Course number: CSC.T341

コンピュータ論理設計
Computer Logic Design

2. ハードウェア記述言語:組合せ回路
Hardware Description Language: Combinational Circuit

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
W621 講義室　月 10:45-12:15, 木 9:00-12:15

Inside code001.v

- モジュールの定義はキーワードmoduleからキーワードendmoduleまで.
- moduleの後にモジュール名を書く. この例ではmainがモジュール名.
- モジュール名の後の括弧内に入出力の端子名を列挙する. ここでは端子は何も定義していない.
- セミコロン(;)で, モジュール名と端子の列挙を終える.
- キーワードinitialにより, シミュレーション開始時(時刻0)に一度だけ実行されることを指定する.
- $writeはシステムタスクの1つで, メッセージを出力する. 書式はC言語のprintfと同様.

Verilog HDL code (code001.v)

```
module main ();
  initial $write("hello, world¥n");
endmodule
```

Simulation output

```
hello, world
```

Verilog HDLのコードは青色で, シミュレーションの出力は黄色で示す.

Inside code002.v

- main.vをcode002.vの内容となるように入力して，シミュレーションする．
- 2つのシステムタスク$writeを用いた出力の例．2つのシステムタスクをブロックとしてまとめている．
- ブロックはキーワードbeginで始まり，キーワードendで終わる．C言語の{ }に対応．
- code002_ng1.vは2番目の$writeがinitialブロックに含まれないので文法エラーとなる．

code002.v

```
module main ();
  initial begin
    $write("hello, world¥n");
    $write("in Verilog HDL¥n");
  end
endmodule
```

```
hello, world
in Verilog HDL
```

code002_ng1.v

```
module main ();
  initial $write("hello, world¥n");
  $write("in Verilog HDL¥n");
endmodule
```

Inside code003.v

- main.vをcode003.vの内容となるように入力して，シミュレーションする．
- モジュール内で複数のinitialを用いても良いので，code002.vとcode003.vの出力は同じとなる．

code002.v

```
module main ();
  initial begin
    $write("hello, world¥n");
    $write("in Verilog HDL¥n");
  end
endmodule
```

```
hello, world
in Verilog HDL
```

code003.v

```
module main ();
  initial $write("hello, world¥n");
  initial $write("in Verilog HDL¥n");
endmodule
```

```
hello, world
in Verilog HDL
```

Inside code004.v

- main.vをcode004.vの内容となるように入力して，シミュレーションする．
- 整数integerとforループを用いた温度変換プログラムの例．
- 整数型のfahr, celsiusを定義．
- C言語の様に演算子++は使えない．fahr++ という記述はエラーとなるので注意．

code004.v

```
module main ();
  integer fahr, celsius;
  initial begin
    for (fahr = 0; fahr <= 300; fahr = fahr + 20) begin
      celsius = 5*(fahr-32) / 9;
      $write("%3d %6d¥n", fahr, celsius);
    end
  end
endmodule
```

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
120     48
140     60
160     71
180     82
200     93
220    104
240    115
260    126
280    137
300    148
```

Inside code005.v

- main.vをcode005.vの内容となるように入力して，シミュレーションする．
- 指定した時間が経過するまで待たせる命令# を用いた例．
- #200 により，ここではシミュレーション開始時（時刻0）から200だけ時間が経過した時刻200に hello, world を表示する．
- #100 により，ここではシミュレーション開始時（時刻0）から100だけ時間が経過した時刻100に in Verilog HDL を表示する．
- 1行目はコメント，Verilog HDLのコメントはC, C++と同様．

code005.v

```
/* sample Verilog code */
module main ();
  initial #200 $write("hello, world¥n");
  initial #100 $write("in Verilog HDL¥n");
endmodule
```

```
in Verilog HDL
hello, world
```

Inside code008.v

- main.vをcode008.vの内容となるように入力して，シミュレーションする．
- システムタスク$timeは，64ビットのシミュレーション時刻を返す．
- このコードでは，それぞれの $write が表示する時刻を表示する．
- 複雑な回路のシミュレーションでは，どの出力がどの時刻に出力されたのかわかりにくい場合がある．その場合，この例のように時刻を出力すると良い．

code008.v

```
module main ();
  initial #200 $write("%3d hello, world¥n", $time);
  initial begin
    #100 $write("%3d in Verilog HDL¥n", $time);
    #150 $write("%3d When am I displayed?¥n", $time);
  end
endmodule
```

```
100 in Verilog HDL
200 hello, world
250 When am I displayed?
```

Inside code009.v

- main.vをcode009.vの内容となるように入力して，シミュレーションする．
- システムタスク$finishは，シミュレーションを終了させる．
- このコードでは時刻210でシミュレーションが終了する．
- Vivadoのデフォルトの設定では1000nsシミュレーションするが，それより短い時間のシミュレーションや，ある条件でシミュレーションを終了させたい場合等に用いると良い．

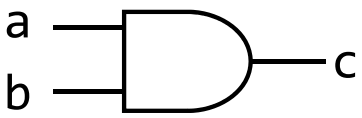
code009.v

