Verilog HDL による回路設計記述

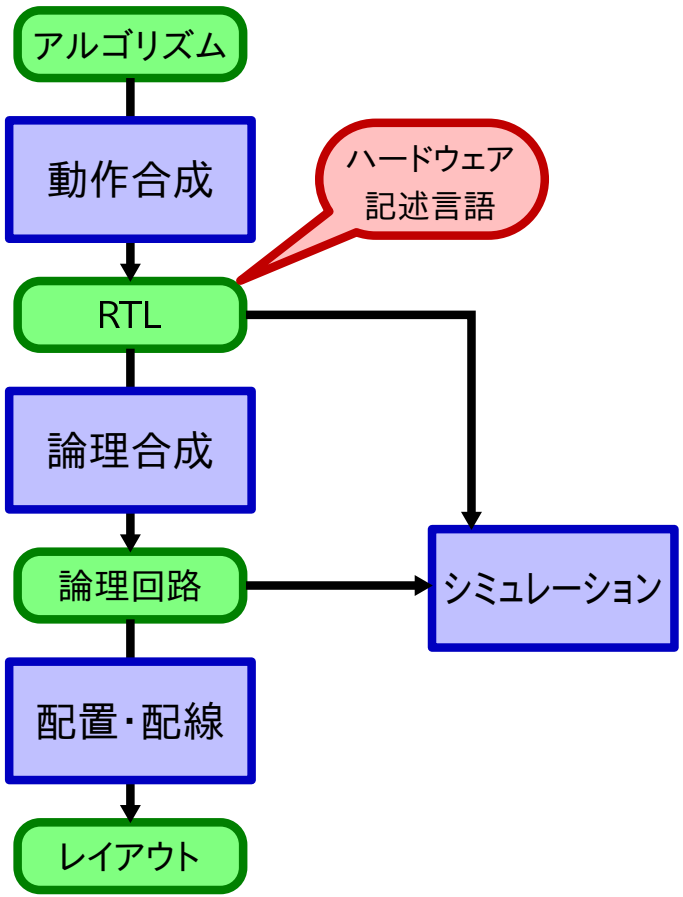
計算機科学実験及演習 3 ハードウェア

京都大学 情報学科計算機科学コース

2020 年 4 月

デジタル回路の設計フロー

- 設計の抽象度をより高く
  - レイアウト
  - ゲートレベル (論理回路図、ネットリスト)
  - レジスタ転送レベル (RTL)
  - 動作レベル
- ハードウェア記述言語 (HDL)
  - RTL、ゲートレベルでの記述
  - HDL による回路記述
    - シミュレーション可能なモデル記述
    - 論理合成可能なデザインエントリ



ハードウェア記述言語

- HDL の開発、標準化
  - VHDL: IEEE Std 1076-1987、プログラミング言語 Ada に似た文法、厳密な型、重厚な言語仕様、IEEE Std 1164-1991
  - Verilog HDL: 1984 年 ツールに搭載、IEEE Std 1364-1995、C に似た文法
  - 日本では SFL (NTT)、UDL/I (JEITA)
  - SpecC, SystemC, SystemVerilog: より抽象度の高いシステムレベル設計へ
  - 普通の C, C++ からの動作合成
- HDL はプログラミング言語ではない
  - イベント駆動で並列動作するコンポーネントの記述
  - レジスタ転送レベル 、ゲートレベルのモデリング
  - 論理合成可能な文法要素は限られる
  - 論理合成可能な論理構造は限られる

