale 1ns/1ps
```

ここでは前者は 1ns(ナノ=10^{-9} 秒)、後者は 1ps(ピコ=10^{-12} 秒) とした。
次に、パラメータ文を使って今回シミュレーションを行う 1 ステップの時間を定義する。

```
parameter STEP = 10;
```

このパラメータ文は実行時に外部から値を設定できるのがミソで、便利なので良く用いられる。ここでは 10nsec を 1 ステップとする。
次に加算器に外部から値を与えるための入力、値を取り出すための出力用の端子を定義してやる。ここで、入力には、値を保持するためレジスタ (reg) として宣言し、出力は単に値を取り出せば良いので端子名を宣言 (wire) してやる。

```
reg ina, inb;
wire outs;
```

次に、adder モジュールの実体を定義して入出力を接続する。先にモジュール文で adder.v 内に定義した加算器のモジュール名を最初に指定し、次に実体名 (ここでは adder_1) を宣言して実体を生成する。一つのモジュールから多数の実体を別の名前で生成することができる。括弧内で、入出力の接続を行う。. の後にモジュール内の入出力名、() の中に接続する外部端子の名前を記述する。ここでは adder 内で宣言された a 入力に外部の ina を、b 入力に外部の inb を、s 出力に外部の outs を接続する。

```
adder adder_1(.a(ina), .b(inb), .s(outs));
```

これで宣言が終わりで、次は initial 文で、シミュレーションの動作を記述する。initial 文は、開始後一回だけ begin から end までの文が順に実行される。

```
    $dumpfile("adder.vcd");
    $dumpvars(0,adder_1);
```

最初に のついた二つの文が実行される。*Verilog* 内で は、シミュレーションを制御する上で必要な特殊なタスク (操作) を示す。この dumpfile と dumpvars では波形ビュアーで見るためのファイル名 (adder.vcd) と、信号を記録する範囲が指定されている。
次にいよいよシミュレーションを開始する。

```
 ina <= 1'b0;
 inb <= 1'b0;
#STEP
 $display("a:%b b:%b s:%b", ina, inb, outs);
 ina <= 1'b0;
 inb <= 1'b1;
#STEP
 $display("a:%b b:%b s:%b", ina, inb, outs);
 ina <= 1'b1;
 inb <= 1'b0;
#STEP
 $display("a:%b b:%b s:%b", ina, inb, outs);
 ina <= 1'b1;
 inb <= 1'b1;
#STEP
 $display("a:%b b:%b s:%b", ina, inb, outs);
 ina <= 1'b0;
 inb <= 1'b0;
$finish;
```

ina, inb への値の代入は、後に解説するノンブロック代入文 <= で行っている。これで 0,1 が順に設定される。#STEP は、10nsec の時間経過を示す。すなわち、以下の文は、

```
 ina <= 1'b0;
 inb <= 1'b0;
#STEP
 $display("a:%b b:%b s:%b", ina, inb, outs);
```

まず、ina, inb に 0 を設定し、10nsec 時間が経過したら、ina, inb, outs を表示する、という意味である。さて、Verilog では定数を以下のように表現する。

<ビット幅>'<基数><数値>

基数は

- b：2 進数
- h：16 進数
- o：8 進数

である。なにも書かないと 10 進数になるので、数を直接書く際は、必ず桁数をはっきりさせて 2 進数、あるいは 16 進数で書くことをお勧めする。ここで 1'b0 は 1bit の 0 を、1'b1 は 1bit の 1 をそれぞれ示す。

```
$display("a:%b b:%b s:%b", ina, inb, outs);
```

display 文はその名の通り、ina,inb,outs の値を表示するためのものである。display 文の記述は C 言語の printf に似ているが、2 進数を表示するための%b を持っている。また、自動的に改行が入る。以下、入力を変えて、それぞれ 10nsec 時間を経過させて結果を出力していく。最後は finish 文でシミュレーションを終了する。

2.5.2　シミュレーションの実行

では、シミュレーションをしてみよう。第 1 章で紹介した ikarus verilog をインストールしてある環境を想定する。

