第12回　より美しく

Verilog記述の改善

今回のVerilog記述は、まず図と文章で、CPUのデータパスのイメージを説明し、ある程度理解してもらった上で、このイメージをなるべく直接Verilogコードに落とすことにした。このため、それぞれのマルチプレクサには、対応する?：を用い、図中の信号線名がそのままコード中に表れるようにした。この書き方は、Verilog本来の基本的書き方であると言えるが、このためにそれぞれの信号毎に条件を決めて、それに対する出力を定義する必要がある。すなわち、この書き方は、図1(a)に示すように、それぞれの出力に対して設計を行う方法であり、結果として、記述がバラバラになって見にくくなる。例として1サイクル動作を行うPOCOのALU周辺は以下のようになる。

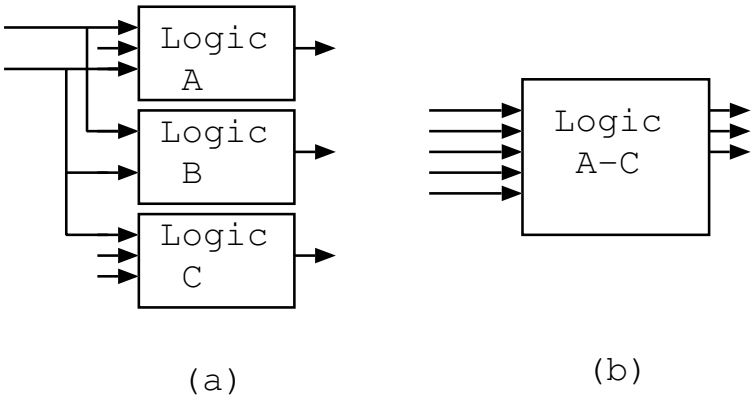

図 1: Verilogモジュールの記述

```
assign st_op = (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = (opcode == ‘OP_REG) & (func == ‘F_LD);
assign jr_op = (opcode == ‘OP_REG) & (func == ‘F_JR);
assign alu_op = (opcode == ‘OP_REG) & (func[4:3] == 2’b00);
assign ldi_op = (opcode == ‘OP_LDI);
assign ldiu_op = (opcode == ‘OP_LDIU);
assign addi_op = (opcode == ‘OP_ADDI);
assign addiu_op = (opcode == ‘OP_ADDIU);
assign ldhi_op = (opcode == ‘OP_LDHI);
assign bez_op = (opcode == ‘OP_BEZ);
assign bnz_op = (opcode == ‘OP_BNZ);
assign bpl_op = (opcode == ‘OP_BPL);
assign bmi_op = (opcode == ‘OP_BMI);
assign jmp_op = (opcode == ‘OP_JMP);
assign jal_op = (opcode == ‘OP_JAL);

assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
               (addiu_op | ldiu_op) ? {8’b0,imm} :
               (ldhi_op) ? {imm, 8’b0} : rf_b;

assign com = (addi_op | addiu_op ) ? ‘ALU_ADD:
               (ldi_op | ldiu_op | ldhi_op) ? ‘ALU_THB: func[‘SEL_W-1:0];
```

```
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op
                | jal_op ;

alu alu_1(.a(rf_a), .b(alu_b), .s(com), .y(alu_y));
```

多くのハードウェアでは、図 1(b) で示すように関連した出力が複数存在するため、入力のグループに対して出力のグループを考えた方が分かりやすい書き方になる。

ところが、Verilog では、図 1(b) の組み合わせ回路を書く標準的な方法が用意されていない。そこで、多くの場合、以下の二つの方法のどちらかを使うことになる。両方共、それぞれ本来想定された使い方とやや異なる方法を流用するため、利点と欠点がある。これを良く認識した上、自分の書き方のスタイルを確立することをお勧めする。

