第2回　Verilogによる ALU の記述

基本的な演算子

前回演習した通り、Verilog は表の演算子を用いて、組み合わせ回路の記述をすることができる。

表 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。以下の優先順位があるが

演算子には優先順位があるので注意されたい。下の方で上位にあるもの程強い。

例 2.1：多数決回路　1 の数が多ければ 1、0 の数が多ければ 0 を出力するのが多数決回路である。3 入力多数決回路は以下のようになる。

```
module major ( input a, b, c, output y);
  assign y = a&b | b&c | c&a ;
endmodule
```

変数：　Verilog では二種類の変数を用いる。

- wire: 信号線に名前を付けたもの。記憶能力を持たない。

表 2: 基本的演算子の優先順位

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

- reg: レジスタ。基本的にはデータを記憶する能力を持つ。次回詳細に説明する。使い方によっては wire と同様、記憶をしない場合もあるが、これも後に説明する。

例えば、wire x; により定義した信号線名 x に対してデータを出力する場合は、出力に対しての場合と同様に assign 文を利用する。

```
assign x = a&b | b&c | c&a;
```

wire、reg 共に、データ幅を定義してまとめて扱うことができる。

```
wire [MSB(Most Significant Bit):LSB(Least Significant Bit)] 名前
```

の形で宣言する。信号線の束をバスと呼ぶが、この方法でバスを定義して名前を付けることができる。なお MSB は一番上の桁、LSB は一番下の桁だが、LSB は通常 0 とする。

```
wire [7:0] a;
```

は、8 ビットのバス a を宣言する。

桁方向（リダクション）演算　バスの桁方向にデータを計算することをリダクション演算と呼ぶ。Verilog では演算子の前に/を付けることでこれを表現する。例えば

```
wire [3:0] x;
wire y;
y = &x;
```

と書いた場合、

```
y = x[3]&x[2]&x[1]&x[0];
```

と同じである。これは 4 ビットなので大した効果が感じられないが、ビット数が長いとこの演算の威力は大きい。

定数：　定数は以下の形で使う。

<ビット幅>'<基数><数値>

基数は

- b：2 進数
- h：16 進数
- o：8 進数

で、指定がなければ 32 ビットの 10 進数となる。

```
8'b11001010
16'hab3d
64'h1111_aaaa_2222_bbbb
```

などのように表す。ちなみに桁が長くなると間違い易いので、_で区切って見やすくする。_は数字の中に出てきても無視される。Verilog はビット幅が違ったもの同士の演算や割り付けが可能なので、ビット幅を省略しても問題なく動作する場合もあるが、なるべくきちんと記述することをお勧めする。

ビット切り出しと連結: ハードウェア設計上、バスで宣言された信号線の一部を切り出したり、連結して新たな信号として扱うことが頻繁に必要になる。Verilog では、切り出しには、宣言と同様に [:] を利用し自然に行う。

```
wire [15:0] y;
wire [7:0] x;
assign x = y[15:8];
```

この記述では y の上位の 8 ビットが x に割り当てられる。
連結には　を利用し、これも容易に記述することができる。

```
wire [15:0] y;
wire [7:0] a,b;
assign y = {a,b};
```

この例では a を上位 8 ビット b を下位 8 ビットを連結して 16 ビットのバス y として扱う。
は二重括弧の中に数字を付けることにより、その数字分の複製を示すこともできる。例えば、

```
{8{2'b10}}
```

は、16'b1010101010101010 と同じである。また、以下の記述を使えば、符号拡張 (sign extension) を示すことができる。符号拡張というのは、数の正負を示す符号を保持しながら、桁数を長くすることで、下の例では 8 ビットの数を 16 ビットに拡張している。

```
wire [15:0] a;
wire [7:0] x;
assign a = {{8{x[7]}},x}
```

Verilog においては、出力に用いることもできる。下は、4 ビットの加算回路の例で、桁上げ出力 c ををを用いて引っ張り出している。

```
/* 4 bit adder */

 module add4 (
  input [3:0] a, b,
  output [3:0] s,
  output c );
```

```
  assign {c,s} = a + b ;  // add a b

endmodule
```

条件演算子による ALU の記述

ハードウェアの設計では、特定の条件が成り立った時に特定の処理を行う場合が多い。例えば、CPU の演算を行う中心部である ALU(Arithmetic Logic Unit) は、制御入力 s のデータによって、行う演算の種類を決める。例えば以下の 16bit ALU を考える。

- S:000 a 入力がそのまま y 出力へ (THA)
- S:001 b 入力がそのまま y 出力へ (THB)
- S:010 a & b 入力が y 出力へ (AND)
- S:110 a + b 入力が y 出力へ (ADD)

この ALU に相当する Verilog 記述が以下の通りである。

```
module alu (
  input [15:0] a, b,
  input [2:0] s,
  output [15:0] y );
  assign y = s==3'b000 ? a:
             s==3'b001 ? b:
             s==3'b010 ? a & b: a + b ;
endmodule
```

ここで用いられるのが、条件演算子

条件`?` 入力`1:` 入力`2;`

である。これは条件が成立すれば入力 1 が、そうでなければ入力 2 が出力される。入力 2 の部分に次々と条件を書いていくと、上記のように複数の条件を選択することができる。条件は上から順にチェックされていく。上記の記述では a+b はそれまでに示された三条件が満足されていない場合は全て出力される点に注意されたい。この構文を用いる場合は、以下のことをお勧めする。

