論理設計の基礎

1　実習の目的

ここでは，ハードウェア記述言語を用いて論理回路を設計する手法について学び，実際に簡単な論理回路（組み合わせ回路，順序回路）を記述してみる．併せて，合成した回路を統合開発環境上で確認する方法や，実際にプログラム可能なハードウェアに転送して動作検証を行う手法についても学習する．

2　ハードウェア記述言語による論理設計

近年，幅広い分野で用いられている電子機器には，大規模半導体集積回路（LSI）が数多く搭載されるようになった．この LSI は回路規模が非常に大きくなってきているため，回路設計のすべてを人手で行うことは現実的ではなく，コンピュータによる支援が不可欠になってきている．コンピュータの性能が向上した 1980 年代以降，回路の動作を言語レベルで記述する**ハードウェア記述言語**（Hardware Description Language; HDL）が広く用いられるようになってきた．

HDL による回路設計の流れを図 1 に示す．HDL では，回路の動作をコンピュータプログラムのようにテキストファイルとして記述する．HDL で作成したハードウェア記述を**論理合成ツール**（logic synthesis tool）によってゲート回路に変換する．論理合成ツールは，設計者が定義した**設計制約条件**（design constraint）[*1] にしたがって，論理回路を最適化する．

HDL でのハードウェア記述がプログラミングにおけるソースコードとすれば，論理合成ツールはコンパイラに相当する．すなわち，コンピュータのプログラミングがハンドアセンブルから高級言語によるプログラミングに進歩したように，回路設計もより高いレベルで行えるようになってきている．

HDL の構文は C 言語などのプログラミング言語の構文とよく似ているが，多くのプログラミング言語が「プログラムの逐次的な動作」を記述するのに対し，HDL では並列動作を記述できるように考慮されている点が大きく異なる．この HDL により，論理ゲートをコンピュータ上で配置するといった旧来の設計手法（論理レベルでの**ゲートレベル設計**）よりも上位のレベルで設計を行うことができるようになった．HDL を用いた設計では，**レジスタ転送レベル**（Register Transfer Level; RTL）という抽象レベルでハードウェアを記述する．このレベルでは順序回路素子であるレジスタを明確に記述した上で，レジスタから次のレジスタへ，その間に存在する各種の組み合わせ回路を通して信号が伝達するという形で論理回路の動作を記述する．クロックに同期するレジスタは設計段階で確定させるため，信号の入力から出力までに要するクロックサイクル数は論理合成後も変化しない．このため，RTL で記述したハードウェアを実装前にシミュレーションできるという利点が得られる．しかし，RTL という一定の抽象レベルで書かれた論理回路を論理合成することになるため，通常のプログラミングと異なり，**言語の文法に沿ってさえいればあらゆる論理回路を合成できるわけではない．**ここが C 言語などを用いたプログラミングと大きく異なる点であり，HDL をプログラミング言語と考えてはいけない理由でもある．

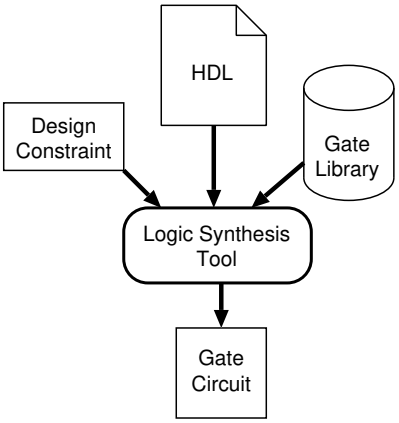

図 1　HDL による回路設計フロー

代表的な HDL には，プログラミング言語 Ada をもとに米国国防総省で開発された VHDL（VHSIC (Very High Speed Integrated Circuit) HDL）[1] や，論理シミュレータ Verilog 用の言語である Verilog HDL[2] などがある．これらは IEEE によって標準化されており，HDL のデファクトスタンダードとなっている．また，Altera 社[*2]が開発した AHDL（Altera HDL）[3] という HDL も存在する．これは，他の HDL と比べて効率的な論理合成が可能であると言われるが，現場でのシェアは大きくない．

本実習では C 言語と記述が似ている Verilog HDL

[*1] クロック周期，回路の面積，入出力のタイミングなど．

[*2] 2015 年に Intel 社が買収し，同社の FPGA 部門となったが，2024 年 2 月に FPGA 部門の新会社として独立した．

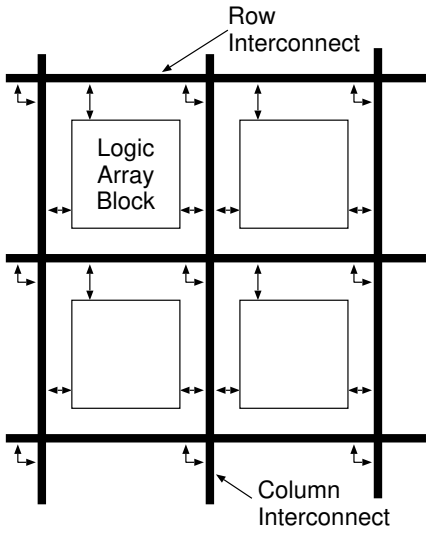

図 2　FPGA の基本構造（一部）

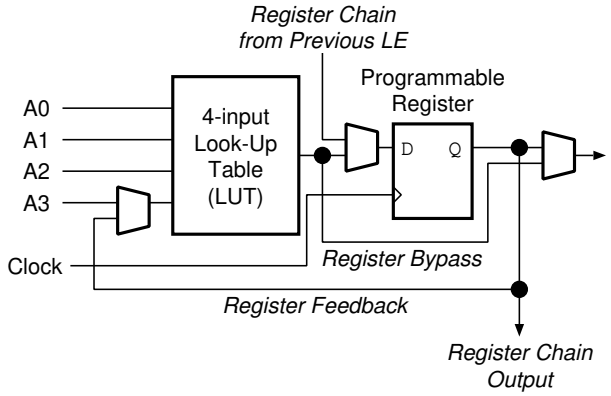

図 3　Cyclone II FPGA デバイスにおけるロジックエレメントの構造（概略）

を用いる．Verilog HDL の基本的な文法については，22 ページからの付録 A にまとめているので参考にしてほしい．

3　FPGA

FPGA は「現場（Field）でプログラム可能（Programmable）なゲートアレイ（Gate Array）」であり，回路構成を電気的に書き込むことができるデバイスである．

FPGA の基本構造を図 2 に示す．FPGA は，いくつかの**ロジックエレメント**（Logic Element; LE）からなるロジックアレイブロック（Logic Array Block; LAB）を格子状に配置したデバイスであり，各 LAB は行列方向のインタコネクト（interconnect; 配線部）と接続している．インタコネクトの接続経路はプログラム可能になっており，LAB 間の接続を自由に変更できる．

LE の内部は，図 3 に示すように，**ルックアップテーブル**（Look Up Table; LUT），プログラム可能レジスタ（programmable register）などで構成されている[*3]．LUT は小規模な RAM[*4]で実現されており，その内容はプログラム可能である．この LUT に，すべての入力パターンそれぞれに対応した出力値を設定しておく．例えば，4 入力 AND を構成したい場合は，入力 A0 から A3 がすべて 1 となる場合にのみ 1 を，それ以外の入力に対しては 0 を出力するように LUT のエントリを定義すればよい．「A0 · A1 + A2 · A3」の論理式で表される回路を構成したい場合は，表 1 のように LUT を定義することになる．この例のように，AND や OR といった基本論理ゲートを複数個用いた回路の動作であっても，LUT にはその結果だけを設定するため，基本論理ゲートと LE は 1 対 1 に対応しないことに注意が必要である．

表 1　論理式「A0 · A1 + A2 · A3」に対応する LUT エントリ

A0	A1	A2	A3	Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

このように，「FPGA をプログラムする」とは LE 内の LUT やプログラム可能レジスタを書き換えたり，インタコネクトの接続を変更することである．これにより回路の動作を自由に変更できるが，個々の LE は小規模であるため，複雑な回路を実現する際は，複数の LE を接続して使用しなければならない．

これ以降の実習では，様々な論理回路を Verilog HDL によって記述する．記述した論理回路は FPGA をプロ

[*3] 実習で用いる Cyclone V では，**ALM**（Adaptive Logic Module）と呼ばれる構成単位を用いている．1 個の ALM は 8 入力の分割可能 LUT と 2 個の全加算器，4 個のレジスタを備えており，より効率良く論理回路を収容できるようになっている．

[*4] 一般的な 4 入力 1 出力 LUT では $2^4 = 16$ bit.

グラムすることで，ハードウェアとしての動作確認が可能になる．実際の作業としては，HDL を記述し，論理合成してから FPGA ボードに転送，動作確認をすることになる．しかし，（似たような作業であっても）これは C 言語などのプログラムをコンパイルし，CPU ボードに転送してソフトウェアとして実行することとは意味がまったく異なることに注意が必要である．

4 実習室に備える FPGA ボード

4.1 概要

実習室に備える FPGA ボード（Terasic 社製 DE1-SoC）の外観図を図 4 に示す．このボードは，FPGA デバイス（Cyclone V 5CSEMA5F31C6N）の他に，7 セグメントディスプレイ 6 桁，赤色 LED 10 個，スライドスイッチ 10 個，プッシュスイッチ 4 個を備えており，入力信号の設定や出力信号の確認に利用できる．また，本実習では使用しないが，ビデオ入力端子（Video In），VGA 出力端子，音声入出力端子（Line In, Line Out），USB 端子，1Gbps 対応 Ethernet 端子，赤外線入出力装置（IrDA; IR-in, IR-out），PS/2 端子，Micro SD カードスロットなどを備えている．これらの入出力デバイスは FPGA に書き込んだ回路から制御できる．

DE1-SoC に搭載されている FPGA デバイス（Cyclone V SE 5CSEMA5 シリーズ）は，全体で 32,070 個の ALM（従来機種の LE 換算で約 8 万 5 千個相当）を搭載しており，4,450K ビットの組み込みメモリも内蔵している．

4.2 スイッチと LED による入出力方法

まず，FPGA ボード左下部のスライドスイッチと赤色 LED を接続し，スイッチをオンにすると対応する LED が点灯する回路を作成してみよう．

図 5 は，スイッチ（入力 `switch`）をオンにすると LED（出力 `led`）が点灯し，switch をオフにすると消灯する回路の Verilog HDL 記述である．

図 5 に示した `control_led` モジュールでは，入力 `switch` と出力 `led` が `assign` 文[*5]によって関連付けられている．しかし，この段階ではボード上のどのスイッチを実際に入力として使い，どの LED を点灯させるかについては定義していない．HDL はハードウェア動作の記述を行うものであるため，入出力の割り当てについては，別のツールで行う．この実習で使用している開発環境（Quartus Prime）では，**ピンプランナ**（Pin Planner）というツールで割り当てを行う．ピンプランナの使い方は 33 ページの付録 B.5 を参照せよ．

[*5] assign 文については，22 ページの付録 A.3.1 を参照せよ．

具体的なピン番号は使用する FPGA デバイスやボードによって異なるため，参考資料 [4] を参照し，スイッチと LED のピン番号をモジュールの入出力に適切に関連付けること．

例えば，右端のスイッチ（SW0）を入力として使い，そのオン／オフによって右端の赤色 LED（LEDR0）を点灯・消灯する場合を考える．信号名 SW[0] に対応する FPGA のピン番号は，参考資料 [4] の Table 3.6 より，PIN_AB12 と分かる．同様に，信号名 LEDR[0] に対応するピン番号は，PIN_V16 である．使用するピン番号が分かれば，ピンプランナを用いて，モジュールの入出力と FPGA のピン番号を対応させる．ピンの設定が終わればコンパイルを行い，FPGA ボードにダウンロードして動作確認を行う．

図 6 は 4 個の LED をスライドスイッチで制御するモジュールの例である．ベクタ表現（23 ページの付録 A.3.3 参照）を用いることで，`assign` 文を 4 個列挙していないことに注目せよ．

なお，DE1-SoC のプッシュスイッチ（KEY0〜KEY3）は，内部でシュミットトリガ（schmitt trigger）回路によりデバウンス（チャタリング対策）されているので，そのままクロック入力やリセット入力のようなエッジトリガ入力として使用できる．スイッチを押すと low（0V），離すと high（3.3V）になる．

5 組み合わせ回路の設計

図 7 のように，入力信号だけで出力信号が決まる論理回路を**組み合わせ回路**（combinatorial logic）という．組み合わせ回路は論理ゲートを組み合わせて作成することになるため，従来のゲートレベル設計では，設計者が論理ゲートを配置・接続することで設計を行っていた．HDL を用いた設計では，設計者は目的とする組み合わせ回路の「動作」を HDL によって記述するだけで，論理ゲートをどのように接続するかについては論理合成ツールが自動生成する[*6]．

なお，C プログラムと同様に，同じ目的に対しても複数の HDL 記述が考えられるため，以下に示す HDL 記述だけがすべてではない．

5.1 デコーダ

デコーダ（decoder）とは入力の組み合わせ（コード）にしたがって，対応する出力信号を生成する回路である．n 入力 $m(=2^n)$ 出力のデコーダを $n \times m$ デコーダという．表 2 は，2 ビットの入力を 4 ビットの出力

[*6] もちろん論理ゲートの接続関係を明示的に記述することも可能である．

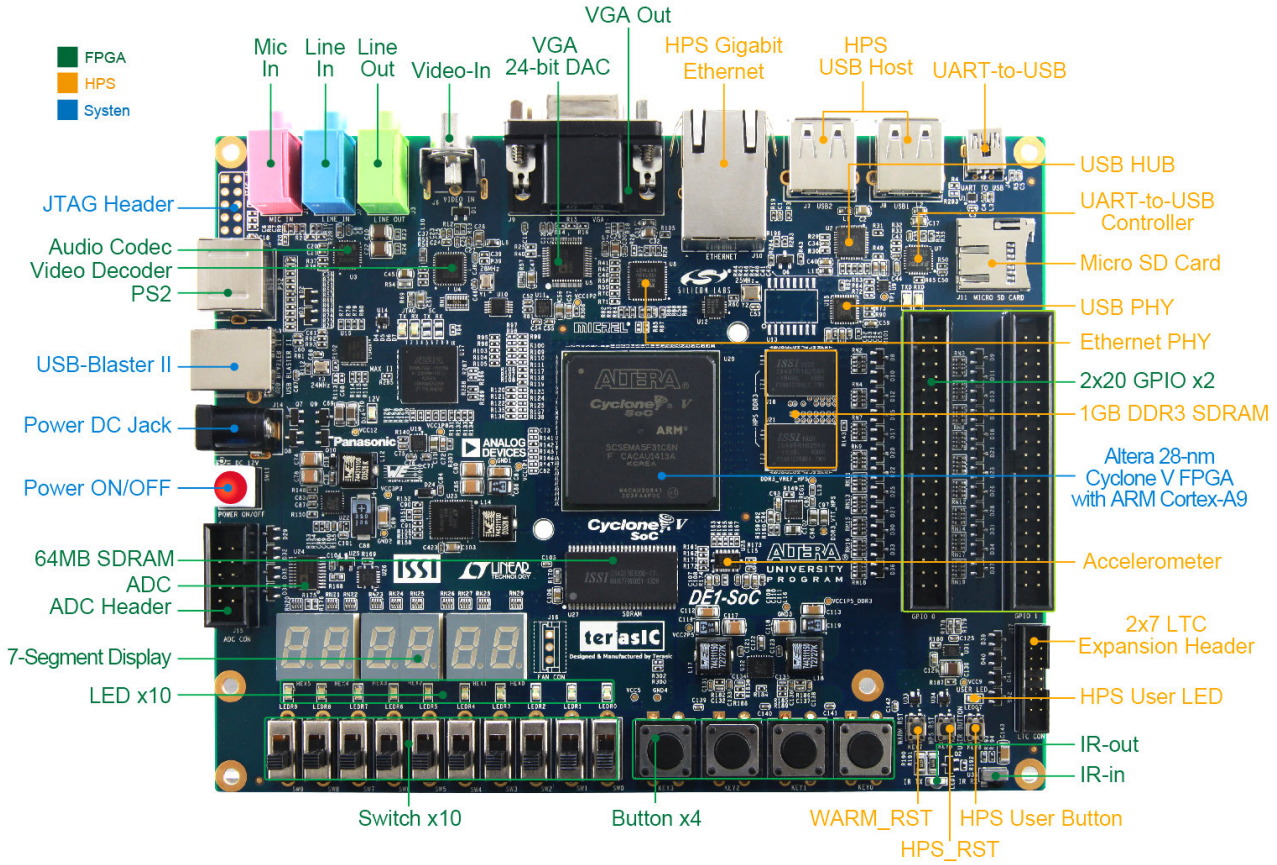

図 4　DE1-SoC FPGA ボード（Terasic の Web サイトより引用）

```
module control_led(switch, led);
  input switch;
  output led;

  assign led=switch;
endmodule
```

図 5　単一 LED を制御するモジュール例

```
module control_4leds(switch, led);
  input [3:0] switch;
  output [3:0] led;

  assign led=switch;
endmodule
```

図 6　4 個の LED を制御するモジュール例

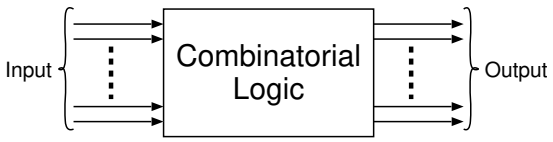

図 7　組み合わせ回路

表 2　2 × 4 デコーダの機能表

D_1	D_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

に複号する 2 × 4 デコーダの機能表であり，図 8 および図 10 は，その組み合わせ回路を Verilog HDL で記述したものである．

基本的に組み合わせ回路は，図 8 のように `assign` 文[*7]で記述できる．図 8 の記述では，Verilog HDL の条件演算子[*8]を用いている．条件演算子により，C 言語の 3 項演算子と同じように，条件を満たすか否かで実際に代入する値を変えることができる．

条件演算子を用いた記述では，条件数が増えるとカッ

[*7] 詳細は 22 ページの付録 A.3.1 を参照せよ．
[*8] 詳細は 26 ページの付録 A.8 を参照せよ．

```
module decoder_2x4(D, Q);
  input [1:0] D;
  output [3:0] Q;
  wire [3:0] Q;

  assign Q =
    (D == 2'b00) ? 4'b0001 : (
    (D == 2'b01) ? 4'b0010 : (
    (D == 2'b10) ? 4'b0100 : (
    (D == 2'b11) ? 4'b1000 : 4'bxxxx)));
endmodule
```

図 8　2 × 4 デコーダの Verilog HDL 記述例（条件演算子使用）

```
module decoder_2x4(D, Q);
  input [1:0] D;
  output [3:0] Q;
  wire [3:0] Q;

  assign Q = func_decoder(D);

  function [3:0] func_decoder;
    input [1:0] in;
    begin
      case (in)
        2'b00: func_decoder = 4'b0001;
        2'b01: func_decoder = 4'b0010;
        2'b10: func_decoder = 4'b0100;
        2'b11: func_decoder = 4'b1000;
        default: func_decoder = 4'bxxxx;
      endcase
    end
  endfunction
endmodule
```

図 9　2 × 4 デコーダの Verilog HDL 記述例（ファンクション使用）

コの段数が深くなり，コードの可読性が低下する．そこで，Verilog HDL のファンクション[*9]を用いて記述すると，図 9 のように書き直すことができる．ここでは，C 言語の `switch-case` 文のように，条件を `case` 文[*10]で列挙して記述している．

