EDA ツールを用いた論理回路設計

実験概要

本実験では，**EDA ツール**（Electric Design Automation ツール，電子回路設計支援ソフトウェア）を用いた**ディジタル LSI** の設計を行う．また本実験では，計算機上での LSI 設計に加え，書き換え可能な論理素子である **FPGA**（Field Programmable Gate Array）を用いた回路の動作実験を行う．

実験と実験課題を通して，論理回路設計，論理シミュレーション，論理合成などの論理回路設計における各段階の基本的な技術を習得する．

実験スケジュール

本実験は全 2 週で，以下のようなスケジュールで行う．

第 1 週（予習：本指導書の 1 章，2 章，3 章，4 章）

ハードウェア記述言語の一つである Verilog HDL による簡単な組合せ回路の設計と，設計した回路の動作実験を行う．回路の動作実験では，FPGA を搭載した実験基板を用いて設計した回路を実際に動作させる．

- 実験 1（簡単な組合せ回路の設計と動作実験）
- 実験 2（加算を行う組合せ回路の設計，組合せ回路の最適化）

第 2 週（予習：本指導書の 2 章，5 章）

Verilog HDL による**組合せ回路**と**順序回路**の記述，EDA ツールを用いた設計制約を考慮した回路の**最適化**についての実験を行う．また，実験課題として，2 進化 10 進（BCD: Binary Coded Decimal）カウンタの設計，系列検出器（有限オートマトン）の設計を行う．

- 実験 3（加算を行う順序回路の設計，順序回路の最適化）
- 実験課題 1（BCD カウンタの設計，実験課題 2 との選択）
- 実験課題 2（系列検出回路の設計，実験課題 1 との選択）

指導書の構成

1 章ではディジタル LSI の設計フローについて述べ，2 章では Verilog HDL による回路記述について述べる．3 章では本実験で用いる EDA ツールの使い方を示す．4 章，5 章では，第 1 週目，第 2 週目に行う EDA ツールを用いた回路設計「基礎編」,「中級編」について述べる．6 章では第 2 週目に行う実験課題を示し，7 章では**レポート**について述べる．8 章は実験基板のピン配置ファイル，クロックの使い方，回路の記述例の付録である．

実験の進め方

実験は，2〜3 人 1 組（各班 2 組で構成）で行う．各組ごとに全ての実験を行う．実験で使用する機器については，班内の組同士で共有する．

実験課題目次

目 次

作成者: 中村 一博，高木 一義
最終更新日: 2012 年 8 月 29 日
第 3.10 版

1　はじめに

1.1　ハードウェア記述言語を用いたディジタル LSI 設計

現代の生活では，多種多様の電気・電子機器が身の回りに存在しており，それらの機器には LSI 回路が多数搭載されている．パソコンやゲーム機に搭載されている CPU，メモリ等のみならず，各種家庭電化製品，音声・画像機器においてもディジタル化が進み，制御，データ処理等の用途で LSI は不可欠なものとなってきている．また，LSI の高集積化，低消費電力化により，携帯情報端末なども実現可能になった．

このような状況は，近年の LSI の製造技術の進歩により，LSI に搭載できる回路の規模が増大し，高度で多様な機能を実現できるようになったため可能になった．LSI の**製造技術**とは，物性・デバイスの技術，微細加工技術である．しかし，回路が大規模で複雑になれば回路設計も複雑で困難なものとなる．製造技術とともに，どのような考え方で回路を構成するか，すなわち，LSI の**設計技術**の進歩もまた，必要不可欠なものである．

初期の LSI 回路設計では，レイアウトのマスクパタンを人手で描いて設計が行われた．その後，レイアウトの自動化が可能になり，トランジスタレベルあるいは論理レベルの回路図を描いて回路を設計する手法がとられた．CAD（Computer Aided Design）システムの発達により，ソフトウェアによる回路図入力の支援や回路シミュレーションが可能となったが，回路が大規模になるに従って，このようなレベルで人間が考えて設計することは困難となる．現在実現可能な，数百万，数千万ゲート規模の回路を，人手で設計することはほぼ不可能であろう．

ソフトウェアの設計において，アセンブラでコードを書いていた時代から，高級言語による設計へと移り変わっていったのと同様に，ハードウェアの設計においても，より抽象度の高い設計入力へと移り変わっていったのは自然な流れである．すなわち，**ハードウェア記述言語**（HDL: Hardware Description Language）を用いた設計フローへの変遷である．

ハードウェア記述言語は，回路の機能を，ソフトウェアのプログラムと同様，動作記述のレベルでプログラムテキストとして記述できる．ハードウェア記述言語は，ハードウェアの**仕様記述**のための言語としての機能を持っていた．即ち，ハードウェア記述言語で記述された仕様と，別に設計された回路との動作の比較を行い，期待された動作を確認する役割を持っていた．しかし，**論理合成**システムと呼ばれる，ハードウェア記述を自動的に回路記述に変換するシステムが実用化されたことにより，ハードウェア記述言語は，ハードウェアの**設計記述**としての役割を持つこととなった．論理合成システムは，ソフトウェアの高級言語におけるコンパイラと同様の役割を持つシステムであり，その発展は，大規模回路の設計期間短縮，設計生産性の向上に大きく寄与した．

特にディジタル回路の設計においては，ハードウェア記述言語による回路設計が現在の主流である．アナログ，高周波回路の設計においては，回路図あるいはレイアウト図を用いた設計が現在でも重要であるが，記述言語による設計フローも発展しつつある．今後は，オブジェクト指向型の設計技術，システムレベルの設計技術の発展が重要になると考えられる．

1.2　ハードウェア記述言語

現在では，HDL により論理回路をレジスタ転送レベル（RTL: Register Transfer Level）で記述し，設計を行うことが多くなっている．HDL はハードウェアの仕様を記述する言語であると同時に，設計を記述する言語でもある．広く普及している HDL としては，VHDL, Verilog HDL が挙げられる．

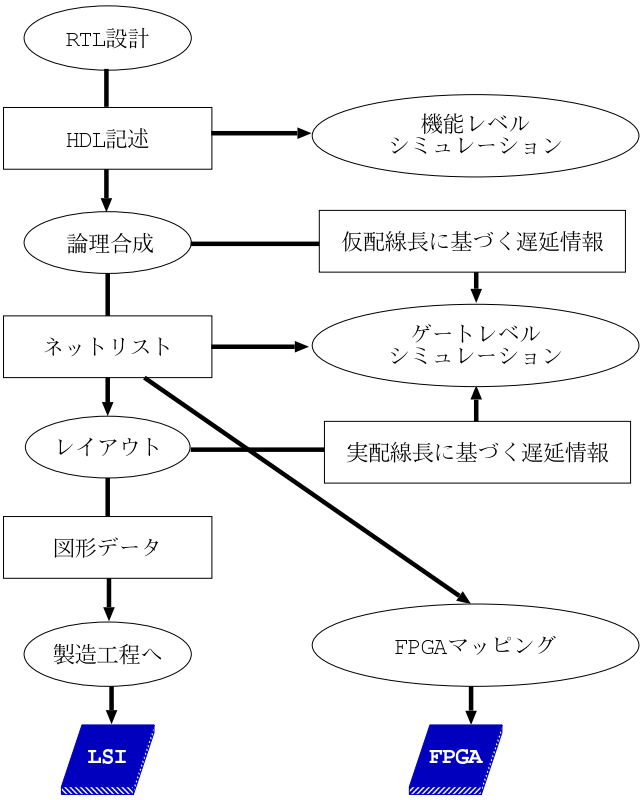


図 1: LSI の設計フロー

VHDL は，米国国防総省の VHSIC（Very High Speed Integrated Circuit）プロジェクトで，ハードウェアの記述言語（VHDL: VHSIC Hardware Description Language）として採用されたものであり，HDL の一つの標準規格である．VHDL は Ada に似た構文を採用している．Verilog HDL は Cadence 社の論理シミュレータ Verilog XL 用の言語として普及してきた．Verilog HDL は C 言語の文法要素を多く採用している．

VHDL は，IEEE Std-1076（VHDL87）及び Std-1164（VHDL93）として早い時期から規格化されている．一方，Verilog HDL はシミュレーション用の言語として事実上の業界標準であったが，IEEE Std-1364 として改めて規格となった．

国内の設計システムとしては，NTT による，記述言語 SFL を用いた LSI 設計システム PARTHENON が挙げられる．PARTHENON と SFL は実際の LSI 設計の実績もあり，研究用，大学等での教育用としても広く使用されてきた．

日本電子工業振興協会の LSI 設計用記述言語標準化委員会で策定された HDL である UDL/I は，処理系がフリーソフトウェアとして配布されており，シミュレーション及び合成ツールが入手可能である．

今後はより上位のアルゴリズムレベルの記述が一般的になっていくと考えられる．現在では，C, C++, Java 等をベースにしたハードウェア設計記述や環境が研究され，すでに実用化されているものもある．

1.3 ディジタル LSI の設計フロー

図 1 に LSI の設計フローと，用いられるツールを示す．設計の工程はいくつかの段階に分けられ，それぞれの設計段階でのハードウェアの表現が存在する．設計フローは，上位の設計段階での回路の表現を，下位での表現に等価変換していく工程である．

1.3.1 回路の動作記述

ハードウェア記述言語を用いて，回路の動作を記述する．データの値を保持する変数（レジスタ）と，それらの間の演算やデータ転送の流れと制御を，言語記述によって指定する．これらはそれぞれ記憶素子，組合せ回路として実現されることになる．このようなレベルでの設計は，レジスタ転送レベル（RTL: Register Transfer Level）設計，あるいは機能設計と呼ばれる．

ソフトウェアと違い，書き方によっては論理合成が不可能な記述となるため，最終的なゲート回路の構造を意識して記述することが重要となる．

1.3.2 機能レベルシミュレーション

設計した回路が要求されている機能を満たしているかどうかを確認する．機能レベルでのシミュレーションを行うことにより，処理手順や論理の誤りを早期に発見することが可能となる．また，一般に，より上位の設計段階でのシミュレーションはより高速に行うことができるため，この点でも機能レベルでシミュレーションを行うメリットがある．

また，回路を実際に動作させる入出力などの環境（テストベンチ）もハードウェア記述言語で記述することが可能であり，柔軟で汎用的なシミュレーション環境の構築が可能である．

1.3.3 論理合成

ハードウェア記述言語で記述した機能を，記憶素子や論理ゲート等の回路素子に置き換え，機能レベルの記述からゲートレベルの記述に変換する．

出力であるゲートレベルの記述は，ネットリストと呼ばれる，論理回路図と等価なテキスト表現であり，実際のゲート等の部品間の接続を記述したものである．

論理合成システムでは，AND，OR，NOT，フリップフロップなどの**基本素子**（**セル**）はあらかじめ設計されているものとして，これらを部品として使った回路を合成する．基本素子の面積，動作速度，消費電力などの設計情報を蓄えておくデータベースはセルライブラリと呼ばれる．LSI 製造のテクノロジ毎に異なるセルライブラリを用いることにより，各テクノロジに対応した回路を合成できる．

論理合成においては，回路全体の面積，動作速度，消費電力などのパラメータを設計者がきめ細かに指定できる．条件を変えて合成することにより，要求を満たす回路が実現可能かどうかを模索する工程が繰り返される．例えば，回路面積の上限を制約条件として，動作速度を最適化するなどの指定をするのが一般的である．ソフトウェアのコンパイルにおいても，性能を左右するコンパイルオプションをいくつか試すことは行われるが，ハードウェアの場合は，出力された回路の性能への要求がより明確であることが多いため，この過程はより重要となる．

1.3.4 ゲートレベルシミュレーション

セルライブラリに記述されている，各セルの機能，面積，動作速度，消費電力などの情報を用いて，合成されたゲートレベルの回路が正しく機能を実現しているかを確認する．配線に起因する回路遅延時間は，回路面積の見積りなどからおおよその配線長を見積ることにより推測される．

機能レベルでの記述が合成後の回路を意識して書かれていない場合，論理合成によって動作が変わってしまうことがあり得るので，ここでのチェックが必要となる．また，機能レベ

