EDA ツールを用いた論理回路設計

実験概要

本実験では、ハードウェア記述言語を用いた簡単な論理回路の設計を行なう。設計課題を通して、 EDA ツールを用いた論理シミュレーション、論理合成などの各段階の基本的な技術を習得する。また、FPGA を搭載した実験基板を用いて回路の動作を観測する。

実験スケジュール

第 1 週 VHDL による組合せ回路記述、機能レベルシミュレーション

第 2 週 VHDL による順序回路記述、論理合成

第 3 週 設計制約と最適化、ゲートレベルシミュレーション

第 4 週 FPGA ボードでの動作実験

実験課題目次

目 次

作成者: 高木 一義、八木 透
最終更新日: 平成 12 年 10 月 11 日
第 0.91 版

1 はじめに

1.1 ディジタル LSI の設計フロー

近年の LSI の大規模化・複雑化にともない、より生産性の高い設計手法が必要とされている。従来の論理回路設計は、論理ゲートレベルの回路図入力によるものが一般的であった。しかし、論理合成系などの設計システムの発達により、回路図に代わってより高位の記述による設計がシステム設計の主流になってきている。

論理回路の動作記述の抽象度のレベルを上げることによって、より大規模な回路を短時間で設計できるようになる。現在一般的である RTL (Register Transfer Level) での設計記述には、主にハードウェア記述言語 (HDL, Hardware Description Language) が用いられる。

RTL で記述された回路は論理合成系によってゲートレベル回路に変換され、ターゲットが LSI であればテクノロジマッピング、レイアウトの工程を経て LSI のマスクパタンに変換される。また、ターゲットが FPGA (Field Programmable Gate Array) などであれば、回路は LUT へのマッピングにより FPGA 構成データに変換される。

本実験では、回路設計における、ハードウェア記述言語 (HDL: Hardware Description Language) による設計入力、論理シミュレーション、論理合成、の各段階の基本的な技術を習得する。

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 RTL (Register Transfer Level) で記述し、設計を行な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。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、実用化されつつある。

1.3 計算機と設計環境

ICE の Sun ワークステーション `iceas, icea{01,02,...,10}` では、以下のツールが利用可能である。

- Cadence 社ソフトウェア一式: Verilog HDL 処理系 (LDV)、配置配線系 (SE)、フレームワーク/アートワーク系 (IC)
- Synopsys 社ソフトウェア一式: VHDL シミュレータ (VSS)、VHDL/Verilog HDL 合成系 (DC)
- Avant! 社ソフトウェア一式: 配置配線系
- Altera 社ソフトウェア一式: FPGA 統合環境

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。

本実験では、Synopsys 社と Altera 社のツールを用いる。各ツールは `/eda2` 以下の各ディレクトリにインストールされている。`~edauser/cadsetup.csh` を読み込むか、これを参考にコマンドパス、環境変数を各自設定すること。

2 VHDL による回路動作記述の基礎

2.1 VHDL 記述の基本構造

図 1 に、2 入力 1 出力のセレクタ回路の VHDL 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、VHDL の基本構造を見ていく。

- "`--`" で始まる行は行末までコメントである。
- 2〜4 行目は使用するライブラリの指定である。IEEE Std-1164 で規定された機能ライブラリを用いている。
- 7〜10 行目はエンティティ宣言である。エンティティ宣言では、エンティティ名 `mux21` を定義し、入出力インタフェースを宣言する。
- 13〜18 行目はアーキテクチャ記述である。アーキテクチャ記述は、エンティティの内部の機能を表す実体であり、エンティティ `mux21` に対するひとつのアーキテクチャ `behavior1` を記述している。同じエンティティに対して、異なる名前で複数のアーキテクチャを対応づけることができ、そのエンティティを用いる部分で実際にどのアーキテクチャを使用するかを指定する。

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、5 ns (ナノ秒) 後に `Y` へ代入することを表す。代入文中の `not, and, or` は論理演算である。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。`after` による回路の遅延時間指定は、シミュレーション上は意味があるが、論理合成を行なう時には無視される。

VHDL 記述のファイル名は `*.vhd` とすることが多い。ツールによってはこれは必須で、更にエンティティ名とファイル名が同じである必要があるものもある。

```
-- mux21.vhd: 2-1 selector
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Multiplexer interface
entity mux21 is
  port (S1, D0, D1: in std_logic;
        Y: out std_logic );
end mux21;

-- Multiplexer body
architecture behavior1 of mux21 is
begin
  Y <= (not S1 and D0) or
       (    S1 and D1) after 5 ns;
end behavior1;
```

図 1: 2-1 セレクタ回路の例

演算子等	意味	使用例
and	AND	a and b
or	OR	a or b
nand	NAND	a nand b
nor	NOR	a nor b
not	NOT	not a
xor	EXOR	a xor b
&	ビット連接	a & b (2 ビットのデータ)
(downto)	多ビット数	a(3 downto 0) は a(3) & a(2) & a(1) & a(0)
(downto)	部分ビット	a(3 downto 2) は a(3) & a(2)
'0', '1'	定数	
""	2 進定数	"0001", "0011"
X""	16 進定数	X"f8a3"