```
module main ();
  initial #200 $write("%3d hello, world¥n", $time);
  initial begin
    #100 $write("%3d in Verilog HDL¥n", $time);
    #150 $write("%3d When am I displayed?¥n", $time);
  end
  initial #210 $finish;
endmodule
```

```
100 in Verilog HDL
200 hello, world
```

Inside code011.v

- reg型の信号a, bを宣言する. reg型はC言語の変数に相当する.
- wire型の信号cを宣言する. wire型はハードウェア記述言語に固有のもの.
- 継続的代入assign は, wire型の信号cをa & bに接続する. initialブロックやalways@ブロックの外に記述する.
- & はANDの論理演算子.
- always@ブロックは, @以降に書かれた事象が発生するたびに繰り返し実行される. always@(*) では, 何らかの入力が変化した事象となる.
- initialブロックの中の <= はノンブロッキング代入と呼ばれ, reg型の信号への代入を表す. a <= 0; はreg型の信号aへの値0の代入を表す. wire型にノンブロッキング代入は使えない.



AND gate

code011.v

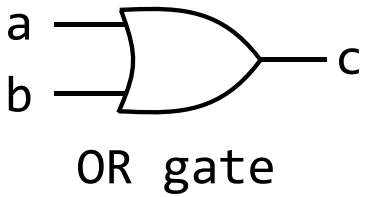
```
module main ();
  reg a, b;
  wire c;
  assign c = a & b;

  initial begin
    #10 a <= 0; b <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d -> %d¥n", $time, a, b, c);
endmodule
```

Simulation output

```
11: 0 0 -> 0
21: 0 1 -> 0
31: 1 0 -> 0
41: 1 1 -> 1
```

Inside code012.v

- | はORの論理演算子.
- 論理演算子には，単項演算子の ~ (NOT), 2項演算子として & (AND), | (OR), ^ (EXOR)がある.
- 算術演算子には，+ (加算), - (減算), * (乗算), / (除算), % (剰余)がある.
- これらの論理演算子，算術演算子はC言語と同じ.



OR gate

code012.v

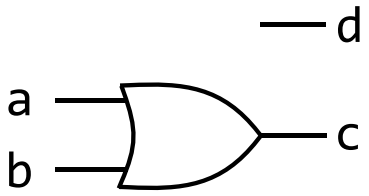
```
module main ();
  reg a, b;
  wire c;
  assign c = a | b;

  initial begin
    #10 a <= 0; b <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d -> %d¥n", $time, a, b, c);
endmodule
```

Simulation output

```
11: 0 0 -> 0
21: 0 1 -> 1
31: 1 0 -> 1
41: 1 1 -> 1
```

Inside code013.v

- wire dはどこにも接続されていない．シミュレーション結果は？
- 信号線の取り得る値には，0, 1, x, z がある．xは不定値を，zはハイインピーダンスを表す．
- どこにも接続されていないwire，あるいは明示的にハイインピーダンスに設定されたwireはzとなる．
- 初期化されていないreg型の信号や，不定値を用いた演算結果などはxとなる．
- 意図的にx, zとしていないのにx, zが出力される場合，コードに記述ミスがあることが多いので，コードの記述を見直すと良い．
- code012.v の | の部分を他の論理演算子や算術演算子に変更してシミュレーションすること．



OR gate

code013.v

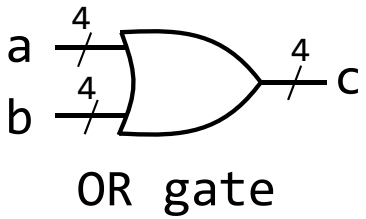
```
module main ();
  reg a, b;
  wire c, d;
  assign c = a | b;

  initial begin
    #10 a <= 0;
    #10 a <= 0; b <= 1;
    #10 a <= 1; b <= 0;
    #10 a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d -> %d %d¥n", $time, a, b, c, d);
endmodule
```

Simulation output