ルでのシミュレーションに比べ，各セルの実現方法と接続関係がより明確であるため，回路の動作速度などの性能をより正確に見積もることができる．

1.3.5 レイアウト設計

LSI 上に実際に作成されるレイアウトのマスクパタンを生成する．レイアウトを設計入力とし，図形エディタ上で各素子を描画していくフルカスタム設計も行われているが，セルライブラリに記述されている基本素子のレイアウトと，ネットリストに記述されている素子の接続関係の情報から，回路全体のレイアウトを半自動的に生成するシステムが広く用いられるようになってきている．セルベースのレイアウト設計では，各素子の位置を決定する配置の段階と，素子間の結線の経路を決定する配線の段階の 2 段階で構成される．

近年では，素子の動作速度に比べ，配線での信号遅延の割合が大きくなってきているため，レイアウトの品質は回路の品質を大きく左右する．

1.3.6 レイアウト後のシミュレーション

セルライブラリの情報に加え，レイアウトの情報から配線長などを抽出し，設計された回路の機能とタイミングを検証する．レイアウト設計により，素子の面積や配線長などが全て決まるので，素子遅延や配線遅延をここで初めて正確に算出することが可能になる．基本的にはゲートレベルシミュレーションと同様であるが，タイミングを含めてより正確なシミュレーションが可能である．

また，より正確な評価のために，マスクパタンからトランジスタ回路を抽出し，トランジスタの動作モデルを用いてシミュレーションを行うことも多い．

1.3.7 FPGA を用いたプロトタイピング

レイアウト設計までで LSI の設計フローは完了するが，回路の動作検証，評価がシミュレーションだけでは不十分であったり，LSI の製造を待たずに周辺の回路と組み合わせた検証，評価をしたい，などの場合には，プロトタイプを作成して動作検証を行うことがある．このために，書き換え可能な論理素子である FPGA（Field Programmable Gate Array）が用いられる．

FPGA は，設計者がその場でプログラムすることが可能な論理素子であり，多くのものは機能の書き換えを何度も行うことができる．FPGA は，機能を書き換え可能な論理ブロックと，組み換え可能な論理ブロック間の配線から成る．FPGA 向けの回路設計では，LSI の設計と同様，ハードウェア記述言語やネットリストを入力とし，各社の FPGA それぞれに専用のマッピングツールを用いて回路が FPGA の構成データに変換される．

FPGA を用いたプロトタイピングでは，設計した LSI のタイミングなどの正確な評価はできないが，ソフトウェアによるシミュレーションに比べ，実機に近い環境でのより高速な動作評価が可能である．

2 Verilog HDL による回路動作記述の基礎

2.1 Verilog HDL 記述の基本構造

Verilog HDL の基本構造や構文要素は，C 言語に類似している．図 2 に，2 入力 1 出力のセレクタ回路の Verilog HDL 記述を示す．この回路は，セレクト入力 S1 が 0 か 1 かによって，データ入力 D0 あるいは D1 の値を Y に出力する．この記述を例に，Verilog HDL の基本構造を見ていく．

- “/* */” で囲んだ部分，及び，“//” から行末まではコメントである．
- Verilog HDL ではモジュールが一つの設計の単位になる．一つのモジュールの宣言は，module，endmodule で囲まれる．
- まずモジュール名 mux21 を定義し，ポート（入出力インタフェース）を ( ) 内に記述する．続いて，各ポートの型 (入力，出力，ビット数) を宣言する．C 言語（ANSI 以前の K&R 風）の関数名，引数と同様である．
- この例にはないが，内部で用いる変数や内部で呼び出すモジュールもここで宣言する．
- 残りはモジュールの動作の本体の記述である．この例では，S1，D0，D1 のいずれかが変化した時に，Y への代入の右辺を計算し，Y へ代入している．代入文中の ~，&，| は論理演算子である．この例のような論理演算による単純な代入は組合せ論理回路の記述となる．

Verilog HDL 記述のファイル名は *.v とすることが多い．ツールによってはこれは必須で，更にモジュール名とファイル名が同じである必要があるものもある．

2.2 基本的な構文と意味

Verilog HDL のデータの種類には，入出力ポートの他にレジスタとネットがある．レジスタは reg, ネットは主に wire として宣言する．reg は記憶素子，wire は配線素子を表わすが，reg として宣言してもフリップフロップが生成されるとは限らない．

```
/*                    *
 * mux21.v            *
 * 2-1 マルチプレクサ  *
 * (2-1 セレクタ回路)  *
 *                    */

module mux21 (S1, D0, D1, Y);  // 入出力ポート
   input  S1, D0, D1;          // 入力 S1, D0, D1
   output Y;                   // 出力 Y

   // Multiplexer body
   // Y = ((not S1) and D0) or (S1 and D1)
   assign Y = (~S1 & D0)       // 出力ポートに対する
            | ( S1 & D1);      // 代入は assign 文で行う
endmodule
```

図 2: 2-1 セレクタ回路

演算子等	意味	使用例
&, |	bitwise AND, OR	a & b
~	bitwise NOT	~a
^	bitwise XOR	a ^ b
{ }	ビット連接	{a, b}
&, |, ^	全桁の AND, OR, XOR	&a は a[n] & a[n-1] & ... & a[0]
~&, ~|	全桁の NAND, NOR	~& a は ~(a[n] & a[n-1] & ... & a[0])
+,-,*,/,%	算術演算	a + b
==, !=, ===	論理等価，非等価演算	a == b
>, >=, <, <=, =	算術比較演算	a > b
? :	条件演算	(a>=b) ? a : b
<<, >>	シフト演算	a << 2
[n:m]	部分ビット参照/代入	a[3:2] は {a[3], a[2]}
数字	10 進定数	0, 1, 29
n'bxx	n ビット 2 進定数	2'b01
n'hxx	n ビット 16 進定数	12'ha5a

図 3: Verilog HDL の演算子等

input，output，inout，reg，wire などはデータ型と呼ばれるが，この区別はデータの扱われ方を示すものであり，Verilog HDL では，データがとり得る値としての型の厳しい区別はない．基本となる 1 ビットのデータは 0，1，x，z の 4 値を持つ．x は不定，z はハイインピーダンスを表わす．

多ビットのデータは，型名の直後に [3:0] のようにレンジを指定する．

演算子は C 言語とほぼ同じのものが使える．多ビットのデータはビットベクトルであるとともに，符号なし整数として扱われる．例に示した論理演算の他に，図 3 に示すような演算がある．C 言語にない演算子としては，連接 （{a，b}） とリダクション（& a ，| a など）が挙げられる．

配列の宣言は，データ名の後に [0:255] などのように記述する．前述の，wire，reg などの型名の後に記述するレンジとは扱いが異なり，配列要素のビットを並べて整数として扱うことはできない．

wire に対する代入は assign 文で行う．これは組合せ回路の記述となる．

条件付き代入は，?: 演算子や，always 構造内での if，case，casex などの構文で記述できる．

2.3 always ブロック

Verilog HDL では，逐次的に解釈されるひとまとまりの機能を always ブロック構造の中に記述する．always ブロックの中では if などの制御構造が記述できる．図 2 の Y への代入文を always ブロックで記述すると以下のようになる．

```
always @(S1 or D0 or D1) begin
  Y = (~S1 & D0) | ( S1 & D1);
end
```

always の次に記述されている @( ) は，イベント制御と呼ばれる．( ) 内のイベント式には，変数名，posedge 変数名，あるいは negedge 変数名 といった表現を or でつないで列挙する．このリスト中の変数が変化した時にこの always ブロック内の文が起動される．

reg に対する代入は always ブロックの中で行う．代入にはブロッキング代入（=）とノンブロッキング代入（<=）がある．ブロッキング代入では，値は左辺に即反映される．ノンブ

ロッキング代入では，begin - end 内の右辺の評価が全て行われた後，全ての代入が同時に実行される．記述の順序に依存しない意味を持つようにするため，reg に対する代入は全てノンブロッキング代入で書くのが望ましい．

代入が全て同時に反映されるということは，例えば，1 つの always 中に

```
Y <= X;
Z <= Y;
```

という記述がある場合，Z に代入される Y の値は，上の行で X の値を代入される前の Y の値である．

基本的には，一つの reg を駆動する always ブロック（ドライバ）は唯一である．つまり，一つの reg に対して，複数の always ブロックで代入を行なうことはできない．

組合せ回路の出力は現在の入力の値のみに依存して決まる．組合せ回路を always ブロックで記述する場合は，always 中で参照しているすべての変数をイベントリストに入れ，また，すべての条件を網羅して代入文を記述する必要がある．入力が変化しても代入が行われない場合があると，出力の値は変化しないことになるため過去の値を保持する必要があり，順序回路となってしまう．この点は誤りやすく，設計者の予期しない記憶素子が合成されてしまうことがあるので，十分に注意する必要がある．組合せ回路を記述していることを確実にするためには，データを wire で宣言し always を使わず assign 文で代入を行うとよいが，同じ条件の下で変化するデータがいくつかある場合には，always でまとめて代入するほうが全体の記述が簡潔になることが多い．

2.4 時間のモデル

ハードウェアの動作記述では並列に動作する回路を記述できるため，ソフトウェアのプログラミング言語と比較すると，時間の概念が特徴的である．

1 つのモジュール内に記述された複数の assign 文や always ブロックは，全て**同時に動作**する．また，前述のように，1 つの always ブロック内のノンブロッキング代入文は全て**同時に値が反映**される．assign Z = X & Y; という記述に対し，処理系は右辺の変数に値の変化（イベント）があるかどうか調べ，もしあれば，Δ 時間（微小時間）後の左辺の値を更新する．時刻 t のシグナル X の値を $X(t)$ と書くと，この式は $Z(t+\Delta) = X(t)$ & $Y(t)$ という意味になる．

2.5 順序回路の記述

前述のように，always 中の分岐の条件により代入が行われないことがあり得るデータは，以前の値を保持する必要があるため，記憶素子（フリップフロップ，レジスタ）として扱われる．

特定のクロックシグナルの立ち上がり，あるいは立ち下がりのイベントによって全ての記憶素子が動作するように記述すれば，単相クロック完全同期式の順序回路の記述となる．つまり，完全な同期式順序回路であれば，全てのフリップフロップは reg で宣言し，値の更新は always @(posedge clock) という形のイベントで制御された構造中に記述すればよい．

非同期リセット付きフリップフロップは以下の例のように記述する．記憶素子としてはリセット / プリセット付きのエッジトリガ式フリップフロップのモデルが用意されていることが多く，通常はクロックの他にリセットシグナルを記述する．

```
/*                *
 * counter2.v     *
 * 2-bit カウンタ *
 *                */

/* reset == 0 のとき，カウンタの値をリセット                    *
 * i0 == 1 のとき，クロック信号 clk に同期してカウントアップ */

module counter2 (reset, clk, i0, y0, y1);  // 入出力ポート
   input  reset, clk, i0;                  // 入力
   output y0, y1;                          // 出力

   // 現在の値を記憶しておく flip-flop の宣言
   reg r0, r1;               // flip-flop (1-bit レジスタ)

   // Counter body
   /* クロック，リセット信号に関係のない *
    * 信号線，出力ポートへの代入         */
   assign y0 = r0;           // 出力ポートに対する
   assign y1 = r1;           // 代入は assign 文で行う

   /* クロックの立上り or リセット信号の立下がりイベント *
    * が発生したときに行う処理                           *
    * flip-flop への代入                                 */
   always @(posedge clk or negedge reset) begin
     if (reset == 1'b0) begin
       // reset == 0 (binary, 桁数 1) のとき，カウンタのリセット
       r0 <= 1'b0;  r1 <= 1'b0;  // r0r1 = 00
     end else begin
       if (i0 == 1'b1) begin
         /* i0 == 1 (binary, 桁数 1) のとき，カウントアップ *
          * r1r0 = 00 → 01 → 10 → 11 → 00 → ...       */
         r0 <= ~r0;                          // r0 = not t0
         // r1 = ((not r0) and r1) or (r0 and (not r1))
         r1 <= ((~r0) & r1) | (r0 & (~r1));
       end
     end
   end
endmodule
```

図 4: 2 ビット同期カウンタ

複数のクロックがある記述，個々の記憶素子が他の論理のイベントで駆動される非同期回路の記述なども Verilog HDL としては正しいが，タイミングの検証を綿密に行う必要があること，また，デバイスによっては実現できない場合があること（FPGA など）に注意する必要がある．