このように，HDL では明示的に論理ゲートの接続関係を記述するのではなく，実現したい機能を高いレベルで記述するだけでも回路を作成できる．

また，組み合わせ回路は `always` 文を用いても記述することができる．例えば，2 × 4 デコーダは図 10 の

[*9] 詳細は 29 ページの付録 A.12 を参照せよ．

[*10] `case` 文の構文については，27 ページの付録 A.10.2 を参照せよ．

```
module decoder_2x4(D, Q);
  input [1:0] D;
  output [3:0] Q;
  reg [3:0] Q;

  always @(D) begin
    case (D)
      2'b00: Q=4'b0001;
      2'b01: Q=4'b0010;
      2'b10: Q=4'b0100;
      2'b11: Q=4'b1000;
      default: Q=4'bxxxx;
    endcase
  end
endmodule
```

図 10　always 文による 2 × 4 デコーダの記述例

```
module decoder_2x4(D, Q);
  input [1:0] D;
  output [3:0] Q;
  reg [3:0] Q;

  always @(D) begin
    case (D)
      2'b00: Q=4'b0001;
      2'b01: Q=4'b0010;
      2'b10: Q=4'b0100;
    endcase
  end
endmodule
```

図 11　case 文の分岐式が不十分な例

ようにも記述できる．図 10 において，`always` に続く `@(D)` は「信号 D が変化した時点で，続く `begin`〜`end` 内のブロックを実行する」ことを示している．このイベント制御文（`@`）で指定されている信号のリストを**センシティビティリスト**（sensitivity list）と呼び，このリストに含まれるいずれかの信号が変化した場合に，文またはブロックが実行される．`always` 文によって組み合わせ回路を記述する場合は，関係するすべての信号をこのセンシティビティリストに列挙しておく必要があるので注意せよ．

図 10 では，信号 D が変化した時に `case` 文を実行し，D の値に対応する 4 ビット値を変数 Q に代入している．ここで，`case` 文中の分岐式に記述漏れがあった場合を考えてみよう．例えば図 11 のように，入力 D が 2'b11 の場合の分岐式が記述されておらず，`default` 文も存在しなかったとする．その場合，もし D として 2'b11 が入力されると，`case` 文中で変数 Q が更新され

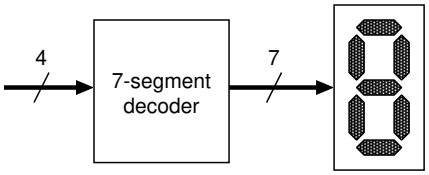

図 12　7 セグメントデコーダ

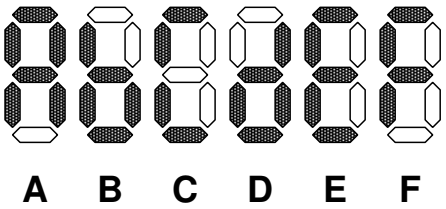

図 13　7 セグメントディスプレイにおける A から F の発光パターン

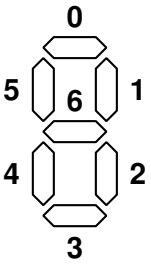

図 14　7 セグメントディスプレイにおける各セグメントの添字と発光位置

ないため，直前の出力値がそのまま出力されることになる．つまり，入力が 2'b11 である時の出力値を一意に決められない（直前の出力値に依存する）ということであり，これは「入力信号が決まれば出力信号が決まる」という組み合わせ回路の条件に反する．Verilog HDL でこうした不完全な記述をしていると，論理合成ツールは警告を表示した上で出力にラッチ（6.1.1 節で後述）を生成してしまう．自分では組み合わせ回路を記述したつもりでも，意図しないラッチが生成されることがあるので，条件の記述漏れには十分に注意せよ．同様の問題は，`if` 文（付録 A.10.1 参照）に対する `else` 節が存在しない場合にも起こりがちであるので，`always` 文で組み合わせ回路を記述する際には十分に注意すること．

[演習 1]　4 個のスライドスイッチで入力した 4 ビットの値を，7 セグメントディスプレイ 1 個に 16 進数 1 桁で表示する 7 セグメントデコーダモジュール（図 12）を Verilog HDL で記述せよ．また，記述した Verilog HDL モジュールの出力信号を FPGA ボードのどのピンに接続すれば良いか検討し，出力信号とピン名の対応を表でまとめよ．なお，A から F の発光パターンは図 13 に従うこと．図 14 中の数字は FPGA ボードにおける 7 セグメントディスプレイのピン名 HEX0_D[n] のインデックス n に相当する値である．例えば，HEX0_D[6] というピンは 7 セグメントディスプレイの中央の横線に対応する．

[ヒント] LED セグメントへの電圧レベルと発光状態の関係は，電圧レベル L で点灯，電圧レベル H で消灯とする．

5.2　マルチプレクサ

マルチプレクサ（multiplexer; MUX）とは，複数個の入力値から 1 個の値を選択して出力する回路である．出力値を選択することから**セレクタ**（selector）ともいう．図 15 は，1 ビットの選択信号 S の値によって 2 個の 1 ビット入力 X, Y を選択して出力する 2-to-1 マルチプレクサであり，表 3 はその機能表である．これを Verilog HDL で記述した例が図 16 である．図 16 では論理ゲートの組み合わせで記述しているが，「選択信号に応じた信号を選択し出力する」という動作に基づき，条件演算子を用いたり，`case` 文を用いても記述できる．

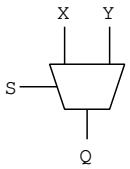

図 15　1 ビット 2-to-1 マルチプレクサ

表 3　1 ビット 2-to-1 マルチプレクサの機能表

S	Q
0	X
1	Y

図 17 は，スイッチと LED を用いて図 16 の動作を確認しやすくしたものである．ここで，モジュール `mux2to1_led` の下位モジュールとして，`mux_1bit_2to1` と `control_led` の 2 つを再利用している．HDL による回路設計では，このように小さな機能をモジュールとして実装し，それらを組み合わせて大きな回路を設計することが一般的である．モジュールの階層構造については 28 ページの付録 A.11.1 を参

```
module mux_1bit_2to1(X, Y, S, Q);
  input X, Y, S;
  output Q;

  assign Q=(~S & X) | (S & Y);
endmodule
```

図 16　1 ビット 2-to-1 マルチプレクサの Verilog HDL 記述例

```
module mux2to1_led(sw, led);
  input[2:0] sw;
  output led;
  wire w;

  mux_1bit_2to1 MUX(sw[2],sw[1],sw[0],w);
  control_led LED(w,led);
endmodule
```

図 17　1 ビット 2-to-1 マルチプレクサ

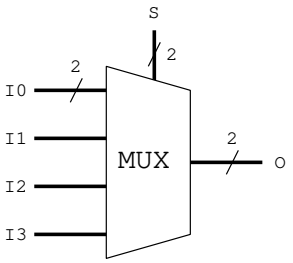

図 18　2 ビット 4-to-1 マルチプレクサ

照せよ．

[実習 1]　図 18 に示すように，各入力が 2 ビット幅の 4 入力（I0〜I3）を選択する 4-to-1 マルチプレクサのモジュールを Verilog HDL で記述し，機能シミュレーションにより動作を確認せよ．シミュレータに与える入力信号は，動作検証に適切なものを設定すること．シミュレータの使用方法は付録 C を参照せよ．選択信号 S を変化させるだけでなく，S を固定した上で，その S に対応する入力信号を変化させてみよ．

シミュレーション結果を報告する際には，入出力信号（信号波形および信号名）を確認できる画面でスクリーンショットを取り，その図を用いて本文中で説明を行うこと．これは以降の実習でも同様である．

6　順序回路の設計

順序回路（sequential logic）とは，ある時刻の出力

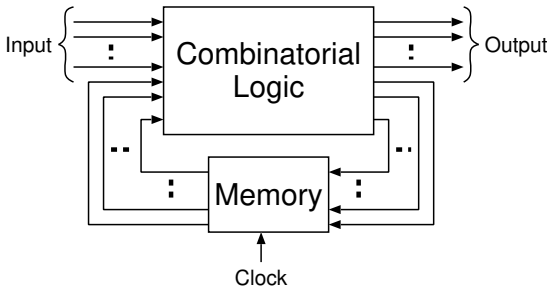

図 19　順序回路

がその時刻での入力と状態（入力履歴）に依存する論理回路である [5]．構成としては，図 19 のように，組み合わせ回路に現状態を記憶するメモリを付加したものになる．

順序回路において，動作のタイミングをとる信号を**クロック**（clock）と呼び，以降の実習で主に取り扱う**同期式順序回路**では，回路の動作がこのクロックに同期する．つまり，同期式順序回路ではクロック入力ごとに状態の変化が起こりうるため，クロックごとに順序回路の入力と出力を決定する．

6.1　順序回路における記憶素子

順序回路の状態を記憶するための機構には，ラッチ（latch）やフリップフロップ (flipflop; FF) がある．これらは論理値の “0” または “1” のいずれかを安定状態として保持する．詳しくは文献 [5] を参照せよ．

6.1.1　D フリップフロップ

ラッチはイネーブル信号が 1 である間，入力を通過させ，イネーブル信号が 0 になった時に，その時点での入力を記憶する素子である．図 20 はラッチを論理ゲートで構成した図である．このラッチの動作を Verilog HDL では図 21，または図 22 のように記述できる．図 21 は図 20 の接続を忠実に，論理式で記述した例である．一方，図 22 はラッチの動作を基にした記述例である．これは，論理ゲートの接続を具体的に指示することなく，「入力信号 D または EN が変化した時に，EN が 1 であれば出力 Q に D を設定する．EN が 0 であれば何もしない（以前の Q を保持する）」という，ラッチの動作をそのまま記述する書き方である．

ラッチでは，イネーブル信号が 1 の期間中に入力信号 D が変化すると，D の変化に連動して即座に出力が変化してしまう．例えば，図 23 において，Clock をイネーブル信号とみなすと，入力 D が変化することで，Q1 のような出力が観測される[*11]．実用上，このまま

[*11] 図 23 では素子内部での遅延時間は考慮していないことに注意すること．

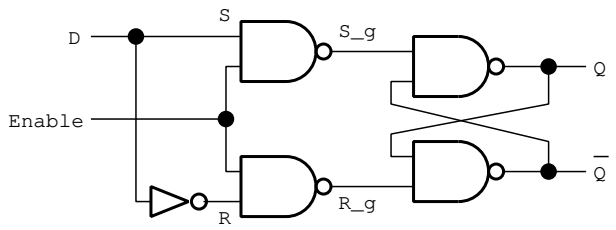

図 20　ラッチの回路構成

```
module d_latch (EN, D, Q);
  input EN, D;
  output Q;
  wire R, S_g, R_g, Q1, Q2;

  assign S_g = ~(D & EN);
  assign R_g = ~(R & EN);
  assign R = ~D;
  assign Q1 = ~(S_g & Q2);
  assign Q2 = ~(R_g & Q1);

  assign Q=Q1;
endmodule
```

図 21　論理式によるラッチの Verilog HDL 記述

```
module d_latch (EN, D, Q);
  input EN, D;
  output Q;
  reg Q;

  always @(EN or D)
    if (EN)
      Q <= D;
endmodule
```

図 22　ラッチの動作に基づく Verilog HDL 記述

では不便であるため，ラッチを 2 つ接続して，図 24 のようにマスタスレーブ型 D-FF を構成する．図 24 では，マスタ側とスレーブ側へのイネーブル信号は NOT 素子により反転しているため，Clock が 1 の期間に入力 D が変化しても，（その期間中はマスタ側では EN=0 になるため）即座に出力 Q を変化させることはない．このように回路を構成することで，図 23 における出力 Q2 のように，Clock が立ち上がる瞬間だけで出力を確定することができる．このように動作する D-FF は，Verilog HDL では図 25 のように記述できる[*12]．図 25 の `always` 文では，イベント制御文として `posedge` を

[*12] `dff` は Quartus Prime のプリミティブとして予約されているため，モジュール名を `d_ff` としている．

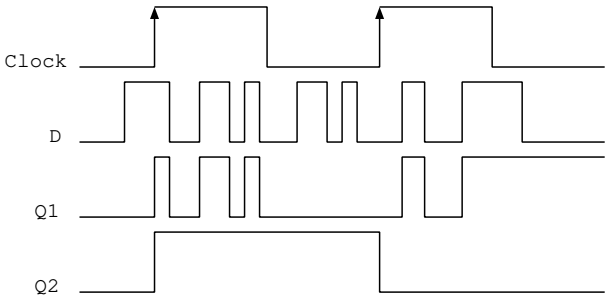

図 23　D-FF のタイミング図

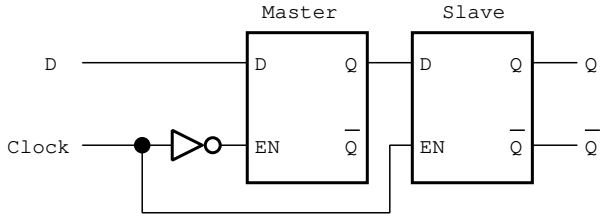

図 24　マスタスレーブ型 D-FF

```
module d_ff (Clk, D, Q);
  input Clk, D;
  output Q;
  reg Q;

  always @(posedge Clk)
    Q <= D;
endmodule
```

図 25　D-FF の Verilog HDL 記述

使っている．これにより，信号 Clk の立ち上がりエッジ（positive edge）のタイミングで，続く代入文を実行できる．なお，立ち下がりエッジ（negative edge）を指定したい場合は，イベント制御文に `negedge` を使用する．

[実習 2]　ラッチを 2 つ接続することで，図 24 のマスタスレーブ型 D-FF を Verilog HDL で記述し，機能シミュレーションで動作を確認せよ．

[ヒント]　ラッチのモジュールを 1 つ定義した上で，2 つの実体（`master_latch` と `slave_latch`）を作成し，それらを適切に接続すればよい．図 23 の Q2 のような出力が得られるかどうかを確認すること．

ただし，以降の実習で D-FF を作成する場合は，実習 2 のようなマスタスレーブ型 D-FF として記述しなくても，図 25 のように記述すればよい．図 25 は「クロック信号（Clk）の立ち上がり（posedge 時）に，入力 D を変数 Q に取り込む」という D-FF の動作を基に

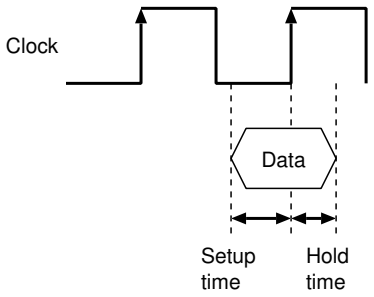

図 26　セットアップ時間とホールド時間

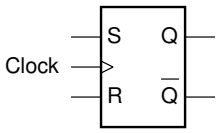

図 27　SR フリップフロップのブロック図

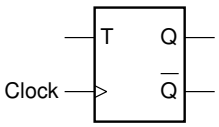

図 28　T フリップフロップのブロック図

記述したものである．

このように，ラッチと異なり，D-FF ではクロック信号の立ち上がりのタイミングでデータを取り込むが，データ信号の与え方には注意が必要である．具体的には，図 26 に示すように，クロック信号の立ち上がりの前後で，データを一定時間安定させる必要がある．クロック信号の立ち上がり前にデータを保持しておかなければならない時間を**セットアップ時間**（setup time），クロック信号の立ち上がり後にデータを保持しておかなければならない時間を**ホールド時間**（hold time）と呼ぶ．これらの時間が満たされていないと，データの変化前か変化後のどちらが読み込まれるか保証できないので，安定した動作が期待できないことになる．

なお，リセット信号などの非同期制御信号に対する用語として，セットアップ時間には**リカバリ時間**（recovery time），ホールド時間には**リムーバル時間**（removal time）がそれぞれ対応している．

6.1.2　その他のフリップフロップ

D-FF 以外にも，SR-FF や T-FF，JK-FF のフリップフロップが存在する．それらのブロック図を図 27，図 28，図 29 に，機能表を表 4，表 5，表 6 にそれぞれ示す．

[演習 2]　T-FF および JK-FF を Verilog HDL で記述せよ．

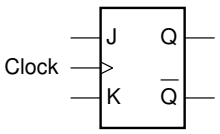

図 29　JK フリップフロップのブロック図

表 4　SR フリップフロップの機能表

S	R	$Q(t+1)$	$\overline{Q(t+1)}$
0	0	$Q(t)$	$\overline{Q(t)}$
0	1	0	1
1	0	1	0
1	1	–	–

表 5　T フリップフロップの機能表

T	$Q(t+1)$	$\overline{Q(t+1)}$
0	$Q(t)$	$\overline{Q(t)}$
1	$\overline{Q(t)}$	$Q(t)$

表 6　JK フリップフロップの機能表

J	K	$Q(t+1)$	$\overline{Q(t+1)}$
0	0	$Q(t)$	$\overline{Q(t)}$
0	1	0	1
1	0	1	0
1	1	$\overline{Q(t)}$	$Q(t)$

6.2　有限状態機械

有限状態機械（Finite State Machine; FSM）は内部に有限個の状態（state）を持ち，その上で状態遷移規則を定義した機構である．FSM の一種として，有限オートマトン（Finite Automaton; FA）や順序機械がある．詳しくは文献 [5] を参照せよ．

図 30 は，クロックごとに入力される信号列が 11 を含む時に 1 を，11 を含まない時に 0 を出力する FSM の状態遷移図（state diagram）である．この FSM には S0, S1, S2 の 3 状態があり，各状態でそれぞれ 0, 0, 1 を出力する（円内の/以降がその状態での出力値を示す）．

この FSM を Verilog HDL で記述すると，図 31 のように書くことができる．図 31 では，`assign` 文により，現状態が S2 にあるときに 1 を出力し，それ以外の状態にあるときは 0 を出力するようにしている．また，現状態 `cur_st` と入力 `I` により遷移先状態 `next_st` を決定し，クロック信号 `Clk` の立ち上がりでその状態に

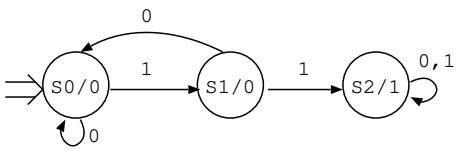

図 30　入力に 11 を含む場合に 1 を出力する FSM の状態遷移図

```
module state_machine (Clk, I, Resetn, Q);
  input Clk, I, Resetn;
  output Q;

  parameter S0=2'b00,
            S1=2'b01,
            S2=2'b10;
  reg [1:0] cur_st, next_st;

  assign Q = (cur_st == S2) ? 1'b1
                            : 1'b0;

  always @(posedge Clk)
    if (!Resetn)
      cur_st <= S0;
    else
      cur_st <= next_st;

  always @(cur_st or I)
    case (cur_st)
      S0: next_st = (I) ? S1 : S0;
      S1: next_st = (I) ? S2 : S0;
      S2: next_st = S2;
      default: next_st=2'bxx;
    endcase
endmodule
```

図 31　図 30 に示した有限状態機械の Verilog HDL 記述例

遷移する．同期リセット信号（Resetn）を 0 にすることで，初期状態（図 30 で二重矢印 ⇒ で指された状態 S0）へ遷移させている．

[実習 3]　300 円のチケットを発券する自動販売機を有限状態機械として設計し，Verilog HDL で記述せよ．FPGA ボード上で動作を確認すること．入力として 50 円硬貨（値 0）と 100 円硬貨（値 1）のみを受け付け，出力はチケットと釣り銭に相当する LED をそれぞれ点灯するものとする．具体的には，投入した金額が 300 円に達した時点でチケットを発券（LEDR[0] を点灯）し，合計で 350 円投入した場合は，チケットと 50 円の釣り銭を出力（LEDR[0] と LEDR[1] を点灯）するように設計せよ．例えば，入力系列として 1100 を与えた場合（合計 300 円を投入した場合）は LEDR[0] のみ点灯し，1101 を与えた場合（合計 350 円を投入した場合）は LEDR[0] と LEDR[1] の両方を点灯させること．チケットや釣り銭の出力後も，継続して入力を受け付けられるようにせよ．レポートでは設計した回路の状態遷移図を示し，State Machine Viewer で表示した状態遷移図と比較せよ[*13]．