```
11: 0 x -> x z
21: 0 1 -> 1 z
31: 1 0 -> 1 z
41: 1 1 -> 1 z
```

Inside code014.v

- Verilog HDLでは，2本以上の信号線の束をバス(bus)と呼ぶ．
- reg型，wire型の信号線をバスとして宣言するには，reg, wire の後に [3:0] の様に本数を指定する．例えば reg [3:0] a は，a[3], a[2], a[1], a[0]の4本から成るバスを宣言する．
- code014.v では，4ビット幅のバスとしてreg型a, bを，4ビット幅のバスとしてwire型cを宣言する．
- 数値を表現するためには，'より前の数字がビット幅を表し，'の後のbが2進法であることを表す（その他，16進法h, 10進法d, 8進法oがある）．例えば，4'b1010 は2進法で示された4ビットの1010となる．数値の表現では大文字，小文字は区別されない．4'b1010 と 4'B1010 と4'hAは同じ値となる．ビット幅を省略すると32ビットとなる．基数を指定しないと10進法となる．
- システムタスク$writeでは，2進法で表示するための %b が利用できる．



OR gate

code014.v

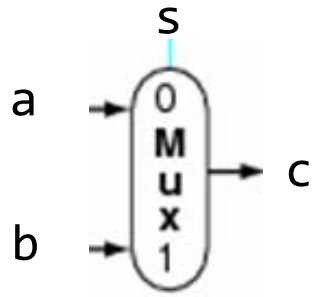
```
module main ();
  reg  [3:0] a, b;
  wire [3:0] c;
  assign c = a | b;

  initial begin
    #10 a <= 4'b1010; b <= 4'b1100;
  end
  always@(*) #1 $write("%2d: %b %b -> %b¥n", $time, a, b, c);
endmodule
```

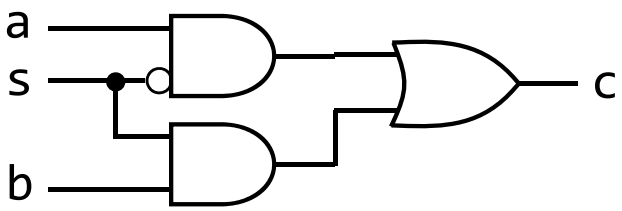
Simulation output

```
11: 1010 1100 -> 1110
```

Inside code015.v

- Multiplexer（マルチプレクサ）のVerilog HDL記述を考える.
  - sが0であればaを出力として，sが1であればbを出力とする回路.





code015.v

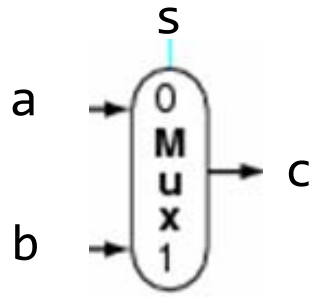
```
module main ();
  reg  a, b, s;
  wire c;
  assign c = (a & ~s) | (s & b);
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d %d -> %b¥n", $time, s, a, b, c);
endmodule
```

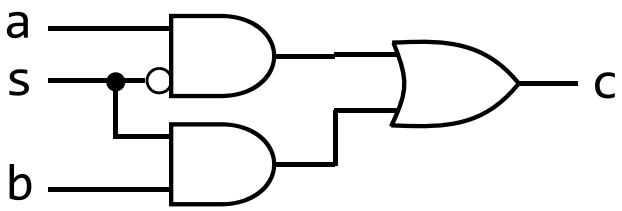
Simulation output

```
11: 0 0 0 -> 0
21: 0 0 1 -> 0
31: 0 1 0 -> 1
41: 0 1 1 -> 1
51: 1 0 0 -> 0
61: 1 0 1 -> 1
71: 1 1 0 -> 0
81: 1 1 1 -> 1
```

Inside code016.v

- マルチプレクサのVerilog HDL記述を考える.
- 3項演算子の条件演算子 (? :) を使っても同じ結果になる. この記述の方が簡潔でわかりやすい.





code016.v

```
module main ();
  reg  a, b, s;
  wire c;
  assign c = s ? b : a;
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d %d -> %b¥n", $time, s, a, b, c);
endmodule
```

Simulation output

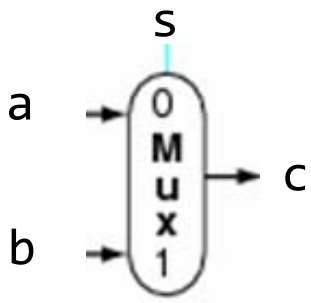
```
11: 0 0 0 -> 0
21: 0 0 1 -> 0
31: 0 1 0 -> 1
41: 0 1 1 -> 1
51: 1 0 0 -> 0
61: 1 0 1 -> 1
71: 1 1 0 -> 0
81: 1 1 1 -> 1
```

Inside code017.v

- モジュールをインスタンス化する例を示す.
- モジュール名とインスタンス名を記述して, その後に入出力端子名を列挙する. C言語の関数呼び出しに似ている. この例では m_mux というモジュール名のインスタンスm_mux0 を生成する.

code017.v