図 4 に 2 ビットの同期カウンタ（$00 \to 01 \to 10 \to 11 \to 00 \cdots$）の例を示す．2 ビットの記憶素子に対応するレジスタを r0，r1 とする．クロック clk, リセット reset, カウントアップするかどうかを指定する信号 i0 を入力として持つ．出力は y0，y1 で，内部状態をそのまま出力している．y0，y1 は出力ポートなので，代入文の右辺には使えない．

この例は，典型的な非同期リセット付き，立ち上がりエッジ同期の順序回路の記述である．Verilog HDL の文法上は同じ意味を持つ書き方が他にも考えられるが，あまり凝った書き方をすると処理できない場合があるので，基本的にはこの枠組での記述，あるいは必要に応じてクロック，リセットの論理を逆にした記述が推奨される．

2.6 状態機械

図 4 の 2 ビットカウンタでは，次状態関数を直接記述したが，有限オートマトンの状態遷移を状態名で記述し，次状態関数の生成を自動的に行なわせることもできる．

図 5 に 4 状態カウンタの例を示す．ここでは st0，st1，st2，st3 を 2 ビット定数として定義し，変数 st に対する代入で状態遷移を表している．

状態毎の動作を書くには case 文が便利である．状態を表わす変数を条件として用い，遷移はこの変数への代入として明示的に記述する．出力は ?: を用いた条件付き代入文で記述しており，この部分は st の変化で起動される組合せ回路となる．

前述のように，複数の always ブロックや assign 文を記述した場合，各構造は**並列に動作**する．データへの参照はどの文からでもできるが，1 つのデータへの代入は単一の always ブロックあるいは assign 文に限られる．

2.7 モジュールと階層設計

ある程度大規模な回路の設計では，まとまった機能単位で 1 つのモジュールを設計し上位のモジュールでそれを部品として用いるといった，階層設計を行うのが普通である．Verilog HDL では，module の呼び出しを用いて階層的な設計を記述する．

2 ビットカウンタを二つ用いた 4 ビットカウンタの記述例を図 6 に示す．図 4 の counter2 を counter2a，counter2b として 2 つ用い，counter2a の出力が "11" になった時に counter2b をカウントアップする．

上位のモジュールでは，中で用いるモジュールを記述したファイルを ‘include で参照し，名前を付けて呼び出す．この際にこの部品の実体 counter2a，counter2b の結線関係を指定する．それぞれの値がポートの宣言順に割当てられる．

2.8 回路の動作環境の記述

回路の動作を確認するには，回路に入力を与えて出力を観測する必要がある．入力を与えるために，HDL シミュレータ固有のコマンドを用いる方法もあるが，回路をテストするための枠組，即ちテストベンチを HDL で記述して用意する方法が汎用性が高く一般的である．テストベンチは，設計された対象回路が本来組み込まれる環境をシミュレートするように記述する．つまり，適当な入力波形を発生し，モジュールとして呼び出した対象回路に入力する動作を記述すればよい．

図 7 に，前述の 2 ビットカウンタ counter2 のためのテストベンチを示す．counter2 を呼び出し，クロック等の入力を発生している．reset，clk，i0 には，"#" で指定された時間の後に値が代入される．always 中の文は指定された時間毎に繰返し実行され，initial 中の文は 1 回だけ実行される．特に，clk は 10ns 毎に反転するため，20ns 周期のクロック波形が生成されることになる．

このテストベンチで用いた # による遅延時間量の記述は論理合成できないことに注意する．これらの記述はシミュレーション上は意味があるが，遅延時間を指定できるデバイスは存在せず，また，論理合成の段階での回路の遅延は合成系に与える制約条件に従って最適化されるパラメータの一つなので，回路の記述としては意味がない．

```
/*                  *
 * counter2st.v     *
 * 状態変数を用いた *
 * 2-bit カウンタ   *
 *                  */

/* reset == 0 のとき，カウンタの値をリセット                    *
 * i0 == 1 のとき，クロック信号 clk に同期してカウントアップ */

// define を用いた状態割り当て
`define st0 2'b00
`define st1 2'b01
`define st2 2'b10
`define st3 2'b11

module counter2st (reset, clk, i0, y0, y1);  // 入出力ポート
   input  reset, clk, i0;    // 入力
   output y0, y1;            // 出力

   // 現在の状態を記憶しておく 2-bit レジスタの宣言
   reg [1:0] st;             // 2-bit レジスタ

   // Counter body
   /* クロックの立上り or リセット信号の立下がりイベント *
    * が発生したときに行う処理                           *
    * flip-flop への代入                                 */
   always @(posedge clk or negedge reset) begin
     if (reset == 1'b0) begin
       // reset == 0 (binary, 桁数 1) のとき，
       // 状態変数に初期状態をセット
       st <= `st0;
     end else begin
       if (i0 == 1'b1) begin
         /* i0 == 1 (binary, 桁数 1) のとき，状態遷移    *
          * st = st0 → st1 → st2 → st3 → st0 → ... */
         case (st)
           `st0: begin
               st <= `st1;
           end
           `st1: begin
               st <= `st2;
           end
           `st2: begin
               st <= `st3;
           end
           `st3: begin
               st <= `st0;
           end
         endcase
       end
     end
   end  // always @(posedge clk or negedge reset) begin

   /* クロック，リセット信号に関係のない *
    * 信号線，出力ポートへの代入         */
   // 出力ポートに対する代入は assign 文で行う
   assign {y1, y0} = (st == `st0) ? 2'b00 : (
                       (st == `st1) ? 2'b01 : (
                         (st == `st2) ? 2'b10 : 2'b11));
endmodule
```

図 5: 状態変数を用いた順序回路の記述

```
/*                              *
 * counter4.v                   *
 * counter2.v を用いた階層設計 *
 * による 4-bit カウンタ        *
 *                              */

`include "counter2.v"  // counter2.v の取り込み

module counter4 (reset, clk, i0, y);  // 入出力ポート
   input  reset, clk, i0;             // 入力
   output [3:0] y;                    // 4-bit 出力

   wire counter2b_in;        // 1-bit 信号線

   // module counter2 (reset, clk, i0, y0, y1) の実体化
   counter2 counter2a(reset, clk, i0, y[0], y[1]);
   counter2 counter2b(reset, clk, counter2b_in, y[2], y[3]);

   // Counter body
   // wire に対する代入も assign 文で行う
   assign counter2b_in = y[0] & y[1];
endmodule
```

図 6: 階層設計による 4 ビットカウンタ

```
/*                              *
 * test_counter2.v              *
 * 2-bit カウンタのテストベンチ *
 *                              */

`timescale 1ns / 1ns    // シミュレーションの単位時間 / 精度
                        // 1 ns = 1/1000000000 sec
`include "counter2.v"   // counter2.v の取り込み

module test ;           // テストベンチモジュール, 入出力ポート無し
   // counter2 の入力用 flip-flop(1-bit レジスタ) の宣言
   reg reset, clk, i0;  // flip-flop

   // counter2 の出力用 wire(信号線) の宣言
   wire y0, y1;         // 1-bit 信号線

   // module counter2 (reset, clk, i0, y0, y1) の実体化
   counter2 counter2a(reset, clk, i0, y0, y1);

   // 周期 20 単位時間のクロック信号の生成
   always begin
       // 10 単位時間毎に値が変化
       #10 clk = ~clk;
   end

   initial begin
       // reset, clk, i0 の初期値
       reset = 1;  clk = 0;  i0 = 0;

       #20  reset = 0;  i0 = 0;   // 20 単位時間 (20 ns) 後
       #20  reset = 1;  i0 = 1;   // 更に 20 単位時間 (20 ns) 後
       #80 $finish;               // 更に 80 単位時間 (80 ns) 後, 終了
   end
endmodule
```

図 7: 2 ビットカウンタのテスト回路

3 EDA ツールと FPGA を用いた回路実現の基礎

3.1 計算機と EDA ツールの環境設定

ICE の Linux マシンでは，以下のツールが利用可能である．

- Synopsys 社ソフトウェア一式: VHDL シミュレータ（VSS），Verilog HDL シミュレータ（VCS），VHDL/Verilog HDL 合成系（DC）
- Cadence 社ソフトウェア一式: HDL シミュレータ（IUS），論理合成系（SPR），配置配線系（SOC），フレームワーク/アートワーク系（IC）
- MentorGraphics 社ソフトウェア一式: LSI 設計検証・回路抽出系（Calibre）
- Altera 社ソフトウェア一式: FPGA 統合環境

上記 4 つの内，上 3 社のソフトウェアは，東京大学大規模集積システム設計教育研究センター（VLSI Design and Education Center, VDEC）の共同利用による．Altera 社ソフトウェアは，Altera University Program で提供されている．

VDEC 共同利用によるツール群は，国内の教育機関であれば申請することにより（今のところ）無料で利用可能なので，インターネットに接続されたパソコンやワークステーションと，数 GB 程度のハードディスクを用意すれば導入できる．ソフトウェアのメディアは年度毎に CD-ROM 等で配布され，ライセンスは使用時にインターネット経由でサーバから取得する．また，VDEC を通して，実際に LSI チップを試作するサービスが提供されており，1 設計あたり数十万円〜数百万円程度で LSI 試作を行える．

Altera University Program は，利用計画・実績を提出し審査を受けることにより，ソフトウェアは無料で利用可能（購入しても比較的安価）であるので，機材の価格（基板一式で数万円〜数十万円程度）のみで FPGA 開発環境が導入できる．他 FPGA メーカでも同様のプログラムがある．

他にも，教育機関向けに特に安価でソフトウェアを提供しているベンダもある．また，現時点では市販のツールには及ばないものの，フリーソフトウェアとして開発が続けられているツール群もあり，今後が期待される．

これらのツールは，実際に企業で使われているものと同一のものであり，企業での LSI 設計環境と同じものが教育機関で低コストで利用可能ということになる．今後のハードウェア設計，LSI 設計の教育環境の一層の普及が期待される．

以下では，本実験で使用する Synopsys 社と Cadence 社，Altera 社の EDA ツールを，ICE の Linux マシンで利用するための設定について述べる．

3.1.1 ホームディレクトリに置く設定ファイル

EDA ツールは `/pub1/jikken/eda2` 以下にインストールされており，これらのツールを使用するには，下記のように，設定ファイル`/pub1/jikken/eda/cadsetup.csh.vdec` を読み込むか，この設定ファイルを参考にコマンドパス，環境変数を各自設定する必要がある．

1. 端末で「`ln -s /pub1/jikken/eda/cadsetup.csh.vdec ~/`」と入力して，同ファイルのシンボリックリンクをホームディレクトリに作成する．
2. 「`source ~/cadsetup.csh.vdec`」と入力して，設定を読み込む．端末を立ち上げる度に `source` により設定を読み込む必要がある．

3.1.2 作業ディレクトリに置く設定ファイル

Synopsys 社の論理合成系 Design Compiler の設定ファイル.synopsys_dc.setup は，Design Compiler を起動して実際に設計作業を行う作業ディレクトリに作成する必要がある．図 8 に.synopsys_dc.setup の記述を示す．この.synopsys_dc.setup の 1, 2 行目は自分の名前の指定，3 から 5 行目は論理合成に用いるライブラリの指定である．合成ライブラリは，AND ゲート，OR ゲート，フリップフロップなどの部品セットの機能と特性が記述されたものであり，回路を実装するデバイス・テクノロジに合ったライブラリを用いる．

```
set designer "nakamura"
set company "Nagoya-U"
set target_library [list lsi_10k.db]
set link_library [list lsi_10k.db]
set symbol_library [list generic.sdb lsi_10k.sdb]
```

図 8: .synopsys_dc.setup

3.2 論理シミュレータ Verilog-XL による論理シミュレーション

Verilog HDL で記述した回路が正しく動作するかどうかを調べるために，**論理シミュレータ**と呼ばれるソフトウェアが用いられる．論理シミュレータは，ハードウェア記述言語による回路記述を中間ファイルに変換する**コンパイル**あるいは**アナライズ**と呼ばれる工程を最初に行う．続いて，この中間ファイルを用いて回路の入出力シグナルをトレースする**論理シミュレーション**が行われる．本節では，Cadence 社の論理シミュレータ Verilog-XL を用いてコンパイル，シミュレーションを行う手順を示す．

1. **コンパイル** 「verilog -c Verilog_HDL_テストベンチ記述ファイル名」で Verilog HDL 記述のファイルをコンパイルする．コンパイルエラーが発生した場合は，エラーの種類と行番号が表示される．