```
iverilog test.v adder.v
```

で、シミュレーションの実行形が生成される。シミュレーション自体は、

```
vvp a.out
```

で実行される。ここでは最初は a 入力、次は b 入力、最後に加算結果が出力される。a.out の名前が気になる方は、

```
iverilog test.v adder.v -o adder
vvp adder
```

としても良い。

```
a:0 b:0 s:0
a:0 b:1 s:1
a:1 b:0 s:1
a:1 b:1 s:0
```

が出力される。次に波形を見てみよう。テストベンチで記述した波形データを入れておくファイルを指定して立ち上げる。

```
gtkwave adder.vcd
```

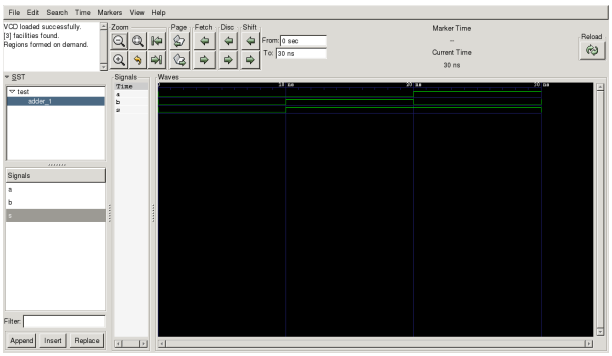

図 2.7: gtkwave による波形表示

左側の SST と書いてある所に test とテストベンチのモジュール名が表示されるのでそこをクリックすると adder_1 という加算器の実体名が表示される。ここをクリックすると信号名が Signals ウインドウに表示される。ここで、見たい信号をクリックしてから Append をクリックすると、Waves ウインドウに波形が表れる。gtkwave を始めて使う時に戸惑うのは、Waves の設定が最小時刻刻みになっているため、最初のきわめて一部しか表示されない点である。Zoom の所の虫眼鏡の-を連打して表示スケールを調整すると、見たい範囲で波形が見られる。gtkwave は優れた GUI を持っているため、勘で結構使える。ここではあまり解説しないが、色々機能を試して欲しい。ちなみに終了の際は File→Quit で確認用の小ウインドウが出てくるので、ここで Yes を押してやる。

演習 1

adder の記述を書き換えて、減算、AND、OR などにして試してみよ (減算は実は加算と同じになる)。

2.5.3 ALU の記述

次にやや難しい ALU を記述して見る。まず、1 ビットはあんまりなのでビット数を拡張することにしよう。ハードウェアでは配線を何本か束にして扱う場合が多い。これをバス (Bus) と呼び、Verilog では大括弧でくくって，MSB:LSB の形でその範囲を表わす。本書では簡単のため LSB は常に 0 としよう。

入出力 16 ビット、コマンド 3 ビットの ALU は次のように定義される。

```
module alu (
  input [15:0] a, b,
```

```
  input [2:0] com,
  output [15:0] y );
```

次に中身を定義する。ALU の場合、入力が特定の条件ならば入力同士の演算の結果を出力する。このような場合、Verilog では以下のように記述する。

```
assign 出力 = 条件? 式 1   :  式 2;
```

条件が真であれば式 1 の結果が出力され、そうでなければ式 2 の結果が出力される。条件を複数書くこともできる。これは case 文と似ている。条件は前から順にチェックされるので、優先順位は先に書いた方が高いことになる。下の例では、条件 1-3 がどれも真でなれば、コロンの後に書いた式 4 の結果が出力される。

```
assign 出力 = 条件 1? 式 1 :
              条件 2? 式 2 :
              条件 3? 式 3 : 式 4;
```

今回の ALU は、以下のような機能を定義した。

```
com     出力
000     A
001     B
010    A AND B                    AND
011    A OR B                     OR
100    A を 1 ビット左シフト      SL
101    A を 1 ビット右シフト      SR
110    A+B                        ADD
111    A-B                        SUB
```

これを Verilog 記述すると、以下のようになる。