```
module m_top ();
  reg  a, b, s;
  wire c;
  initial begin
    #10 s <= 0; a <= 0; b <= 0;
    #10 s <= 0; a <= 0; b <= 1;
    #10 s <= 0; a <= 1; b <= 0;
    #10 s <= 0; a <= 1; b <= 1;
    #10 s <= 1; a <= 0; b <= 0;
    #10 s <= 1; a <= 0; b <= 1;
    #10 s <= 1; a <= 1; b <= 0;
    #10 s <= 1; a <= 1; b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d %d -> %b¥n", $time, s, a, b, c);
  m_mux m_mux0 (a, b, s, c);
endmodule

module m_mux (a, b, s, c);
  input  wire a, b, s;
  output wire c;
  assign c = s ? b : a;
endmodule
```



Simulation output

```
11: 0 0 0 -> 0
21: 0 0 1 -> 0
31: 0 1 0 -> 1
41: 0 1 1 -> 1
51: 1 0 0 -> 0
61: 1 0 1 -> 1
71: 1 1 0 -> 0
81: 1 1 1 -> 1
```

Some rules for following exercises

- モジュールの名前には m_ から始まる名前を使う. wire型の信号の名前には w_ から始まる名前を使う. reg型の名前には r_ から始まる名前を使う.
- シミュレーションの最上位のモジュール(トップモジュール)には m_top という名前を使う.
  - $display などのシステムタスクは m_top の中でしか用いてはいけない.
- 論理合成のトップモジュールには m_main という名前を使う.
- *Name*という名前のモジュールのインスタンス名には *Name*に数字を付加した名前を使う.

ルールを適用したVerilog HDL記述の例

```
module m_top ();
  reg  r_a, r_b, r_s;
  wire w_c;
  initial begin
    #10 r_s <= 0; r_a <= 0; r_b <= 0;
    #10 r_s <= 0; r_a <= 0; r_b <= 1;
  end
  always@(*) #1 $write("%2d: %d %d %d -> %b¥n", $time, r_s, r_a, r_b, w_c);
  m_mux m_mux0 (r_a, r_b, r_s, w_c);
endmodule

module m_mux (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b, w_s;
  output wire w_c;
  assign w_c = w_s ? w_b : w_a;
endmodule
```

Inside code018.v

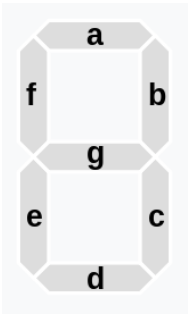
- 0～9を表示する seven-segment LED decoder の例を示す.
- 場合分けの処理を記述するための case文 がある. 記述はC言語と同様.
- モジュールm_7segledでは, 入力の値により, 点灯させるLEDのビットを1とする.
  - r_led の MSBから, LEDのabcdefgのセグメントを割り当てる.

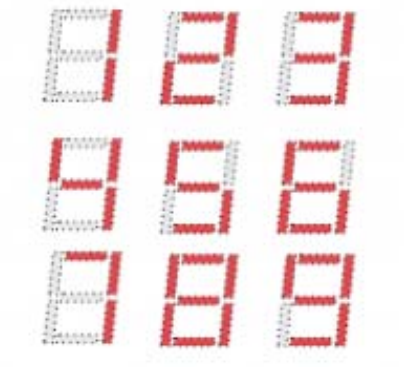
code018.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

```
module m_top ();
  reg  [3:0] r_in;
  wire [6:0] w_led;
  integer i;
  initial
    for (i = 0; i <= 15; i = i + 1) #10 r_in <= i;

  initial       $write("      abcdefg¥n");
  always@(*) #1 $write(" %x -> %b¥n", r_in, w_led);

  m_7segled m_7segled0 (r_in, w_led);
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```





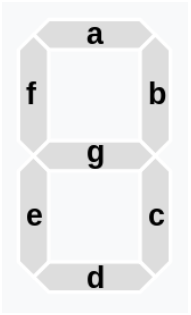
Inside code019.v

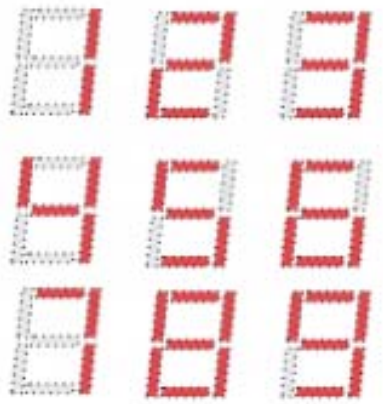
- Seven-segment LED decoder の別の例を示す.
- 関係演算子(==)は等しい時に1'b1となり，そうでなければ1'b0となる.
- code019.v ではreg型は使っていない．このため，m_7segled からは組合せ回路が合成される.
- code018.v の m_7segled から，組合せ回路が合成されるか？順序回路が合成されるか？
  - reg型の信号が常にレジスタに合成されるという訳ではない.