2. **シミュレータの GUI の起動** 「verilog +gui Verilog_HDL_テストベンチ記述ファイル名」で起動する（コンパイルも行われる）．シミュレータの GUI である SimVision が起動すると “Design Browser 1 - SimVision” ウインドウと “Console - SimVision” ウインドウが現れる．

3. **トレースする信号の指定**

   (a) “Design Browser 1 - SimVision” ウインドウ左部の Design Browser 部で，トレースする信号を含むモジュールを選択（複数可）し，ウインドウ上部の波形のボタン（黒と白の波形が書かれたアイコン）を押して，“Waveform 1 - SimVision” ウインドウを出す．

4. **シミュレーションの実行**

   (a) “Console - SimVision” ウインドウ上部のメニュー「Simulation → Reinvoke Simulator...」でシミュレータを reinvoke する．Arguments の確認ウインドウが表示されたら，そのまま「Yes」を押す．

(b) “Console - SimVision” ウインドウ上部のメニュー「Simulation → Set Breakpoint → Time...」で “SimVision: Set Breakpoint” ウインドウを出し，Stop at exact time: に停止時刻を入力して,「OK」を押す．単位は Verilog HDL テストベンチ記述で指定した timescale である．

(c) “Console - SimVision” ウインドウ上部のメニュー「Simulation → Run」でシミュレーションを実行する．

5. **シミュレーションの再実行** 4.(a), 4.(b), 4.(c) でシミュレーションを再実行する．

6. **シミュレータの終了** “Design Browser 1 - SimVision” ウインドウ上部のメニュー「File → Exit SimVision」でシミュレータを終了させる．正常に終了されない場合は端末で「killall verilog」としシミュレータを強制終了する．

3.3 論理合成系 Design Compiler による論理合成

論理シミュレーションによって，ハードウェア記述言語で記述した回路が期待通りに動作することが確認できたら，次は，**論理合成系**と呼ばれるソフトウェアが用いられる．論理合成系は，ハードウェア記述言語により記述された回路を最適化し，ゲートレベルの記述であるネットリストへ変換する**論理合成**を行う．本節では，Synopsys 社の論理合成系 Design Compiler を用いて論理合成を行う手順を示す．

1. **デザインビジョンの起動** 「`design_vision`」と入力し Design Compiler の GUI を起動する．Design Compiler のユーザインタフェースは `design_vision` の他に，Tcl(Tool Command Language) ベースのコマンドライン言語による CLI(Command Line Interface) がある．`design_vision` のコマンド入力ダイアログに CLI のコマンドを入力することもできる．以下では `design_vision` の操作の後に CLI でのコマンドを併記する．

2. **設計の読み込み** 「File → Read...」（`read_file -f verilog Verilog_HDL_`記述ファイル名）でソースファイルを読み込む．ここで警告やエラーが出たらソースを修正する．レジスタの推定などの情報も表示される．

3. **制約条件の指定** 回路の遅延時間や面積（回路規模）の制約を与える．詳細は，5 章の EDA ツールを用いた回路設計「中級編」で述べる．

4. **最適化** 「Design → Compile Design...」（`compile`）で 最適化を行う．その後，回路の遅延時間，面積を表示し評価する．詳細は，5 章の EDA ツールを用いた回路設計「中級編」で述べる．

5. **ネットリストの出力** 「File → Save as...」（`write -hierarchy -f verilog -o` 出力ファイル名 ）で回路を出力する．出力フォーマットは合成後の用途に合わせる．

詳しくは,「`sold`」と入力し，オンラインドキュメンテーションを読むこと． 「Japanese-Language Documents」をクリックすると，日本語のオンラインドキュメンテーションが表示される．

3.4 FPGA を用いた回路実現

論理シミュレーションや論理合成が通っても，それが本当に実際のハードウェア上で動くかどうかは100%保証の限りではない．そのため設計した回路が正しく動作するかどうかを，FPGA（書き換え可能なゲートアレイ）を搭載した評価ボードを用いて確認する．本実験では，Altera DE2 ボードを用いる．

3.4.1 Altera DE2 ボード

図 9 に FPGA（Altera Cyclone II）を搭載した Altera DE2 ボードの写真を示す．FPGA は，入力として 18 個のトグルスイッチと 4 個のプッシュスイッチを，出力として 26 個の個別の LED と 8 桁の 7 セグメント LED を利用できる．プッシュスイッチは押していない状態が 1（high），押した状態が 0（low）で，LED はついた状態が 1，消えた状態が 0 である．また Cyclone II は，下記のような PS/2 コネクタ（マウス／キーボードを接続）と XSGA コネクタ（ディスプレイを接続）とも接続されている．更に Cyclone II は，USB コントローラを介し USB Device ポート，USB Host ポート，Audio CODEC を介し Mic In, Line In, Line Out ポート，TV デコーダを介し Video In ポート，イーサネットコントローラを介しイーサネットポートとも接続されている．それ以外に，RS-232 ポート，SD カードポート，IrDA トランシーバ，LCD，8MB SDRAM，512KB SRAM，4MB フラッシュメモリとも接続されている．

PS/2 コネクタ PS/2 コネクタは 6-pin mini-DIN コネクタで，5, 2 pin は VCC, GND である．1, 3 pin は MOUSE_CLK, MOUSE_DATA 信号で FLEX10K に接続されている．MOUSE_CLK はマウス，キーボードに与えるクロック信号，MOUSE_DATA はマウス，キーボードから DE2 ボードに送られるデータである．

XSGA コネクタ XVGA コネクタは 15-pin D-sub コネクタである．1, 2, 3 pin はそれぞれ RED, GREEN, BLUE 信号，13, 14 pin は HORIZ_SYNC, VERT_SYNC，4, 5, 9, 15 pin は No Connect，6, 7, 8, 10, 11 pin は GND である．

なお，DE2 ボードと Linux マシンは，USB インタフェースを介して接続する．DE2 ボードの電源には，専用の AC アダプタを使用する．DE2 ボード左上の赤いプッシュスイッチが電源スイッチである．

3.4.2 Altera QuartusII

Altera QuartusII は Altera FPGA 用の開発ソフトウェアである．本節では QuartusII を使い，設計した回路を論理合成し，DE2 ボードにダウンロードする手順を示す．

0. **コンパイル及びダウンロード前の準備**

- Verilog HDL で記述された回路の入出力ポートと DE2 ボード上の FPGA の実際の入出力ピンとの対応関係などを記述したピン配置設定ファイル「プロジェクト名.qsf」を用意する（実験中に配布予定）.
- PC と DE2 ボードが USB ケーブルを介して接続されていること，PC から DE2 ボード上の FPGA へ転送するストーム・アウト・ファイルの名前などを記述した設定ファイル「プロジェクト名.cdf」を用意する（実験中に配布予定）.

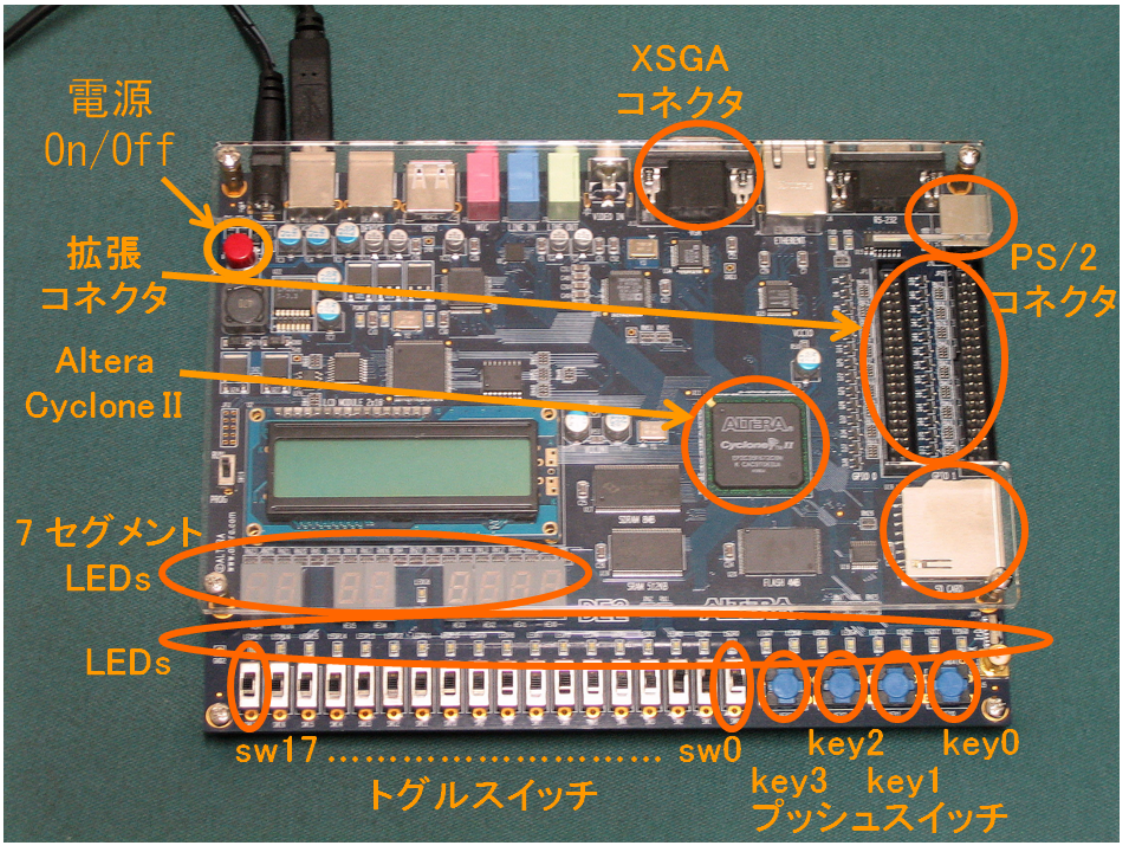


図 9: Altera DE2 ボード

1. **コンパイル** 端末上で「`quartus_sh --flow compile プロジェクト名`」とし，論理合成，FPGA マッピングを行う．コンパイルにより，ストリーム・アウト・ファイル「プロジェクト名`.sof`」が得られる．

2. **ダウンロード** DE2 ボードの電源を On にした後，端末上で「`quartus_pgm プロジェクト名.cdf`」とし，FPGA へのダウンロードを行う．ダウンロードは，`quartus_pgm` を実行したマシンに接続されている DE2 ボードに対して行われる．`quartus_pgm` を実行する前に，DE2 ボードが接続されていること，DE2 ボードが他の人により使用中でないことを確認すること．上記手順の 1 は DE2 ボードが接続されていないマシンでも行える．

FPGA 用の開発ソフトウェアに関して何か変更が生じた時は下記の URL にその情報を掲載するので，その時はここを参照すること．

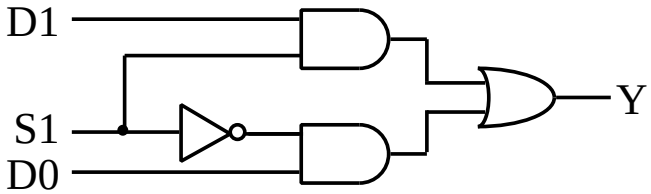
`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/index.html`

4 EDA ツールを用いた回路設計「基礎編」

第 1 週目の実験では，簡単な組合せ回路の設計と回路の動作実験を行う．回路の動作実験では，FPGA を搭載した実験基板を使い，FPGA 上に設計した回路を実現する．ここでは，EDA ツールを用いた LSI 設計の設計フローの理解を目指す．

実験 1 次の 2 入力 1 出力セレクタ回路の設計ならびに動作実験を，下記の手順で行いなさい．

- 設計する 2 入力 1 出力セレクタ回路の仕様
  - 入力: データ D0, D1（それぞれ 1 ビット），セレクト信号 S1（1 ビット）
  - 出力: データ Y（1 ビット）
  - 機能：セレクト信号 S1 の値が 0 か 1 かにより，データ D0, D1 の値を Y に出力
  - 図 10 はこのセレクタ回路の回路図と真理値表
- 回路設計および動作実験の手順
  1. Verilog HDL による回路記述とテストベンチの作成
     4.1 節を参考に，セレクタ回路の Verilog HDL 記述 `mux21.v` と，テストベンチ（論理シミュレーション用の回路の動作環境の記述）`test_mux21.v` を作成する．
  2. 論理シミュレーション
     4.2 節を参考に，セレクタ回路の論理シミュレーションを行う．
  3. 論理合成
     4.3 節を参考に，セレクタ回路の論理合成を行う．
  4. FPGA を用いた回路実現
     4.4 節を参考に，セレクタ回路を DE2 ボード上の FPGA にダウンロードし，実際の動作を観察する．なお，DE2 ボードの sw0 が S1 に， key0, key1 がそれぞれ D0, D1 に対応する．DE2 ボードの最も左の LED が Y に対応する．