ハードウェア記述言語 Verilog HDL

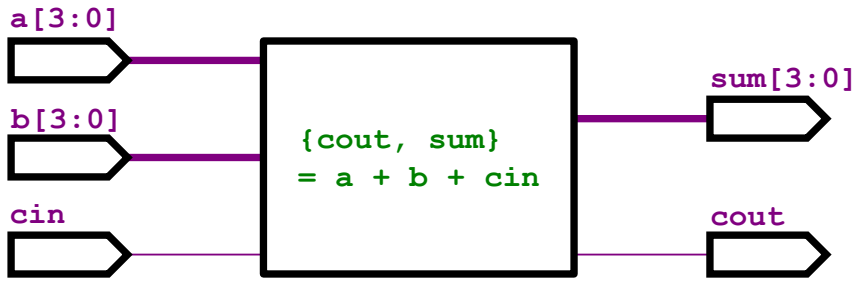
- シミュレーションと論理合成のための仕様記述
  - 論理シミュレータ用の記述言語として開発 [1984]
  - IEEE 規格 [1995]
  - HDL 論理合成系が普及 [1990 年代]
  - IEEE 規格改訂 [2001] (本稿は一部 Verilog-2001 前提)
- 文法はだいたい C 言語と同じ
  - 構文、演算子、型のユルさなど
  - ブロックが `{ }` じゃなくて `begin end` なぐらいの違い
- 抽象度は回路図より少し上ぐらいの気持ちで
  - レジスタ転送レベル: レジスタ間の論理を記述
  - 「テキストでブロック図を書く」つもりで
  - どんな回路が合成されるかイメージを持って書く

まずは記述例: 4 ビット加算器

adder4.v

```
module adder4 (
  input [3:0] a, b,
  input cin,
  output [3:0] sum,
  output cout );

  assign {cout, sum}
         = a + b + cin;
endmodule
```



- module が設計単位
  - 入出力ポートの宣言
  - パラメータ、内部変数等の宣言
  - 動作の記述

データの種類と値の表現

- 1 ビットは 0，1，x，z の 4 値
  - x: 不定 (シミュレーションでのみ使用)
  - z: ハイインピーダンス
- データはビットベクタであり整数でもある
  - 基本は符号なし、signed も指定可能 (使い方に注意)
  - リテラルはビット数、基数 (b,o,h,d) を指定

```
4'b1010、16'h0a3f
```

  - 普通に 10 進数を書くと 32 ビット値になる
  - x，z のビットを含むことも可能
  - '_' は無視されるので読み易さのために桁区切りに使用

```
8'bxx10_111x
```

  - ビット数が合わなくても演算、代入可能 (最大幅に拡張)
- データオブジェクトの種類
  - ネット wire、レジスタ reg — 配線や FF のモデル
  - 定数 parameter、整数 integer、他

データオブジェクトとポートの宣言

- データオブジェクト
  - ネット wire: 値を継続的に代入される配線

    'wire data' は 1 ビットのネット
  - ビットベクタの宣言/参照はレンジ [n:m] を指定

    (宣言) 'wire [3:0] data' は 4 ビットのネット
    (参照) 'data' は 4 ビット全体、'data[3:1]' は上位 3 ビット
  - レジスタ reg: 値を保持する変数

    (宣言) 'reg [3:0] data [0:255]' は 4 ビットのレジスタ 256 個の配列
    (参照) 'data[5]' は 4 ビット、'data[5][3:1]' は上位 3 ビット
  - wire は配線要素、reg は記憶要素にだいたい相当
  - reg が必ずしもフリップフロップにはならない
- モジュールのポート
  - input, output, inout のいずれかで宣言
  - モジュール内で wire として参照 (暗黙の宣言)
  - output を reg として使いたい場合は明記

演算と継続的代入

- C 言語と同様の演算子
  - 算術演算 (+, −, *, /, %)、ビット毎論理演算 (&, |, ^, ~)、条件演算 (?:)、論理/関係演算 (&&, ==, !=, >=, ...|)
- Verilog HDL 特有の演算子
  - 連接 {a, b}
    - 1 つのビットベクタとして扱う
    - 式の左辺にも使用可能
  - リダクション &a, |a (全ビットの論理積、論理和) など
- wire への継続的代入は assign 文
  - 組合せ回路の記述になる
  - 場合分けは条件演算で
  - 条件が複雑な場合や、同じ場合分けで複数の信号へ代入する場合は、function や always ブロックで