```
 assign y =  com==3'b000 ? a:
             com==3'b001 ? b:
             com==3'b010 ? a & b:
             com==3'b011 ? a | b:
             com==3'b100 ? a<<1:
             com==3'b101 ? a>>1:
             com==3'b110 ? a + b: a - b ;
```

Verilog は、表 2.1 に示す演算を記述できる。ここでは AND, OR, 左右のシフト、加算、減算を使っている。AND, OR では、16 ビットの入力のそれぞれの桁同士で演算が行われ、答えも 16 ビットになる。

表 2.1: Verilog の基本的演算子

ビット演算	~	NOT
	&	AND
	~&	NAND
	|	OR
	~|	NOR
	^	Ex-OR
	~^	Ex-NOR
シフト演算	<<	左シフト
	>>	右シフト
等号、関係演算	==	等しい
	!=	等しくない
	===	等しい (x,z も比較)
	!==	等しくない (x,z も比較)
	<	小さい
	<=	以下
	>	大きい
	>=	以上
算術演算	+	加算
	−	減算
	*	乗算
	/	除算
	%	剰余算
論理演算	!	否定
	&&	論理積
	||	論理和

ここで、論理演算は、C 言語で用いるのと同様に、条件の論理的な真偽に対する演算を行い、結果は 1 ビットの値となる。ここでは比較演算子==を使っている。比較対象は 3 ビットの 2 進数なので、3'b000 と書く。

演算子には優先順位があるので注意されたい。表 2.2 で上位にあるもの程強い。

2.5.4 define 文を使って格好を付ける

先に示した ALU の記述はきちんと動作し、合成もできるが、あまり格好良いと見なされない。これは、コード中に 3'b000 とか 15:0 など定数が直に記述されているからである。これらのハードウェアの記述に必要な定数 (マジックナンバーと呼ばれる

表 2.2: 基本的演算子の優先順位

高い	^ ~ !&| + - (単項演算子)
	*/ %
	+ -
	<<>><<<>>>
	<<=>>=
	==! ====! ==
	& ~& ~ ~^
	| ~|
	&&
	||
低い	

こともある) は、直接コード内に書かない方が良い。これは、コードが読みにくくなることと、仕様変更があった際にコード中のあちこちを変更しなければならないためである。そこで、Verilog では C 言語と同様 define 文でこれらの定数をあらかじめ定義しておく。C 言語と違って define 文の前と定義された文字列を使う場合はバックシングルコーテーション (‘) を使う。下記は 4 種類の演算のみこの変換を行った例である。

```
‘define DATA_W 16  // bit width
‘define SEL_W 3 //control width
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010

module alu (
  input [‘DATA_W-1:0] a, b,
  input [‘SEL_W-1:0] s,
  output [‘DATA_W-1:0] y );
  assign y = s==‘ALU_THA ? a:
             s==‘ALU_THB ? b:
             s==‘ALU_AND ? a & b: a + b ;
endmodule
```

演習 2

上記の記述を全ての演算について行え。
define 文は parameter 文よりも変更が少なく、基本的に決めたら滅多に変えないも

のを定義するのに使う。ただ両者の特徴は微妙で、これについては、web のうんちくを参照されたい。

2.5.5 ALU のテストベンチ

下の test.v は ALU をテストするシミュレーション用のテストベンチである。基本的なやり方は、加算器用と同じである。

```
/* test bench */
‘timescale 1ns/1ps
‘define DATA_W 16  // bit width
‘define SEL_W 3 //control width
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010
‘define ALU_ADD ‘SEL_W’b110

module test;
parameter STEP = 10;
   reg [‘DATA_W-1:0] ina, inb;
   reg [‘SEL_W-1:0] sel;
   wire [‘DATA_W-1:0] outs;

   alu alu_1(.a(ina), .b(inb), .s(sel), .y(outs));
   initial begin
      $dumpfile("alu.vcd");
      $dumpvars(0,alu_1);
      ina <= ‘DATA_W’h1111;
      inb <= ‘DATA_W’h2222;
      sel <= ‘ALU_THA;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_THB;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_AND;
     #STEP
      $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
      sel <= ‘ALU_ADD;
```