S1	D0	D1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

図 10: 2 入力 1 出力セレクタ回路の回路図と真理値表

4.1 Verilog HDL による回路記述

Verilog HDL による回路記述ならびにテストベンチの作成は，Emacs, vi 等のエディタで行う．

4.1.1 セレクタ回路の記述

図 10 のセレクタ回路に対応する Verilog HDL 記述は 2.1 節の図 2 の通りである．2.1 節の Verilog HDL 記述の基本構造に関する説明を理解したのち，こセレクタ回路の Verilog HDL 記述を作成する．ファイル名はモジュール名 mux21 に合わせて mux21.v とする．

4.1.2 テストベンチの作成

記述したセレクタ回路の動作を確認するためのテストベンチ（回路の動作環境の記述） を図 11 に示す．2.8 節，第一段落の回路の動作環境記述に関する説明を理解したのち，セレクタ回路のテストベンチを作成する．ファイル名は test_mux21.v とする．

```
/*                                *
 * test_mux21.v                   *
 * 2-1 セレクタ回路のテストベンチ *
 *                                */

‘timescale 1ns / 1ns    // シミュレーションの単位時間 / 精度
                        // 1 ns = 1/1000000000 sec
‘include "mux21.v"      // mux21.v の取り込み

module test ;           // テストベンチモジュール，入出力ポート無し
   // mux21 の入力用 flip-flop(1-bit レジスタ) の宣言
   reg S1, D0, D1;  // flip-flop

   // mux の出力用 wire(信号線) の宣言
   wire Y;          // 1-bit 信号線

   // module mux21 (S1, D0, D1, Y) の実体化
   mux21 mux21a(S1, D0, D1, Y);

   initial begin
       // S1, D0, D1 の初期値
       S1 = 0; D0 = 0; D1 = 0;

       // 20 単位時間 (20 ns) 後
       #20  S1 = 0; D0 = 1; D1 = 0;

       // 更に 20 単位時間 (20 ns) 後
       #20  S1 = 1; D0 = 1; D1 = 0;

       // 更に 20 単位時間 (20 ns) 後
       #20  S1 = 0; D0 = 0; D1 = 0;

       // 更に 80 単位時間 (80 ns) 後，終了
       #80 $finish;
   end
endmodule
```

図 11: 2-1 セレクタ回路のテストベンチ

4.2 論理シミュレーション

3.1 節の通りに環境設定をしたのち，3.2 節を参考に，以下の手順でセレクタ回路の論理シミュレーションを行う．

1. 端末で「verilog +gui test_mux21.v」と入力して，シミュレータの GUI を起動する（コンパイルも行われる）．

(a) 起動後，初期状態では，シミュレーションの各操作を行うための “Design Browser 1 - SimVision” ウィンドウと “Console - SimVision” ウィンドウが現れる．

(b) “Design Browser 1 - SimVision” ウィンドウ左部の Design Browser 部で，値の変化を観察したい，トレースする信号を含むモジュールを選択（複数可）し，ウィンドウ上部の波形のボタン（黒と白の波形が書かれたアイコン）を押して，“Waveform 1 - SimVision” ウィンドウを出す．

(c) “Console - SimVision” ウィンドウ上部のメニュー「Simulation → Reinvoke Simulator...」でシミュレータを reinvoke する．Arguments の確認ウィンドウが表示されたら，そのまま「Yes」を押す．

(d) “Console - SimVision” ウィンドウ上部のメニュー「Simulation → Set Breakpoint → Time...」で “SimVision: Set Breakpoint” ウィンドウを出し，Stop at exact time: に適当な停止時刻を入力して,「OK」を押す．単位は Verilog HDL テストベンチ記述で指定した timescale である．

(e) “Console - SimVision” ウィンドウ上部のメニュー「Simulation → Run」でシミュレーションを実行する．“Waveform 1 - SimVision” ウインドウに結果の波形が表示される．

(f) “Design Browser 1 - SimVision” ウインドウ上部のメニュー「File → Exit SimVision」でシミュレータを終了させる．正常に終了されない場合は端末で「killall verilog」としシミュレータを強制終了する．

4.3 論理合成

論理シミュレーションによりセレクタ回路が正しく動作することを確認したのち，3.3 節を参考に，以下の手順でセレクタ回路の論理合成を行う．

1. 端末で「`design_vision`」と入力し，デザインビジョンを起動する．図 12 に起動後の初期画面を示す．Design Compiler のユーザインタフェースは GUI の `design_vision` の他

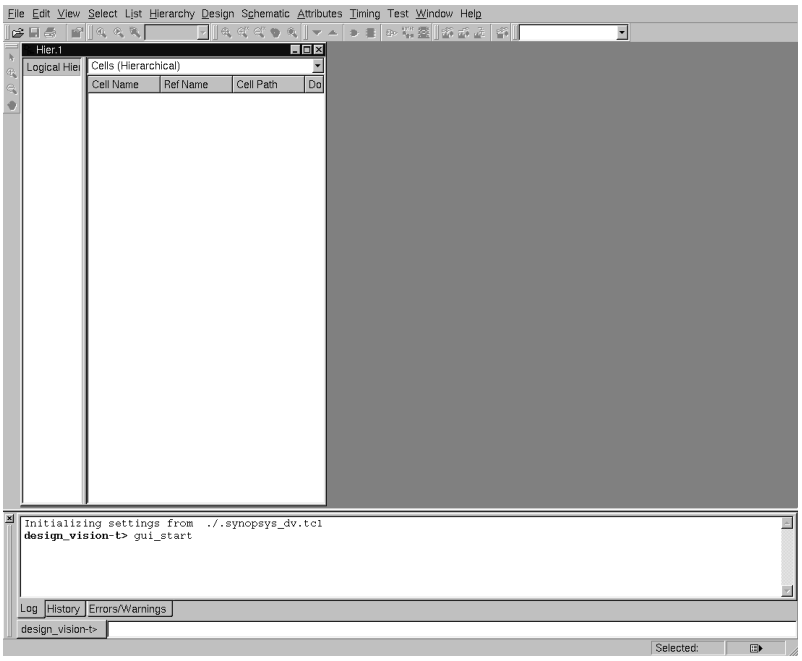


図 12: デザインビジョン起動後の初期画面

に，Tcl(Tool Command Language) ベースのコマンドライン言語による CLI(Command Line Interface) がある．図 12 下部の「design_vision-t>　　」が CLI のコマンド入力ダイアログである．以下では design_vision の操作の後に CLI でのコマンドを併記する．

2. 「File → Read...」（「read_file -f verilog mux21.v」）でソースファイル mux21.v を選択し,「OK」をクリックする．設計の読み込みが正しく終了すると図 13 のような画面が表示される．コマンド入力ダイアログ「design_vision-t>　」の上に隣接す

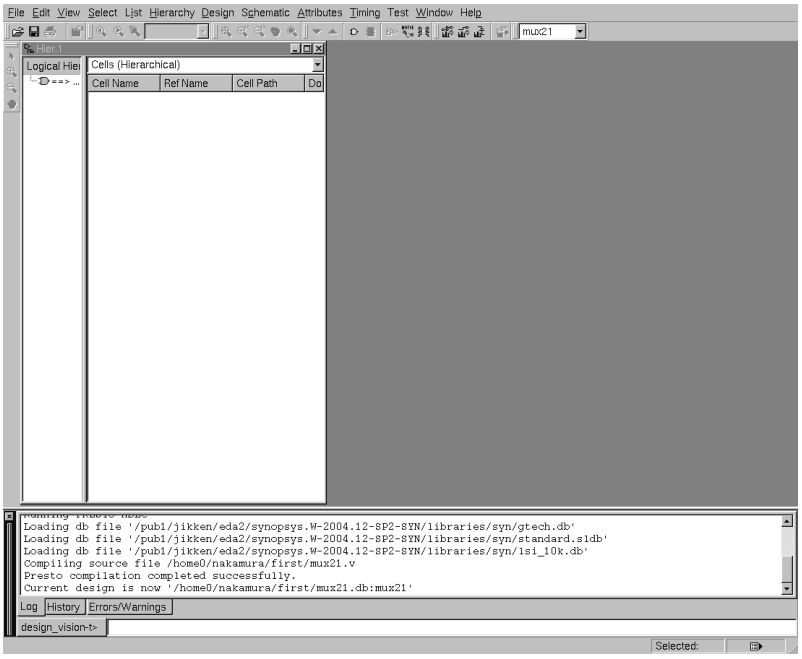


図 13: mux21.v 読み込み後の画面

るメッセージ表示フレーム内に警告やエラーが表示された場合，その内容を読み必要に応じてソースを修正する．エラーがない場合には「compilation completed successfully.」と表示される．論理合成の際に使用されるライブラリに関する情報やレジスタの推定に関する情報も表示される．

3. 次に Verilog HDL で作成した回路の回路図を表示するための操作について述べる．

   (a) 「Select → Cells → Top Design」で読み込んだ回路のトップモジュールを選択する．それから,「Schematic → New Symbol View」で選択されたモジュールのシンボルビューを表示する．すると図 14 のように，モジュール名とモジュールの入出力ポートの情報を示す図が表示される．

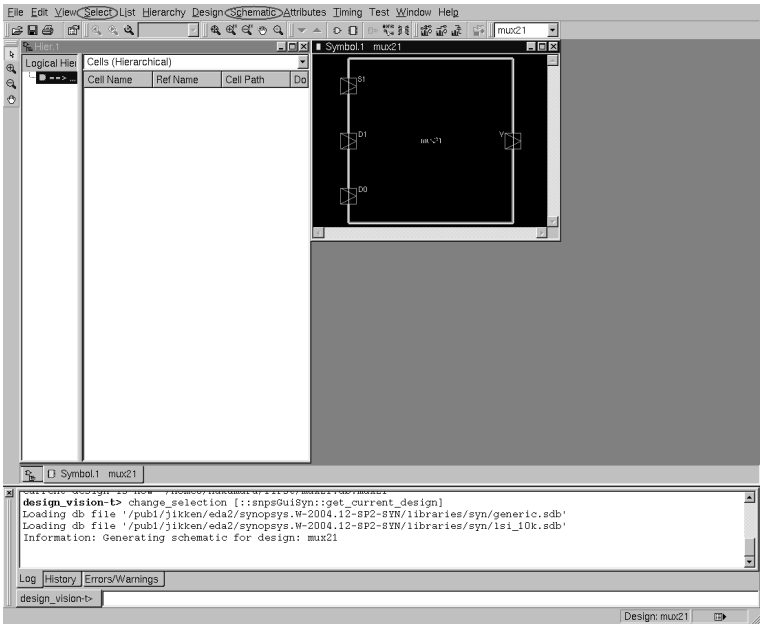


図 14: mux21 のシンボルビュー

(b) 「Schematic → New Design Schematic View」で回路図を表示する．すると図 15 のような mux21 の回路図が表示される．

