# 吉本研究室

computational science and computer science

# 第6回説明

## ハードウェア記述言語(HDL)

## この講義では(System)Verilogを使う。他にVHDLが著名である。

Verilogによる基本の回路

課題5-3をVerilogで書いた例

Verilogによるバブルソート回路

Verilogの各種情報へのリンクを参考情報に用意している。例えば規格書へのリンクがある。Verilogの文法の詳細を確認したいのであれば、規格書を読むべきである。

## ハードウェア記述言語について

- HDLの目的
  - シミュレーションによるハードウエアの動作の検証を行う。（元々はこちら）
  - 論理合成によって、ハードウエアを生成する。（この講義の課題で行うのはこちら）
- HDLの記述レベル
  - gate level、register transfer level、behavior level
  - 論理合成できるレベル、テストベンチに使うレベル
  - HDLはハードウェアの振る舞いを記述するが、すべてのHDL記述が回路に合成可能ではなくサブセットのみが合成可能である。
- ハードウェア＝論理回路を記述しているときの注意
  - C言語のような手続き型プログラミング言語で書いたプログラムを逐次翻訳しようとしないこと。
  - 回路の静的な存在を宣言することがやるべきこと。回路はただそこに存在しているだけ。
    - 組み合わせ回路は、入力信号から出力信号を生成する静的なルール（＝計算式）と等価。
    - 動作を書くように見える文を使うところがあるが、これは「ある動作をする回路が存在する」ことを宣言するように使う。
      - 書くべき「動作」は「クロックエッジが来た時、（有効化されているのなら）入力信号をレジスタに固定する」だけに限定できる。
    - 静的な存在の宣言なので、要素の宣言の順番に意味はないし、意味が出来てはならない。
  - HDLを書く前に回路の上の情報の流れが図面で書けている必要がある。
    - 回路の構成要素＝単純な組み合わせ回路またはレジスタ

    - 情報の流れ＝構成要素の入出力の接続関係
- 記述した論理回路の動作検証のためのテストベンチを書くこともでき、こちらは手続き型プログラミング言語同様、情報処理の手順＝論理回路を動作させ結果を点検する手順、を書くことになるが、この講義では、与えられているテストベンチを改変して自分のデバッグに役立てる場合だけ、この使い方をすることになる。

## Verilogの言語要素

- コメント
    - 単一行のコメントは「`//`」で始める。
    - 複数行のコメントは「`/*`」と「`*/`」で囲む。
- リテラル値
    - (bit幅)'(radix)(数値)の形

| | |
|---|---|
| 7'b1111000 | 7bitの数の2進数表現1111000 |
| 32'h1a2b3c4d | 32bitの数の16進数表現1a2b3c4d |
| 8'd134 | 8bitの数の10進数表現134 |
| 12'o3725 | 12bitの数の8進数表現3725 |
| data = '0; | dataの全bitを0にする。System Verilog |
| data = '1; | dataの全bitを1にする。System Verilog |

    - bit幅を指定せずに、3などと定数を書くとinteger型のbit幅(規格では32以上、通常32)を持つと解釈される。
- シンボルとしての定数
    - **parameter** 上位moduleから定義を引き継ぐ物
    - **localparam** 上位moduleから定義を引き継がない物
    - **'define** 文字列置換されるマクロの定義
    - 例

```
'define PARAM0 0

module example #(parameter param1 = 10)
  (input wire a, output wire b)
   ...
   localparam param2 = 20;
   ...
   if (a == PARAM0) begin
      ...
   end
   ...
endmodule
```

- ネット型とレジスタ型
  - ネット型(wire)の信号は配線の状態に対応する。それ自身で値を保持することは無く、それにつながっている回路の出力値で値が決まる。assign文はこの接続関係を記述する。
  - レジスタ型(reg)は状態が保持される可能性のある信号を表現する。実際に保持する場合にはラッチやフリップフロップに相当する。結果論的に状態を保持する必要が無く、組み合わせ回路の出力に相当する場合もある。
    - Verilogではinitialとalwaysなどの中でのみ代入できる。
    - System Verilogのlogic型はregと等価である。ただしSystem Verilogではalwaysとinitialの中に加えて、継続代入（ただし一ヶ所からだけ）できる。このためSystem Verilogでは多くのケースでwireに変えてlogicを利用できる。
  - wireとreg(logic)には、1 bitにつき可能な状態がH=1,L=0,X,Zの4つある。Xは未定義状態、Zはhigh impedance状態。System Verilogの場合、2値(H,L)しか可能な値がないbit型もある。
  - 信号にbit幅を持たせる場合、

```
wire [7:0] a;
wire [15:0] b;
reg [31:0] c;
```