```
    #STEP
     $display("a:%h b:%h s:%h y:%h", ina, inb, sel, outs);
    $finish;
  end
endmodule
```

今回は、桁数が 16bit になっており、a,b 入力には１６進数を、com には２進数を使って定義をしている点に注意されたい。16 進表示は C 言語の 0x ではなく h を使う。

```
    a <= 16'h1111;
    b <= 16'h2222;
    com <= 3'b000
```

同様に、display 文も 16 進表示は%x でなくて、%h である。では、加算器と同様にシミュレーションをしてみよう。

```
iverilog test.v alu.v
```

で、シミュレーションの実行形が生成される。シミュレーション自体は、

```
vvp a.out
```

で実行される。ここでは最初は a 入力、次は b 入力、最後に加算結果が出力される。

```
a:1111 b:2222 com:0 y:1111
a:1111 b:2222 com:1 y:2222
a:1111 b:2222 com:6 y:3333
```

演習

AND,OR, 減算のシミュレーションを行い、結果を確認せよ。

2.5.6　データパスの記述

図 2.4 のデータパスを記述してみよう。このためには、レジスタを記述する必要がある。レジスタはクロックに同期してデータを格納する点でやや特殊な素子であり、ALU での論理演算とは雰囲気が違っている。

Verilog でレジスタを宣言する際はそのものずばり

```
reg [15:0] acum;
```

とすれば良い。ここでのアキュムレータは 16 ビットのデータを記憶するので、入出力同様バスの形にして宣言する。図 2.4 を見ると、信号線を区別するため alu_y などの名前が付いている。Verilog では、wire 文で信号の名前を宣言する。

```
wire [15:0] alu_y;
```

レジスタと違って、alu_y は ALU の出力データそのものを示していて、データを記憶することはできない。ALU 同様、直接数字を書かないことにして、ここでは、

```
reg [‘DATA_W-1:0] accum;
wire [‘DATA_W-1:0] alu_y;
```

と記述する。ここで ALU の時と違って、この define 文を 1ヶ所にまとめて def.h としておく。そして

```
‘include ‘‘def.h’’
```

とすれば、様々なファイルで同じ定義を共有することができる。この辺の発想は、C 言語と同じである。ただし、include 文の先頭にはシングルバックコーテーション (‘) が付く点に注意されたい。def.h の中身は以下の通りである。

```
‘define DATA_W 16
‘define SEL_W 3
‘define ADDR_W 8
‘define DEPTH 256
‘define ALU_THA ‘SEL_W’b000
‘define ALU_THB ‘SEL_W’b001
‘define ALU_AND ‘SEL_W’b010
‘define ALU_ADD ‘SEL_W’b110
‘define ALU_SUB ‘SEL_W’b111
‘define ENABLE 1’b1
‘define DISABLE 1’b0
‘define ENABLE_N 1’b0
‘define DISABLE_N 1’b1
```

では、全体の記述を見てみよう。

```
‘include "def.h"
module datapath(
input clk, input rst_n,
input [‘DATA_W-1:0] datain,
input [‘SEL_W-1:0] com,
```

```
output [‘DATA_W-1:0] accout);

reg [‘DATA_W-1:0] accum;
wire [‘DATA_W-1:0] alu_y;

assign accout = accum;
alu alu_1(.a(accum), .b(datain), .s(com), .y(alu_y));

always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) accum <= ‘DATA_W’b0;
    else accum <= alu_y;
end
endmodule
```

このデータパスは、アキュムレータの中身をメモリに書き込むために外部に出さなければならない。このような場合、assign 文を用いる

```
assign accout = accum;
```

この文は、accum のデータを dataout に出力すると考えてもよいし、accum を出力 dataout に接続すると考えても良い。さて、ALU は別モジュールとして宣言されているので、これを部品として使う。これが以下の記述である。

```
alu alu_1(.a(accum), .b(datain), .com(com), .y(alu_y));
```

alu がモジュール名であり、alu_1 が実体 (インスタンス) 名である。同じモジュールから実体を多数作ることも良く行われる。先に述べたように、Verilog では、モジュール内の入出力名を.(ピリオド) で表し、それに接続する外部信号名を括弧の中に書いて対応を付ける。つまり、ALU の a 入力には accum を、b 入力には datain を、com 入力には com を接続し、y に alu_y を接続して出力を取り出す。ここで、com は内部も外部も同じ名前を使っているが、module alu と module datapath の両方で宣言されていれば問題ない。

