計算機科学実験及演習3 ハードウェア　資料
～Verilog-HDLによる回路記述～

京都大学 工学部情報学科 計算機科学コース
計算機科学実験及演習3
ハードウェア担当
le3hw@kuis.kyoto-u.ac.jp

ゲートレベルからレジスタ転送レベルへ

- 実験2ではゲートレベル（回路図）でALU及び任意の順序回路を設計した
  - 複雑な回路を回路図で記述するのは生産性が低い
- より高い抽象度で論理回路を設計したい
  - レジスタ転送レベルで回路を記述できるハードウェア記述言語（HDL, Hardware Description Language）を用いれば良い
    - ✓順序回路のまとまりをレジスタとしてブラックボックス化し，その間の転送を記述する
    - ✓実験3では，Verilog-HDLを用いてCPUを設計する

HDLの二大勢力

- VHDL
  - 元々は仕様記述言語だったため，文法が厳格
  - ALGOLに端を発し，PL/1, Adaの系譜に連なる
- Verilog-HDL
  - 元々はシミュレーション記述言語であったため，いい加減
    - ✓ 極めて書きやすいが，バグが出やすい
    - ✓ いい加減に書いても動いてしまうことが利点でもあり欠点でもある
  - デファクトスタンダード

- ハードウェア設計者（実際的で合理的）の考え
  - どちらかで書ければ問題ない．実際，論理合成系の発達により混在していても設計可能．結局**何を設計するかが問題**

Verilog-HDL入門

module

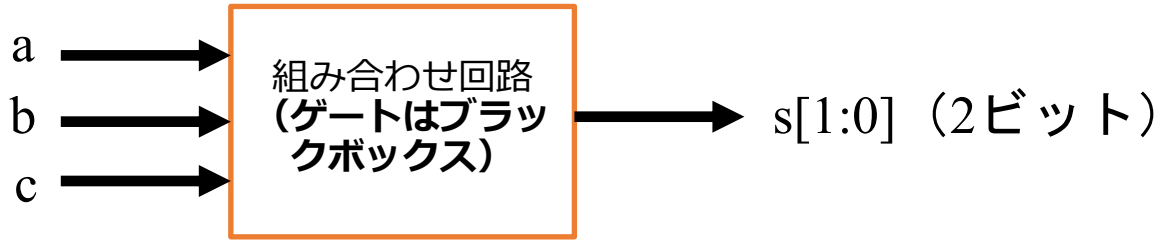
- 回路を記述する基本単位がモジュール
- 回路表現，テストベンチ等すべてこの中で記述

```
module module_name (input signal1,
                    input signal2,
                    output signal3);
    //回路の記述
endmodule
```

assign文

```
wire s[1:0];
assign s = a + b + c;
```

- 信号の出力，接続をassign文で表現
- 組み合わせ回路を記述
- wire型の変数に対してのみ書ける



具体例：4ビット加算器

バス表現（4ビット）

```
module adder4 (input [3:0] a, b,
               input cin,
               output [3:0] sum,
               output cout);
    assign {cout, sum} = a + b + cin;
endmodule
```

連接．上位cout，下位sumで合計5ビット

- たったこれだけ
- 論理合成により自動でゲートに変換してくれる

always文

- 基本的に順序回路を記述（例外ありだが，推奨しない）
- 基本構文：クロックが立ち上がった時or非同期リセットが立ち下がった時に値が変化

```
always @ (posedge clk or negedge rst_n)
begin
    if (~rst_n) begin
        リセット時の動作;
    end else begin
        クロックに同期した動作;
    end
end
```

具体例：4ビットカウンタ

```
module count4 (input rst_n, clk,
               output reg [3:0] data);
    always @ (posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            data <= 4'b0000;
        end else begin
            data <= data + 1;
        end
    end
endmodule
```

- reg型の変数に対して代入する（FFになる）
- 代入には=ではなく<=を使う

三項演算子とマルチプレクサ構文

- assignで使える三項演算子が便利

```
assign a = (b == 1’b0) ? c : d; //b==0のときc，そうでないときdを代入
```

- これを組み合わせることでマルチプレクサを記述可能（マルチプレクサ構文）

```
assign a = (b == 2’b00) ? c:
           (b == 2’b01) ? d:
           (b == 2’b10) ? e:
           f;
```

bの値に応じて出力が変化

define文

- コード中に直接数を書かないようにできる
- 文頭，またはmoduleの前に置く

```
`define DATA_WIDTH 4
module count4 (input rst_n, clk,
               output reg [`DATA_WIDTH - 1:0] data);
    always @ (posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            data <= `DATA_WIDTH' b0000;
        end else begin
            data <= data + 1;
        end
    end
endmodule
```

ヘッダファイル

- 定義をまとめてヘッダファイルとし，includeすることも可能

```
`define DATA_WIDTH 4
`define …
…
```

def.h

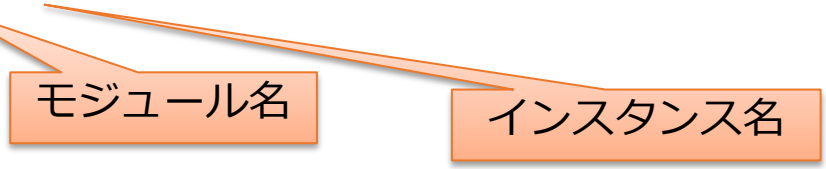
```
`include “def.h”
module (…);
…
endmodule
```