always、イベント制御、逐次ブロック

- always
  - 継続的に実行される文を記述
  - 制御構文 (後述) を使用可能
  - イベント制御 @... と合わせて使用が基本
    - イベント (値の変化など) 発生の度に実行
  - ブロック begin ... end と合わせて使用が基本
    - ブロック内の文は順に実行 (代入文の値の反映については後述)
- initial
  - 最初に 1 回だけ実行される文 (ブロック) を記述
  - 回路記述では使用せず、テストベンチで使用
  - テストベンチでは時間経過 #n と合わせて使用

function 定義

function [レンジ] 関数名; ... endfunction

- assign で書くには複雑な組合せ論理を記述
- always での組合せ回路記述とは一長一短

function decode2_4

```
function [3:0] decode2_4;
input [1:0] a;
begin
  case (a)
    0: decode2_4 = 4'b0001;
    1: decode2_4 = 4'b0010;
    2: decode2_4 = 4'b0100;
    default: decode2_4 = 4'b1000;
  endcase
end
endfunction
...
assign x = decode2_4(x_bin);
```

- レンジと関数名を定義
- 入力信号、内部信号を宣言
  - モジュール内の信号は宣言しなくても参照できるので注意
- 返り値は関数名の信号に代入

function、always 中の制御構文

- for, while, repeat
  - 繰り返し構造の回路を記述 (処理の繰り返しではないのに注意)
- if 文

```
if （式） begin
   ...
end else begin
   ...
end
```

- case 文
  - case 重複無し、全場合網羅 (default 記述) を推奨 (重複有の記述は if ... else if ... のが分かりやすいかも)
  - 式中で '?' (= 'z') を don't care として使える casez もある (更に 'x' も don't care になる casex もあるが非推奨)

```
case （式）
   式 1: 文 // 上から優先
   式 2: 文
   ...
   default: 文
endcase
```

下位モジュールのインスタンス

submodule instantiation

```
...
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));
...
```

- 別途定義したモジュールをインスタンス化
- ポートに wire、reg を接続
- 並列に動作する回路要素となる

モジュールの構造

module structure

```
module モジュール名 (
  ポート宣言
  ... );
  wire/reg 宣言;
  ...

  ...
  assign 文、always ブロック、下位モジュールインスタンス
  ...
endmodule
```

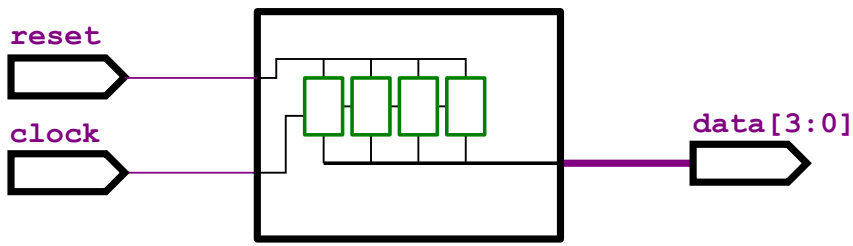
- assign 文、always ブロック、下位モジュールインスタンスは全て並列に動作する回路要素
- wire、reg に複数の回路要素で値を代入することはできない

順序回路の記述例: 4 ビットカウンタ

count4.v

```
module count4 (
  input reset, clock,
  output reg [3:0] data );

  always @(posedge clock or
      negedge reset) begin
    if (reset == 1'b0) begin
      data <= 4'b0000;
    end else begin
      data <= data + 1;
    end
  end
endmodule
```



- reg でレジスタとして宣言
  - 同名で内部変数を改めて宣言してもよい
- always ブロック内でレジスタに値を代入

always ブロック

- 時刻 0 で実行開始し無限ループ
- '@(...)' 内はセンシティビティリスト
  - 指定したイベント (信号値の変化、立ち上がり、立ち下がり) の発生まで待機
- クロック同期式順序回路
  — 基本的には全てこの書き方で