のように宣言する。それぞれ8bit信号a、16bit信号b、32bit信号cである。

  - 信号を符号付きとして宣言する。

```
wire signed [7:0] sa;
reg signed [15:0] sb;
```

  - 信号の中の1つのbitの取り出し。

```
wire [31:0] x;
wire a;
assign a = x[15];
```

  - 信号の一部のbit列の取り出し。bit範囲を[ * : * ]で指定すれば良い。単一のbitが必要な時には不適切なので注意すること。

```
wire [31:0] x;
wire [15:0] lo;
wire [15:0] hi;
```

```
assign lo = x[15:0];
assign x[31:16] = hi;
```

- +: を使うと、一定幅の信号を取り出すのに便利であり、またCAD側に取り出される信号の幅が定数であることを宣言することが出来る。以下の例ではwordからbyteを取り出している。i=0の時は[7:0]の範囲、i=1の時は[15:8]の範囲になる。Vivadoは[i*8+7 : i*8]を合成可能とみなさないので注意。(2018.3)

```
wire [7:0] byte;
wire [31:0] word;
integer i;
...
byte = word[i*8 +: 8];
```

- 配列の宣言。配列はしばしばメモリを意味する。
  - 16bit、1024要素の配列ram

```
reg [15:0] ram[1024];
wire [15:0] data;
wire [9:0] addr;
...
always @(posedge clk) begin
   ram[addr] <= data;
end
```

- 手続き代入：レジスタ型（値を保持する可能性がある型）に対する代入
  - always, initial, task, functionの中で使う代入
  - ブロッキング代入「 =」：代入文が上から順番に評価される。ある文の評価結果は次の文の右辺に反映される。組み合わせ回路の記述ではこちらを使う。
  - ノンブロッキング代入「<=」；代入文は並列に評価される。ある文の評価結果はその時刻で評価される他の文の右辺に影響せず、その次の時刻から評価結果は反映される。シミュレータ的には同時刻のイベント処理の終わりで評価結果がまとめて代入される。alwaysによる順序回路の記述ではこちらを使うのが基本。
- 継続代入：ネット型に対する代入
  - assign文
    「assign “ネット型の信号” = 式; 」の形
  - and(), or(), not()などgate primitiveの引数
  - moduleのportの引数
  - 常時、代入先＝代入元となるよう代入先が駆動されることを意味する。つまり組み合わせ論理的。
- procedure
  - always, initial, task, functionで宣言される構造
  - 手続き的な記述をする。その内部で、if文、case文が使える。

- always文
  「 always “センシティビティリスト” 式」の形。センシティビティリストで指定した信号変化があった時に式を評価する、という意味を持つ。この講義の課題ではセンシティビティリストに”@(posedge クロック信号)”の形だけを使うはずである。クロック信号の立ち上がりにおいて「式」を評価する、という意味になる。（注意：論理回路を書いている時は、このように動作する論理回路が存在すると宣言する意味）
- initial、task、functionはこの講義の課題で使うことはないはず。
- 式としてbegin endで囲まれるブロックを通常伴う。

```
always @(posedge clk) begin
    ...
    ...
end
```

- module
  - 入力portと出力portをもつ、部分回路を表現する。内包する回路は、以下を用いて表現する。
    - 内部で使う信号の宣言文
    - assign文
    - always文
    - 他のmoduleのインスタンスの宣言
    - gate primitive
  - module内部の記述に使うparameterを引数の形態で指定できる。
  - module の宣言と引用。

```
module an_instance_of_module
   # ( parameter A_PARAM = 10,
       parameter B_PARAM = 20 )
     ( input wire a,
       input wire b,
       output wire c );
   ...
endmodule

module an_parent_module
   ...
   an_instance_of_module #(12,24) u1(x,y,z); // parameterを変更
   an_instance_of_module u2(p,q,r); // defaultのパラメータ
   ...
endmodule
```