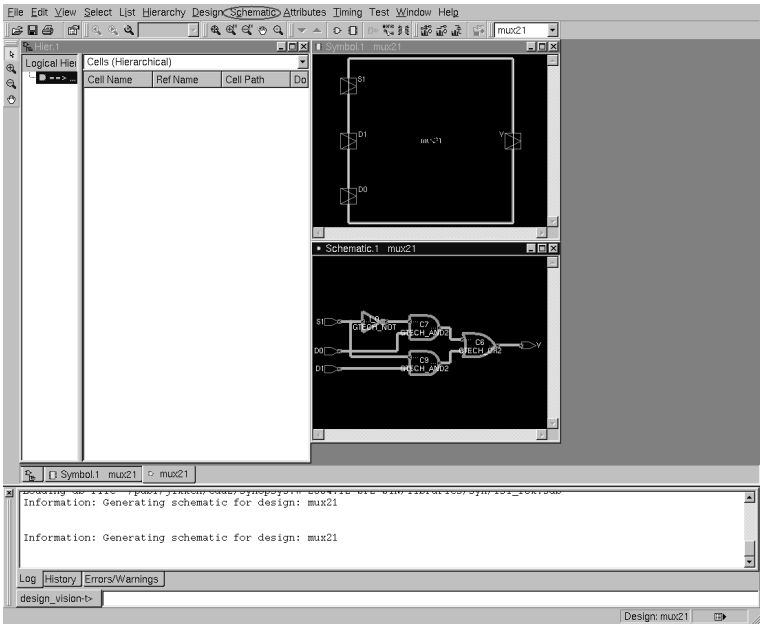


図 15: mux21 の回路図

4. 表示された回路は，LSI 製造のテクノロジに依存しない，汎用の基本素子（セル）を使って構成されたもので，まだこの段階では，読み込まれた回路の論理を中間表現用のセルを使って表現しただけである．この中間表現から，回路を実現するデバイスに応じ，そのデバイス用のセルを使った回路を生成する必要がある．以下の操作により，ターゲットのセルを使った回路の生成と最適化を行う．最適化により，使われるセルの面積や動作速度，駆動能力などが考慮され，回路規模ができるだけ大きくならないように，またスピードが遅くならないように回路が構成される．

(a) 「Design → Compile Design...」(「`compile`」) を選ぶと，新たにダイアログウィンドウが表示される．何も変更せずにそのまま「OK」ボタンを押す．すると回路が自動的に変換，最適化され，最適化前のシンボルビューと回路図が消える．メッセージウィンドウが現れたら，そのまま「OK」ボタンを押す．

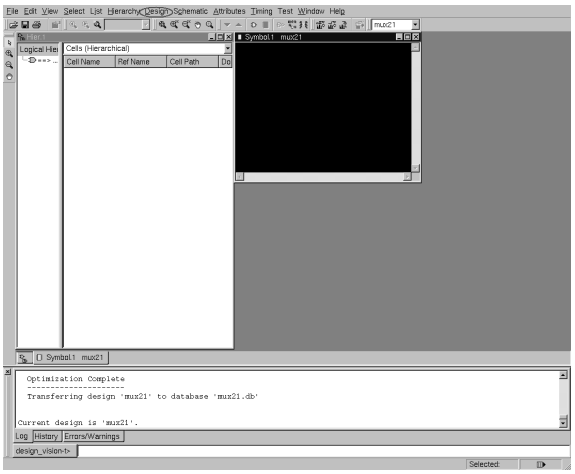


図 16: 最適化操作後の状態

(b) 前述した「トップモジュール選択」,「回路図表示」の作業を行うと，図 17 のような最適化後の回路の回路図が表示される．

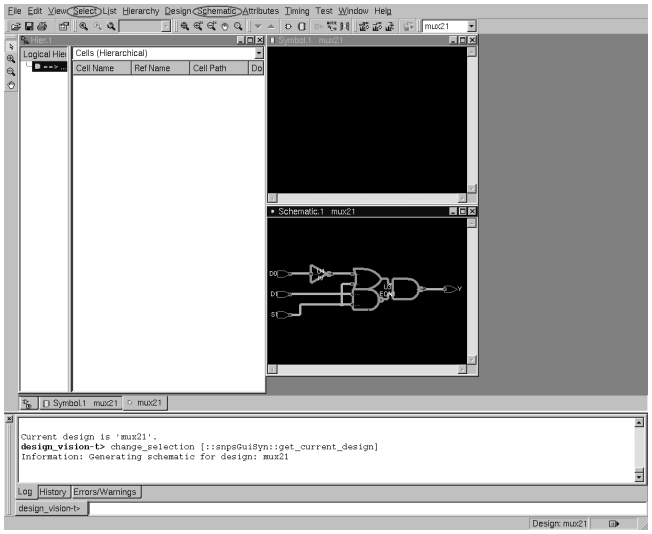


図 17: 最適化後の mux21

5. 最後に，最適化後の回路について，回路の面積や遅延時間の表示を行う．

(a) 「Design → Report Area...」(「`report_area`」) を選ぶと，新たにダイアログウィンドウが表示される．何も変更せずにそのまま「OK」ボタンを押す．すると図 18 のように，回路の面積に関する情報が表示される．

(b) 「Timing → Report Timing...」(「`report_timing`」) を選ぶと，新たにダイアログウィンドウが表示される．何も変更せずにそのまま「OK」ボタンを押す．すると図 19 のように，回路の最大遅延時間に関する情報が表示される．

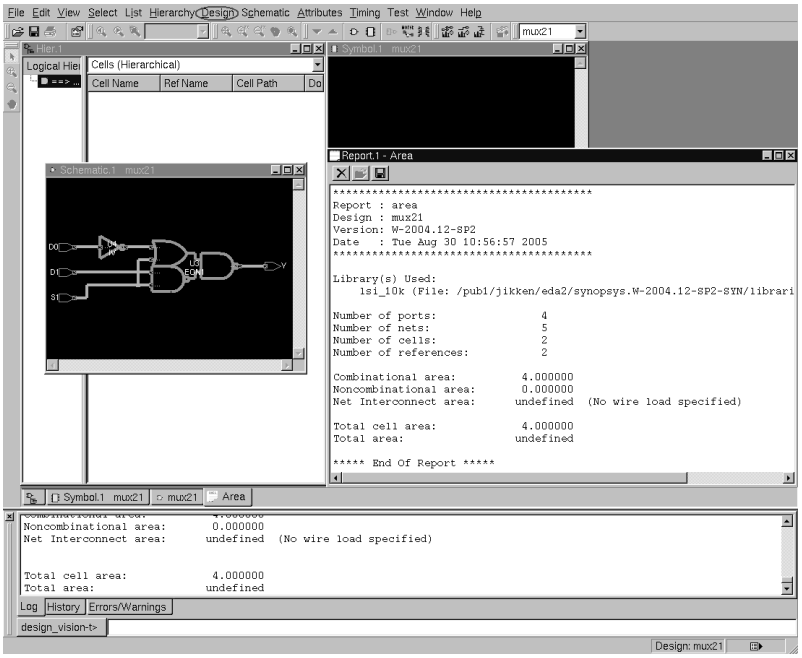


図 18: 面積に関する情報を表示した状態

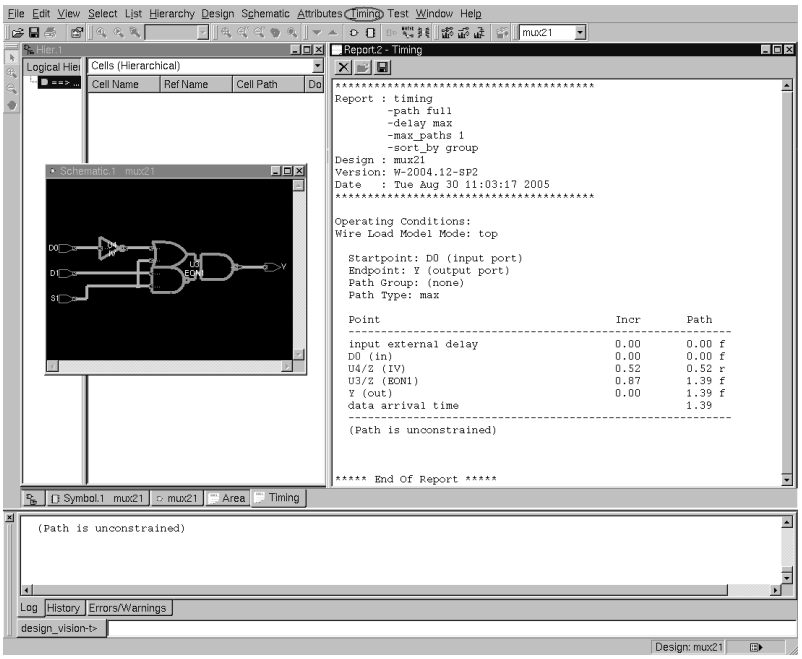


図 19: 遅延時間に関する情報を表示した状態

- 最適化後の回路が必ずしもよい回路とは限らない．つまり，動作速度が遅かったり，セルの数が多い回路かもしれない．最適化の操作は何回でも行える．もう一度最適化を行うと回路がどのように変わるだろうか？ なお，読み込んだ回路を初めて最適化するときは，ターゲットのセルを部品とする回路の生成と最適化が行われるが，二回目以降は，最適化だけが行われる．

- 遅延時間や面積（回路規模）の制約を与えて最適化を行うことにより，所望の性能を有した回路を論理合成により得ることができる．詳細は，5 章の EDA ツールを用いた回路設計「中級編」で述べる．

- 必要に応じて,「File → Save as...」(「`write -hierarchy -f verilog -o` 出力ファイル名`.v` 」で最適化後の回路をファイルに出力することもできる.

4.4 FPGA を用いた回路実現

論理シミュレーションと論理合成を行ったのち，3.4 節を参考に，以下の手順でセレクタ回路を DE2 ボード上の FPGA にダウンロードし，動作させる.

1. 設計したセレクタ回路を FPGA 上に実現して動かすためには，回路の各ポート S1, D0, D1, Y を，FPGA の数百本ある IO ピンのうちのどのピンに割り当てるかを指定する必要がある．DE2 ボード上の FPGA で，`mux21.v` を動かすためのピン配置設定ファイル `mux21.qsf` が用意済なので，以下の URL からダウンロードし，回路設計を行っている作業ディレクトリに置く.

   `http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/first/mux21.qsf`

2. 端末上で「`quartus_sh --flow compile mux21`」とし，コンパイル（論理合成，FPGA マッピング）を行う．コンパイルによりストリーム・アウト・ファイル `mux21.sof` が得られる.

3. `mux21.sof` を FPGA へダウンロードするための設定ファイル `mux21.cdf` が用意済なので，以下の URL からダウンロードし，`mux21.sof` と同じディレクトリに置く.

   `http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/first/mux21.cdf`

4. 端末上で「`quartus_pgm mux21.cdf`」とし，FPGA へのダウンロードを行う.

5. 仕様どおりに動作することを確認する．なお，LED はついた状態が 1 ついていない状態が 0，KEY0,1 は押した状態が 0 押していない状態が 1，SW0 は上た状態が 1 下げた状態が 0 である.

FPGA 用の開発ソフトウェアに関して何か変更が生じた時は下記の URL にその情報を掲載するので，その時はここを参照すること.

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/first/index.html`

5　EDA ツールを用いた回路設計「中級編」

本実験では，加算を行う回路の設計を例題に，組合せ回路と順序回路の記述，EDA ツールを用いた設計制約を考慮した回路の最適化についての実験を行う．

5.1　組合せ回路の設計と最適化

組合せ回路の設計と最適化についての実験を行う．

実験 2 次の 16 ビット加算回路の設計および最適化を下記の手順で行いなさい．

- 設計する 16 ビット加算回路の仕様
  - 入力: 被演算数 x, y（各 16 ビット），桁上げ入力 cin（1 ビット）
  - 出力: 和 sum（16 ビット），桁上げ出力 cout（1 ビット）
  - 機能：x + y を計算し，和と桁上げを出力する組み合わせ回路
- 回路設計および設計制約を与えた最適化の手順
  1. Verilog HDL による回路記述とテストベンチの作成
     2.2 節から 2.4 節を読み，Verilog HDL の基本的な構文と意味，always ブロック，時間のモデルについて理解したのち，16 ビット加算回路の Verilog HDL 記述 `adder16.v` と，テストベンチ `test_adder16.v` を作成する．
  2. 論理シミュレーション
     4.2 節を参考に，加算回路の論理シミュレーションを行う．入力値をいくつか与えて加算回路が正しく動作することを確認する．
  3. 組合せ回路の論理合成と最適化
     論理回路の性能は主に遅延時間と面積で評価される（他に消費電力も）．ここでは，組合せ回路の遅延時間と面積の制約条件を与えた最適化を行う．
     (a) 4.3 節と同様の手順で `design_vision` を起動したのち，`adder16.v` の最適化を行う．
     (b) `report_timing`, `report_area` で回路の遅延時間と面積を確認する．
     (c) 「Select → Paths From/Throgh/To...」で，クリティカルパス（最も遅延が大きい経路）が回路図上に表示できる．このパスが `report_timing` のパスと一致していることを確認する．
     (d) コマンドウィンドウ上で「`set_max_delay 5 -to [all_outputs]`」と入力し，遅延時間 5ns の制約を与える．
     (e) 更に，コマンドウィンドウ上で「`set_max_area 5`」と入力し，面積 5 の制約を与える．
     (f) 再び最適化を行い，回路図ならびに回路の遅延時間，面積を確認する．制約条件を与えない場合と与えた場合で，最適化後の回路に変化があるか？