- 条件をなるべく整理して、排他的（ある条件が成り立つ時はそれ以外の条件が成り立たない）に書くことが望ましい。
- 入力 1 の所に、条件演算子を書いて入れ子にすると、非常に読みにくいので避ける方が良い。常に、条件が成り立たない時に出力される入力 2 に対してさらに条件を加えて行く

上記は、case 文などの条件を書く場合の一般的な注意だが、？ :は、これを守らないと特に読み難くなるので、一層の注意が必要である。

define 文と parameter 文　上に示した alu の記述は様々な所に数字が直接出てきてこの点では良くない記述である。これを置き換えるには二つ方法がある。一つは define 文でモジュールの外にセミコロン無しに記述する。C 言語の define 文に似ているが井桁ではなくて、シングルバックコーテーションをいう普段あまり使わない記号を使うので注意。これを文中で利用する時も同じ記号が必要となる。

```
‘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
```

もう一つの方法は parameter 文を用いることである。module 宣言の最初の部分に井桁の後のパラメータリストの部分に宣言して使うことができる。

```
module alu #(parameter DATA_W=16,
        SEL_W=3,
        ALU_THA=3’b000,
        ALU_THB=3’b001,
        ALU_AND=3’b010 )
 (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
```

define 文は、プリプロセッサによって文字列が置き換えられるため、コード中の全ての数字を入れ替えることができ、一つのファイルにまとめておけるメリットがある。一方、parameter 文は宣言時が文法チェックの対象となること、外部から値を与えられることなどのメリットがある。STARC 指定の設計スタイルガイドでは parameter 文を推奨しているが、天野個人および天野研は define 文にどっぷり浸かっている。

function 文による ALU の記述

Verilog HDL は、module 文内の通常の記述では、if 文、case 文など条件を指定する文を使うことができない。実際は、条件演算子を使えば大抵の場合なんとかなるのだが、可読性を重視する場合、function 文を使う方法がある。

function 文は、以下の構文を持つ。

```
function [ビット幅指定]  名前;
 input 入力 (複数);
begin

ここで if 文、case 文が使える

end
endfunction
```

先の ALU の記述例を示す。

```
‘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 );

function [‘DATA_W-1:0] alu;
  input [‘DATA_W-1:0] in1, in2;
  input [‘SEL_W-1:0] sel;
  begin
     if(sel==‘ALU_THA) alu= in1;
     else if(sel==‘ALU_THB) alu= in2;
     else if(sel==‘ALU_AND) alu= in1 & in2;
     else alu= in1 + in2 ;
  end
endfunction

assign y = alu(a,b,s);
endmodule
```

if 文と else 文の使い方は、C 言語と同じで、複雑な入れ子を作る場合は，begin end で構造を作って入れ子にする。しかしあまり複雑な構造は、条件演算子と同様、お勧めしない。

function 文を呼び出す場合は、入力は宣言した順に与える。function 文の出力は一つなので、複数の出力を取り出す場合 を用いる。

function 文内では case 文も使うことができる。

```
‘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 );

function [‘DATA_W-1:0] alu;
  input [‘DATA_W-1:0] in1, in2;
  input [‘SEL_W-1:0] sel;
  begin
    case (sel)
     ‘ALU_THA: alu= in1;
     ‘ALU_THB: alu= in2;
     ‘ALU_AND: alu= in1 & in2;
     default: alu= in1 + in2 ;
   endcase
  end
endfunction
assign y = alu(a,b,s);
endmodule
```

function 文は、条件演算子よりも C 言語っぽく書けるので、ファンが多いが、落とし穴もある。まず、入力信号の混乱である。上記の記述は function 文内で使うのは、全て入力として宣言した信号だが、実は、function 文内では、宣言しないでも function 外の入力が使えてしまう。例えば上の例は、

```
module alu (
  input [‘DATA_W-1:0] a, b,
  input [‘SEL_W-1:0] s,
  output [‘DATA_W-1:0] y );

function [‘DATA_W-1:0] alu;
  input [‘SEL_W-1:0] sel;
  begin
    case (sel)
     ‘ALU_THA: alu= a;
     ‘ALU_THB: alu= b;
     ‘ALU_AND: alu= a & b;
     default: alu= a + b;
   endcase
  end
endfunction
assign y = alu(s);
endmodule
```

と書いても、動作する。むしろこの場合は、こっちの方が良いような気もするが、一般的にはこれが理由で混乱することがある。また、複数出力に代入するようになると、ビット幅を間違えて酷い目に合うこともあるのだが、これはまた後に解説する。

本日の演習 (演習 2)　ここに示した ALU のどれか好きなものを土台として、以下の機能を加えた本格的な ALU を作り、テストベンチで動作を確認せよ。

- S:000 a 入力がそのまま y 出力へ (THA)
- S:001 b 入力がそのまま y 出力へ (THB)
- S:010 a & b 入力が y 出力へ (AND)
- S:011 a | b 入力が y 出力へ (OR)
- S:100 a<<1 が y 出力へ (SL)
- S:101 a>>1 が y 出力へ (SR)
- S:110 a+b が y 出力へ (加算)
- S:111 a-b が y 出力へ (減算)

web 上の test.v を使うと iverilog 時に warning が出るが、これは加算をやったため、出力が 1 ビット増えたのを無視しているためである。今回は気にしないでやって下さい。