```
always @(posedge clk)
    accum <= alu_y;
```

最後の部分が、アキュムレータ accum に ALU 出力 alu_y を覚えさせる記述である。この書き方がとっつきにくいのが、Verilog が入門者に嫌われる理由の一つになっているが、決った使われ方しかしないので、「おまじない」と思えば良い。

コンピュータでは、レジスタやメモリにデータを覚えさせるのは、一定の信号の変化に合わせて行なう。この信号をシステムクロックあるいは単にクロックと呼ぶ。3GHz で動作する CPU とか呼ぶが、これはクロックの周波数を示している。これが高い程、CPU は高速で動くので宣伝文句として使われる。[2]

さて、

```
always @(posedge clk)
```

は、always=いつも、@=at つまり.. の時、(posedge clk)=クロックが L レベルから H レベルに変化する (立ち上がり：positive edge) という意味である。つまり、「clk の立ち上がりの時にはいつも」ということを示す。ちなみに立ち下がりで動作する際は、negedge：negative edge を使う。

```
    accum <= alu_y;
```

<= は、ノンブロッキング代入文と呼び、レジスタにデータを覚えさせる時に使う。先のテストベンチでも使っていた。ここには同じクロックの立ち上がりで動作する複数の文を begin end で使って書くことができる。これは後に紹介する。

2.5.7 データパスのシミュレーション

ここまでのデータパスにはメモリが含まれていない。後に解説するがメモリは普通は論理合成の対象としないため、テストベンチの中に入れて記述しよう。メモリはレジスタ同様 reg 文で記述するが、名前の後に

最初の番地 : 最後の番地

の形で、記憶できる範囲を示す。

```
reg [15:0] dmem [0:15];
```

この場合、dmem は 0 番地から 15 番地まで記憶できるアドレスを 16 持つ。ということはアドレスは 4 ビットで表すことができる。

```
reg [15:0] dmem [0:1023];
```

と書けば、1024=1K の範囲で、アドレスは 10 本になる。原則として範囲は 2 のべき乗で示す。メモリからの読み出しは、4 ビットの daddr を宣言しておき、それを直接括弧の中い入れれば良い。

```
dmem [daddr];
```

[2] しかしクロック周波数は性能と直接結びつかないので注意。これは後に取り上げる

書き込みはレジスタ同様 always 文を用いるが、メモリは、データを書き込む場合と、何も書き込まず単に読み出しだけを行う場合があるので、we(write enable) 端子でこれを指示する。we=1 とした時のクロックの立ち上がり時にデータが書き込まれる。そこで、記述は下のようになる。

```
always @(posedge clk)
   if(we) dmem[daddr] <= ddataout;
```

if 文は C 言語など多くのプログラミング言語と同様、括弧内の条件が真になった時に次に書く動作が実行される。最初は奇妙に思われるかもしれないが Verilog は always 文など限られた構造の中でのみ、if 文の利用が許される。

さて、このメモリの記述を含むデータパスのテストベンチは以下のようになる。

```
/* test bench */
‘timescale 1ns/1ps
‘include "def.h"
module test;
parameter STEP = 10;
   reg clk, rst_n;
   reg [‘DATA_W-1:0] datain;
   reg [‘SEL_W-1:0] com;
   reg [‘ADDR_W-1:0] addr;
   reg we;
   wire [‘DATA_W-1:0] accout;
   wire [‘DATA_W-1:0] dmem2;
   reg [‘DATA_W-1:0] dmem [‘DEPTH-1:0];
   always @(posedge clk)
   begin
      if(we) dmem[addr] <= accout;
   end

   always #(STEP/2) begin
            clk <= ~clk;
   end
   datapath datapath_1(.clk(clk), .rst_n(rst_n), .com(com),
               .datain(dmem[addr]), .accout(accout));
   initial begin
      $dumpfile("datapath.vcd");
      $dumpvars(0,test);
      $readmemh("dmem.dat", dmem);
```