code019.v（m_topの記述はcode18.vと同じ）

```
module m_7segled (w_in, w_led);
  input  wire [3:0] w_in;
  output wire [6:0] w_led;

  assign w_led = (w_in==4'd0) ? 7'b1111110 :
                 (w_in==4'd1) ? 7'b0110000 :
                 (w_in==4'd2) ? 7'b1101101 :
                 (w_in==4'd3) ? 7'b1111001 :
                 (w_in==4'd4) ? 7'b0110011 :
                 (w_in==4'd5) ? 7'b1011011 :
                 (w_in==4'd6) ? 7'b1011111 :
                 (w_in==4'd7) ? 7'b1110000 :
                 (w_in==4'd8) ? 7'b1111111 :
                 (w_in==4'd9) ? 7'b1111011 :
                 7'b0000000;
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```





Inside code020.v

- Seven-segment LED decoder の別の例を示す.
- code020.v の m_7segled から，組合せ回路が合成されるか？順序回路が合成されるか？
  - w_in が 4'ha の時に，どうして 7'b1111011 が出力されるのか？

code018.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 0000000
b -> 0000000
c -> 0000000
d -> 0000000
e -> 0000000
f -> 0000000
```

code020.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
    endcase
  end
endmodule
```

```
      abcdefg
0 -> 1111110
1 -> 0110000
2 -> 1101101
3 -> 1111001
4 -> 0110011
5 -> 1011011
6 -> 1011111
7 -> 1110000
8 -> 1111111
9 -> 1111011
a -> 1111011
b -> 1111011
c -> 1111011
d -> 1111011
e -> 1111011
f -> 1111011
```

Inside code021.v

- Seven-segment LED decoder の別の例を示す.
- case文ではなく，if文 (if else) を用いて記述することもできる.
  - code18.v と code21.v は同じ出力となる.

code018.v

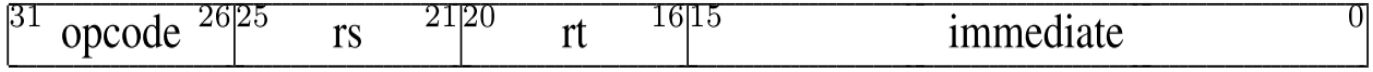
```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    case (w_in)
      4'd0   : r_led <= 7'b1111110;
      4'd1   : r_led <= 7'b0110000;
      4'd2   : r_led <= 7'b1101101;
      4'd3   : r_led <= 7'b1111001;
      4'd4   : r_led <= 7'b0110011;
      4'd5   : r_led <= 7'b1011011;
      4'd6   : r_led <= 7'b1011111;
      4'd7   : r_led <= 7'b1110000;
      4'd8   : r_led <= 7'b1111111;
      4'd9   : r_led <= 7'b1111011;
      default: r_led <= 7'b0000000;
    endcase
  end
endmodule
```

code021.v

```
module m_7segled (w_in, r_led);
  input  wire [3:0] w_in;
  output reg  [6:0] r_led;
  always @(*) begin
    if      (w_in==4'd0) r_led <= 7'b1111110;
    else if (w_in==4'd1) r_led <= 7'b0110000;
    else if (w_in==4'd2) r_led <= 7'b1101101;
    else if (w_in==4'd3) r_led <= 7'b1111001;
    else if (w_in==4'd4) r_led <= 7'b0110011;
    else if (w_in==4'd5) r_led <= 7'b1011011;
    else if (w_in==4'd6) r_led <= 7'b1011111;
    else if (w_in==4'd7) r_led <= 7'b1110000;
    else if (w_in==4'd8) r_led <= 7'b1111111;
    else if (w_in==4'd9) r_led <= 7'b1111011;
    else                 r_led <= 7'b0000000;
  end
endmodule
```

Inside code022.v

- ビット選択の例を示す．バスは多ビットの束で表現されるので，バスから選択するビットの範囲を指定する．
- MIPSアーキテクチャの命令で用いられるI形式の命令から各フィールドを選択する例．
- レジスタの宣言時に初期値を指定できる．



code022.v

```
module m_top ();
  reg  [31:0] r_ir = 32'h1464fffe;
  wire [5:0]  w_op;
  wire [4:0]  w_rs;
  wire [4:0]  w_rt;
  wire [15:0] w_imm;
  initial #1 $write(" %x -> %x %x %x %x ¥n", r_ir, w_op, w_rs, w_rt, w_imm);
  m_decode m_decode0 (r_ir, w_op, w_rs, w_rt, w_imm);
endmodule

module m_decode (w_ir, w_op, w_rs, w_rt, w_imm);
  input  wire [31:0] w_ir;
  output wire [6:0]  w_op;
  output wire [4:0]  w_rs;
  output wire [4:0]  w_rt;
  output wire [15:0] w_imm;
  assign w_op  = w_ir[31:26];
  assign w_rs  = w_ir[25:21];
  assign w_rt  = w_ir[20:16];
  assign w_imm = w_ir[15:0];
endmodule
```

Simulation output