[ヒント] 実際の動作の流れとしては，入力値（0 または 1）を設定してからクロック信号を与える（FPGA ボードでは，入力値をスライドスイッチ，クロック信号をプッシュスイッチで与える）ことを繰り返すことになる．

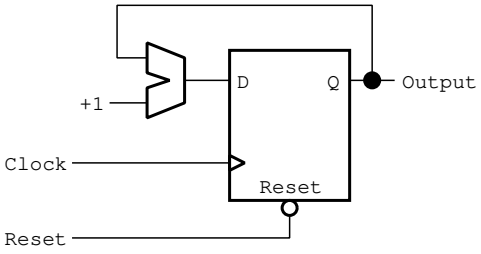

図 32　カウンタの構造

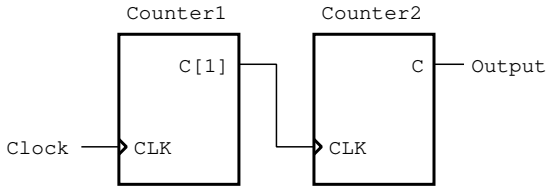

図 33　非同期カウンタの例

6.3　カウンタ

カウンタ（counter; 計数器）はクロックに同期して値を加算（または減算）していく回路である．現在のカウント値を記憶する D-FF と，クロックごとに 1 加算する加算器を組み合わせて図 32 のように構成する[*14]．クロックの立ち上がりごとに現在のカウント値をインクリメントし，新しいカウント値を D-FF に記憶する．

カウンタは値を数えるといった通常の用途の他，クロック周波数を落とす目的や，回路内部の状態を記憶するためにも用いられる．

[*13] State Machine Viewer の使い方は，34 ページの付録 B.6.2 を参照せよ．

[*14] ブロック図の端子に ○ が付いている場合は，それが負論理で機能することを示す

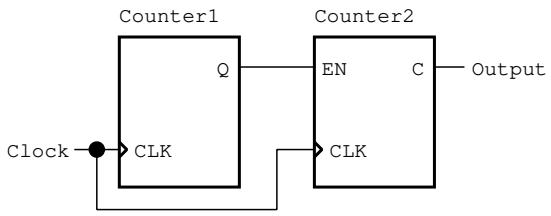

図 34　同期カウンタの例

クロックの周波数を落とす[*15]目的でカウンタを使用する場合は，クロック信号の扱いに注意する必要がある．図 33 では，2 ビットカウンタモジュールである Counter1 が高速なクロック信号 (Clock) をカウントし，カウント値の最上位ビット C[1] を出力することで分周している．後段にあるカウンタ（Counter2）は，Counter1 からの出力をクロック信号として入力し，本来のクロック信号 Clock よりも長い間隔でカウントしている．以下に具体例を示す．Counter1 はクロック入力（Clock）の立ち上がりごとに内部のカウント値をインクリメントする．カウント値の最上位ビット C[1] は，元のクロック周期の 4 倍の周期で変化するため，Counter2 のクロック入力ピン（CLK）には信号 Clock の 1/4 の周波数の信号が入力されることになる（「信号 Clock を分周比 4 で分周している」）．このような，各フリップフロップのクロック信号が共通ではない回路を**非同期回路**という．図 33 のような非同期カウンタでは，カウンタの段数を増やすとそれぞれのカウンタで生じた遅延時間が累積し，最初段のカウンタと最後段のカウンタでは大きな時間差が生じることがある．

これに対し，共通のクロック信号ですべてのフリップフロップを駆動する回路を**単相同期回路**という．論理合成ツールでは，入出力段にあるフリップフロップの間で遅延解析を行い，回路の最適化を行うため，各フリップフロップは共通のクロック信号で駆動するように（単相同期回路として）設計する必要がある．

図 34 は同期カウンタの例である．2 つのカウンタ（Counter1, Counter2）に共通のクロック信号（Clock）を与えて，同じタイミングで動作させている点が図 33 との違いである．後段のカウンタ（Counter2）は「Enable 付きカウンタ」であり，Enable 入力が 1 の時に限り，クロック入力の立ち上がりでインクリメントを行う．つまり，Counter2 がインクリメントするかどうかを，Counter1 の出力 Q が制御している．例えば，Counter1 のカウント値が 99 になっているタイミングで出力 Q を 1 にするとする．Counter1 は次のクロック入力でカウント値を 0 に戻し，インクリメントを継続する．こうすることで，Counter1 のカウント値が 99 になっている時（クロックの立ち上がり 100 回につき 1 回）だけ，Counter2 の Enable 入力（EN）が 1 になる．結果的に，Counter2 ではクロック信号（Clock）の立ち上がり 100 回ごとにインクリメントを 1 回だけ行うことになり，Clock 信号の 1/100 の周波数でカウントしていることになる．

[*15] 周波数を $1/n$ にすることを，「分周比 n で**分周**する」という．

```
module binary_counter(Clock, Q);
  input Clock;
  output [3:0] Q;
  wire [3:0] Q;
  reg [3:0] Count;

  assign Q = Count;
  always @(posedge Clock)
  begin
      Count <= Count + 1'b1;
  end
endmodule
```

図 35　4 ビットカウンタのモジュール例

Verilog HDL では，カウント動作を算術演算子によって記述できる（カウント動作に基づく記述）．図 35 は，Clock 信号の立ち上がりのタイミングで変数 `Count` をインクリメントするカウンタの記述例である．

[実習 4]　PWM (Pulse Width Modulation) 制御により，赤色 LED の輝度を調節する回路を作成せよ．輝度はスライドスイッチ SW0 から SW7 の 8 ビット値で指定できるようにし，指定された輝度で FPGA ボード上の赤色 LED を発光させること．回路は Verilog HDL で記述し，FPGA ボード上で実際に輝度が変更できることを確認せよ．レポートには RTL Viewer の結果も示すこと．

[ヒント]　LED の輝度は，LED に与えるパルスのデューティ比（duty cycle; パルス周期に対するパルス幅の割合）を変化させることで調節できる（ON の時間を長くするほど明るくなる）．例えば，図 36 のように，LED を ON にする時間を長くすれば LED の輝度を明るくできる．具体的には，FPGA ボードが供給する 50MHz クロック信号を利用して 8bit カウンタを作成し，カウンタの値がスライドスイッチで指定された輝度値に達していない時のみ LED を光らせればよい．

図 36　デューティ比の変化による輝度の変更

参考文献

[1] IEEE Computer Society: “IEEE Standard VHDL Language Reference Manual,” IEEE Std 1076-2008, Jan. 2009.

[2] IEEE Computer Society: “IEEE Standard for Verilog Hardware Description Language,” IEEE Std 1364-2005, Apr. 2006.

[3] Altera Corporation: “Altera MAX+PLUS II AHDL,” Oct. 2011.

[4] Terasic Technologies Inc.: “DE1-SoC User Manual V2.0.4,” Chapter 3. Using the DE1-SoC Board, 2019.

[5] 柴山 潔：“コンピュータサイエンスで学ぶ論理回路とその設計,” 第 3 章「同期式順序回路とその設計」，近代科学社，1999.

演算回路

7　実習の目的

本実習では，算術演算を行うための論理回路について学ぶ．一般に，コンピュータ内では数を 2 進数で表現するため，2 進数に対する算術演算が行われる．論理回路の観点からこの算術演算についてながめると，その基本は 1 ビットの加算回路（加算器）にある．本実習では，加減算器および乗算器を設計する．

8　全加算器

2 つの 2 進数の和を計算する回路を**半加算器**（half adder）と呼ぶ．さらに，下の桁からの桁上げ（キャリ; carry）を考慮して 2 つの 2 進数の和を計算する回路を**全加算器**（full adder; FA）と呼ぶ．

表 7 に 1 ビットの全加算器の真理値表を示す．A と B が被加数と加数の入力で，C_{in} が下の桁からの桁上げ入力（キャリイン; carry-in）を表す．また，S はその桁の和の値であり，C_{out} は上の桁への桁上げ出力（キャリアウト; carry-out）を表す．この全加算器は組み合わせ回路で構成することができる．

[演習 1]　1 ビットの全加算器を論理ゲートを用いて構成せよ．表 7 の出力変数を入力変数の論理式として示し，回路図も示せ．結果だけでなく，導出過程も説明すること．

表 7　1 ビット全加算器の真理値表

入力			出力	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

9　補数と減算

2 進数で負の数を表現する方式にはいくつかの方式があるが，一般には**2 の補数表現**（two's complement）が用いられる．例えば，3 を 3 ビットの 2 進数で表現すると [011] であるが，この左端に符号を表すビットを 1 ビット付加して [0011] と表現する．符号ビットは，値が 0 または正のとき 0 とし，負のときには 1 と定める．次に，-3 がどのように表現されるかをみてみよう．2 の補数表現では，符号反転は，2 の補数表現で表現された 2 進数の各ビットを反転し，その最下位桁に 1 を加えることで得ることができる．例えば，[0011] の各桁を反転すると [1100] となり，これに 1 を加えると [1101] となる．つまり，-3 の 2 進数表現は [1101] であり，確かに符号ビットの値は 1 となっている．逆に，-3 の符号を反転して 3 となる様子をみてみよう．[1101] の各ビットを反転すると [0010] となり，これに 1 を加えると [0011] となる．

ここで注目すべき点は，元の値が正か負かにかかわらず，同一の手続きで符号の反転が行えるという点である．そして特に重要なのは，この手続きの中で符号ビットを特別扱いしていない点である．補数表現を採用する利点はこの点にあり，この特徴は加算を行う場合にも当てはまる．例えば 7 と -4 の加算を考えよう．7 は [0111]，-4 は [1100] とそれぞれ表される．これらの値を符号付き整数，つまり 2 の補数表現による値とみなさずに，符号なし整数，すなわち全 4 ビットで正の整数を表しているものとみなすことにしよう．符号なし整数としてみた場合，[0111] はやはり 7 であるが，[1100] は 12 である．そこで [0111] と [1100] を加算すると結果は [10011] となり，10 進数における加算 7+12=19 と一致する．今は 4 ビットで数を表現することにしているので，最上位桁からの桁上げを無視して，[0011] を結果として残すことにすると，[0011] は 3 を表しており，つまり，7+(−4)=3 が求まったことになる．

このように，2 の補数表現で表現された 2 つの値を加算する場合には，それらを符号なしの整数とみなして加算を行い，最上桁からの桁上げが生じても無視すればよいことがわかる[*16]．ここでも符号ビットは特別扱いされていない点に注意されたい．演算回路を設計する立場からみると，符号なしの整数の加算とみなして回路を設計してよいことになる．そして，負の数の

[*16] この事実についてより数学的に理解したければ，文献 [1] および [2] を参照せよ．

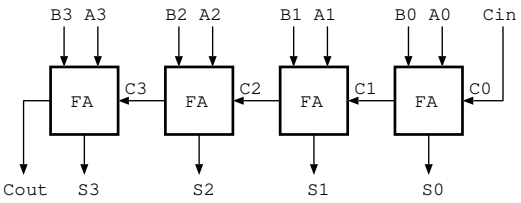

図 37　4bit リプルキャリ加算器

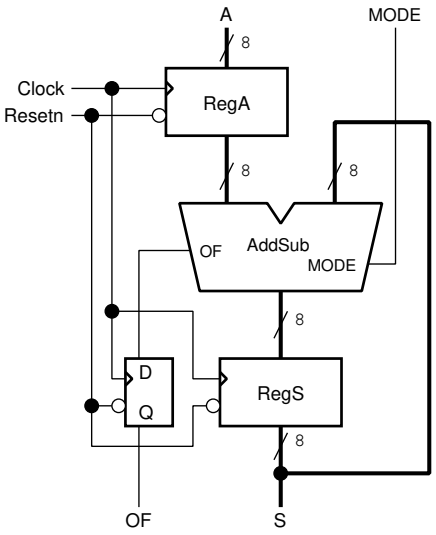

図 38　8 ビット符号付き加減算器

加算が可能であることから，減算は，被減数と減数の補数とを加算することで実現できることもわかる．

10　加減算器の設計

図 37 に 4bit リプルキャリ加算器の回路を示す．この回路では 2 個の 4 ビット値 A ($[A_3A_2A_1A_0]$) と B ($[B_3B_2B_1B_0]$) を加算し，和 S ($[S_3S_2S_1S_0]$) を求めている．この方式では，下位ビット（図の右側）から全加算器（図中では FA と表記）で加算を行い，桁上げ（キャリ）が生じた場合は上位ビットに伝達する．キャリが波（リプル; ripple）のように伝搬することから**リプルキャリ方式**と呼ばれる．

図 38 の AddSub は 8 ビット符号付き加減算器を示している．2 の補数表現で表される 8 ビット値 A を入力し，前回の結果に A を加減算した結果を S として出力する．AddSub は 1 ビットの命令入力信号 MODE を持ち, MODE=0 のときは加算（$S+A$）を, MODE=1 のときは減算（$S-A$）を行うものとする．1 ビット出力 OF は**オーバフロー**（overflow；桁あふれ[*17]）を表

[*17] 演算結果が規定のビット数に納まりきらない場合をオーバフローと呼び，通常はエラー扱いとする．キャリ（桁上げ）とは異なることに注意すること．例えば，4 ビットの符号付き整数において，4+6 の結果 10 は 4 ビット符号付き整数として扱える数の範囲（-8〜7）を超えているのでオーバフローとなる．

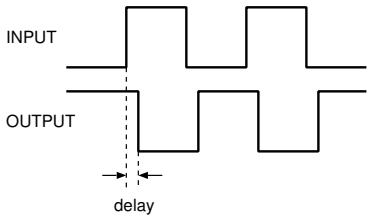

図 39　NOT 素子における信号遅延

しており，S が正しい符号付き値ではない場合に 1 にする．この AddSub 部は組み合わせ回路であり，クロック入力を取らないことに注意せよ．

図 38 では，A の値を一度 8 ビット幅のレジスタ[*18]RegA に保持してから，AddSub 部に供給している．同様に，AddSub 部からの出力をレジスタ RegS で受けている．これらのレジスタにはリセット入力を接続し，任意のタイミングで 0 にできるようにしておく．

[実習 1]　図 38 の 8 ビット符号付き加減算器をリプルキャリ方式により Verilog HDL で記述せよ．動作確認は機能シミュレーションにより行い，正数および負数の加減算が正常に行えることを確認すること．また，オーバフロー出力 OF が適切に変化することもチェックせよ．

[ヒント] 今回, レジスタと D-FF の構造上の差異は扱うビット幅だけなので，Verilog HDL の `parameter`（29 ページの付録 A.11.2 を参照せよ）を用いると簡潔に記述できる．

11　信号遅延の問題

これまでの実習では，信号の遅延（delay）について特に考慮していなかったが，実際の回路では AND, OR, NOT などの回路素子を通過するごとに遅延が生じる[*19]．図 39 は NOT 素子における信号遅延の例を示しており，入力が High になってから一定の遅延時間の後に出力が Low に反転していることがわかる．

回路内部において，レジスタ間を伝搬する信号の経路（パス; path）のうち，遅延時間が最も大きい経路を**クリティカルパス**（critical path）という．クリティ

[*18] クロック信号の立ち上がりのタイミングでデータを取り込む機構.

[*19] 厳密には論理ゲートを接続する信号線においても遅延（配線遅延）が存在する.

カルパスにおける遅延時間[*20]がクロック周期を下回れば，そのクロック周期で回路は正常に動作することになる．逆に遅延時間が上回ると誤動作が生じる可能性があるため，設計時には十分な**タイミング解析**を行う必要がある．

FPGA の場合，個々の LE では実現できる機能に限りがあるため，単純な回路を除けば，通常は複数の LE を組み合わせて目的とする回路を構成することになる．ここで，LE を縦続接続した場合，LE での遅延時間が積算されるだけでなく，LAB に含まれる個数以上の LE を組み合わせると，別の LAB との接続が必要になるため，配線遅延の影響がさらに大きくなってしまう．このように，FPGA での論理設計においては，論理合成しようとする回路が，実際にはどのようにデバイス上に展開されるかを意識しておくことが重要である．

FPGA 開発ツールでは，回路の設計者が指示する「タイミングに関する設計要求（**タイミング制約**; timing constraint）」を用いて，最終的な回路の最適化を行う．タイミング解析の結果，タイミング制約の範囲内であれば，回路は設計要求を満たしていると判断できる．逆に，タイミング制約を満たさなければ，設計要求を満たしていない（要求されたタイミングで動作しない）ことになるため，回路構成を変更するなどの修正をしなければならない．

[実習 2]　Timing Analyzer を用い，実習 1 で作成した回路が動作可能な最大周波数 f_{max} を調べよ[*21]．コンパイル後に Timing Analyzer のレポートを参照し，指定した動作要求周波数においてセットアップ時間とホールド時間が十分に確保されていることを確認せよ．また，回路のどの部分がクリティカルパスになっているかを調べよ．確認方法は付録 B.8 を参照せよ．

[演習 2]　論理回路一般において，最大動作周波数を高める工夫として考えられる手法を調べよ．

12　乗算器の設計

ここでは 4 ビット乗算器を設計しよう．乗算回路には組み合わせ回路だけで構成する方法もあるが，ここでは順序回路によって部分積を累算していくアルゴリズムを採る．まず，簡単のため，乗数と被乗数はどちらも 4 ビット符号なし数（すなわち，4 ビットすべてを用いて 0 または正の整数を表現する）とする．被乗数を $X = [x_3x_2x_1x_0]$，乗数を $Y = [y_3y_2y_1y_0]$ とし，乗算結果を 8 ビット符号なし整数 M に格納するものとする．このとき，積 M を求める式は以下のように変形できる．

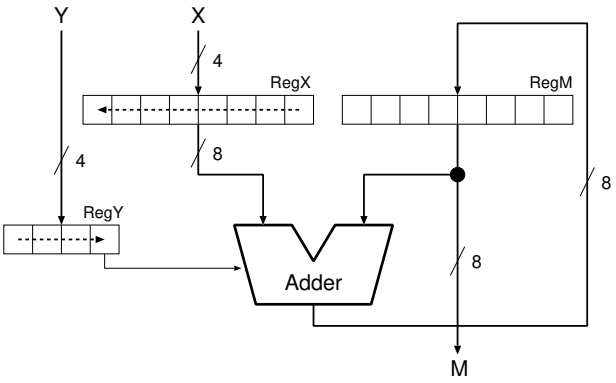

図 40　4 ビット乗算器のブロック図

$$
\begin{aligned}
M &= X \cdot Y \\
&= X \cdot (y_3 \cdot 2^3 + y_2 \cdot 2^2 + y_1 \cdot 2^1 + y_0 \cdot 2^0) \\
&= y_3 \cdot (X \cdot 2^3) + y_2 \cdot (X \cdot 2^2) + y_1 \cdot (X \cdot 2^1) \\
&\quad + y_0 \cdot (X \cdot 2^0)
\end{aligned}
$$

この式において，$X \cdot 2^i$ は X を i ビットだけ左シフトした部分積に相当する．また，y_i のビットは 0 または 1 のいずれかである．$y_i = 0$ であれば，当然 $y_i \cdot (X \cdot 2^i)$ は 0 になるため，無視できる．$y_i = 1$ であれば，X を i ビットだけ左シフトした値（つまり $X \cdot 2^i$）を M に加算すればよい．このようにして乗算を行う回路のブロック図を図 40 に示す．