図 20 と図 21 に加算回路とテストベンチの記述例をそれぞれ示す．

```
/*                  *
 * adder16.v        *
 * 16 ビット加算回路 *
 *                  */

module adder16 (x, y, cin, sum, cout);
   input [15:0] x, y;
   input cin;
   output [15:0] sum;
   output cout;

   assign {cout, sum} = x + y + cin;
endmodule
```

図 20: 16 ビット加算回路

```
/*                                *
 * test_adder16.v                 *
 * 16 ビット加算回路のテストベンチ *
 *                                */

‘timescale 1ns / 1ns    // シミュレーションの単位時間 / 精度
                        // 1 ns = 1/1000000000 sec
‘include "adder16.v"    // adder16.v の取り込み

module test;
   reg [15:0] x, y;
   reg cin;
   wire [15:0] sum;
   wire cout;

   adder16 adder16a(x, y, cin, sum, cout);

   always begin
      #10 x = x + 100;
   end

   always begin
      #5 y = y + 300;
   end

   initial begin
      x = 0 ; y = 0 ; cin = 0;
   end
endmodule
```

図 21: 16 ビット加算回路のテストベンチ

5.2 順序回路の設計と最適化

次は，順序回路の設計と最適化についての実験を行う．順序回路は組合せ回路と記憶素子からなり，順序回路 = 組合せ回路 + 記憶素子（フリップフロップ，レジスタ）である．フリップフロップは，クロック入力のエッジ（立ち上がり，立ち下がり）のイベントに同期して動作する．回路全体が一つのクロックシグナルのイベントによって動作する順序回路を単相クロック同期式順序回路という．ここでは，この単相クロック同期式順序回路を設計する．

実験 3 次の 16 ビット加算回路の設計および最適化を下記の手順で行いなさい．

- 設計する 16 ビット加算回路の仕様
  - 入力: クロック clk（1 ビット），リセット reset（1 ビット），被演算数 x, y（各 16 ビット），桁上げ入力 cin（1 ビット）
  - 出力: 和 sum（16 ビット），桁上げ出力 cout（1 ビット）
  - 機能：x + y を計算し，和と桁上げを出力する．ただし，入力値を加算した結果は次のクロックの立ち上がりで出力に反映される．また，リセットが 0 になると出力の各ビットは 0 になる（非同期リセット）．
- 回路設計および設計制約を与えた最適化の手順
  1. Verilog HDL による回路記述とテストベンチの作成
     2.5 節から 2.7 節を読み，Verilog HDL による順序回路の記述，状態機械，モジュールと階層設計について理解したのち，順序回路版 16 ビット加算回路の Verilog HDL 記述 `adder16s.v` と，テストベンチ `test_adder16s.v` を作成する．
  2. 論理シミュレーション
     実験 2 と同様に，加算回路の論理シミュレーションを行う．
  3. 順序回路の論理合成と最適化
     (a) 4.3 節と途中まで同様の手順で `adder16s.v` を開く．
     (b) 「`create_clock -period 5 -name clk [get_ports clk]`」，
     「`set_max_delay 5 -to [all_outputs]`」，
     「`set_max_delay 5 -to [all_registers -data_pins]`」
     と入力し，遅延時間 5ns の制約を与える．
     (c) 更に，「`set_max_area 5`」と入力し，面積 5 の制約を与える．
     (d) その後，最適化を行い，回路図ならびに回路の遅延時間，面積，クリティカルパスを確認する．また，遅延時間 100ns の制約条件を与えた場合，最適化後の回路に変化はあるか？

順序回路の最適化では，クロックの波形（クロック周期）と信号線名の指定を `create_clock` で行う必要がある．また，`set_max_delay` で遅延時間の制約を与えるとき，回路の出力ポートへの遅延だけではなく，レジスタの入力への遅延も指定する必要がある．図 22 と図 23 に順序回路版 16 ビット加算回路とテストベンチの記述例をそれぞれ示す．

```
/*                                *
 * adder16s.v                     *
 * 順序回路版 16 ビット加算回路 *
 *                                */

module adder16s (clk, reset, x, y, cin, sum, cout);
   input [15:0] x, y;
   input clk, reset, cin;
   output [15:0] sum;
   output cout;

   reg [15:0] r0, r1;

   assign {cout, sum} = r0 + r1 + cin;

   always @(posedge clk or negedge reset) begin
      if (reset == 1'b0) begin
         r0 <= 0 ; r1 <= 0;
      end else begin
         r0 <= x; r1 <= y;
      end
   end
endmodule
```

図 22: 順序回路版 16 ビット加算回路

```
/*                                              *
 * test_adder16s.v                              *
 * 順序回路版 16 ビット加算回路のテストベンチ *
 *                                              */

`timescale 1ns / 1ns    // シミュレーションの単位時間 / 精度
                        // 1 ns = 1/1000000000 sec
`include "adder16s.v"    // adder16.v の取り込み

module test ;
   reg reset,clk, cin;
   reg [15:0] x, y;

   wire [15:0] sum;
   wire cout;

   adder16s  adder16sa(clk, reset, x, y, cin,sum, cout);

   always begin
      #5 clk = ~clk;
   end

   always begin
      #8  x = x + 100;
          y = y + 200;
   end

   initial begin
      reset = 1; clk = 0; x = 0; y = 0; cin = 0 ;
      #20 reset = 0;
      #20 reset = 1;
   end
endmodule
```

図 23: 順序回路版 16 ビット加算回路のテストベンチ

6 実験課題

実験課題 1 （実験課題 2 との選択）

次の 2 つの順序回路を Verilog HDL で記述し,「論理シミュレーション」,「論理合成」を行いなさい．また，論理合成の際に，設計制約により最適化後の回路の遅延時間，面積がどのように変化するかを確認しなさい．

1. 1 桁の BCD コード（4 ビット）を出力する BCD カウンタ（0 → 1 → ... → 8 → 9, 9 の次は 0 → 1 → 2 → ...）の設計を行いなさい．
   - 設計する 1 桁の BCD カウンタの仕様
     - 入力: クロック clk（1 ビット），リセット reset（1 ビット），カウントアップ信号 x（1 ビット）
     - 出力: カウンタ値出力 bcd1_out （4 ビット）
     - 機能：リセットが 0 になると出力の各ビットは 0 になる（非同期リセット）．出力値は，clk が 0 から 1 に変化かつカウントアップ信号が 1 になるたびにカウントアップし，0000 → 0001 → 0010 → ... → 1001 → 0000 → 0001 → ... のようになる．
2. 2 桁の BCD コード（8 ビット）を出力する BCD カウンタ（0 0 → 0 1 → ... → 0 8 → 0 9, 0 9 の次は 1 0 → 1 1 → 1 2 → ...）の設計を，**階層設計**により行いなさい．
   - 設計する 2 桁の BCD カウンタの仕様
     - 入力: クロック clk（1 ビット），リセット reset（1 ビット），カウントアップ信号 x（1 ビット）
     - 出力: カウンタ値出力 bcd2_out（8 ビット）
     - 機能：リセットが 0 になると出力の各ビットは 0 になる（非同期リセット）．出力値は，clk が 0 から 1 に変化かつカウントアップ信号が 1 になるたびにカウントアップし，0000 0000 → 0000 0001 → 0000 0010 → ... → 1001 1001 → 0000 0000 → 0000 0001 → ... のようになる．

実験課題 1 に関するヒントが必要な場合は下記の URL に情報を掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/problem1/index.html`

実験課題 2 （実験課題 1 との選択）

次の順序回路を Verilog HDL で記述し,「論理シミュレーション」,「論理合成」を行いなさい．また，論理合成の際に，設計制約により最適化後の回路の遅延時間，面積がどのように変化するかを確認しなさい．

1. 0011 および 0010 という入力系列が入力される毎に 1 を出力する系列検出回路（有限オートマトン）の設計を，状態機械による順序回路記述により行いなさい．

   - 設計する系列検出器の仕様
     - 入力: クロック clk（1 ビット），リセット reset（1 ビット），データ入力 x（1 ビット）
     - 出力: 検出結果出力 y （1 ビット）
     - 機能：リセットが 0 になると出力は 0 になる（非同期リセット）．出力値は，0011 または 0010 という系列を検出したときのみ 1 になる．
       動作例: 入力として 100111 ··· がシリアルに入ってきたとき
       入力 1 に対して 0 を出力
       入力 0 に対して 0 を出力
       入力 0 に対して 0 を出力
       入力 1 に対して 0 を出力
       入力 1 に対して 1 を出力（← 0011 を検出）
       入力 1 に対して 0 を出力
       ・
       ・
       ・

実験課題 2 に関するヒントが必要な場合は下記の URL に情報を掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-eda/problem2/index.html`

7 実験と実験課題のレポートについて

1. 実験 1，実験 2，実験 3，実験課題 1（2 との選択），実験課題 2（1 との選択）について，実験の概要，使用機器ならびにソフトウェア，以下の各段階の説明，実験の考察を，文章ならびに図，表を交えてまとめなさい．

   - 回路の Verilog HDL 記述（どんな回路かの説明）
   - 論理シミュレーション（テストベンチとシミュレーション結果の説明）
   - 論理合成（論理合成の際の制約条件により，最適化後の回路の遅延時間，面積がどのように変わるかの説明）
   - 動作実験（どんな動作だったか，実験 1 のみ）

2. 「論理回路の遅延時間と面積の最適化」について調査し，本実験の結果もふまえて説明しなさい．

8 その他

8.1 FPGA のピン配置の変更

実験で使用する FPGA のピン配置ファイルは，ほぼ全てあらかじめ用意されているが，以下のようにしてピン配置を変更することもできる．

入出力の設定例として，実験で使用した「mux21.qsf」を例に説明する．このファイルを編集して直接入出力の指定を書き込むことができる．ファイルに以下のように書き込む．

```
set_location_assignment PIN_N25 -to S1
set_location_assignment PIN_G26 -to D0
set_location_assignment PIN_N23 -to D1
set_location_assignment PIN_AD12 -to Y
```

これでプッシュスイッチ key0, key1 がそれぞれ入力 D0, D1 に，トグルスイッチ 0 が入力 S1 になる．Y は最も左の LED に出力される．ピン番号とボード上の各入出力デバイスとの対応は DE2 ボードのマニュアルに詳しく書かれている．

8.2 Altera DE2 ボードのクロックの使い方

DE2 ボードには発振器がついており，クロックを入力として使うことができる．入力としてクロックを使いたいときは,「.qsf」で `set_location_assignment PIN_N2 -to clk` と設定する．ただし，周波数が 50MHz のため，早すぎて視覚できないので，適当に分周して使わなければならない．以下に，分周回路の Verilog HDL 記述の例を示す．

```
module divider (clk, sysreset, clkin);
   input clk, sysreset;
   output clkin;
   reg [23:0] cnt;
   always @(posedge clk or negedge sysreset) begin
      if(sysreset == 1'b0)  cnt <= 0;
      else  cnt <= cnt + 1;
    end
    assign clkin = ~cnt[22];
endmodule
```

最後の assign 文のレジスタ「cnt」の何ビット目を使うかを変更することで周波数を 2^n ずつ変えることができる.

8.3 種々の回路の Verilog HDL 記述

Verilog HDL を使って実験課題の回路を記述するにあたり，課題と直接は関係しないが，記述スタイルの参考になると思われる回路の記述例をいくつか示す.