  - 回路に合成させるmodule内部の記述にそのmodule自身を使うことはできない。つまり再帰呼び出しはできない。
  - moduleのportの引数は継続代入の関係を意味している。

- Verilogではmoduleの入力ポートはネット型でなければならない。System Verilogであればlogic型も使える。ただしこの講義の範囲内の場合、Verilogとの互換性を捨ててまでこの機能を使う理由はないはずである。
- moduleの出力ポートに常にレジスタを取り付ける場合、そのポートをoutput regと宣言する。何も書いていない場合wireになっている。そのため実引数もwireでなくてはならない。wireの場合、initialの中でブロッキング代入を用いて値を直接変化させられない。必要なら一旦regを用意して、このregとwire型の実引数をassignで接続し、regの値を代入で変化させなければならない。
- 回路のシミュレーションをする場合にのみ、initial文によって、シミュレーションが始まってから順に実行する手続き的な処理を記述する。要するにinitial文によって、手続き的な処理で仮想回路の振る舞いを記述できる。例えば、テストのための入力信号源を記述するのに使える。

- リダクション演算子(Verilogに特有)
  信号xはnビット信号 x[n-1:0] と宣言されているものとする。以下の演算子&, |, ^はxから1bitの信号yを計算する。

| 表記 | 意味 |
|---|---|
| y = &x | y = x[n-1] AND x[n-2] AND … AND x[1] AND x[0] |
| y = \|x | y = x[n-1] OR x[n-2] OR … OR x[1] OR x[0] |
| y = ^x | y = x[n-1] XOR x[n-2] XOR … XOR x[1] XOR x[0] |

- bit 演算子
  信号x,y,zはnビット信号 x[n-1:0] , y[n-1:0], z[n-1:0]と宣言されているものとする。

| 表記 | 意味 |
|---|---|
| z = x & y | z[i] = x[i] AND y[i] for i = 0, …, n-1 |
| z = x \| y | z[i] = x[i] OR y[i] for i = 0, …, n-1 |
| z = x ^ y | z[i] = x[i] XOR y[i] for i = 0, …, n-1 |
| z = ~x | z[i] = NOT x[i] for i = 0, …, n-1 |

- シフト演算子
  xとsをそれぞれ信号とする。sが定数でなくても回路に合成できる。

| 表記 | 意味 |
|---|---|
| x<<s | xを左(MSB側)にs bit論理シフト |
| x>>s | xを右(LSB側)にs bit論理シフト |
| x<<<s | xを左(MSB側)にs bit算術シフト。以下の>>>と同じく使い方に注意すること。 |
| x>>>s | xを右(LSB側)にs bit算術シフト。xと出力先の両方が符号付きでないと、符号なし(論理シフト)で計算されて |

しまうので注意すること。必要なら$signed()を使って$signed($signed(x)>>>s)とする。

- 連接演算子{}。いくつかの信号をbit順に連結した信号を作る。

```
wire [7:0] x;
wire [15:0] y;
wire c;
wire [24:0] z;
assign z = {c,x,y};
```

なら、

```
z[24] = c
z[23] = x[7], z[22] = x[6], ..., z[16] = x[0]
z[15] = y[15], z[14] = y[14], ..., z[0] = y[0]
```

となる。連接演算子の中で同一信号の繰り返しを表現することもでき、

```
wire [7:0] x;
wire [23:0] y;
assign y = {3{x}};
```

とすると、

```
y[23] = x[7], y[22] = x[6], ..., y[16] = x[0]
y[15] = x[7], y[14] = x[6], ..., y[8] = x[0]
y[7] = x[7], y[6] = x[6], ..., y[0] = x[0]
```