function 文を用いる方法

上記の ALU 周辺の信号線、alu_b, com, rf_c, rwe は、ir 中のフェッチした命令、状態 stat、pc によって関連して変化する。これらを function 文を使えば、case 文でまとめてスマートに記述することができる。

```
function [‘FUNC_ALU-1:0] aludec; // rwe, rf_c, com, alu_b, cadr
   input [‘OPCODE_W-1:0] opcode, func;
   input [‘DATA_W-1:0] alu_y, ddatain, rf_b, pc;
   input [‘REG_W-1:0] rd;
   input [‘IMM_W-1:0] imm;
   reg [‘DATA_W-1:0] pc1;
   begin
   pc1 = pc + 1;
    case(opcode)
     ‘OP_REG:
          if(func == ‘F_LD)
                aludec = {‘ENABLE, ddatain, func[‘SEL_W-1:0], rf_b, rd};
          else if(func[4:3] ==2’b00)
                aludec = {‘ENABLE, alu_y, func[‘SEL_W-1:0], rf_b, rd};
          else aludec = {‘DISABLE, alu_y, func[‘SEL_W-1:0], rf_b, rd};
     ‘OP_LDI: aludec = {‘ENABLE, alu_y, ‘ALU_THB, {{8{imm[7]}},imm},rd} ;
     ‘OP_LDIU: aludec = {‘ENABLE, alu_y, ‘ALU_THB, {8’b0,imm}, rd} ;
     ‘OP_LDHI: aludec = {‘ENABLE, alu_y, ‘ALU_THB, {imm,8’b0},rd} ;
     ‘OP_ADDI: aludec = {‘ENABLE, alu_y, ‘ALU_ADD, {{8{imm[7]}},imm},rd} ;
     ‘OP_ADDIU: aludec = {‘ENABLE, alu_y, ‘ALU_ADD, {8’b0,imm},rd} ;
     ‘OP_JAL: aludec = {‘ENABLE, pc1, func[‘SEL_W-1:0], rf_b, 3’b111} ;
     default: aludec = {‘DISABLE, alu_y, func[‘SEL_W-1:0], rf_b, rd} ;
    endcase
    end
endfunction

assign {rwe, rf_c, com, alu_b, cadr} =
                aludec(opcode, func, alu_y, ddatain,
                                        rf_b, pc, rd, imm);
```

function 文では、入力に宣言した順番に引数を指定する。出力は一つしか取れないので、いくつかの出力を連接して取り出す。

ここで pc1 は、function 文中で仮に信号名を付ける目的で利用される。function 文中では wire でなく reg 宣言を行うが、これは後で示す always 文で用いる reg 宣言同様で、function 文中のいかなる条件でも値を保持しておくことを意味しており、本当にレジスタが発生するわけではない。

同様に pc についても以下のように書くことができる。

```
// PC nexT
function [‘DATA_W-1:0] pcnext; // rwe, rf_c, com, alu_b, cadr
   input [‘OPCODE_W-1:0] op;   // reg for function
   input [‘BODY_W-1:0] body;
   input [‘DATA_W-1:0] pc;
   input [‘DATA_W-1:0] rf_a;
   if(op==‘OP_JAL| op == ‘OP_JMP)
     pcnext = pc +{{5{body[10]}},body}+1 ;
   else if ((op==‘OP_BEZ & rf_a == 16’b0 ) | (op==‘OP_BNZ & rf_a != 16’b0) |
              (op==‘OP_BMI & rf_a[15] == 1 ) | (op==‘OP_BPL & rf_a[15] == 0))
     pcnext = pc +{{8{body[7]}},body[7:0]}+1 ;
   else if((op==‘OP_REG)&(body[4:0]==‘F_JR))
     pcnext = rf_a;
   else
     pcnext = pc+1;
endfunction

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else
     pc <= pcnext(opcode,idatain[‘BODY_W-1:0], pc, rf_a);
end
```

このように、レジスタの値を決める場合、リセット以外の条件を独立して function 中に書くことができる。このように、function 文は複雑な条件をスマートに記述することができない。

function 文は出力は一つしか取ることができない。このため、複数の出力を扱う場合、これらを一度に連接して出力する。この方法は、行数が少なくて済む上、出力の定義し忘れがないという点で優れている。この方法の問題点は以下の二つである。

- 連接した出力のビット数は、全ての出力の和とならなければならない。ここでは、FUNC_ALU を 1+2*DATA_W+SEL_W+REG_W として定義している。これが狂うと出力がずれて異なった所に接続されてしまう。このミスにより発見しにくいバグが発生する。
- ここでは全ての入力を引数として与えているが、これを行わないと、入力がグローバル信号として与えられる場合がある。これも発見しにくいバグとなる。