32 ビット入力 8 ビット出力の 2 ビット左シフトモジュール

```
/********************/
/* shifter32_8_l2.v */
/********************/

//              +----+
// sh_a[31:0]->|    |->sh_y[7:0]
//              +----+

module shifter32_8_l2 (sh_a, sh_y);  // 入出力ポート
  input  [31:0]  sh_a;          // 入力 32-bit
  output  [7:0]  sh_y;          // 出力  8-bit

  //Body
  //2-bit 左シフト
  assign sh_y = {sh_a[5:0], 2'b00};
endmodule
```

32 ビットの 2 入力 1 出力セレクタモジュール

```
/**************/
/* mux32_32_32.v */
/**************/

//           +----+
// d0[31:0]->|    |
// d1[31:0]->|    |->y[31:0]
//         s->|    |
//           +----+

module mux32_32_32 (d0, d1, s, y);  // 入出力ポート
  input  [31:0] d0;          // 入力 32-bit d0
  input  [31:0] d1;          // 入力 32-bit d1
  input         s;          // 入力 1-bit  s
  output [31:0]  y;          // 出力 32-bit  y

  // Multiplexer body
  // if (s == 0) y = d0; else y = d1;
  // 出力ポートに対する代入は assign 文で行う
  assign y = (s == 1'b0) ? d0 : d1;
endmodule
```

CPU の 16 ビット入力 32 ビット出力の符号拡張モジュール

```
/******************/
/* signext16_32.v */
/******************/

//            +----+
// a16[15:0]->|    |->y32[31:0]
//            +----+

module signext16_32 (a16, y32);  // 入出力ポート
  input   [15:0]  a16;          // 入力 16-bit
  output  [31:0]  y32;          // 出力 32-bit

  //Body
  //符号拡張
  assign y32 = {a16[15], a16[15], a16[15], a16[15],
                a16[15], a16[15], a16[15], a16[15],
                a16[15], a16[15], a16[15], a16[15],
                a16[15], a16[15], a16[15], a16[15],
                a16[15:0]};
endmodule
```

CPU の 32 ビット ALU モジュール

```
/*********/
/* alu.v */
/*********/

//                   +----+
//     alu_a[31:0]->|    |
//     alu_b[31:0]->|    |->alu_y[31:0]
//   alu_ctrl[2:0]->|    |->alu_iszero
//                   +----+

// 命令セット
// lw(load word)
// sw(store word)
// add
// sub
// and
// or
// slt(set on less than)
// beq(blanch on equal)

// alu_ctrl[2:0], 実行する演算
// 010,           add
// 110,           sub
// 000,           and
// 001,           or
// 111,           slt

‘define     ADD  3’b010
```

```
`define     SUB  3'b110
`define     AND  3'b000
`define      OR  3'b001
`define     SLT  3'b111

module alu (alu_a, alu_b, alu_ctrl, alu_y, alu_iszero);  // 入出力ポート
  input  [31:0]  alu_a;          // 入力 32-bit      a
  input  [31:0]  alu_b;          // 入力 32-bit      b
  input   [2:0] alu_ctrl;        // 入力  3-bit   ALU 制御コード

  output [31:0]  alu_y;          // 出力 32-bit      y
  output    alu_iszero;          // 出力  1-bit iszero (y==0 ? 1:0)

  reg    [31:0] result;
  reg           iszero;

  always @(alu_a or alu_b or alu_ctrl) begin
    case (alu_ctrl)
      `ADD: begin
        result = alu_a + alu_b;
      end
      `SUB: begin
        result = alu_a - alu_b;
      end
      `AND: begin
        result = alu_a & alu_b;
      end
      `OR: begin
        result = alu_a | alu_b;
      end
      `SLT: begin
        result = (alu_a < alu_b) ? 32'h00000001 : 32'h00000000;
      end
      default: begin
        result = 0;
      end
    endcase
  end

  always @(alu_a or alu_b or alu_ctrl or result) begin
    if (result == 0) begin
        iszero = 1;
    end else begin
      iszero = 0;
    end
  end

  assign alu_y = result;
  assign alu_iszero = iszero;
endmodule
```

CPU の PC 用 4 加算モジュール

```
/***********/
/* plus4.v */
/***********/

//             +----+
// inc_a[7:0]->|    |->inc_y[7:0]
//             +----+

module plus4 (inc_a, inc_y);  // 入出力ポート
  input  [7:0]  inc_a;          // 入力 8-bit
  output [7:0]  inc_y;          // 出力 8-bit

  assign inc_y = inc_a + 4;
endmodule
```

CPU の PC モジュール

```
/********/
/* pc.v */
/********/

//               +----+
//        clock->|    |
//        reset->|    |
// pc_next[7:0]->|    |->pc[7:0]
//               +----+

module pc (clock, reset, pc_next, pc);  // 入出力ポート

  input      clock, reset;   // 入力 クロック, リセット
  input  [7:0]    pc_next;   // 入力 8-bit 次にPCにセットする値
  output [7:0]         pc;   // 出力 8-bit PC

  reg    [7:0]     pc_reg;   // PC用レジスタ

  // Always ブロック: プログラムカウンタ
  // 入力: clock, reset, pc_next
  // 出力: pc_reg
  // レジスタ: pc_reg
  always @(posedge clock or negedge reset) begin
    if (reset == 1'b0) begin
        pc_reg <= 8'b00000000;
    end else begin
        pc_reg <= pc_next;
    end
  end

  assign pc = pc_reg;
endmodule
```

CPU の 32 ビット × 16 ワードレジスタファイルモジュール

```
/***************/
/* registers.v */
/***************/

//                        +----+
//                clock->|    |
//                reset->|    |
//   reg_read_idx1[3:0]->|    |
//   reg_read_idx2[3:0]->|    |->reg_read_data1[31:0]
//   reg_write_idx[3:0]->|    |->reg_read_data2[31:0]
//     reg_write_enable->|    |
// reg_write_data[31:0]->|    |
//                        +----+

module registers (clock, reset,
  reg_read_idx1, reg_read_idx2,
  reg_write_idx, reg_write_enable, reg_write_data,
  reg_read_data1, reg_read_data2);

  input               clock, reset;       // 入力 クロック, リセット
  input   [3:0]       reg_read_idx1;      // 読みアドレス 1
  input   [3:0]       reg_read_idx2;      // 読みアドレス 2
  input   [3:0]       reg_write_idx;      // 書き込みアドレス
  input               reg_write_enable;   // 書き込み (1)/読み (0)
  input  [31:0]       reg_write_data;     // 書き込みデータ
  output [31:0]       reg_read_data1;     // 読みデータ 1
  output [31:0]       reg_read_data2;     // 読みデータ 2

  // Registers (regs_0 = 0)
  reg [31:0] regs_1;   reg [31:0] regs_2;
  reg [31:0] regs_3;   reg [31:0] regs_4;
  reg [31:0] regs_5;   reg [31:0] regs_6;
  reg [31:0] regs_7;   reg [31:0] regs_8;
  reg [31:0] regs_9;   reg [31:0] regs_10;
  reg [31:0] regs_11;  reg [31:0] regs_12;
  reg [31:0] regs_13;  reg [31:0] regs_14;
  reg [31:0] regs_15;
  //
  // 読み 1 (regs[0] は常に 0)
  // assign reg_read_data1 = regs[1〜15];
  //
  assign reg_read_data1 = (reg_read_idx1 == 4'b0000) ? 0 : (
    (reg_read_idx1 == 4'b0001) ? regs_1 : (
    (reg_read_idx1 == 4'b0010) ? regs_2 : (
    (reg_read_idx1 == 4'b0011) ? regs_3 : (
    (reg_read_idx1 == 4'b0100) ? regs_4 : (
    (reg_read_idx1 == 4'b0101) ? regs_5 : (
    (reg_read_idx1 == 4'b0110) ? regs_6 : (
    (reg_read_idx1 == 4'b0111) ? regs_7 : (
    (reg_read_idx1 == 4'b1000) ? regs_8 : (
    (reg_read_idx1 == 4'b1001) ? regs_9 : (
```

```
  (reg_read_idx1 == 4'b1010) ? regs_10 : (
  (reg_read_idx1 == 4'b1011) ? regs_11 : (
  (reg_read_idx1 == 4'b1100) ? regs_12 : (
  (reg_read_idx1 == 4'b1101) ? regs_13 : (
  (reg_read_idx1 == 4'b1110) ? regs_14 : (regs_15))))))))))))));

//
// 読み2（regs[0]は常に0）
// assign reg_read_data2 = regs[1〜15];
//
assign reg_read_data2 = (reg_read_idx2 == 5'b00000) ? 0 : (
  (reg_read_idx2 == 5'b00001) ? regs_1 : (
  (reg_read_idx2 == 5'b00010) ? regs_2 : (
  (reg_read_idx2 == 5'b00011) ? regs_3 : (
  (reg_read_idx2 == 5'b00100) ? regs_4 : (
  (reg_read_idx2 == 5'b00101) ? regs_5 : (
  (reg_read_idx2 == 5'b00110) ? regs_6 : (
  (reg_read_idx2 == 5'b00111) ? regs_7 : (
  (reg_read_idx2 == 5'b01000) ? regs_8 : (
  (reg_read_idx2 == 5'b01001) ? regs_9 : (
  (reg_read_idx2 == 5'b01010) ? regs_10 : (
  (reg_read_idx2 == 5'b01011) ? regs_11 : (
  (reg_read_idx2 == 5'b01100) ? regs_12 : (
  (reg_read_idx2 == 5'b01101) ? regs_13 : (
  (reg_read_idx2 == 5'b01110) ? regs_14 : (regs_15))))))))))))));

// Always ブロック: 書き込み
// 入力: clock, reset, reg_write_idx, reg_write_enable, reg_write_data
// 出力: regs_1〜regs_15
// レジスタ: regs_1〜regs_15
always @(posedge clock or negedge reset) begin
  if (reset == 1'b0) begin
     regs_1 <= 0;   regs_2 <= 0;   regs_3 <= 0;   regs_4 <= 0;
     regs_5 <= 0;   regs_6 <= 0;   regs_7 <= 0;   regs_8 <= 0;
     regs_9 <= 0;  regs_10 <= 0;  regs_11 <= 0;  regs_12 <= 0;
    regs_13 <= 0;  regs_14 <= 0;  regs_15 <= 0;
  end else begin
  if (reg_write_enable == 1'b1) begin
    //
    // 書き込み（regs[0]は常に0）
    // regs[1〜15] = reg_write_data;
    //
    if (reg_write_idx == 4'b0001) begin
      regs_1 <= reg_write_data;
    end if (reg_write_idx == 4'b0010) begin
      regs_2 <= reg_write_data;
    end if (reg_write_idx == 4'b0011) begin
      regs_3 <= reg_write_data;
    end if (reg_write_idx == 4'b0100) begin
      regs_4 <= reg_write_data;
    end if (reg_write_idx == 4'b0101) begin
      regs_5 <= reg_write_data;
    end if (reg_write_idx == 4'b0110) begin
```

```
        regs_6 <= reg_write_data;
      end if (reg_write_idx == 4'b0111) begin
        regs_7 <= reg_write_data;
      end if (reg_write_idx == 4'b1000) begin
        regs_8 <= reg_write_data;
      end if (reg_write_idx == 4'b1001) begin
        regs_9 <= reg_write_data;
      end if (reg_write_idx == 4'b1010) begin
        regs_10 <= reg_write_data;
      end if (reg_write_idx == 4'b1011) begin
        regs_11 <= reg_write_data;
      end if (reg_write_idx == 4'b1100) begin
        regs_12 <= reg_write_data;
      end if (reg_write_idx == 4'b1101) begin
        regs_13 <= reg_write_data;
      end if (reg_write_idx == 4'b1110) begin
        regs_14 <= reg_write_data;
      end if (reg_write_idx == 4'b1111) begin
        regs_15 <= reg_write_data;
      end
      end
    end  // End: if (reg_write_enable == 1'b1) begin
  end  // End: always @(posedge clock or negedge reset) begin

endmodule
```

参考文献

[1] http://www.vdec.u-tokyo.ac.jp/ 東京大学大規模集積システム設計教育研究センター（VDEC）.

[2] VDEC 監修, 浅田邦博. ディジタル集積回路の設計と試作. 培風館, 2000.

[3] 深山正幸, 北川章夫, 秋田純一, 鈴木正國. HDL による VLSI 設計 – Verilog-HDL と VHDL による CPU 設計 –. 共立出版株式会社, 1999.

[4] 白石肇. わかりやすいシステム LSI 入門. オーム社, 1999.

[5] 桜井至. HDL によるデジタル設計の基礎. テクノプレス, 1997.

[6] James O. Hamblen and Michael D. Furman. Rapid Prototyping of Digital Systems. Kluwer Academic Publishers, 2000.

[7] パターソン&ヘネシー 著, 成田光彰 訳. コンピュータの構成と設計（上巻）. 日経 BP 社, 1999.