となる。

- System Verilogでのみ使える演算子

| | |
|---|---|
| ++ | インクリメント |
| — | デクリメント |

Verilogへの互換性が問題なら使用しないこと。

- Verilogでの信号の比較

    - シミュレーション上では信号に0(L),1(H)だけではなくXやZの状態があるため、信号の比較時にX,Zを認識して比較する===や!==を使わなければならない。==と!=は比較対象にXやZが入っている場合、値がXになるので意図していない結果になる。ただし===と!==は回路には合成できない。そもそもXやZは本当の電気信号ではない。
    - ==と!=は回路に合成できる。
- 信号の整数値としての解釈
  四則演算のオペランドとなる時など、整数値として信号が解釈される時、なにも宣言されていなければ、unsignedとして扱われる。信号xをsignedとして扱いたい時には`$signed(x)`と書く。signed関連の詳細はこちら。
- bit長の諸問題
    - 混乱しやすいので、できるだけ計算で使われるbit長が自明になるようにコードを書くこと。このためにbit長の拡張や一部のbitの取り出しは明示的にするべきである。
    - 詳細はこちら

## Verilogによる論理回路の表現

- 組み合わせ回路の表現
    - 内部状態は無く、入力に対して出力が一意に定まる＝出力は入力から即時に決定される。
    - assign文は継続的代入を意味する。つまり式の左辺にある入力の変化が即時に文で指定した計算式を通して出力に反映される。したがって、assign文は組み合わせ回路を表現する。
    - 複数のassign文は順不同に（並列に）評価されてもよいように記述しなければならない。シミュレータは実際にこうなっているかどうかを点検しないことがある(Vivado 2017.4では点検できない)ので気をつけること。
        - 同一変数に対するassign文が複数あると、評価の順が問題になってしまう。Vivadoで合成するとこのような場合「`multi-driven net`」というエラーになる。
    - 条件演算子(A?B:C)を使えば、真理値表をassign文で書くこともできる。
        - case的if文の記述例

```
assign x1 = (a == p1) ? def1 :
            (a == p2) ? def2 :
            (a == p3) ? def3 : def4;
assign x2 = (a == p1) ? def5 :
            (a == p2) ? def6 :
            (a == p3) ? def7 : def8;
assign y = (b == q1) ? x1 : x2;
```

    - 一つの式でかけない複雑な関数を表現したい場合には、一旦その関数をfunctionで書き表しておいて、assign文の右辺にそのfunctionを使う。（この講義では必要ないはず）
    - 組み合わせ回路をcase文で表現する場合（この講義では必要ないはず）
        - 回路の可能なすべての入力パターンに対して、case文が値を定義しなければならない。必要ならdefaultを使う。
- 条件を指定する信号selによって、生成される信号を切り替えたい時

- sel = Hの時の式とsel = Lの時の式を計算する組み合わせ回路を両方用意して、selによってそれらの出力のいずれかを選択回路で選択すれば良い。
- 例: selによって加算と減算を切り替える

```
assign out_add = a + b;
assign out_sub = a - b;
assign out = sel ? out_add : out_sub ;
```

- 順序回路の表現
  - always文はsensitivity listに現れる信号が変化したときに、always文を評価して回路の状態を変化させることを表現する。
    - 複数のalways文はそれぞれ並列に動作する回路に対応できなければならない。つまりシミュレーターは順不同でこれらalways文を評価するが、そのことによって結果が変わってはならない。したがって、異なるalways文から同一の信号に代入してはならない。例えノンブロッキング代入であっても。条件処理で実は代入が排他的になっている場合も含めて許されないので注意せよ。Vivadoでは「multi-driven net」という合成時のエラーになる。
    - 同一のalways文のノンブロッキング代入はそれぞれの文が評価された順番で実行される。したがって、最後の代入が有効になる。
  - sensitivity listにclock edgeを指定するとclockに同期する回路が記述される。
    - 現在のclockが終わった時に変化する信号を現在のclockの信号から決定する方法を記述すれば、同期式順序回路の記述になる。この時ノンブロッキング代入が便利である。
    - 条件eの元でのノンブロッキング代入は、enable付きD flip-flopに対応する。つまりD flip-flopのenable信号e、入力信号q_next、出力信号qとするとこのようになる。

```
if (e) begin
   q <= q_next;
end
```