```
      clk <= `DISABLE;
      rst_n <= `ENABLE_N;
      {we,com,addr} <= {`DISABLE,`ALU_THB,`ADDR_W'h00}; // LD 0
   #(STEP*1/4)
   #STEP
      rst_n <= `DISABLE_N;
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
     $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
   #(STEP*1/2)
      {we,com,addr} <= {`DISABLE,`ALU_ADD,`ADDR_W'h01};  // ADD 1
   #(STEP*1/2)
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
     $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
   #(STEP*1/2)
      {we,com,addr} <= {`DISABLE,`ALU_ADD,`ADDR_W'h02};   // ADD 2
   #(STEP*1/2)
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
     $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
   #(STEP*1/2)
      {we,com,addr} <= {`ENABLE,`ALU_THA,`ADDR_W'h02};   // ST 2
   #(STEP*1/2)
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
     $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
   #(STEP*1/2)
      {we,com,addr} <= {`DISABLE,`ALU_THA,`ADDR_W'h02};  // NOP
   #(STEP*1/2)
      $display("we:%b com:%h addr:%h accout:%h", we, com, addr, accout);
     $display("dmem[0]:%h dmem[1]:%h dmem[2]:%h", dmem[0], dmem[1], dmem[2]);
     $finish;
   end
endmodule
```

まず、クロック信号 clk が一定の間隔で L,H,L,H を繰り返すようにしている。これが下記の文である。

```
   always #(STEP/2) begin
            clk <= ~clk;
   end
```

この場合、STEP の半分で反転するということは、2 回反転すればもとのレベルに戻ることから、クロック周期は STEP となることがわかる。この記述でクロックを発振させるには、clk を reg で定義して、初期化する必要がある。

initial 文以降は以前の記述と似ているが、データメモリに初期値を設定するための文が必要である。

```
$readmemh("dmem.dat", dmem);
```

この場合、データメモリの実体 dmem に対して dmem.dat という名前のファイル中の 16 進データ (readmemh だから) が初期設定される。あらかじめ dmem.dat に以下のフォーマットでデータを入れておく。

```
0002
0003
0004
0001
```

ちなみに、2 進数のデータを読み込むためには、readmemb を用いる。dmem.dat はシミュレーションを起動するディレクトリに置いておく必要がある。次に、以下の記述に注目されたい。

```
{we,com,addr} <= {‘DISABLE,‘ALU_THB,‘ADDR_W’h00};
```

Verilog では中括弧は、信号をくっつけて一つにして扱うことを示す。すなわち、we,com.addr はこの順番にくっついて一連の信号線として扱われる。ここでは、we=0, com=001, addr=0000 が設定される。この場合、データパスに対する命令を操作の部分とアドレスの部分に分離している。ここでは、一定の時間間隔で、LD 0, ADD 1, ADD 2, ST 2 の命令が順に与えられる。このため、2+3+4=9 が dmem[2] に入る。

それぞれの display 文で、クロックが立ち上がる度に、アキュムレータの値とメモリの 0-2 番地を表示する。最初に STEP*1/4 としているのは、値を設定する時刻と表示する時刻をクロックの立ち上がりとずらしてやるためである。シミュレーション上完全に同じ時刻に行ったことは、どのような順番でシミュレーションが実行されるかわからなくなるため、困ったことになる。これを防ぐためである。

先ほどと同じように、

```
iverilog test.v datapath.v alu.v
vvp a.out
```

と打ち込みシミュレーションを行い結果が正しいことを確認せよ。

演習

A を 0 番地、B を 1 番地のデータとして、$A << 1$ OR $B << 1$ のデータを 2 番地にしまう命令の実行をテストベンチを改造してシミュレーションせよ。

演習

A を 0 番地、B を 1 番地、C を 2 番地のデータとして、(A+B) OR (A-B) の結果を 3 番地にしまう命令の実行をテストベンチを改造してシミュレーションせよ。

演習

このデータパスは、ST 命令の実行時にも accum 中のデータが壊されずに動作する。これはなぜだろう？