```
1464fffe -> 05 03 04 fffe
```

Inside code023.v

- ビットの連結 (concatenation) の例を示す.
- 連結演算子 ({ }) は, 幾つかの信号を連結してビット長の大きい1つのバスにできる. 4ビットの信号 w_a, w_b を連結するには {w_a, w_b} と記述する. 4ビットの信号 w_a, w_b, w_c を連結するには {w_a, w_b, w_c} と記述する.
- ある信号を複製してビット長の大きい1つのバスにできる. 例えば, 4ビットの信号 w_a を3回複製して連結するには {3{w_a}} と記述する. 例えば, {4{w_a}} と {w_a, w_a, w_a, w_a} は同じビット列となる.
- 最後の例で示した下位ビットのMSBを複製して上位ビットを補填する操作は, 2の補数で表現された符号付きの整数を符号拡張する際に用いられる. 後の講義で解説する.

code023.v

```
module m_top ();
  reg [3:0] r_a = 4'b1001;
  reg [3:0] r_b = 4'b0101;
  reg [3:0] r_c = 4'b1111;
  initial #1 begin
    $write("%b¥n", {r_a, r_b});
    $write("%b¥n", {r_a, r_b, r_c});
    $write("%b¥n", {2{r_a}});
    $write("%b¥n", {3{r_a}});
    $write("%b¥n", {4{r_a}});
    $write("%b¥n", {{4{r_a[3]}}, r_a});
    $write("%b¥n", {{4{r_b[3]}}, r_b});
  end
endmodule
```

Simulation output

```
10010101
100101011111
10011001
100110011001
1001100110011001
11111001
00000101
```

w_bの値を赤色で強調した.

Inside code024.v

- 関係演算子 (>, <, >=, <=, ==, !=) の例を示す.
- 例えば, w_a >= w_b は, w_a の値が w_b の値以上であれば 1'b1, そうでなければ 1'b0 となる.
- C言語と同様.
- ノンブロッキング代入の演算子 <= と 関係演算子 <= は同じ記述だが, 文法的に区別できる.
  この演習では (w_a >= w_b) の様に, 関係演算子の比較の前後に ( ) を追加して明示的に区別する.

code024.v

```
module m_top ();
  reg [3:0] r_a = 4'd7;
  reg [3:0] r_b = 4'd8;
  initial #1 begin
    $write("%b¥n", (r_a> r_b));
    $write("%b¥n", (r_a< r_b));
    $write("%b¥n", (r_a>=r_b));
    $write("%b¥n", (r_a<=r_b));
    $write("%b¥n", (r_a==r_b));
    $write("%b¥n", (r_a!=r_b));
  end
endmodule
```

Simulation output

```
0
1
0
1
0
1
```

Inside code025.v

- 論理シフト演算 (>>, <<) の例を示す．算術シフトについては別の演習で．
- C言語と同様．
- 例えば，w_a << 3 は，w_a の値を左に3ビット移動させ，下位の3ビットは0となる．同様に，w_b >> 2 では，w_bの値を右に2ビット移動させ，上位の2ビットは0となる．
- 論理シフト演算では，シフトさせるビット数としてワイヤ型やレジスタ型の信号を用いてもよい．

code025.v

```
module m_top ();
  reg [7:0] r_a = 8'b11110101;
  reg [2:0] r_s = 3'd3;
  initial #1 begin
    $write("%b¥n", (r_a>>0));
    $write("%b¥n", (r_a>>1));
    $write("%b¥n", (r_a<<1));
    $write("%b¥n", (r_a>>r_s));
    $write("%b¥n", (r_a<<r_s));
  end
endmodule
```

Simulation output

```
11110101
01111010
11101010
00011110
10101000
```

FPGA constraint file, XDC (Xilinx Design Constraints)

- 制約 (constraint) を与えるためのファイル.
- project_1 で用いた main01.xdc の内容の制約ファイルでは，ledという信号を H17 というピンに割り当てる制約，btn という信号を P18 というピンに割り当てる制約を追加している.
  - led, btn は論理合成のためのトップモジュールとしてVerilog HDL記述で列挙した信号の名前
- 信号をピンを割り当てる制約が無い場合，その信号はVivadoによって適切なピンに割り当てられる.
- また，これらのピンを LVCMOS33 (low voltage CMOS 3.3V) とする制約を追加している. この制約について，本演習では詳細を理解する必要はない.

main01.xdc