  - 電源投入時の順序回路の状態は一般論としては不定になる。そのため最初にリセットして状態を確定させなければならない。ただしFPGAのフリップフロップは初期値が通常確定している。
  - Xilinx FPGAでのリセット
    - 同期リセットを使う方が良い。この講義で使うのは同期リセットだけ。
    - 7シリーズとUltrascaleの場合、内臓D-FFのリセットは正論理になっている。
  - alwaysによる同期リセットの記述
    - リセット要求が即時にではなくクロックエッジで処理されるのが同期リセット
    - センシティビティリストはクロックエッジのみ

```
always @(posedge clk) begin
   内部
end
```

    - 内部は

```
if (resetする) begin
   リセット処理
end else begin
   本来の処理
end
```

- alwaysによる非同期リセットの記述
    - リセット要求がクロックエッジとは無関係に即時に処理されるのが非同期リセット
    - センシティビティリストがクロックエッジまたはリセット要求の形

```
always @(posedge clk or negedge resetn) begin
   内部
end
```

    - 内部は

```
if (resetする) begin
   リセット処理
end else begin
   本来の処理
end
```

- always文と組み合わせ回路
    - sensitivity listに組み合わせ回路のすべての入力をもれなく書き、これらから出力を記述すれば組み合わせ回路が記述される。
    - System Verilogの場合、代わりにalways_comb文を使う。この文は必ず組み合わせ回路に対応するのでsensitivity listはない。ただし、内部の記述が実際にすべての入力パターンに対して出力を決定するようにするのは記述者の責任であるので気をつけること。

## Verilogの注意点

- 繰り返し構文(for, while, repeat)
    - 合成可能であるためには、これら構文の繰り返し範囲が静的に決まっていなければならない。つまり合成ソフトが読んだとき、これを繰り返し構文なしで展開できなければならない。
    - この講義の課題では使わなくても問題ないはず。信号の並びの扱いは他の手段で間に合う。
- レジスタ型の初期化構文
    - initialブロックで初期化する他、宣言文中でもインライン初期化できる。
    - Verilogの場合、各initialブロックとインライン初期化は時刻0でalwaysブロックの処理とまとめて順不同に実行されるので、初期化同士に依存性がある場合には注意すること。なお一つのinitialブロックの中では順番が保持される。

    - System Verilogの場合、インライン初期化はinitialやalwaysよりも前、時刻0よりも前に処理される。したがってインライン初期化の結果をinitialやalwaysの中で参照できる。
    - **原則として**レジスタの初期化はこれらinitialなどではなくreset信号によって行うべきである。(FPGAは例外であるが)物理的存在の論理回路の電源投入直後の状態は不定であり、reset信号およびそれで駆動されるリセット回路によってリセットしてやることで、設計上意図された初期状態になるものである。
- 桁上がり(carry)や負になる減算の場合、必要なだけMSB側にbitを補って演算をする。補うためには連接演算子「{}」が使える。
- 連接演算子はassignの左辺にも使える。つまり一つのassignで複数の信号をまとめて扱える。
- 回路の生成
    - parameterによって合成される回路を変えたいことがある。例えばデータのbit幅をparameterで可変にするなどである。この様な場合、generate ブロックを使う。
    - genvar で宣言された変数をif文やfor文に用いて合成される回路を変更できる。
    - 変更は静的に行われる。つまり論理合成される時点でどのような回路になるかは決まってしまう。
- システムタスク
    - テストベンチを記述するための機能。回路には合成できない。シミュレーターに対して指示する物。

| | |
|---|---|
| `$finish;` | シミュレーターの実行を終了する。 |
| `$display("フォーマット",引数1,引数2,...);` | メッセージを出力する。 |
| `$dumpvars(0);` | 記録をとる変数の範囲を呼び出しモジュール以下のすべてに指定する。 |
| `$dumpfile("ファイル名");` | 変数の記録を格納するファイルを指定する。 |
| `$urandom()` | unsigned 32bitの乱数を発生させて返す関数。 |
| `$readmemh("ファイル名",変数名)` | 16進数表記のファイルからデータを読み込んで変数にセットする。 |