X の値は乗算の途中に i ビットだけ左シフトする必要があるため，X レジスタは 8 ビット幅で構成する．Y レジスタ（RegY）の最下位ビット（y_0）の値を参照し，1 の場合は X レジスタ（RegX）と M レジスタ（RegM）の加算を行う．加算結果は M レジスタに格納する．

最初に，$y_0 \cdot (X \cdot 2^0)$ の部分積が求まれば，Y レジスタを 1 ビット右シフト，X レジスタを 1 ビット左シフトする．すると，Y レジスタは $[0y_3y_2y_1]$ となり，最下位ビットが y_1 に変化する．X レジスタの内容も $[000x_3x_2x_1x_0]$ になり，これは $X \cdot 2^1$ の値である．あとは同様に，Y レジスタの最下位ビット y_1 が 1 であれば X レジスタの値を M レジスタの値に加算すればよい．以降は，この手順の繰り返しである．

図 41 に，$X = [1010]$, $Y = [1001]$ とした場合の計

[*20] この遅延時間には，組み合わせ回路の論理ゲートで生じる遅延だけでなく，前段のレジスタにおけるクロック入力から出力までの遅延時間と後段レジスタに要するセットアップ時間を含めておく必要がある．

[*21] Timing Analyzer を用いて最大動作周波数を確認する方法は，35 ページの付録 B.7 および B.8 を参照せよ．

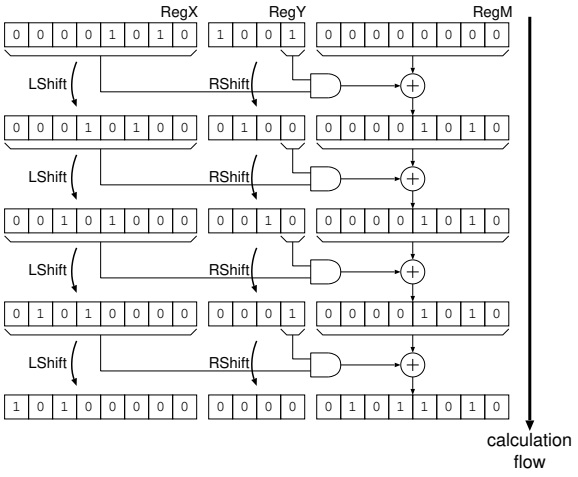

図 41　乗算の計算過程

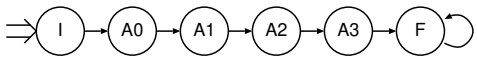

図 42　4 ビット乗算器の状態遷移

表 8　乗算器の状態コード割り当て

状態名	状態コード
I	000
A_0	001
A_1	010
A_2	011
A_3	100
F	101

算過程を示す．部分積を 4 回加算することで，最終的に積 [01011010] が M レジスタに格納されていることがわかる．

この乗算器の動作を見ると，まず各レジスタを初期化し，次に部分積和を計算することがわかる．初期化では，X レジスタおよび Y レジスタに被乗数 X と乗数 Y をそれぞれ読み込み，M レジスタを 0 にすればよい．

部分積和の計算は 4 回行うことになる．また，次の計算の準備として Y レジスタの右シフトと X レジスタの左シフトを行う．

以上の手順をそれぞれ状態 $I, A_i (0 \leq i \leq 3)$ とする．状態 A_3 で結果が求まった後は，その結果を出力し続けたいので，そのための状態 F も用意する．以上をまとめると，図 42 に示すように状態を遷移させればよい．各状態を識別するために，表 8 に示すような状態コードを割り当てる．

次は，図 40 の各部分が図 42 の各状態でどのような動作をすべきかを記述すればよい．

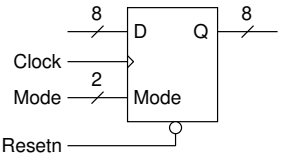

図 43　モード入力信号をもつレジスタのブロック図

表 9　レジスタの動作コード割り当て

動作	コード
ロード（Load）	00
右シフト（RShift）	01
左シフト（LShift）	10
ホールド（Hold）	11

まず，図 40 の X レジスタと Y レジスタの動作を考えよう．これらのレジスタでは，状態 I の時に被乗数または乗数をロード，状態 A_i の時にシフトし，状態 F の時は値を保持（ホールド）すればよい．ただし，X レジスタに被乗数を取り込む際は，4 ビット値 X の上位 4 ビットに 0 を補う必要があることに注意せよ．

最後に，M レジスタを設計して乗算器の完成である．M レジスタでは，状態 I でリセット（値 0 をロード），状態 A_i でロード，状態 F でホールドをそれぞれ行う．

ここで，各レジスタに対して表 8 の状態コードを渡し，レジスタ側でその状態に応じた動作を行うように設計してしまうと，状態コードに依存したレジスタモジュールになってしまい，Verilog コードの再利用が難しくなる．そこで，レジスタに状態コードを渡すのではなく，動作（ロード／ホールド／左右シフト）の種類を渡すようにすれば，状態コードに依存しないモジュール構成にできる．この場合はメインモジュール内でその状態で行う動作を決定し，各レジスタに指示すればよい．例えば，図 43 のような 2 ビットのモード入力信号をもつレジスタを構成する．Mode 入力に対しては，行う動作にあわせて表 9 のコードを指定する．

[実習 3]　図 40 の乗算器を Verilog HDL で記述し，FPGA ボード上で動作を確認せよ．レジスタ X, Y, M の値をそれぞれ 7 セグメントディスプレイ上に表示すること．また，最大動作周波数 f_{max} を調べて報告せよ．レポートには RTL Viewer の結果も示せ．

[ヒント] 図 40 ではクロック信号やリセット信号が省略されているので注意すること．

[実習 4]　実習 3 までは，被乗数 X および乗数 Y は

符号なし整数（$0 \leq X, Y \leq 15$）としてきた．ここでは X および Y を 4 ビットの符号付き整数（2 の補数表現）として，負数も扱えるように拡張せよ．ただし，被乗数 X にのみ負数を許し（$-8 \leq X \leq 7$），乗数 Y には 0 または正数のみを与える（$0 \leq Y \leq 7$）ものとする．拡張した乗算器を Verilog HDL で記述し，動作を機能シミュレーションで確認せよ．

[演習 3] 今回作成した乗算器は，図 42 のように 6 状態で制御している．状態数を減らすことによって，乗算器の構成をさらに単純化できるかどうか検討せよ．

参考文献

[1] 柴山 潔：“改訂新版 コンピュータアーキテクチャの基礎,” 第 3 章「コンピュータにおける数表現」, 近代科学社, 2003.

[2] 柴山 潔：“改訂新版 コンピュータアーキテクチャの基礎,” 第 6 章「演算アーキテクチャ」, 近代科学社, 2003.

プロセッサ

13　実習の目的

本実習ではこれまでの実習で学んだ事項の総まとめとして，単純な 8 ビットプロセッサ[*22]をハードウェア記述言語を用いて設計する．今回扱うプロセッサの規模は，現在，市販されているプロセッサと比べると極めて小さく，性能面でもまったく比較にならないものである．しかし，ここで行うプロセッサの設計は，単に論理回路についての理解を深めるだけでなく，一般的かつ複雑な論理回路を設計する際に十分に応用できる内容を含んでいる．実習の分量が多いので，しっかりと予習をした上で取り組んでもらいたい．

14　アーキテクチャ

本実習で作成するプロセッサの仕様を表 10 に示す．このプロセッサは，8 ビットの汎用レジスタを 4 個（R0〜R3）備えている．DIN（Data IN）から入力されたデータは，8 ビット幅のマルチプレクサを経由して，これらのレジスタに取り込まれる．

簡略化したブロック図を図 44 に示す．図 44 では，**制御ユニット**（control unit）から各構成要素への信号線（制御信号やクロック信号）は一部を除いて省略している．具体的には，各レジスタへの入力制御信号（Rn_{in}，A_{in}，G_{in}，IR_{in}），マルチプレクサの選択信号（Rn_{out}，G_{out}，DIN_{out}），算術論理演算ユニット（Arithmetic Logic Unit; ALU）の演算選択信号 *Mode*，および各レジスタへのクロック信号が必要であるが，IR_{in}，G_{in} と IR へのクロック信号以外は省略している．実際にはその他の信号線を接続する必要があるので注意すること．

例えば，DIN 入力からのデータをレジスタ R0 にセットする場合，マルチプレクサでは DIN を選択して（具体的にはマルチプレクサの選択信号 DIN_{out} を 1 にすることで），バスに DIN からのデータを出力する．バスには複数のレジスタ（R0〜R3，A）が接続しているが，そのままではすべてのレジスタがバスからデータを取り込んでしまうため，R0 だけにデータを送ることができない．そこで，図 45 に示すように，レジスタに 1 ビットの入力制御信号（イネーブル信号 EN）を用意し，その信号が 1 の時のみ 8 ビットの入力データを取り込むようにする．つまり，前回の実習までは入力制御信号が常に 1（常にデータを取り込む状態）になっていたと考えれば良い．制御ユニットは，レジスタ R0 の入力制御信号 $R0_{in}$ だけを 1 にして，それ以外のレジスタ（R1〜R3，A）の入力制御信号は 0 にする．こうすることで，レジスタ R0 だけがバスからデータを取り込めるように制御すればよい．

このプロセッサは，非同期入力 Run が 1 になると DIN から命令を入力し，実行を開始する．命令の実行が完了した時点で，Done 信号を 1 にする．以下，この繰り返しで命令列を実行する．また，非同期リセット入力 Resetn を 0 にすることで，プロセッサ内部の状態（制御ユニットの状態やレジスタの内容など）を初期

表 10　本実習で作成するプロセッサの仕様

内部バス幅	8 ビット
命令数	4
汎用レジスタ数	4

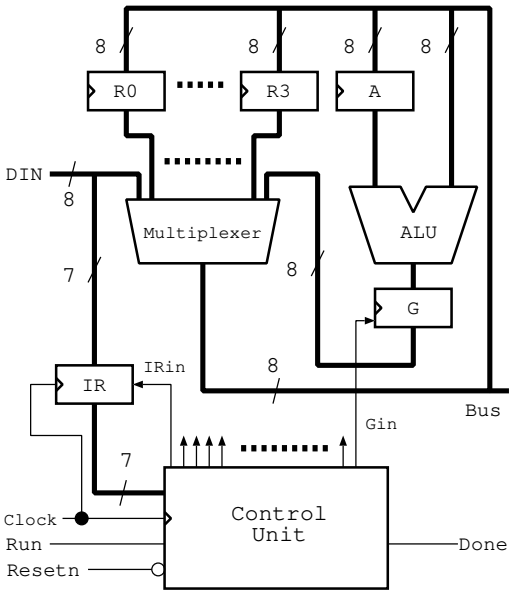

図 44　プロセッサのブロック図

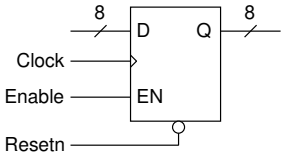

図 45　入力制御信号をもつレジスタのブロック図

[*22] 本プロセッサの仕様は文献 [1] をもとに本実習用に構成し直したものである．

表 11　命令セット

命令コード	命令	動作
000	**mv** Rx, Ry	Rx←[Ry]
001	**mvi** Rx, #D	Rx←D
010	**add** Rx, Ry	Rx←[Rx]+[Ry]
011	**sub** Rx, Ry	Rx←[Rx]−[Ry]

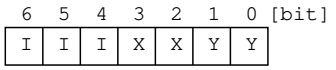

図 46　命令構成

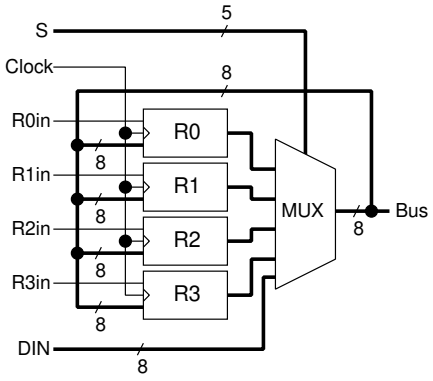

図 47　マルチプレクサからレジスタへの入力

化する．

15　プロセッサの設計

一般的なプロセッサでは外部のメモリから命令を読み出して処理を進めるが，メモリを接続する代わりに，8 ビット幅の DIN 入力から命令やデータを順次読み出すものとする．DIN 入力の下位 7 ビットは**命令レジスタ**（Instruction Register; IR）に接続し，制御ユニットが入力制御信号 IR_{in} を 1 にした時に，IR に値を取り込む．

このプロセッサでは，表 11 に示す 4 種類の命令を使用できることにする．表 11 の Rx と Ry は，レジスタ R0 からレジスタ R3 までのいずれかを指すものとする．また，D は 8 ビットの即値を示している．

このプロセッサの命令構成を図 46 に示す．図 46 において，III は命令コード 3 ビット，XX および YY は 4 個のレジスタ R0〜R3 を識別する 2 ビット値を表している．現時点では命令を 4 種類しか定義していないため，命令コードは 2 ビットで表現できるが，将来の命令追加を考慮し，3 ビットまで表現できるようにしておく．2 ビットで命令コードを表現する場合は，下位 2 ビットだけを用いればよく，最上位ビット（図 46 中の第 6 ビット）は使用しない（常に 0 にしておく）．例えば，R2 の内容を R1 に転送する命令（mv R1, R2）は，2 進数で表記すると [0000110] となる．

なお，mvi 命令では 8 ビットの値を Rx に読み込むため，図 46 の YY 部（第 0 ビットと第 1 ビット）は指定する必要がない．そのため，mvi 命令においては図 46 の YY 部に意味はない．

mvi 命令に対する D は，mvi 命令に続けて DIN から取り込む．ただし，前述の通り，DIN は 8 ビット幅しかないため，mvi 命令とそれに続く値 D までを一度に読み取ることができない．そこで，命令コードを IR に取り込んだ後，次のサイクルで D を取り込めばよい．

15.1　レジスタ部の設計

これまで見てきたように，このプロセッサでは DIN から入力した命令コードに従って，データを汎用レジスタと A レジスタ間で移動させ，ALU で演算を行う．そのためにマルチプレクサの出力を適切に選択しなければならない．

[実習 1]　図 47 のように，4 個の汎用レジスタ（R0〜R3）と入力 DIN をマルチプレクサ（MUX）に接続し，選択信号[*23]S で指定したレジスタまたは DIN の値を MUX から出力させる．MUX からの出力（Bus）は各レジスタの入力に接続し，入力制御信号（$R0_{in}$〜$R3_{in}$）が 1 になっているレジスタにのみ入力できるようにする．また，各レジスタにはリセット信号を接続し，リセット入力によりレジスタの内容がゼロクリアできるようにする．この回路を Verilog HDL で記述せよ．動作確認は機能シミュレーションにより行うこと．

15.2　ALU 部の設計

一般的な ALU では論理演算，算術演算，シフト演算の機能を持つが，本実習では基本的な算術演算（加減算）だけを行うものとする．演算の種類を選択するための制御信号（演算選択信号）は，扱う演算が加減算の 2 種類しかないため，1 ビット幅でよい．この演算選択信号を，ALU モジュールへの入力として接続せよ．

[実習 2]　ALU モジュールと A レジスタおよび G レジスタを Verilog HDL で記述し，実習 1 で作成した回路に追加せよ．A レジスタと汎用レジ

[*23] ここでは 5 本の各入力に対応させた 5 ビット幅の選択信号としている．

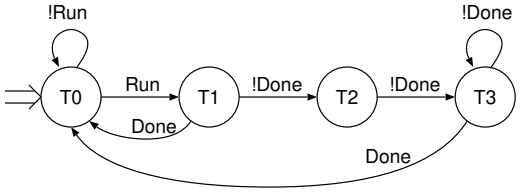

図 48　タイムステップを管理する有限状態機械の状態遷移図

スタを用いて ALU で演算を行い，結果が G レジスタに格納されることを機能シミュレーションによって確認すること．

[ヒント] G レジスタを追加することで，マルチプレクサの選択信号線を増やす必要があることに注意せよ．

ここまでの段階で，適切な制御信号を与えれば，レジスタ間のデータ転送と加減算処理が可能になっている．

15.3　制御ユニットの設計

次に，与えられた命令に従って，制御信号を生成する制御ユニットを作成する．この制御ユニットは，DIN から IR に入力した命令を解釈し，どのタイミングで値をレジスタに取り込むか，またマルチプレクサではどの入力を選択するかといった制御を行う．なお，命令によっては，実行完了までに数サイクルを要するものがある．例えば，加減算命令はレジスタと ALU の間で，複数回データを転送しなければならない．このため，制御ユニットは命令の実行ステップ（現在どのタイムステップを実行しているか）を管理し，各ステップごとに適切なタイミングで制御信号を出力する．

現在の実行ステップは，制御ユニット内部に有限状態機械（9 ページの 6.2 節を参照せよ）を構成し，そこで管理すればよい．具体的には，図 48 のような状態遷移を行う有限状態機械を構成する．Run 信号が 1 になると動作を開始し，命令が完了するまで（Done 信号が 0 の間），次のタイムステップに遷移する．命令の実行が完了し，Done 信号が 1 になれば，初期ステップ T_0 に戻り，Run 信号が 1 になるまで待機する．

表 12 は，表 11 で定義した命令を実行する場合，制御ユニットが各タイムステップで出力する制御信号を示したものである．最初のタイムステップ T_0 では，IR_{in} を 1 にすることで，DIN から IR に命令を取り込む．表 12 のように，現在実行している命令と，現在のタイムステップが分かれば，どの制御信号を 1 にするかを一意に決定することができる．つまり，制御ユニット内部のこの部分に関しては，組み合わせ回路で構成できることになる．この制御ユニットを作成することで，DIN から入力した命令列を順番に処理できるようになる．

表 12　各タイムステップで出力する制御信号

命令	T_0	T_1	T_2	T_3
mv	IR_{in}	RY_{out}, RX_{in}, *Done*	-	-
mvi		DIN_{out}, RX_{in}, *Done*	-	-
add		RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
sub		RX_{out}, A_{in}	RY_{out}, G_{in}, *Mode*	G_{out}, RX_{in}, *Done*

表 13　サンプルプログラム

Address	Object Code	Source Code
00	10H 05H	**mvi** R0, #05
02	04H	**mv** R1, R0
03	21H	**add** R0, R1
04	30H	**sub** R0, R0

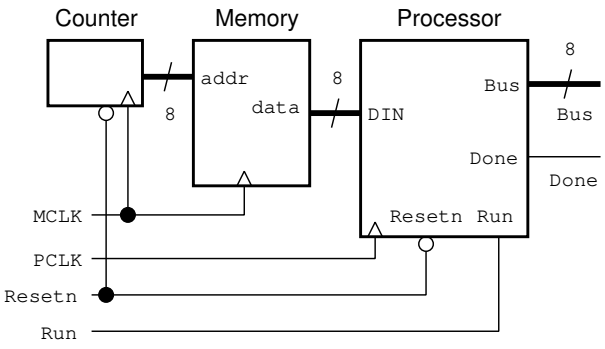

図 49　外部メモリとプロセッサの接続

[実習 3]　表 12 に基づき，制御ユニットの動作を Verilog HDL で記述し，実習 2 の回路に追加せよ．動作確認は，表 13 に示すサンプルプログラムを用いて，機能シミュレーションによって行うこと．

[ヒント] タイムステップの遷移を管理する部分（有限状態機械；順序回路）と，各タイムステップで入力制御信号等を制御する部分（組み合わせ回路）は，分離して記述するとコードの見通しが良くなる．

[選択課題 1]　実習 3 までで作成したプロセッサの動

作を FPGA ボード上で確認せよ．

[演習 1] このプロセッサのレジスタ数を 16 個に増やすとすると，プロセッサの内部構成と命令構成をどのように変更しなければならないか，具体的に考察せよ．

[選択課題 2] これまで作成したプロセッサでは実行する命令を適切なタイミングで DIN 入力に与える必要があった．人間が DIN 入力に情報を与える代わりに，図 49 に示すように外部メモリ[*24]を接続し，そのメモリから情報を DIN に与えるように改良せよ．メモリへのアドレス入力（addr）はカウンタ回路で生成すればよい．カウンタとメモリへのクロック入力（MCLK）は，プロセッサへのクロック入力（PCLK）と分離して構わない．

[選択課題 3] 選択課題 2 で作成した回路の動作を FPGA ボード上で確認せよ．

[演習 2] 実習 3 までで作成したプロセッサに条件分岐命令を追加する場合，プロセッサの内部構成や命令セットにどのような変更が必要になるか，具体的に考察せよ．レポートには図 44 のようなブロック図を示し，新たに用意すべき機構との接続関係も説明せよ．

参考文献

[1] Altera Corporation: “Digital Logic – Laboratory Exercises,” Lab 9 A Simple Processor, 2011.