```
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports { led }];
set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33} [get_ports { btn }];
```

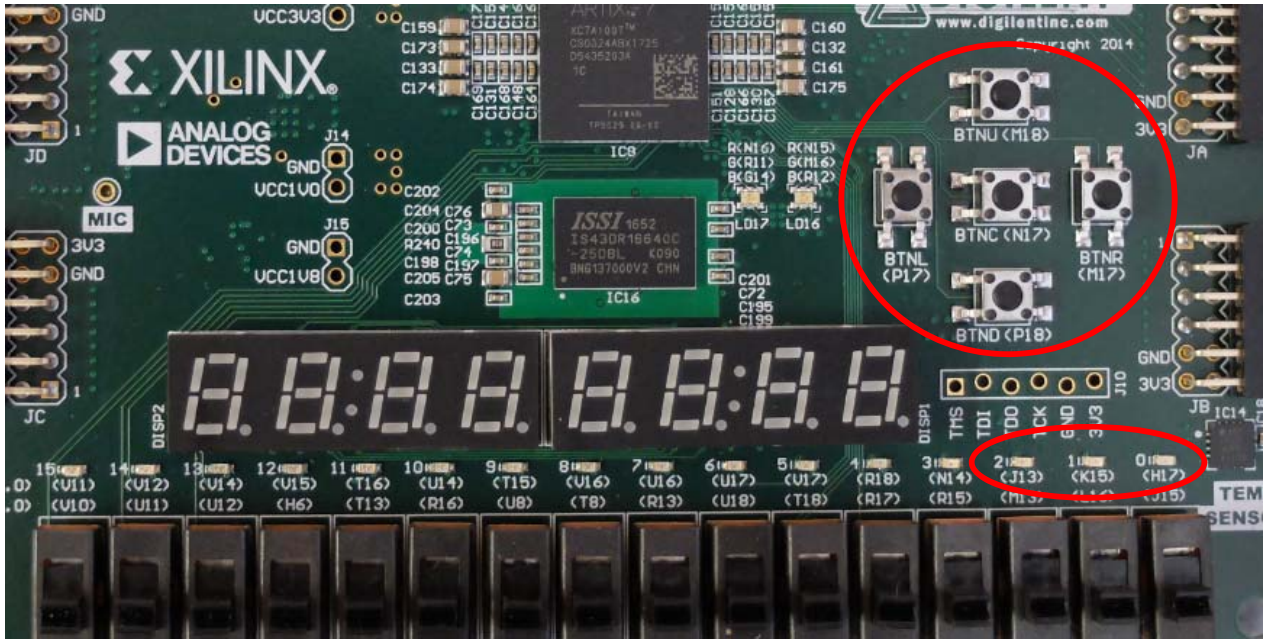
```
module main (btn, led);
  input  wire btn;
  output wire led;

  assign led = ~btn;
endmodule
```

FPGA constraint file, XDC (Xilinx Design Constraints)

- main.xdcをmain02.xdcの内容となるように入力する.

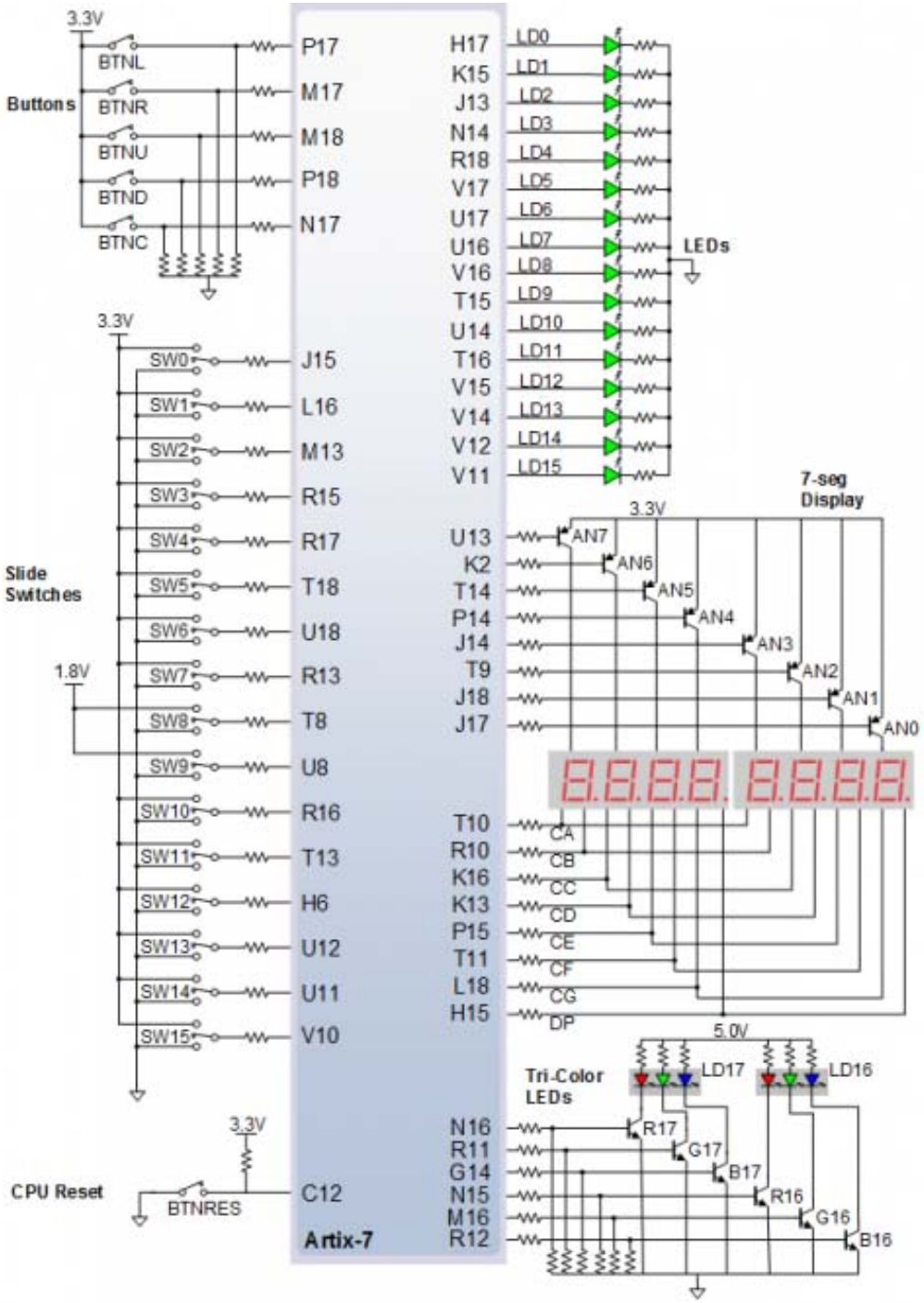
main02.xdc