両方共、合成時にワーニングが生じるため、注意深く合成ログをチェックすることで発見可能である。

また、この記述方法は、今まで示した基本的な方法に比べて、合成結果が改善されるわけではない。web 上の記述を合成して結果を確認しよう。

always 文を用いる方法

always 文は、今までは括弧内、すなわち waiting list には posedge clk と記述して、クロックの立ち上がりに同期してレジスタに値を格納する場合にのみ用いてきた。しかし、この方法を使って組み合わせ回路を記述することもできる。この場合、まず、本来組み合わせ回路の出力で wire で宣言すべき信号を reg で宣言する。次に always の括弧内を*すなわち全ての信号が変化した際に内部が反応するようにする。以下が、先の記述に対応する always 文の記述である。

```
reg [‘DATA_W-1:0] rf_c;          //wire for always
reg [‘DATA_W-1:0]  alu_b;        //wire for always
reg [‘SEL_W-1:0] com;            //wire for always
reg [‘REG_W-1:0] cadr;            //wire for always
reg rwe;                         // wire for always
always @( * ) begin
    case(opcode)
     ‘OP_REG:
          if(func == ‘F_LD) begin
                rwe <= ‘ENABLE; rf_c <= ddatain;  cadr <= rd;
                com <= func[‘SEL_W-1:0]; alu_b <= rf_b; end
          else if(func[4:3] == 2’b00)  begin
                rwe <= ‘ENABLE; rf_c <= alu_y; cadr <= rd;
                com <= func[‘SEL_W-1:0]; alu_b <= rf_b; end
          else begin
                rwe <= ‘DISABLE; rf_c <= alu_y; cadr <= rd;
                com <= func[‘SEL_W-1:0]; alu_b <= rf_b; end
     ‘OP_LDI: begin
                rwe <= ‘ENABLE; rf_c <= alu_y;  cadr <= rd;
                com <= ‘ALU_THB; alu_b <= {{8{imm[7]}},imm} ; end
     ‘OP_LDIU: begin
                rwe <= ‘ENABLE; rf_c <= alu_y;  cadr <= rd;
                com <= ‘ALU_THB; alu_b <= {8’b0,imm} ; end
     ‘OP_LDHI: begin
                rwe <= ‘ENABLE; rf_c <= alu_y;  cadr <= rd;
                com <= ‘ALU_THB; alu_b <= {imm,8’b0} ; end
     ‘OP_ADDI: begin
                rwe <= ‘ENABLE; rf_c <= alu_y;  cadr <= rd;
                com <= ‘ALU_ADD; alu_b <= {{8{imm[7]}},imm} ; end
     ‘OP_ADDIU: begin
                rwe <= ‘ENABLE; rf_c <= alu_y;  cadr <= rd;
                com <= ‘ALU_ADD; alu_b <= {8’b0,imm} ; end
     ‘OP_JAL: begin
                rwe <= ‘ENABLE; rf_c <= pc+1;  cadr <= 3’b111;
                com <= func[‘SEL_W-1:0]; alu_b <= rf_b ; end
     default: begin
                rwe <= ‘DISABLE; rf_c <= alu_y;  cadr <= rd;
                com <= func[‘SEL_W-1:0]; alu_b <=  rf_b ; end
    endcase
```

```
end
```

この場合、always 文は全ての入力変化に対して反応する。このため、全てのケースにおいて出力が決定すれば、それはレジスタではなく組み合わせ回路の出力と同じである。合成系もそのように判断して組み合わせ回路を生成してくれる。function 文と同じようにこの方法も、case 文でスマートで記述することが可能で、一つの出力に連接する必要はないことから記述はより自然となる。しかし、やはり以下の問題点がある。

- waiting list 内に入力を書き込む場合、入力を書き忘れると発見が困難なバグを生成してしまう。このためここには*を使うことをお勧めする。
- 入力の全ての組み合わせで出力を定義しないと、定義されていない入力に対しては出力を保持する必要があると判断されてしまう。このため、合成時に一種の記憶素子であるトランスペアレントラッチが生成されてしまう。これは多くの場合、トラブルの元となる。

always 文を用いる方法は、以前は合成系がうまく働き、優れた合成結果を得ることができることから特に FPGA ベンダによって推奨された。これは、複数の入力と出力を同じ条件で記述することから、合成と圧縮が楽であったことに起因する。しかし、現在の合成系は、マルチプレクサ (? :) を用いてばらばらに条件を分けて書いても always 文を使うのと同様に合成してくれる。web 上の記述を合成し、基本的な記述の合成 (8 回) と比較してみよう。

マルチプレクサを用いる方法の問題点

今まで、function 文を用いる方法の問題点、always 文を用いる方法の問題点を述べてきた。公平を期すため、マルチプレクサを用いる基本的な方法の問題点を述べておこう。