[*24] 読出し専用メモリ．ROM

付録 A　Verilog HDL の文法

ここでは Verilog HDL の基本的な文法を説明する．紙面の都合上，すべての文法を説明することはできないので，より詳しくは文献[*25]などの参考書を参照するとよい．

A.1　HDL の基本構文

Verilog HDL では C 言語と同様，自由に改行，スペース，タブを用いて記述することができる．コメントは，複数行にわたるものは/*と*/で囲む．入れ子にはできない．1 行コメントは// で始め，その行末までをコメントとみなす．

Verilog HDL において，基本的な論理値は，0，1，x，z の 4 種類である．0，1 はそれぞれ偽（FALSE），真（TRUE）を表す．

x は論理が 1 か 0 か確定しない状態（不定値）を表し，例えば電源投入直後で信号の明確な初期化を行っていない状態を表す．また，信号線上で 1 と 0 が競合した場合，どちらの論理値になるかわからない状態も表す．

z はハイインピーダンス（high impedance）を意味し，信号が接続していない状態を表す．

A.2　識別子

Verilog HDL での**識別子**とは，C 言語における変数名や関数名のように，信号や変数，モジュールに付ける名前である．識別子には，英数字とアンダスコア (_)，$が使えるが，先頭は英字またはアンダスコアでなければならない．例えば，4bit_counter のようなモジュール名では，先頭が数字のためにエラーになるので注意せよ．英字の大文字と小文字は区別される．C 言語と同様，予約語は使用できない．

A.3　信号と変数

Verilog HDL で設計を行う際に，特に重要な概念が信号と変数である．

A.3.1　信号

信号（signal）はハードウェア間を接続する配線に相当する概念であり，Verilog HDL では **wire** で宣言する．wire は，文字通り部品間を接続するワイヤ（配線）の役割を持つ．ゲートなどでドライブされていないワイヤはハイインピーダンス状態になる．

wire の値を設定する（ワイヤの接続を定義する）場合は assign 文を用いる．assign 文の右辺に現れる信号が変化すると，即座に再評価されて左辺の信号に代入される．assign 文による記述は，組み合わせ回路の接続関係を定義していることになる．複数の assign 文を記述する場合，その実行順序は記述順序と無関係である．

```
wire S;
wire Q, X, Y;
assign Q = S ? Y : X;
```

図 50　Verilog HDL における assign 文の記述例

組み合わせ回路を assign 文で記述する際に，後述する if 文や case 文といった手続き文を使用したい場合は，ファンクション（A.12 節）として記述する必要がある．

assign 文の記述例を図 50 に示す．ここでは 1 ビット幅の信号線 Q に対し，選択信号 S が真であれば Y 信号を代入し，S が偽であれば X 信号を代入する．この動作は 1 ビット 2-to-1 マルチプレクサ（6 ページの図 15）に相当する．

A.3.2　変数

Verilog HDL における**変数**はデータを記憶する素子の役割を持ち，値が代入されると，再度値が代入されるまで前の値を保持する．変数は **reg** として宣言する．

変数への代入（手続的代入）は，A.10 節で後述する initial 文や always 文の手続き文で行う．

“=” による**ブロッキング代入**では，式の評価と同時に代入を行う．一般的なプログラムと同様に，記述した順番で代入が行われていくため，複数の代入文がある場合，記述順が異なれば結果も異なることになる．これに対し，“<=” による**ノンブロッキング代入**では，右辺の評価が終了した時点では代入は行わず，その時刻で行われる他の代入文が評価し終わるまで代入は遅延される（シミュレーションでは，次の時刻に進む直前に，一度に左辺へ代入される）．

例えば，図 51 のようにブロッキング代入を行うことを考える．ここでは o1 へ in を代入した後，その o1 を o2 へ代入するため，最終的にはどちらの出力も in の値をもつ．論理合成を行うと図 52 の結果が得られる[*26]．出力 o1，o2 が共に入力 in と等しいことがわかる．一方，図 53 のようにノンブロッキング代入に書き換えると，1 段目のレジスタ（o1）への代入と 2 段目のレジスタ（o2）への代入は同じ時刻における並列動作となる．すなわち，o1 が o2 へ代入されるタイミン

[*25] 桜井 至 著「HDL によるデジタル設計の基礎」（テクノプレス，1997）

[*26] 論理合成ツールによっては，1 個の D-FF からの出力を o1,o2 の 2 出力に分岐して接続するものもある．

```
input in, Clock;
output o1, o2;
reg o1, o2;

always @(posedge Clock) begin
  o1 = in;
  o2 = o1;
end
```

図 51　ブロッキング代入の例

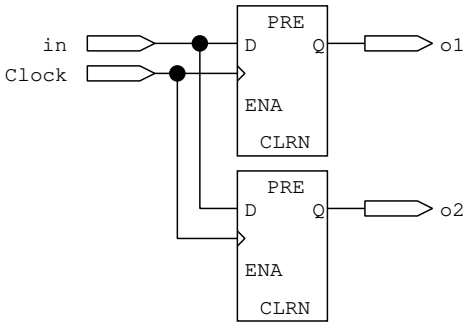

図 52　図 51 の論理合成結果

```
input in, Clock;
output o1, o2;
reg o1, o2;

always @(posedge Clock) begin
  o1 <= in;
  o2 <= o1;
end
```

図 53　ノンブロッキング代入の例

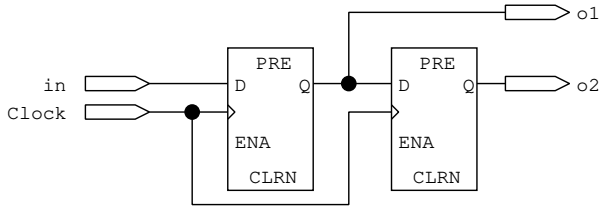

図 54　図 53 の論理合成結果

グは，Clock の次の立ち上がりであるため，論理合成の結果は図 54 のように 2 個の D-FF によるシフトレジスタになる．

複数の `always @()` 構文を用いた記述では，同じ時刻に多数の代入が行われる場合，シミュレーションの際にどの代入文が先に実行されるかは不確定である．つまり，ブロッキング代入では，シミュレータによっては結果が異なる可能性がある．よって，実際にハードウェア化を想定して `always @()` 構文で代入する場合

```
reg a, b;

always @(...) begin
  a = 1; b = 0; // initialize
  ...
  a = b;
  b = a;
  // a=0, b=0
```

図 55　ブロッキング代入の例 2

```
reg a, b;

always @(...) begin
  a = 1; b = 0; // initialize
  ...
  a <= b;
  b <= a;
  // a=0, b=1
```

図 56　ノンブロッキング代入の例 2

は，ノンブロッキング代入が適していると言える．

もう 1 つの例を見てみよう．図 55 では，変数 a に変数 b の内容（0）を上書きした後，変数 b に変数 a の内容（上書きされた 0）を代入する．つまり結果はどちらの変数も 0 である．しかし図 56 では，次のシミュレーション時刻に進む直前に一度に 2 つの代入が行われるため，**変数 a と変数 b の内容が入れ替わり，**変数 a の内容は 0，変数 b の内容は 1 になる．また，図 55 の代入順を入れ替えると結果も変わるが，図 56 の代入順を入れ替えても結果は変わらない．

A.3.3　宣言とベクタ表現

2 進数を構成する 1 桁はビット（bit）であり，このビットを複数並べたものを**ベクタ**（vector）と呼ぶ．ベクタに対し，ビットは**スカラ**（scalar）とも呼ばれる．Verilog HDL では，論理値 `TRUE`（真）と `FALSE`（偽）はそれぞれスカラ値 `1'b1`，`1'b0` と等価である．ベクタの最上位ビットを **MSB**（Most Significant Bit），最下位ビットを **LSB**（Least Significant Bit）と呼ぶ．

HDL で用いる信号や変数はベクタで表現する．Verilog HDL では信号（wire）と変数（reg）は図 57 のように宣言する．カッコ `[]` 内の 2 つの数値は，コロン（`:`）の左側が MSB のインデックス，右側が LSB のインデックスを表す．MSB のインデックス値を LSB のインデックス値より大きくするか小さくするかはどちらでも構わないが，混乱を避けるためにシステム内では統一しておくべきである．

図 57 において，信号 `DATA` の MSB を選択するには

```
wire CLK;   // 1bit scalar signal
wire [7:0] DATA; // 8bit vector signal
wire [3:0] w1, w2; // two 4bit signals
reg [1:16] R1;    // 16bit vector variable
```

図 57　信号と変数の宣言例

DATA[7]，変数 R1 の LSB を選択するには R1[16] と表記する．なお，変数 R1 の上位 8 ビットだけを取り出したい場合は，R1[1:8] と表記する．

また，ベクタは 2 進数だけではなく，8 進数，10 進数，16 進数で表記することもできる．ベクタの用途によって適切な基数を選択すると読みやすくなる．Verilog HDL では「ビット幅’基数 数値」のように表現する．ここで，ビット幅指定の直後の文字は single quote (’) であり，backquote (‘) ではないので間違えないように注意すること．基数は 2 進数（b），8 進数（o），10 進数（d），16 進数（h）で指定する．例えば，10 進数での 27 を 8 ビットベクタで表記すると，それぞれ次のようになる．

```
8'b00011011, 8'o33, 8'd27, 8'h1b
```

桁数が多いベクタは非常に読みにくくなるため，読みやすくするために，適宜 “_” を挿入しても構わない（この場合，途中に挿入した “_” は無視される）．例えば 4 ビットずつ区切って，8'b0001_1011 のように書くことができる．

ビット幅を指定しない整数は 32 ビットベクタと等価であり，整数とベクタの相互代入が可能である．代入時に，左辺のビット幅が狭い場合は，右辺の上位ビットが除去される．逆に，左辺のビット幅が広い場合は，右辺の上位に 0 が挿入される．また，演算を実行する際には**ビット幅の小さい値はビット幅の大きい方に幅を揃えてから**行われる．

この動作には注意が必要で，例えば図 58 のようなコードでは意図しない動作をする．図 58 では 8 ビット幅のカウンタ count をカウントアップすることで，カウンタ値が 8'hff から 8'h00 に変化するタイミングで信号 out を 1 にしようとしている[*27]．count + 1 の部分で，1 はビット幅を指定していない整数値であるため，32 ビット幅で（32'h00000001 として）解釈されることになる．そうすると，本来 8 ビット幅である count は一度 32 ビット幅に拡げられて（count 値が 8'hff のときであれば 32'h000000ff のようにして）から 1 を

```
wire [7:0] count;
wire out;
assign out = (count + 1 == 0) ? 1 : 0;
```

図 58　ビット幅を指定しないと問題が生じるコード例

```
wire [7:0] count;
wire out;
assign out = (count + 8'h01 == 8'h00) ?
             1'b1 : 1'b0;
```

図 59　ビット幅を明示的に指定したコード例

加算する．その結果，count+1 は 32'h00000100 となり，0 と比較しても結果は偽となる．

この問題を回避するためには，整数値には明示的にビット幅を指定するようにすればよい．図 59 は整数値にビット幅を明示的に指定するように書き換えたものである．ここでは，count + 8'h01 の演算は一貫して 8 ビット幅で行われることになり，先の例で示したように 8 ビットを超えての桁上がり問題は無くなる．count + 8'h01 の部分は count + 1'b1 としても同じである[*28]．

A.3.4　宣言に関する注意事項

Verilog HDL では未宣言の信号線はビット幅 1 の wire として扱われる．この場合，警告メッセージとして「Verilog HDL Implicit Net warning at foo.v(xx): created implicit net for “unknown_wire”」のように表示され，unknown_wire という名前の未宣言信号が 1 ビットで扱われていることを示している．

A.4　算術演算子

算術演算子として，加算（+），減算（–），乗算（*），除算（/），剰余演算（%）がある．整数の除算では小数点以下は切り捨てである．

Verilog HDL でこれらの演算子を使うと，暗黙にうちに対応するハードウェア（加算器，減算，乗算器，除算）を生成することになる．実際に必要とする規模以上に大きな（多ビットに対応した）回路構成になることがあるので，使用する場合は注意すること．特に回路の遅延時間が大きすぎる場合は，意図していない演算回路が構成されていないかを RTL Viewer で確認するとよい．

A.5　関係演算子

関係演算子として，大小比較（<, <=, >, >=）と等号（==, ! =, ===, ! ==）がある．比較結果が真であ

[*27] 実際には，この部分で count の値は変更していない．

[*28] 1'b1 を count のビット幅である 8 ビット（8'b00000001）に拡大してから加算する．

```
A = 4'b1010;
B = ~A;  // B = 4'b0101
C = !A;  // C = FALSE (1'b0)
```

図 60　Verilog HDL における否定の記述例

れば論理値 1，偽であれば 0 を返す．比較対象に不定値 x が含まれる場合は，結果は不定 x となる．== や != では，比較対象に不定値 x やハイインピーダンス z が含まれる場合は結果が不定 x になる．一方，=== や !== では x や z を比較することができる．

A.6　論理演算子

論理演算子として，否定（NOT），積（AND），和（OR）が使える．論理式は左から評価され，真偽が確定した時点でそれ以降の評価は行われない．ベクタを対象とする場合，全ビットが 0 であれば偽，そうでなければ真として評価される．

A.6.1　論理否定（NOT）

Verilog HDL では，論理演算子!で単項の論理否定を得る．ベクタに対して論理演算!を行うと，ベクタ値が 0 であれば真（TRUE），非 0 であれば偽（FALSE）のスカラ値が得られる（各ビットの反転にならないことに注意）．

A.6.2　論理積（AND）

複数の関係演算や等号・不等号演算の結果の論理積（AND）は&&により求めることができる．なお，論理積否定（NAND）は論理積&&の結果を論理否定!することで求める．

A.6.3　論理和（OR）

AND と同様に，複数の関係演算や等号・不等号演算の結果の論理和（OR）は||により求めることができる．論理和否定（NOR）は論理和||の結果を否定!することで求める．

A.7　ビット演算子

A.7.1　ビット NOT

ベクタの各ビットを反転したい場合はビット演算~を行えばよい．スカラに対しては，論理演算!とビット演算~による結果の違いは無い．

記述例を図 60 に示す．

A.7.2　ビット AND

ベクタの各ビットで論理積を求める場合はビット演算子&を，論理積否定を求める場合は!&を用いる．

記述例を図 61 に示す．

A.7.3　ビット OR

ベクタの各ビットで論理和を求める場合はビット演算子|を，論理和否定を求める場合は!|を用いる．

```
if (c > 0 && S == 2'b00) // 論理演算(AND)
...
A=3'b010; B=3'b110;
Y = A & B;    // Y=3'b010
Y = A !& B;   // Y=3'b101
```

図 61　Verilog HDL における論理積の記述例

```
if (c > 0 || S == 2'b00) // 論理演算(OR)
...
A=3'b010; B=3'b110;
Y = A | B;    // Y=3'b110
Y = A !| B;   // Y=3'b001
```

図 62　Verilog HDL における論理和の記述例

```
A=3'b010; B=3'b110;
Y = A ^ B;    // Y=3'b100
Y = A ~^ B;   // Y=3'b011
```

図 63　Verilog HDL における排他的論理和の記述例

```
A = 3'b100;
B = &A; // B=A[2] & A[1] & A[0] = 1'b0
C = |A; // C=A[2] | A[1] | A[0] = 1'b1
```

図 64　リダクション演算子の使用例

```
A = 4'b0101;
B = (A << 1); // B=4'b1010
C = (A >> 2); // C=4'b0001
```

図 65　シフト演算子の使用例

記述例を図 62 に示す．

A.7.4　ビット XOR

ベクタの各ビットで排他的論理和（XOR）を求める場合，^により求めることができる．排他的論理和否定（XNOR）は~^で求める．

記述例を図 63 に示す．

A.7.5　リダクション演算子

リダクション演算子には&，|，^，~^がある．リダクション演算は単項演算であり，1 つのオペランドに対してビットごとの演算を行い，結果を 1 ビットの値として返す．使用例を図 64 に示す．

A.7.6　シフト演算子

左シフト演算子（<<）と右シフト演算子（>>）があり，シフトしたいビット数を指定する．シフトによって生じた空きビットには 0 が補われる．使用例を図 65 に示す．

```
wire [2:0] A, B;
wire [5:0] C = {A, B};
wire [3:0] D = {A[1:0], B[1:0]};
wire [1:0] E, F;
wire [3:0] G;
assign {E, F} = D;
assign G = {2{E}};
```

図 66　連接演算子の使用例

```
out1=(enable==1) ? data : 8'bz;

out2=(sel==2'b00) ? data[0] : (
    (sel==2'b01) ? data[1] : (
    (sel==2'b10) ? data[2] : data[3]));
```

図 67　条件演算子の使用例

表 14　演算の優先順位

順位	演算子	演算
1	! ~ & ~& | ~| ^ ~^	論理否定 リダクション（単項演算）
2	* / %	乗除・剰余
3	+ -	加減算
4	« »	シフト
5	< <= > >=	比較
6	== != === !==	等号
7	& ^ ~^	ビット AND ビット XOR, XNOR
8	|	ビット OR
9	&&	論理 AND
10	||	論理 OR
11	? :	条件演算

A.7.7　連接演算子

連接はビットを結合する演算である．連接演算子の使用例を図 66 に示す．6 ビット信号 C は 2 個の 3 ビット信号 A, B を連接しており，C の上位 3 ビットが A の値，下位 3 ビットが B の値になる．4 ビット信号 D は A, B それぞれの下位 2 ビットずつを連接している．また，2 ビット信号 E, F には，D の上位 2 ビットと下位 2 ビットがそれぞれ代入される．

連接の繰り返し回数を指定することもできる．例えば，図 66 の 4 ビット信号 G は，2 ビット信号 E が 2 回連接された信号となる．つまり，`{2{E}}` は `{E, E}` と等価である．

A.8　条件演算子

条件演算子は C 言語の 3 項演算子と同様に，

（条件式）? 真の場合 : 偽の場合

と記述し，条件式を満たした場合は “?” の直後の式，満たさなかった場合は “:” 後の式を実行する．使用例を図 67 に示す．図 67 において，out2 への代入のように複数の条件が連続する場合は，付録 A.10.2 で説明する `case` 文を使うと読みやすく記述できる．

A.9　演算の優先順位

表 14 に演算の優先順位を示す．優先順位の高い演算から計算され，同じ優先順位の場合は左から右に計算される．演算の優先順位を変更したい場合は括弧を用いる．

A.10　手続きブロック

`if` 文や `case` 文といった手続き文は手続きブロック内部で使用する．C 言語プログラミングと同様の感覚で，手続き文を module 内部に直接記述しても文法エラーになるので注意すること．組み合わせ回路を `assign` 文で記述しようとして，`if` 文や `case` 文を使いたい場合は，A.12 節で後述するファンクションとして記述すれば良い．