```
set_property -dict { PACKAGE_PIN M18 IOSTANDARD LVCMOS33} [get_ports { w_btn[0] }];
set_property -dict { PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports { w_btn[1] }];
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports { w_btn[2] }];
set_property -dict { PACKAGE_PIN M17 IOSTANDARD LVCMOS33} [get_ports { w_btn[3] }];
set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33} [get_ports { w_btn[4] }];

set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports { w_led[0] }];
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports { w_led[1] }];
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports { w_led[2] }];
```



Nexys 4 DDR Basic I/O

- 5 buttons
- 16 slide switches
- 16 LEDs
- 7-segment display
- Tri-color LEDs
- CPU Reset button



From Nexys 4 DDR Reference Manual

Inside code026.v

- main.xdcをmain02.xdcの内容となるように入力する.
- main.vをcode026.vの内容となるように入力する. リダクション演算子(&, |, ^) の例.
  例えば ^ はバスの全てのビットの排他的論理和となる. コメントを参照.
- Vivadoで合成してビットファイルを生成し, FPGAをコンフィギュレーションする. 次のスライド参照.

code026.v

```
module m_top ();
  reg  [4:0]  r_btn;
  wire [2:0]  w_led;
  initial begin
    #10 r_btn <= 5'b00000;
    #10 r_btn <= 5'b11111;
    #10 r_btn <= 5'b00010;
  end
  always@(*) #1 $write(" %b -> %b¥n", r_btn, w_led);
  m_main m_main0 (r_btn, w_led);
endmodule

module m_main (w_btn, w_led);
  input  wire [4:0] w_btn;
  output wire [2:0] w_led;
  assign w_led[0]  = &w_btn; // same as w_btn[0] & w_btn[1] & w_btn[2] & w_btn[3] & w_btn[4]
  assign w_led[1]  = |w_btn; // same as w_btn[0] | w_btn[1] | w_btn[2] | w_btn[3] | w_btn[4]
  assign w_led[2]  = ^w_btn; // same as w_btn[0] ^ w_btn[1] ^ w_btn[2] ^ w_btn[3] ^ w_btn[4]
endmodule
```

教科書

コンピュータの構成と設計 第5版、パターソン＆ヘネシー（成田光彰　訳）、日経BP社

1. コンピュータの抽象化とテクノロジ
2. 命令：コンピュータの言葉
3. コンピュータにおける算術演算
4. プロセッサ

A. アセンブラ，リンカ，SPIMシミュレータ

B. 論理設計の基礎

References

- Computer Logic Design support page
  - http://www.arch.cs.titech.ac.jp/lecture/CLD/
- 情報工学系計算機室
  - http://www.csc.titech.ac.jp/
- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Nexys 4 DDR Atrix-7 FPGA
  - https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
  - https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog
- Frix (Feasible and Reconfigurable IBM PC Compatible SoC)
  - http://www.arch.cs.titech.ac.jp/a/Frix/