立ち上がりエッジ同期

```
always @(posedge clock)
```

非同期リセット付き ('reset==0' でリセット)

```
always @(posedge clock, negedge reset)
```

- 組合せ回路を記述する場合の always ブロック
  - 右辺の信号を全てセンシティビティリストに含む ('@*' と略記も可、むしろ略記推奨、always_comb も良し)
  - 全ての条件でもれなく代入する必要

ブロッキング/ノンブロッキング代入

- ブロッキング代入 (=)
  - 値は左辺に即反映
  - `wire` への代入はブロッキングで
- ノンブロッキング代入 (<=)
  - ブロック内の式の評価が全て終了後、同時に左辺に反映
  - ブロック内の式の記述順に依存しない
  - `reg` への代入は全てノンブロッキングで書くと間違いが少ない

Blocking

```
...
A = B;
B = A; // これは上の行の値
...
```

Nonblocking

```
...
A <= B;
B <= A; // A と B を交換
...
```

ラッチとフリップフロップの推定

always ブロックは、内容により記憶素子が推定され、意図しない回路になりがちなので注意 (function でどうなるか謎: シミュレーション系や合成系によるかも)

クロック同期

```
always @(posedge clock) begin
  s = x + y + c;
end
```

ラッチを推定

```
always @(x or y) begin
  s = x + y + c;
end
```

組合せ回路

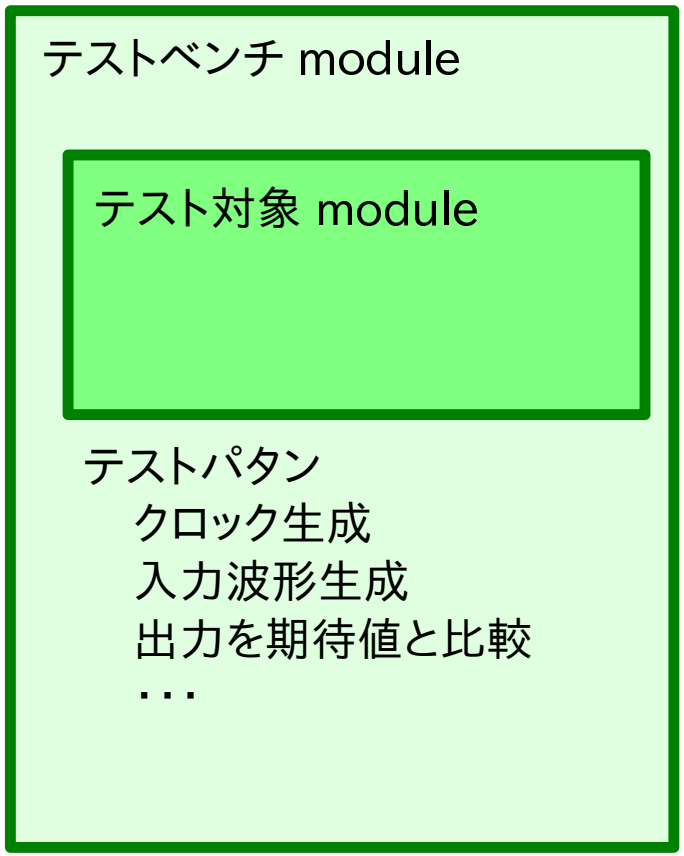
```
always @(x or y or c) begin
  s = x + y + c;
end
```

ラッチを推定

```
always @* begin
  if (c==0) s = x + y;
end
```

回路のシミュレーション

- テストベンチを用意
  - テストベンチも HDL で記述
  - 設計したモジュール (テスト対象) を下位モジュールとしてインスタンス化
  - 入力テストパタンを記述
  - 出力パタンの期待値を記述、あるいは出力パタンを観測
- シミュレーション
  - RTL シミュレーション: HDL 記述をシミュレーション
  - ゲートレベルシミュレーション: HDL 記述を論理合成し、合成結果の回路をシミュレーション