手続きブロックは `initial` 文または `always` 文で構成される．`initial` と `always` はどちらも時刻 0（シミュレーションの開始時）で実行を開始するが，`initial` で記述されたブロックは 1 回しか実行されないという違いがある．ただし，`initial` 文は本来，シミュレーションの開始時に信号の初期化やテストベクタの設定のために用いるものであり，ここに記述した内容は**論理合成の対象外**となる．そのため，`initial` 文は**論理合成の対象とするモジュールで使用してはならない**．`initial` 文の使い方は 付録 D を参照せよ．

一方，`always` で記述されたブロックは，ブロック末尾まで実行すると先頭に戻り，何度も実行を繰り返す．`always` 文はイベント制御文（`@`）を併せて指定することで，特定のイベントが発生した時に実行が始まるように指定することができる．記述例を図 68 に示す．

モジュールには複数の `always` ブロックを記述することができる．この場合，それらのブロックは独立して動作する．これによって並行して動作するハードウェアを記述できる．

A.10.1　if 文

図 69 に示すように，`if` 文は指定された条件（enable が 1）が満たされた場合に文（`Q <= d`）を実行する．条件を満たさない場合は `else` に続く文（`Q <= ~Q`）を実行する．条件判断の結果，2 つ以上の文を実行させる場合は，それらの文を `begin` と `end` で囲むこと．

```
reg a,b;

initial begin
  // シミュレーションの初期設定のため,
  // 1回だけ実行される.
  // この部分は論理合成されない.
  a = 1'b0;
  b = 1'b1;
end

always @(posedge Clock) begin
  // クロックの立ち上がりごとに実行
  a = b;
end
```

図 68　initial 文と always 文の記述例

```
always @(posedge Clock) begin
  if (enable == 1'b1)
    Q <= d;
  else
    Q <= ~Q;
end
```

図 69　if 文の記述例

```
always @(posedge Clock) begin
  case (inst)
    2'b01: OUT = IN[1];
    2'b10: OUT = IN[2];
    default: OUT = IN[0];
  endcase
end
```

図 70　case 文の記述例

A.10.2　case 文

case 文は式の値と合致する分岐式があれば，対応する文を実行する．複数の分岐式と合致する場合は，最初に合致した分岐式に対応する文を実行する．どの分岐式とも合致しない場合は，default に対応する文を実行する．if 文と同様に，複数の文を実行したい場合はそれらの文を begin と end で囲む必要がある．

図 70 の例では，inst が 2'b01 であれば OUT=IN[1] を，inst が 2'b00 または 2'b11 であれば default の OUT=IN[0] を実行する．

A.10.3　always 文記述の際の注意点

always 文を使った基本的な記述スタイルを表 15 に示す．always 文には表 15 以外の記述をすることも可能で，シミュレーションを行うこともできるが，必ずしも論理合成が可能なわけではない．例えば，図 71 で

```
module clk_rst (Clk, Resetn, d, q);
  input Clk, Resetn, d;
  output q;
  reg q;

  always @(posedge Clk or negedge Resetn)
    begin
      q <= d;
    end
endmodule
```

図 71　レジスタとして論理合成できない記述

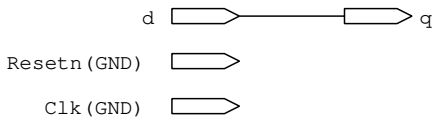

図 72　図 71 を RTL Viewer で表示した結果

は，always のイベント制御文に複数のエッジトリガを記述している．このような記述をした場合，論理合成ツールはレジスタを合成しようとする．しかし，このコードからは Clk と Resetn のどちらがクロック信号であるかを判断できない[*29]ので，レジスタとして論理合成できず，図 72 のような組み合わせ回路を生成してしまう[*30]．図 72 を見れば，Clk と Resetn の両信号が動作に何の影響も及ぼさないことがわかる．

そこで，表 15 にしたがって，図 71 の記述を図 73 のように書き換えると，図 74 のような非同期リセット付きレジスタとして合成できる．このように，RTL Viewer を用いると自分が記述した回路がどのように論理合成されたかを確認できる．

また，組み合わせ回路を記述する際には，if 文や case 文の条件がすべての場合を網羅していることに注意する必要がある．分岐する条件が存在しないと，出力が以前のデータを保持することになってしまい，組み合わせ回路を記述したつもりでも，意図しないラッチが生成されてしまう．

A.11　モジュール

Verilog HDL において，回路を記述する基本構造となるものが**モジュール**（module）である．モジュールは図 75 に示すように，module から endmodule までの間に記述する．ポートリストには入出力信号（**入出**

[*29] 信号名は判断材料にならない．

[*30] Quartus Prime では “can't infer register for assignment in edge-triggered always construct because the clock isn't obvious. Generated combinational logic instead” の警告が表示される．

表 15　always 文の基本的な記述スタイル

型	記述
エッジトリガ	`always @(posedge Clk) begin ... end`

エッジトリガ＋非同期リセット	`always @(posedge Clk or negedge Resetn) begin`
	`  if (!Resetn) begin ... end`
	`  else begin ... end`
	`end`

エッジトリガ＋同期リセット	`always @(posedge Clk) begin`
	`  if (!Resetn) begin ... end`
	`  else begin ... end`
	`end`

レベルトリガ	`always @(Clk or Data) begin`
	`  if (Clk) begin ... end`
	`end`

レベルトリガ＋非同期リセット	`always @(Clk or Resetn or Data) begin`
	`  if (!Resetn) begin ... end`
	`  else if (Clk) begin ... end`
	`end`

```
module clk_rst (Clk, Resetn, d, q);
  input Clk, Resetn, d;
  output q;
  reg q;

  always @(posedge Clk or negedge Resetn)
    begin
      if (!Resetn)
        q <= 0;
      else
        q <= d;
    end
endmodule
```

図 73　図 71 を非同期リセット付きレジスタに書き換えた記述

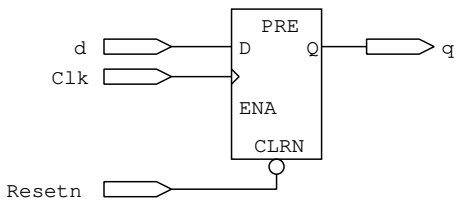

図 74　図 73 を RTL Viewer で表示した結果

力ポート）をカンマ区切りで列挙する．各ポートの入出力方向やビット幅は `module` 文直後のポート記述で定義する．ポートリストの直後にはセミコロンが必要なので注意すること．

```
module モジュール名（ポートリスト）;
  ポート記述
  変数宣言
  ネット宣言
  パラメータ宣言

  モジュール構成要素
endmodule
```

図 75　モジュール構造

A.11.1　モジュールのインスタンス化

Verilog HDL では定義済みのモジュールを，別のモジュール内部で**インスタンス化**（instantiation; 実体化）することで，モジュールの階層構造を作成できる．ここで最上位に位置するモジュールを**トップモジュール**と呼ぶ．FPGA のピンと接続できるのは，トップモジュールの入出力ポートだけである．

下位のモジュールに信号を接続する方法には，**ポート順接続** と**ポート名接続** の 2 通りがある．ポート順接続では，下位モジュールのポートリストで定義されたポート順と等しい順番で上位モジュールの信号を接続する．ポート名接続では，下位モジュールでのポート名をピリオドに続けて指定し，接続する上位モジュールの信号をカッコ内に記述する．このため，ポート名接続ではポートの記述順は任意である．

モジュールのインスタンス化の例を図 76 に示す．図 76 では 2 入力 1 出力の下位モジュール `submod1` を `mainmod` 内部で 2 個，インスタンス化している．これ

```
// 下位モジュール
module submod1(IN1,IN2,OUT);
  input IN1, IN2;
  output OUT;

  ...
endmodule

// 上位モジュール
module mainmod(I1,I2,I3,Q);
  input I1,I2,I3;
  output Q;
  wire W1,W2;

  assign ...

  always @(....) begin
    ...
  end

  // インスタンス化は always文の外側
  submod1 U1(I1,I2,W1); // ポート順接続
  submod1 U2(.OUT(W2),.IN1(I2),.IN2(I3));
                        // ポート名接続
endmodule
```

図 76　モジュールインスタンス化の例

らのインスタンスには U1 と U2 というインスタンス名が与えられ，U1 はポート順接続，U2 はポート名接続している．

モジュールの構造としては, 図 77 のように, トップモジュールである `mainmod` モジュールの内部に `submod1` というモジュールの 2 つの実体が存在するようにイメージすればよい．

モジュールのインスタンス化は，記述上は C 言語の関数呼び出しのように見えるかもしれないが，関数呼び出しとは性質が全く異なることに注意すること．always 文などの手続きブロックの内部には記述できない．

A.11.2　モジュールパラメータ

下位モジュールをインスタンス化する際に，上位モジュール側から下位モジュール側のパラメータ（parameter）を書き換えることができる, 例えば, モジュールが扱うビット幅をパラメータとしておけば，そのモジュールをインスタンス化する時に実際のビット幅を指定することができる．

図 78 にモジュールパラメータの使用例を示す．ここでは，2 種類の下位モジュール（`submod2`，`submod3`）に対して, パラメータ値を置き換えてインスタンス化し

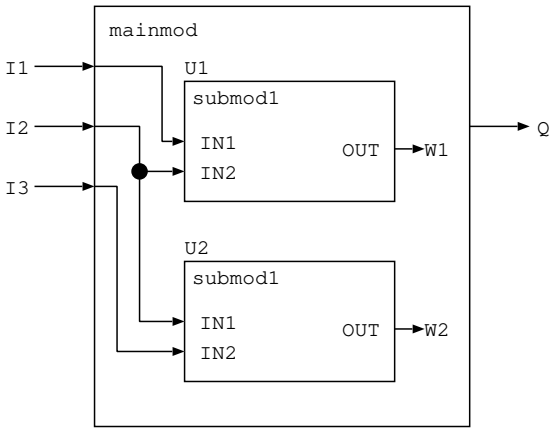

図 77　図 76 のモジュール構造

ている．モジュール `submod2` はデフォルトでは 8 ビット幅の入出力を行うモジュールとして定義されている．トップモジュール `mainmod` 内で `submod2` をインスタンス化する際，インスタンス U0 のようにビット幅を指定しなければ，パラメータ `bitwidth` はデフォルト値（8）のままになる．モジュール `submod2` のインスタンス U1 では，パラメータ `bitwidth` を 16 で上書きしてインスタンス化しているため，`submod2` 自体を書き換えることなく，U1 では 16 ビット幅の入出力ポートを使えるようになる．インスタンス U2 は，2 つのパラメータ `in_width` と `out_width` を持つモジュール `submod3` に対し，それぞれ異なる値（8 と 16）を設定して作成している．このように複数のパラメータを設定する場合は，`#( )` 内に，下位モジュールで宣言された順番で値を列挙する．

A.12　ファンクション

C 言語と同様に，Verilog HDL でも `if` 文や `case` 文を使用することができるが，A.10 節でも説明したように，これらの手続き文は `assign` 文の右辺に記述することができない．組み合わせ回路を `assign` 文で記述する際に，`if` 文や `case` 文を使いたい場合は，ファンクションとして記述する必要がある．

ファンクションの構文を図 79、使用例を図 80 にそれぞれ示す．ファンクションは 1 個以上の入力パラメータとただ 1 つの戻り値をもつ．戻り値幅は省略可能で，省略した場合は 1 ビット幅となる．

ファンクションを定義すると，内部にファンクション名と同じ名前のレジスタが作成される．このレジスタに値を代入することで，戻り値を設定できる．

A.13　コンパイラ指示子

コンパイラ指示子（compiler directives）は，Verilog HDL のコンパイラに対して指示を与えるために用

```
// 下位モジュール
module submod2(IN1,IN2,OUT);
  parameter bitwidth = 8; // default
  input [bitwidth-1:0] IN1, IN2;
  output [bitwidth-1:0] OUT;
  ...
endmodule

module submod3(IN1,IN2,OUT);
  parameter in_width = 4; // default
  parameter out_width = 8; // default
  input [in_width-1:0] IN1, IN2;
  output [out_width-1:0] OUT;
  ...
endmodule

// 上位モジュール
module mainmod(I1,I2,I3,Q);
  input [7:0] I1,I2,I3;
  output Q;
  wire [15:0] W1,W2,C1,C2;

  assign C1 = {I1,I2}; // 連接演算子
  assign C2 = {I2,I3};
  ...
  submod2 U0 (I1,I2,W1[7:0]);
              // 8bit (default) version
  submod2 #(16) U1 (C1,C2,W1);
              // 16bit version
  submod3 #(8,16) U2 (I2,I3,W2);
              // in=8bit, out=16bit
endmodule
```

図 78　モジュールパラメータの例

```
function 戻り値幅 ファンクション名;
  入力宣言
  局所データ宣言

  手続き文
endfunction
```

図 79　ファンクション構文

いる．コンパイラ指示子は backquote (‘) で始まる．single quote (’) ではないので，間違えないようにすること．なお，通常の HDL 構文とは異なり，行末のセミコロンは不要である．一度設定した指示子は，モジュールやファイルの境界に関わらず，別の指示子によって上書きされるかリセットされるまで有効である．

文字列置換　マクロ名を，指定した文字列で置き換える．コード中で参照する場合は，マクロ名の前に backquote (‘) を付加すること．‘define で一度定義したマクロは，‘undef により未定義にできる．

‘define マクロ名 文字列

‘undef マクロ名

ファイル取り込み　任意の場所で指定したファイルの内容を取り込む（インクルードする）．

‘include “**ファイル名**”

条件付きコンパイル　‘ifdef 直後に指定したマクロ 1 が定義されていれば文 1 を，定義されておらず，マクロ 2 が定義されていれば文 2 をコンパイルする．どちらのマクロも定義されていなければ文 3 をコンパイルする．‘ifndef を用いると，指定したマクロが定義されていない場合に直後の文を実行する．

‘ifdef ‘マクロ名 1
**　文 1**
‘elsif ‘マクロ名 2
**　文 2**
‘else
**　文 3**
‘endif

リセット　すべてのコンパイラ指示子をデフォルト値にする（マクロを未定義にする）．

‘resetall

```
function [2:0] selector_func;
  input s;
  input [2:0] i0, i1;
  reg [2:0] select;

  begin
    if (s == 0)
      begin
        select = i0;
      end
    else
      begin
        select = i1;
      end
    selector_func = select; // 戻り値
  end
endfunction

// ファンクションの呼び出し
assign out=selector_func(sel, sig1, sig2);
```

図 80　ファンクションの記述例

```
‘define T0 1’b0
‘define T1 1’b1

‘include "commondef.v"

‘ifdef ‘DEBUG
  ...
‘endif
```

図 81　コンパイラ指示子の使用例

使用例を図 81 に示す．ここでは，マクロ T0 に 1 ビット値 0 を，マクロ T1 に 1 ビット値 1 をそれぞれ定義している．また，‘include 指示子の位置にファイル commondef.v の内容を取り込む．マクロ DEBUG が定義されていれば，‘endif までの間の文をコンパイルする．

付録 B　Quartus Prime の基本的な操作方法

本実習で使用する CAD システム Quartus Prime の操作方法を簡単に説明する．

B.1　起動方法

Windows のスタートメニューから [Intel FPGA 20.1.1.720 Lite Edition] とたどると，Quartus (Quartus Prime 20.1) のショートカットが見つかるはずである．実験室の PC には，デスクトップ上にもショートカットを設置している．それをクリックすると Quartus Prime が起動する．Quartus の初回起動時には図 82 のようなダイアログボックスが表示されるので，2 番目の「Run the Quartus Prime software」を選択して OK を押す．その後，もう一度 Quartus Prime を起動する．

B.2　デザインフロー

Quartus Prime におけるデザインフロー[*31]を図 83 に示す．[Analysis & Synthesis] の実行により，**ネットリスト**（netlist；回路における端子間の接続関係）を生成し，同時に機能シミュレーションに必要なファイルを生成する．フルコンパイル（[Start Compilation] の実行）により，DE1-SoC に転送する構成情報ファイル（*.sof）が得られる．なお，[Analysis & Elaboration] を実行すると，Quartus Prime は記述した Verilog HDL ファイルの構文エラーとセマンティックエラーをチェックする．図 83 では [Analysis & Synthesis] の上流に位置する工程である．

B.3　新規プロジェクトの作成

Quartus Prime を起動した直後に [Create a New Project (New Project Wizard)] のボタンがあるウィンドウが表示された場合は，そのボタンをクリックする．起動直後にそのウィンドウが表示されなかった場合は，メニューの [File]-[New Project Wizard] を選択すると，New Project Wizard が起動する．

New Project Wizard では以下の手順で新規プロジェクトの設定を行う．

1. New Project Wizard のウィンドウ（図 84）に，プロジェクトを作成する作業ディレクトリ，プロジェクト名，トップモジュール（Top level design

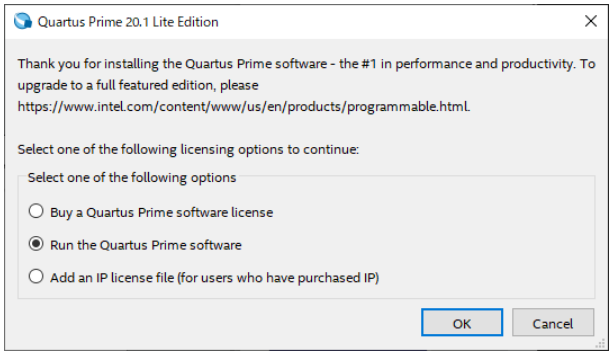

図 82　Quartus Prime ライセンスの確認ダイアログボックス

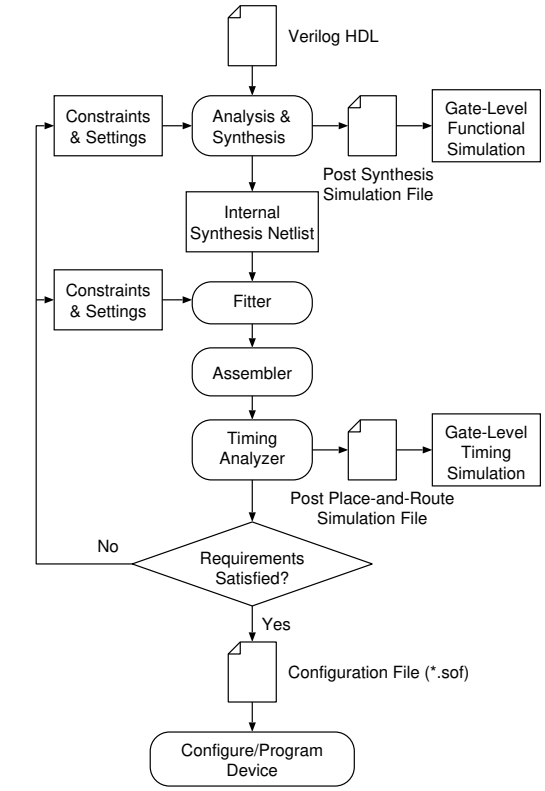

図 83　Quartus Prime デザインフロー

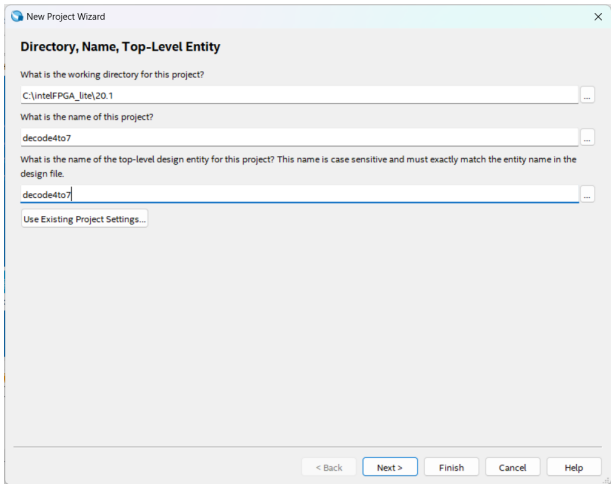

図 84　作業ディレクトリとプロジェクト名，トップモジュールの設定

[*31] Altera Corporation “Quartus II Integrated Synthesis” (2012) Figure 16-1 より抜粋.