モジュールの中で別モジュールを使う

- top_moduleの中でadder4を使う

```
module top_module (input …,
                   output …);
    wire [7:0] op1, op2, sum;
    wire c0, c4, c8;
    adder4 a0(.a(op1[3:0]), .b(op2[3:0]), .cin(c0), .sum(sum[3:0]), .cout(c4));
    adder4 a1(.a(op1[7:4]), .b(op2[7:4]), .cin(c0), .sum(sum[7:4]), .cout(c8));

endmodule
```



データの表現

- ビットは0, 1, x, zのどれかの値
  - x: 不定（0か1か決まらない状態．初期値を与えていないか，バグによって生じることが多い）
  - z: ハイインピーダンス（電気的な絶縁状態．本実験では使わない）

データの表現

- wire/reg変数
  - 符号なしが基本．signedと書くことで符号つきにできる
  - ビット数，基数（b, o, h, d)を指定して値を書く
    - ✓ 4’b1010は4ビット2進数1010
    - ✓ 8’o377は8ビット8進数377
    - ✓ 8’ha0は8ビット16進数a0
    - ✓ 4’d5は4ビット10進数5
    - ✓ 指定しないと，32ビット10進数になる
- ‘_’は無視されるので読みやすさのために使用可能
  - 例：8’b1001_0011

演算子

- 算術演算
  - 加算 +
  - 減算 -
  - 乗算 *
  - 除算 /
  - 剰余 %
  - 冪乗 **
- ビット毎演算
  - AND &
  - OR |
  - XOR ^
  - NOT ~
  - XNOR ~^
- 比較・等号演算
  - ==
  - !=
  - <=
  - >=
  - <
  - >

演算子

- シフト演算
  - 論理右シフト　信号名 >> シフト幅
  - 論理左シフト　信号名 << シフト幅
  - 算術右シフト　信号名 >>> シフト幅
  - 算術左シフト　信号名 <<< シフト幅

- 論理シフトは符号を考慮しない．空いたビットはゼロで埋める
- 算術シフトは符号を考慮．符号付きに対してのみ行う．符号ビット以外の空いたビットはゼロで埋める

おすすめのルール

- Verilog HDLはいい加減のため適当に書いても動いてしまうが，そのせいで後に深刻なバグに悩まされる
- 書き方のルールを決めるとバグが出にくい

- 組み合わせ回路はすべてassignで書く
- alwaysはすべてposedge clk or negedge rst_n
- 一つのalwaysブロック内では一つの出力に対する記述だけ書く

回路のシミュレーション

- テストベンチ（これもVerilog-HDLで書く．実は実験2でもやった）を用意

```
`timescale 1ns / 1ps // 1タイムスケールあたりの実時間 / 最小単位（丸めの精度）
module my_test_bench; //入出力はなし
    reg rst_n, clk;
    wire [3:0] data;

    count4 c1(rst_n, clk, data); //テストしたいモジュール

    always begin
        #10 clk = ~clk; // クロック
    end
    initial begin
          rst_n = 1; clk = 0; #35
          rst_n = 0; #200
          rst_n = 1;
          $display(“%d %h %h %h”, $time, rst_n, clk, data); // 値の観測が可能
          #1000    $stop;
    end
endmodule
```

回路のシミュレーション

- クロックはalways文で生成，その他のインプットはinitial文で記述すればよい
- $displayで値を観測可能（下のtranscription部分に出力される）
- $stopでシミュレーションを停止
- Test Bench Template Writerでででてくる@eachvecのalways文は無視してOK（中に何も書かなくてよい）

```
`timescale 1ns / 1ps // 1タイムスケールあたりの実時間 / 最小単位（丸めの精度）
module my_test_bench; //入出力はなし
    reg rst_n, clk;
    wire [3:0] data;

    count4 c1(rst_n, clk, data); //テストしたいモジュール

    always begin
        #10 clk = ~clk; // クロック
end
```

FPGA向け初期値の設定

- 回路の初期値はリセット時の動作として記述するのが基本

```
always @ (posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            data <= 4’b0000;
        end else begin
            data <= data + 1;
        end
end
```

- FPGAでは，電源投入時の値を書くことも可能

```
reg [3:0] data = 4’b0000;
```

or

```
initial begin
    data <= 4’b0000;
end
```

その他の文法

- function文
  - 複雑な組み合わせ回路を記述するのに向くが，必須ではない
- parameter文
  - パラメータを設定することができる

- 詳細はhttps://isle3hw.kuis.kyoto-u.ac.jp/hdl/index.htmlを参照

困ったら？

- TA・教員に聞く
- ググる
- 演習室にある参考書を読む
- 図書館や書店でより詳しい参考書をゲットし読む

参考書

- D.A.パターソン，J.L.ヘネシー著, 成田光彰訳『コンピュータの構成と設計』“パタヘネ”
  - CPUのアーキテクチャが詳しく書いてある（ただしSIMPLEアーキテクチャではない）
- 富田眞治，中島浩『コンピュータハードウェア』
  - SIMPLEアーキテクチャについて書いてある
- 小林優『入門Verilog HDL記述』
  - 初心者向けVerilogの解説

参考書

- S. L. Harris and D. M. Harris『ディジタル回路設計とコンピュータアーキテクチャ』”ハリスハリス“
  - だいぶ詳しくアーキテクチャが解説されている.
  - MIPS版，ARM版，RISC-V版などいろんな版がある
- 天野英晴・西村克信『作りながら学ぶコンピュータアーキテクチャ』
  - 絶版
  - CPUを設計する過程を体験できる実用的な本
  - Verilogのコード例も豊富

HDLてにをは集

- https://isle3hw.kuis.kyoto-u.ac.jp/hdl/index.html
- 本資料では説明していないfunction文やparameter文などについても載っています