テストベンチ

- 時間の単位、シミュレーション精度を `‘timescale` で設定
- 設計ファイルを `‘include` で呼び出し
  - いずれも先頭文字はバッククオート
- 設計をインスタンス化
- `#n` で時間経過を指定
- `initial` ブロックは時刻 0 で 1 回だけ実行
- `$monitor` システムタスクでイベント毎の値を観測、あるいは波形ビューアで観測

test_count4.v

```
‘timescale 1ns / 1ns
‘include "count4.v"
module t;
  reg reset, clock;
  wire [3:0] data;

  count4 c1(reset, clock, data);

  always begin
    #10 clock = ~clock;
  end
  initial begin
    $monitor("%d %h %h %h",
      $time, reset, clock, data);
            reset = 1; clock = 0;
    #35     reset = 0;
    #5      reset = 1;
    #200    reset = 0;
    #140    reset = 1;
    #1000   $finish;
  end
endmodule
```

FPGA 向けの初期値の設定

- 回路の初期値はリセット入力時の動作として記述するのが基本
- FPGA の場合は電源投入時の値を決められるので以下の記述が有効 (でもリセット動作も書いておこう)
  - reg の宣言時に値を設定

```
reg [3:0] count = 4'b0001;
```

  - initial ブロックで値を設定

```
reg [3:0] count;
...
initial begin
  count <= 4'b0001;
  ...
```

よくある問題と対策 1/3

— コンパイルが通らない —

- コードはきれいに。整理整頓。ソフトと同じ。
- ソフトと違って、過度の抽象化は危険。あくまで RTL での記述。ブロック図をテキストで描くぐらいの意識で。
- エラーメッセージを上からちゃんと読む。警告も読む。
- どの場合に `wire`/`reg` を使うか原則を決める。
- 各 `wire`/`reg` をどの回路要素 (`assign`、`always`、下位モジュール) が駆動しているか把握する。複数の回路要素で駆動することはできない。いつも心に回路図を。

よくある問題と対策 2/3

— 思ってた回路と違う —

- 制御構文では全ての場合を網羅する。`if` では `else` も書く。`case` では `default` を書く。
  - 組合せ回路では、全ケースで同じセットに代入が必須。
  - 順序回路では、代入無しは「前の値を保持」の場合のみ。
- 順序回路は完全同期式で書く。センシティビティリストにはクロックと、必要ならリセットのみ。
- 論理合成レポートを見て、意図した通りのレジスタが推定されているかなどチェックする。
- 演算や代入のビット数はできるだけ明記して合わせる。0 拡張、符号拡張を自動に任せると間違いやすい。
- signed と unsigned を演算すると unsigned。10 進定数も 32-bit `unsigned`。signed の部分ビット参照も unsigned。

よくある問題と対策 3/3

— RTL シミュレーション、ゲートレベルシミュレーション、実機で動作が違う —

- 前ページの制御構文の注意を参照。特に `function` 内の記述に注意。
- 合成後はドントケアが 0/1 に確定する。実機に不定値は存在しない。
  - ドントケアの指定が適切か確認。
  - 不定値にマッチする条件を書いていないか確認。
  - シミュレーションで不定値が出ていないか確認。
- 動作クロックは適切か。巨大な演算器や、クリティカルパスに影響する逐次処理を生成していないか。

設計に向けて

- 設計は計画的に。とりあえず部品から書き殴っていってもゴミが溜まるだけ。トップダウンの視野を持って。ソフトと同じ。
- 初期段階から設計と検証のフローを考えること。設計が小さいうちにテストする。部品毎に検証、ダミー部品を入れたトップモジュールを検証。ソフトと同じ。
- ボードの表示系を活用し、実機検証、デバッグの手順を早い段階で確立すること。
- 教科書やネット上の情報は古かったり間違っていたりすることも多いので注意 (この資料も)。

本資料は高木一義先生が作成されました。
ここに深く感謝の意を表します。