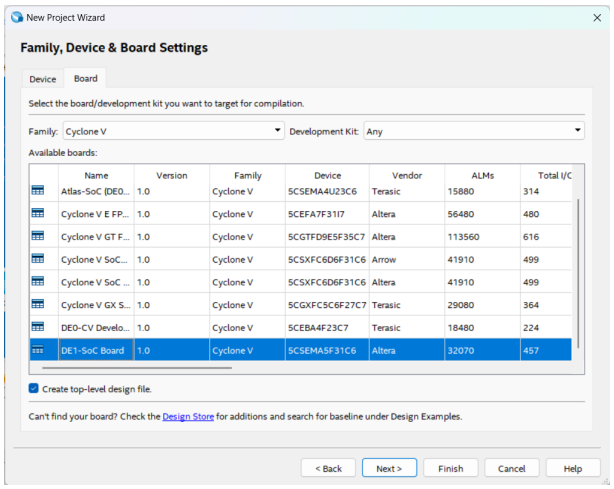

図 85　FPGA デバイスの設定

entity）の名前を入力する．**プロジェクトは，実習用作業ディレクトリ**[*32]**の下に，実習課題ごとにサブディレクトリを分けて作成すること**（複数のプロジェクトを同一ディレクトリ内に混在させると，シミュレーション時に問題が出る場合がある）．図 84 では作業ディレクトリの初期値（C:¥intelFPGA_lite¥20.1）が表示されているが，このディレクトリ下にプロジェクトを作成すると，ファイルの書き込み権限に関係する不具合が発生する可能性がある．必ず正しい実習用作業ディレクトリに修正すること．なお，プロジェクト名とトップモジュール名を一致させる必要は無いが，トップモジュールとして指定した名前のモジュールが存在しないとエラーになる．大文字と小文字の違いでも別モジュールとして認識されるので注意すること．後でトッププモジュール名を変更したい場合は，メニューから [Assignments]-[Settings] を選び，[General] カテゴリの [Top level entity] を変更する．

2. 既存のファイルをプロジェクトに追加する場合は，Add Files の画面でファイルを追加しておく．他のプロジェクトのために作成したモジュールを再利用したい場合は，ここから追加する．なお，プロジェクトを新規作成した後でもファイルを追加することは可能である．この段階で追加しない場合は，そのまま [Next] ボタンを押す．
3. 使用する FPGA デバイスを選択する．[Board] タブを開き，Family を Cyclone V とし，Available boards のリストから DE1-SoC Board を選択する（図 85）．Family の選択項目に Cyclone V が出てこない場合は，インストール時に間違った FPGA デバイスファイルを選択した可能性がある．
4. （この手順は通常の方法でインストールした場合は設定不要．後から ModelSim を単体で追加インストールした場合のみ設定する必要がある）EDA ツールの設定画面ではサードパーティ製の EDA ツールを選択する．シミュレータとして ModelSim を使用するため，ModelSim の実行ファイルへのパスが正しく設定されているか[*33]，確認しておく．
5. 最後に確認画面が表示されるので，間違いがなければ [Finish] を押して終了する．誤りがあれば [Back] で戻れるので修正する．

B.4　Verilog HDL ファイルの編集

プロジェクトに Verilog HDL のソースファイルを追加する場合は，メニューから [File]-[New] を選択して，表示されたリストボックスの中から [Verilog HDL File] を選択し，[OK] ボタンを押す．その後，新規ファイルが作成されるので編集し，[File]-[Save As] で適切なファイル名で保存する．Verilog HDL ファイルの拡張子は *.v とする．

B.5　ピンの割り当て

Verilog HDL で作成したモジュールの入出力ピンを DE1-SoC のピンと関連付けることで，ボード上の LED やスイッチを制御できるようになる．

まず最初に Quartus Prime のメニューから [Processing]-[Start]-[Start Analysis & Elaboration] を実行する．この段階でエラーが出た場合はメッセージに従って修正すること．いくつかの警告が出る場合があるが，ライセンス上制限されている機能[*34]についての警告であれば無視して構わない．それ以外の警告は必ずチェックして，修正を試みること．

次にメニューの [Assignments]-[Pin Planner] を選択し，ピンプランナを起動する．起動後のウィンドウ上部には対象としている FPGA デバイスが表示されるの

[*32] 実習用ノート PC では C:¥Users¥LDesign¥Altera¥を作業ディレクトリとして想定している．存在していなければ作成すること．このディレクトリ名に含まれる “LDesign” は Windows にログインするユーザ名である．自宅 PC など，異なるユーザ名の環境で実行する場合は，適宜読み替えること．

[*33] 通常の手順でインストールすると，ModelSim の実行ファイルは C:¥intelFPGA¥20.1¥modelsim_ase¥win32aloem に置かれている．

[*34] 実習で使用している Lite Edition（ライセンス不要）に対し，フル機能が提供される Standard Edition, Pro Edition（それぞれ有償ライセンス必要）が存在する．機能によっては Lite Edition では使えないものがある．

で，正しいデバイスが表示されていることを確認する．ウィンドウ下部にはトップモジュールの入出力ピンが一覧されている．この表の Location 列をダブルクリックすると，ピン選択のドロップダウンリストが現れるので，接続したいピンを選択するか，ピン名をキーボードから入力する．入力する場合，大文字と小文字は区別されない．また，先頭の `PIN_` は省略できる（自動的に補完される）．

すべてのピン割り当てが終了したら，ピンプランナのメニューから [File]-[Close] を選択し，ウィンドウを閉じる．

なお，ピンの割り当ては他のプロジェクトから読み込む（インポートする）ことも可能である．まず，Quartus Prime のメニューから，[Assignments]-[Import Assignments] を選択する．表示されたダイアログボックスの [...] ボタンを押して，読み込み元となるプロジェクトの設定ファイル（*.qsf）を選択後，[OK] を押す．

ピンの割り当ては，qsf ファイルからだけでなく，CSV ファイルからも可能である．CSV ファイルから読み込みたい場合は，元のプロジェクトでピンプランナを起動し，メニューから [File]-[Export] を選択する．ピンプランナから書き出した CSV ファイルは，必要に応じてテキストエディタなどで編集することができる．元のプロジェクトから信号名が変わっている場合は，一度 CSV ファイルに書き出し，編集してからインポートすればよい．

【注意】 割り当てるピンによっては，コンパイル時に “Error: Can’t place multiple pins assigned to pin location Pin_xx” のようなエラーが表示されることがある．これは指定したピンが多目的に使えるようになっており，デフォルトでは通常の入出力用途に使えない設定になっているためである．この場合は，Quartus Prime のメニューから [Assignment]-[Device] とたどり，表示されたウィンドウの中央付近にある [Device and Pin Options] ボタンを押す．ダイアログボックスが表示されるので，左側の Category から [Dual Purpose Pins] を選び，右側のリストボックスの右列をダブルクリックして [Use as regular I/O] になるように設定を変更する．

B.6 回路構成の確認

設計した回路の構成を確認するために，Quartus Prime には RTL Viewer，State Machine Viewer，Technology Map Viewer が用意されている．

B.6.1 RTL Viewer

Quartus Prime の RTL Viewer を使うと，HDL で記述した回路をグラフィカルに表示することができる．これにより，HDL で記述したコードから期待通りの論理回路が得られるか，接続に誤りが無いかを確認することができる．この段階ではまだ最適化を行っていないため，表示される内容は HDL 記述に近いものと言える．設計ミスを視覚的に発見できるため，早い段階，特にシミュレーションを行う前に RTL Viewer を使うと良い．RTL Viewer を使うためには，[Analysis & Elaboration] を実行しておく必要がある．

RTL Viewer は，Quartus Prime のメニューから [Tools]-[Netlist Viewers]-[RTL Viewer] と選択すると起動できる．

B.6.2 State Machine Viewer

Quartus Prime の State Machine Viewer は，有限状態機械（FSM）における状態遷移[*35]と遷移条件（condition）を表示する．

State Machine Viewer を使用する場合も，RTL Viewer と同様，あらかじめ [Analysis & Elaboration] を実行しておく必要がある．

State Machine Viewer の起動は，Quartus Prime のメニューから [Tools]-[Netlist Viewers]-[State Machine Viewer] と選択するか，付録 B.6.1 の RTL Viewer 内で FSM を表す黄色のブロックをダブルクリックする．

Quartus が有限状態機械として認識するには条件があり，その条件を満たしていないと State Machine Viewer には表示されない．認識する条件を以下に示す．

- 状態を識別する値は `parameter` で設定していること．
- 状態変数に対して直接整数値を代入していないこと．
- 状態遷移に算術演算を行っていないこと．
- 状態変数を出力 (`output`) にしていないこと．
- 状態変数を符号付き変数 (`integer`) にしていないこと．

B.6.3 Technology Map Viewer

Technology Map Viewer により，Analysis & Synthesis による合成結果を表示したり，設計した回路を対象デバイスのどこに配置し，どう配線するかを決定[*36]

[*35] 遷移元（source state）と遷移先（destination state）

[*36] この配置配線の処理を**フィッティング**（fitting）と呼ぶ．フィッティングを行う機構はフィッタ（fitter; 図 83 参照）であ

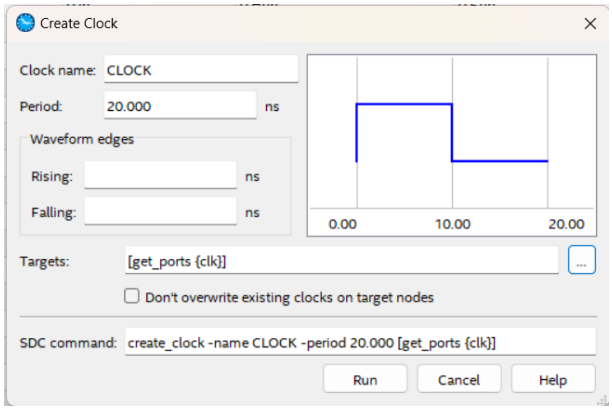

図 86　クロック信号の設定

した後の結果を表示することができる．この時点では，論理合成ツールによる最適化の結果として，ポート名が変更されたり削除されたりする場合があるので注意せよ．

Technology Map Viewer は，Quartus Prime のメニューから [Tools]-[Netlist Viewers]-[Technology Map Viewer] と選択することで起動できる．メニューにおいて，(Post Mapping) は Analysis & Synthesis を実行した後の状態，(Post Fitting) はさらにフィッティングまで実行した後の状態[*37]を表している．

B.7　タイミング制約の設定

以下では，希望するクロック周波数（**動作要求周波数**）で回路が動作するように，タイミング制約を設定する方法を示す．論理合成ツールは，ここで指定された制約に基づき，回路の構成や FPGA デバイス上での配置などを最適化する．

1. Quartus Prime のメニューから [Tools]-[Timing Analyzer] を選び，Timing Analyzer を起動する．
2. Timing Analyzer のウィンドウ左側にある Tasks リストの [Create Timing Netlist] をダブルクリックし，タイミングネットリストを作成する．
3. メニューの [Constraints]-[Create Clock] を選択する．
4. Create Clock というダイアログボックス（図 86）が開くので，[Clock name:] には CLOCK，[Period] には動作要求周波数から求めたクロック周期を指定する．
5. 同じダイアログボックス内の [Target] フィールドの右にある [...] ボタンを押す．

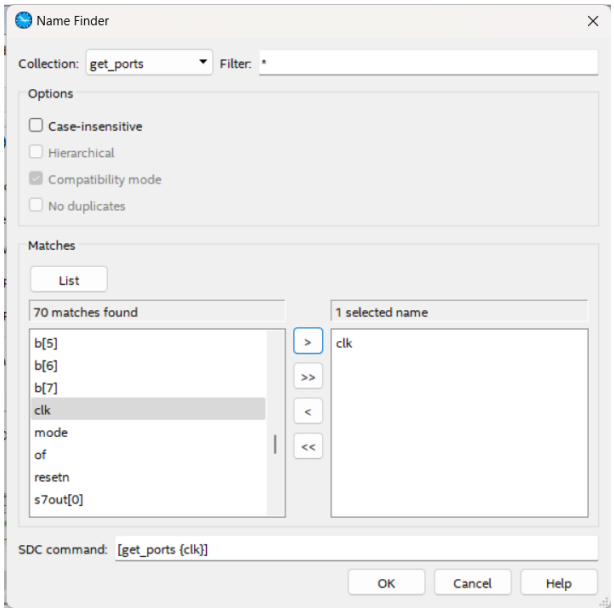

図 87　入出力信号の設定

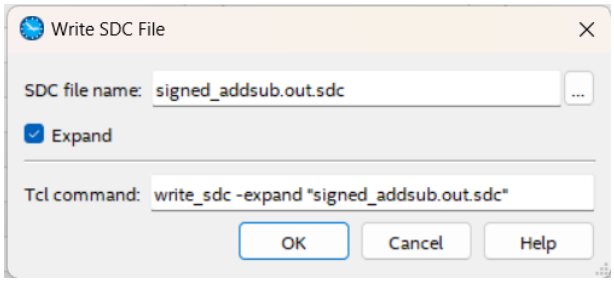

図 88　SDC ファイルへの書き込み

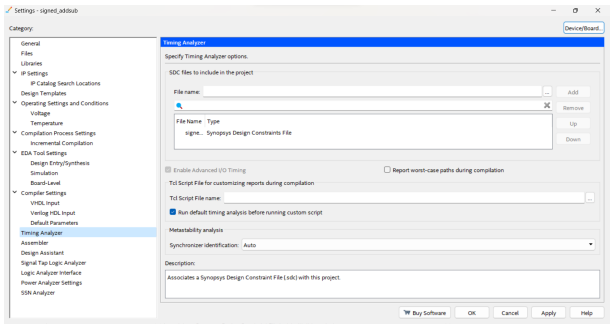

図 89　Timing Analyzer で使用する SDC ファイルの指定

6. Name Finder のウィンドウ（図 87）が開くので，中央の [List] ボタンを押す．これにより左側のリストボックスにすべてのポートが一覧される．
7. その一覧からクロック信号に該当する信号を選択し，[>] ボタンを押す．右側のリストボックスに選択した信号が現れていることを確認し，[OK] を押す．

[*37] タイミング制約が指定されていれば，それも考慮した結果．

8. Create Clock のウィンドウに戻るので，[Run] を押す．
9. メニューの [Constraints]-[Write SDC File] を選び，「プロジェクト名.out.sdc」のようなファイル名[*38]で保存する（図 88）．
10. Timing Analyzer を終了し，Quartus Prime のメインウィンドウに戻る．
11. Quartus Prime のメニューから [Assignments]-[Settings] を選び，左側の Category から [Timing Analyzer] を選択する．
12. プロジェクトにインクルードする SDC ファイルを選択する（図 89）．ここでは先ほど Timing Analyzer で作成した SDC ファイルを指定すれば良い．SDC ファイルを選択した上で，ウィンドウ中央のリストボックスに [Add] する．
13. SDC ファイルを追加したら，[OK] ボタンを押す．
14. 回路を再コンパイルする．

これにより，Quartus Prime のコンパイラとフィッタは，指定された制約を満たすような最適化（最適な論理合成と配置配線）を行うようになる．

B.8 最大動作周波数の確認

Verilog HDL のコンパイルが終了した際に表示されるレポートの左側に，「Timing Analyzer」というフォルダがある．その左側にある ▷ 記号をクリックして展開すると，「Slow 1100mV 85C Model」というフォルダがあるので，同様に ▷ 記号をクリックして展開する．そこに現れる「Fmax Summary」を選択すると，タイミング制約を満たすように合成した回路における**最大動作周波数** F_{max} がわかる．

クリティカルパスを確認する場合は，Timing Analyzer を開いて Tasks リスト（ウィンドウ上部のメニューバーではなく，ウィンドウ左側のリストであることに注意）から，[Custom Reports]-[Report Timing...] をダブルクリックする．ウィンドウが開くので，From clock と To clock のドロップダウンリストから，B.7 節の手順 4 で指定したクロック名を選択する．最後に [Report Timing] ボタンを押せば，タイミングレポートが表示される．ここで，余裕時間（slack）が最小となる経路（パス; path）がクリティカルパスを表している．右クリックして表示されるメニューから [Locate Path] を選択すれば，付録 B.6.3 の Technology Map Viewer 上で経路をグラフィカルに確認することができる．なお，slack の値が負のときは赤文字で表示され，指定したタイミング制約を満たしていないことを示す．

セットアップ時間およびホールド時間の slack を確認する場合は，Tasks リストから [Reports]-[Slack]-[Report Setup Summary] または [Report Hold Summary] をダブルクリックすればよい．ここで表示される [Slack] 列が，セットアップ時間およびホールド時間に対する Slack（ナノ秒単位）を表している（セットアップ時間やホールド時間そのものではないことに注意）．Slack 列が黒字（正の値）であれば，タイミング制約を満たしていることを示している．

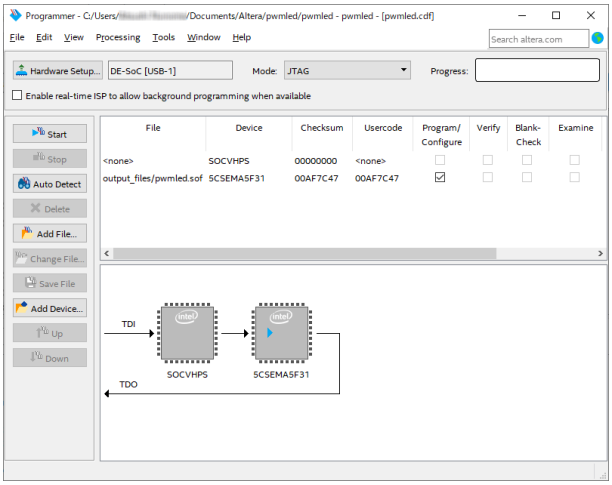

図 90　FPGA プログラムダイアログボックス

B.9 コンパイルと FPGA ボードへの転送

Quartus Prime のメニューから [Processing]-[Start Compilation] を選択すると，配置・配線やタイミング解析を含むフルコンパイルが始まる．ここでもエラーや重大な警告[*39]が出た場合は修正すること．

コンパイルが正常終了すると，Compilation Report が表示される．使用した ALM 数は “Logic utilization (in ALMs)” の項目で確認できる．生成した回路の構成情報（configuration bit stream）は，次の手順で FPGA ボードに転送する．

1. FPGA ボードに AC アダプタが接続してあることを確認する．ここではまだ電源は入れないこと．
2. USB ケーブルを DE1-SoC の USB Blaster II ポートに接続する．USB Blaster II ポートの位

[*38] SDC とは Synopsys Design Constraints の略で，米国 Synopsys 社の設計ツールで使われる制約条件ファイル（テキスト形式）である．タイミング制約を記述する業界標準フォーマットとして広く使われている．