マルチプレクサは、条件 1？式 1：条件 2？式 2... 条件 n?式 n:式 n+1 の形を守れば、case 文とほとんど同じである。ここで、条件 1 の方が条件 2 よりも優先順位が高いが、このことを利用する記述は止めて、case 文同様、すべての条件は排他的に書くことが望ましい。

マルチプレクサ記述で問題となるのは、入れ子を使う場合である。すなわち、条件 1？(条件 2？式 2：式 3)：式 4 の形で書くことができるのだが、この入れ子を何重にも用いると、非常に混乱した記述になる。このため、入れ子はたとえ二重であっても絶対に使ってはならない。入れ子にする必要がある際は、信号名をもう一つ設け、マルチプレクサを分離して書くこと。この場合、複数の縦列接続するマルチプレクサ間で、条件が排他的になるようにすること、式の入力等が混乱しないようにすることに注意する必要がある。マルチプレクサ記述が敬遠されるのは、入れ子を使った記述が横行しているためであり、上記を気をつければこのテキストで学んだ方法は決して判り難い書き方ではない。しかし、マルチプレクサ構文は、基本的に一つの出力のためのものであり、複数入力、複数出力をスマートに書けないことから、複雑な回路を設計する場合、always 文を使う方法、funciton 文を使う方法のどちらかを使う場合が多い。

おわりに：データパスと制御系

always 文と function 文を利用する場合、制御系の信号 (rwe) と、データパスの切り替え (rf_c など) を同じ枠組みで書くことができる。この考え方を推し進めて行くと、データパスと制御系を分離して考える必要はないのではないか、と思えてくる。つまり、（ある命令が来たら）（ある操作をする）という方法で書くわけだ。Verilog は、レジスタの書き方と組み合わせ回路の書き方が違うので、この書き方で全体を押し通すのは難しい。しかし、うまく書くとこれに近いことはできる。今回、授業ではこのような記述方式は取らなかった。これは、レジスタファイル、ALU など CPU を形成するブロックとデータパスの構造を意識しながら設計した方が、アーキテクチャの授業としては理解が深まり、分かりやすいと考えたためである。しかし、今回の授業のようにかなり構造を意識して書いても、図で示した通りのハードウェアが合成されるわけではなく、実は合成系と指定の仕方によって様々なハードウェアが生成される。この辺、図で示した CPU もある意味で一つのモデルである点注意されたい。

さて、Verilog HDL は世界中で利用されており、これを用いて膨大な量のハードウェアが設計されているにも関わらず、標準的な記述法がないというのは驚くべきことである。これは、ハードウェア設計者の多くが、問題は設計すべき対象であって言語や記法ではないと思っており、自分のスタイルを他人に押しつけようとしないためだろう。実際、どの方法で記述しても対象のシステムがちゃんとしており、合成がきちんとできれば問題はない。逆に言うと、ハードウェア設計の世界は、良く言えばおおらか、悪く言えばいいかげんな所がある。皆さんも、あるレベルまで来れば、記法にはさほどこだわらず、設計対象に対して思う存分独創性を揮っていただければ幸いである。

試験対策問題

1. 16bit RISC POCO で下の命令を順に実行した。

```
LDI  r1, #0x80
ADDI r1, #05
LDIU r0, #0
ST r1,(r0)
```

a) 上記の機械語命令を示せ。またそれぞれの opcode フィールド、function(func) フィールド (あれば) はどうなるかを示せ
b) r0、r1 の値はどのように変化するかを示せ。また、最終的に、どの番地にどのような値が書き込まれるかを示せ。

2. 16bit RISC POCO で　a)ADDI 命令、b)BEZ 命令を実行する際に、各制御信号線をどのように設定すれば良いか？表に付け加えよ。

表 1: 各命令の制御信号

	pcsel	comsel	alu_bsel	rf_csel	rwe	we
ADDI						
BEZ						

3. 16bit RISC POCO について以下のプログラムをアセンブリ言語で記述せよ。機械語に変換する必要はない。また飛び先にはラベルを利用せよ。
(a) データメモリの 0 番地から 7 番地までに格納されている 8 個のデータを 8 番地から 16 番地までに移動するプログラムを記述せよ。
(b)r0 に与えられた番地から格納されている 8 個数分のデータを r1 から与えられた番地へ移動するサブルーティンを記述せよ。
(c) スタックポインタが r6 であり既に値が設定されているとして、サブルーティン内で破壊するレジスタを退避して復帰するコードを (b) に付け加えよ。
4. POCO の Verilog コードを改造し、STINC rd,(rs) 命令を取り付けよ。この命令は ST を実行した後に rs に 1 を加える命令である。