[*39] ウィンドウ下部の “Critical Warninag” タブをクリックして確認せよ．

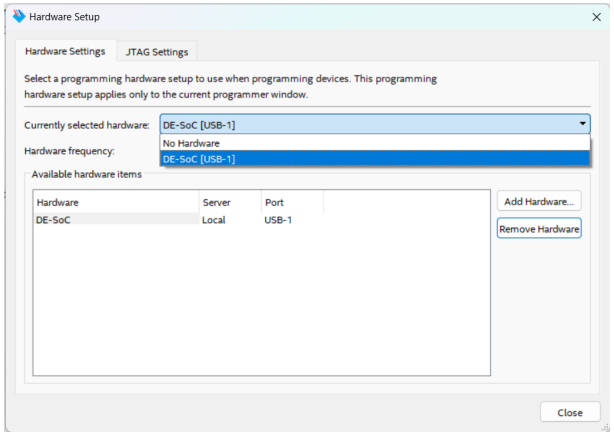

図 91　FPGA ハードウェアセットアップ

置は 4 ページの図 4 を参照せよ．

3. 電源スイッチ（4 ページの図 4 左側の Power ON/OFF）を押す．この時点では構成情報は転送していないので，デフォルト動作（DE1-SoC のデモンストレーション）が始まる．
4. Quartus Prime のメニューから [Tools]-[Programmer] を選び，Programmer を起動する（図 90）．
5. Programmer のウィンドウ内にある [Hardware Setup] ボタンを押し，DE-SoC を選択する（図 91）．Programmer ウィンドウに戻り，[Hardware Setup] ボタンの右側に，選択した DE-SoC が表示されていることを確認する．
6. Programmer の Mode: 設定を JTAG にする（既定値）．
7. Programmer ウィンドウの右下部に SOCVHPS と SCSEMA5F31 のチップが直列に接続されているかを確認する．
   - SOCVHPS しか表示されていなければ，[Auto Detect] ボタンを押す．Device の選択画面が表示されるので，5CSEMA5 を選択する．
   - SCSEMA5F31 しか表示されていなければ，一度 SCSEMA5F31 を delete してから，[Auto Detect] ボタンを押す．Device 選択画面で 5CSEMA5 を選択する．
8. Programmer ウィンドウ中央部にあるリストに転送する構成情報ファイル（*.sof）が 5CSEMA5F31 のデバイスに設定されていること，Program/Configure 欄にチェックが入っていることを確認する．
   - 5CSEMA5F31 の File 欄が none の場合は，右下の 5CSEMA5F31 のチップの図を右クリックし，Change File を選択する．その後，プロジェクトの output_files ディレクトリ内にある*.sof ファイルを指定する．最後に Program/Configure 欄にチェックを入れる．
9. Programmer ウィンドウ左側の [Start] ボタンを押して DE1-SoC に構成情報を転送する．
10. Programmer ウィンドウを閉じる．

FPGA ボード上で動作確認が終わったら，FPGA ボードの電源スイッチを押して電源をオフにする．転送した構成情報は電源をオフにすることで消去される．

付録 C　Quartus Prime でのシミュレーション手順

Quartus Prime には ModelSim というシミュレーション環境が用意されている．入力波形の設定や出力波形の確認は，Simulation Waveform Editor というツールを使い GUI 上で行う．

1. シミュレーションを行う前に，Quartus Prime で作成しているプロジェクトをコンパイル（[Processing]-[Start Compilation]）しておく．ここでエラーや重大な警告が見つかれば修正する．
2. テストベクタを作成するために，[File]-[New...]-[University Program VWF] を選択する（図 92）．Simulation Waveform Editor が起動する．
3. メニューの [Edit]-[Set End Time...] から，シミュレーションの終了時刻を設定する．
4. メニューの [Edit]-[Insert]-[Insert Node or Bus] を選択する．「Insert Node or Bus」ダイアログ（図 93）が開くので，[Node Finder...] ボタンを押す．環境によっては Node Finder ウィンドウが開かないことがある．表示されるエラーメッセージにしたがっても Node Finder ウィンドウが開かない場合は，明示的に [Name:] 欄にノード名を入力し，手順 6 へ進む．
5. Node Finder ウィンドウが開くので，[List] ボタンを押した後，左側のノード一覧から，観測したいノードを選択し，[>] を押して右側のリストボックスに追加する（図 94）．[OK] を押すと，Waveform Editor ウィンドウに選択したノードが表示されている．
6. 信号波形を変更したい部分をマウスドラッグで選択し，[Edit]-[Value] メニューまたはツールバーのボタンを押して，希望の値に変更する．クロック信号を編集する場合は，入力信号名を選択した後，[Edit]-[Value]-[Overwrite Clock...] からクロック周期とデューティ比の設定を行う．また，信号名を右クリックした後，[Radix] を選ぶと，値の表示形式を選択できる．例えば，[Radix]-[Hexadecimal] を選ぶと 16 進数表記，[Radix]-[Signed Decimal] を選ぶと符号付き 10 進数表記で表示される．信号の用途に応じて適切な表記を設定すること．
7. すべての入力信号の設定が終わったら，[File]-[Save] から設定を保存する．ファイルの拡張子は *.vwf とする．
8. Waveform Editor の [Simulation]-[Simulation Settings] から，シミュレーションの詳細設定を行う．ほとんどの項目はデフォルトのままで良い．HDL Language が Verilog（デフォルト値）になっていることを確認する．Quartus Prime のバージョンによっては vsim コマンドの-novopt オプションがエラー扱いされる[*40]ため，この後の手順でエラーが表示された場合は，ModelSim Script 欄の 5 行目付近にある vsim から始まる行から -novopt を削除する（図 95）．削除した場合は右下の [Save] ボタンを押して保存する．
9. 実行するシミュレーション方式（想定した論理機能が実現できているかを確認する**機能シミュレーション**（functional simulation），または素子と配線の遅延時間を考慮する**タイミングシミュレーション**（timing simulation））に合わせて，メニューの [Simulation]-[Run Functional Simulation] または [Simulation]-[Run Timing Simulation] を選択し，シミュレーションを実行する．
10. シミュレーションが正常終了すると，Simulation Waveform Editor が起動し，出力波形の確認ができる．このウィンドウ内の波形は read only であり，変更はできない．

なお，入力波形における値 X と，出力波形における値 U は不明（unknown）を表している．

[*40] 過去のバージョンで使われていた-novopt オプションは最近のバージョンで廃止されたため，新しいバージョンではエラーになる．

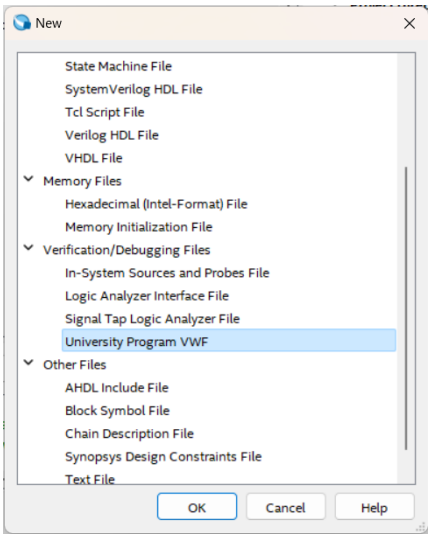

図 92　VWF ファイルの新規作成

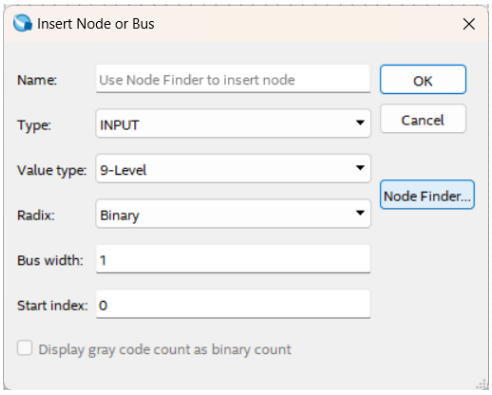

図 93　ノードとバスの設定

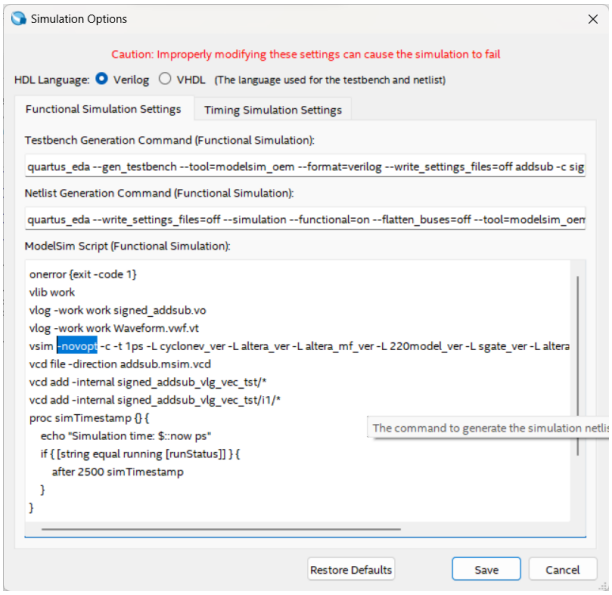

図 95　ModelSim 起動スクリプトの修正

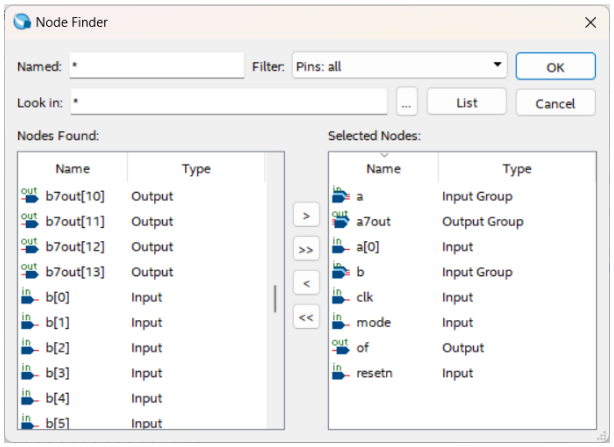

図 94　ノードファインダによる観測対象ノードの指定

付録 D　iverilog と gtkwave によるシミュレーション

ここでは Quartus Prime 以外のシミュレーションツールである iverilog と波形ビューア gtkwave を紹介する．iverilog (Icarus Verilog) は Icarus 社の Verilog シミュレーションツールであり，Windows/macOS/Linux/FreeBSD 向けのパッケージやソースファイルが公開されている．iverilog はテスト対象の Verilog HDL ファイルを入力とし，シミュレーション実行用ファイル[*41]を出力する．シミュレーション実行用ファイルを起動する（または vvp へのコマンドライン引数に与えて実行する）と，シミュレーション結果を vcd 形式のダンプファイルに出力する．gtkwave は，このダンプファイルを読み込んで，信号波形をグラフィカルに表示する．gtkwave も iverilog と同様に，多くの OS をサポートしている．

iverilog と gtkwave によるシミュレーションの流れを図 96 に示す．テスト対象の Verilog HDL ファイルは，どのタイミングでどのような入力をテスト対象に与えるかを記述した**テストベンチ**と共に iverilog へ入力し，シミュレータのファイルを得る．シミュレータはランタイムエンジン vvp によって実行され，シミュレーション結果を vcd 形式で出力する．gtkwave はこの vcd ファイルを読み込み，グラフィカルに表示する．

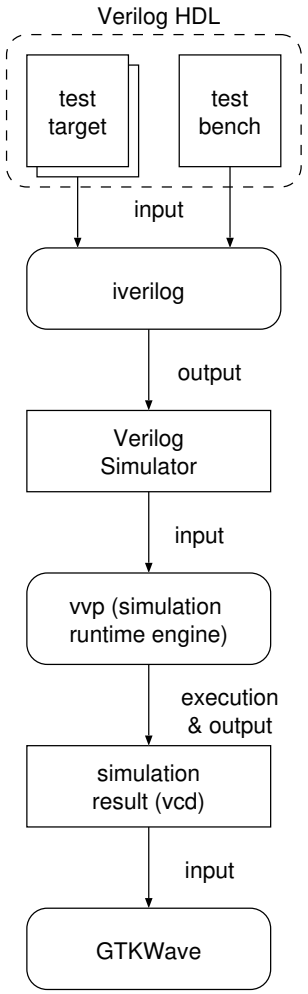

図 96　iverilog と gtkwave によるシミュレーションフロー

D.1　ダウンロードとインストール

iverilog および gtkwave は次の URL からダウンロードできる．

```
iverilog  https://steveicarus.github.io/
      iverilog/
gtkwave  http://gtkwave.sourceforge.net
```

各 OS への具体的なインストール手順はネットで調べてもらいたい．

D.2　テストベンチの準備

Quartus Prime では Simulation Waveform Editor を使って入力波形をグラフィカルに設定するが，iverilog を使う場合は Verilog HDL の記述を用いてテキスト形式でテストベンチを作成することになる．

テストベンチは，テスト対象の回路記述と同じファイルに記述することも可能である．ただし，テストベンチの記述はシミュレーションの段階だけで必要になるもので，最終的な回路の論理合成時には不要なものである．よって，テストベンチは独立したファイルに分離して記述することを推奨する．

テキスト 11 ページの図 35 をシミュレーションするためのテストベンチ例を図 97 に示す．

テストベンチを記述する際の注意点を以下に示す．

- テストベンチも 1 つのモジュールとして記述する．テストベンチ自体には入出力が無いため，モジュール名直後のポートリストは記述しない．
- 入力信号は `reg`，出力信号は `wire` として宣言する．
- 1 行目の ‘timescale によって，遅延時間の単位とシミュレーション精度を設定できる．数値は 1, 10, 100，単位は s, ms, us, ns, ps などが指定できる．
- #によって遅延時間を指定できる．ここで指定する時間は直前の文からの相対時間であり，絶対時刻を表していないことに注意せよ．

[*41] バイナリの実行可能ファイルではなく，vvp というランタイムエンジンへの入力ファイルを出力する．vvp がこのファイルを読み込んで，実際のシミュレーションを行う．

```
‘timescale 1ns/100ps

module test_bc;
  reg Clk, Clr;
  wire [3:0] out;

  binary_counter bc (Clr, Clk, out);

  initial begin
    $dumpfile("out.vcd");
    $dumpvars(0, bc);
    $monitor("%t: Clk=%b, Clr=%b, out=%b",
      $time, Clk, Clr, out);
  end

  initial begin
    #0  Clr = 1;
    #10 Clr = 0;
    #5  Clr = 1;
    #5000 $finish;
  end

  initial begin
    #0 Clk = 0;
  end

  always begin
    #50 Clk <= ~Clk;
  end
endmodule
```

図 97　テストベンチの例

- テスト対象のモジュールは明示的にインスタンス化する．
- initial ブロックはシミュレーション開始時に一度だけ実行される（initial ブロックは複数個記述可）．この中でテスト対象の回路に与える信号（テストベクタ）を規定する．initial ブロック内の記述は，論理合成の対象外になることに注意せよ．
- `$dumpfile` により出力するダンプファイル名（vcd 形式）を指定する．
- `$dumpvars` でダンプ対象を指定する．第 1 引数で階層レベル，第 2 引数以降でダンプ対象モジュールのインスタンス名を指定する．階層レベルは，0 で対象の全階層，$n(n > 0)$ で最上位から第 n 階層までの意味になる．
- `$monitor` を使うと，指定した信号が変化するたびにフォーマット化されたメッセージを表示できる．フォーマット文字列を表 16 に示す．

表 16　$monitor システムタスクでのフォーマット文字列

書式指定子	意味
`%t`	シミュレーション時刻
`%b`	2 進数表示
`%o`	8 進数表示
`%d`	10 進数表示
`%h`	16 進数表示

- `$time` は現在のシミュレーション時刻を表す．
- `always` 文を使って，周期的に変化するクロック信号を記述できる．`always` 文も複数個記述することができ，それぞれが並列に評価される（並列動作する）．

図 97 では 1 行目の ‘timescale 指定により，1 [ns] を単位として 100 [ps] の精度でシミュレーションを行うようにしている．つまり，テストパターン中に記述された遅延時間 #1 が 1 [ns] を表し，論理合成ライブラリ内でより細かい遅延時間が設定されていても，100 [ps] の精度に丸められる．ここでは説明のためにより細かい精度を設定しているが，機能シミュレーションが目的であれば遅延時間単位と同じ 1 [ns] に設定して差し支えない．また，initial ブロックではシミュレーション開始 10 [ns] 後に Clr 信号を 0 にし，その 5 [ns] 後に 1 に戻している（リセット操作）．シミュレーションはリセット操作完了の 5000 [ns] 後で終了（`$finish`）するようにしている．クロック信号（Clk）は 50 [ns] ごとに High, Low と交互に変化させている（クロック周期 100 [ns]）．

D.3　iverilog の実行方法

コマンドラインで以下のオプションを付けて iverilog を実行する．

`iverilog -o` 出力ファイル名 `-s` トップモジュール名 `verilog` ファイル（複数可）

例えば，D.2 節で示した例では，

```
iverilog -o bincounter -s test_bc
bincounter.v testbench.v
```

のように実行することで，2 進カウンタの定義（bincounter.v）とテストベンチ（testbench.v）を入力として，シミュレータファイル bincounter が出力として得られる．

次に，得られたシミュレータを実行する[*42]．この例ではシミュレータが bincounter というファイル名で得

[*42] 正確にはランタイムエンジン vvp に対してシミュレータ bincounter を与えて実行する．

```
VCD info: dumpfile out.vcd opened for output.
                   0: Clk=0, Clr=1, out=xxxx
                 100: Clk=0, Clr=0, out=0000
                 150: Clk=0, Clr=1, out=0000
                 500: Clk=1, Clr=1, out=0001
                1000: Clk=0, Clr=1, out=0001
                1500: Clk=1, Clr=1, out=0010
                2000: Clk=0, Clr=1, out=0010
                2500: Clk=1, Clr=1, out=0011
                3000: Clk=0, Clr=1, out=0011
（以下略）
```

図 98　iverilog の実行例

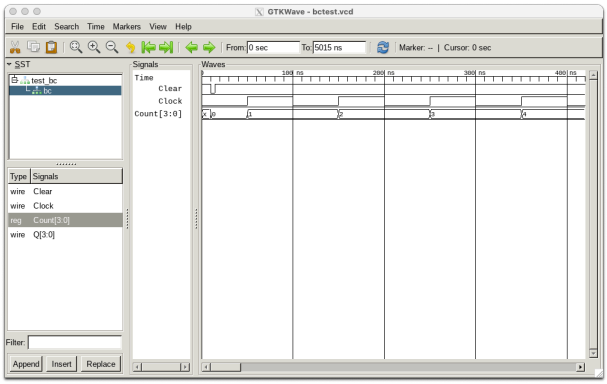

図 99　gtkwave の表示例

られているので，コマンドラインから

```
vvp bincounter
```

と入力して実行する．実行すると，図 98 のような出力がコンソールに表示される．

ここでは 100 [ps] 精度でシミュレーションしているため，`$monitor` 文によって表示している行の 1 列目は 100 [ps] 単位となっている．

D.4　gtkwave による波形表示

テストベンチにおいて`$dumpfile` 文で指定したダンプファイルに，信号波形に関する情報が出力されている．このダンプファイルを例えば out.vcd とすると，以下のように gtkwave の入力として起動することで波形表示ができる．

```
gtkwave out.vcd
```

gtkwave の表示は図 99 のようになる．gtkwave のウィンドウ内左上にモジュール構成がツリー表示されているので，観測したいモジュールを選択する．そうすると，その下の領域にそのモジュールに関連する信号が表示されるので，観測したい信号を選んで右側ウィンドウに追加する．これで観測したい信号波形が右側ウィンドウに現れるので，ズームイン・ズームアウトボタン（虫眼鏡の形）で適切な表示スケールを選択せよ．

索引