図 2: VHDL の演算子等

2.2 基本的な構文と意味

VHDL は厳格な型を持つ言語である。型は機能ライブラリの中で定義されている。図 1 の入出力ポートの宣言で用いている std_logic は、0, 1, X (unknown), Z (high impedance) などの 9 値をとることができる標準の型で、IEEE 1164 のライブラリで定義されている。

例に示した論理演算の他に、図 2 に示すような演算がある。

VHDL では、値を保持する要素として、シグナル (signal で宣言) と変数 (variable で宣言) の 2 種類がある。多くの場合、シグナルは記憶素子、変数は記憶を持たない配線素子に対応するが、必ずしもシグナルがレジスタになり変数が配線になるとは限らない。代入文の記号は、シグナルへの代入では <=、変数への代入では := を用いる。

2.3 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。シグナルへの代入 <= は同期代入と呼ばれる。1 つのアーキテクチャ内に記述された同期代入文は全て同時に代入が反映さ

れる。Z <= X and 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 $Z(t+\Delta) = X(t)$ and $Y(t)$ という意味になる。

2.4 プロセス

VHDL のプロセス文は、内部が逐次的に解釈されるひとまとまりの機能を記述する。プロセス文の中では if などの制御構造が記述できる。図 1 の Y への代入文をあえてプロセス文で記述すると以下のようになる。

```
process (S1, D0, D1) begin
  if S1 = '0' then
    Y <= D0;
  else
    Y <= D1;
  end if;
end process;
```

process の次に記述されている変数のリストはセンシティビティリストと呼ばれ、このリスト中の変数が変化した時にこのプロセスが起動される。一つのプロセス文で複数のシグナルへの代入を記述できるが、前述のように、代入は全て同時に反映される。例えば、プロセス中に

```
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、シグナルを駆動するプロセス (ドライバ) は唯一である。つまり、一つのシグナルに対して、複数のプロセス (あるいはプロセス外の代入文) 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。組合せ回路をプロセス文で記述する場合は、プロセス中で参照しているすべての変数をセンシティビティリストに入れ、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なわれない場合があると、出力のシグナル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、プロセスを使わず代入文や条件付き代入文を使うとよいが、同じ条件の下で変化するシグナルがいくつかある場合には、プロセスのほうが全体の記述が簡潔になることが多い。

2.5 順序回路の記述

前述のように、プロセス文中の分岐の条件により代入が行なわれないことがあり得るシグナルは、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。記

```
-- counter2.vhd: 2bit counter
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter2 is
  port (reset, clk, input: in std_logic;
        y0, y1: out std_logic);
end counter2;

architecture behavior1 of counter2 is
    signal r0, r1 : std_logic := '0';
begin
  y0 <= r0;
  y1 <= r1;
  P1: process (clk, reset)
    variable t0, t1 : std_logic;
  begin
    if reset = '0' then
      r1 <= '0';  r0 <= '0';
    elsif clk'event and clk = '1' then
      t0 := r0;  t1 := r1;
      if (input = '1') then
        r0 <= not t0;
        r1 <= ((not t0) and t1) or (t0 and (not t1));
      end if;
    end if;
  end process;
end behavior1;
```

図 3: 2 ビット同期カウンタ

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。

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 VHDL としては正しいが、タイミングの検証を綿密に行なう必要があること、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 3 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シグナルを `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 `input` を入力として持つ。variable の使用例を示すために `t0`, `t1` を介して `r0`, `r1` を更新しているが、直接 `r0`, `r1` を用いても等価である。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。VHDL 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。

```
-- counterst.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counterst is
  port (reset, clk, input: in std_logic;
        y: out std_logic_vector(1 downto 0));
end counterst;

architecture behavior1 of counterst is
  TYPE state_type is (s0, s1, s2, s3);
  signal st: state_type;
begin

  transition: process (clk, reset) begin
    if reset = '0' then
      st <= s0;
    elsif clk'event and clk = '1' then
      if (input = '1') then
        case st is
          when s0 =>
            st <= s1;
          when s1 =>
            st <= s2;
          when s2 =>
            st <= s3;
          when s3 =>
            st <= s0;
        end case;
      end if;
    end if;
  end process;

  y <= "00" when st = s0 else
       "01" when st = s1 else
       "10" when st = s2 else
       "11";

end behavior1;
```

図 4: 状態変数を用いた順序回路の記述

2.6 状態機械

図 3 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、状態割り当てを自動的に行なわせることもできる。

図 4 に 4 状態カウンタの例を示す。ここでは s0, s1, s2, s3 の値をとる数え挙げ型 state_type を定義し、state_type 型の変数 st に対する代入で状態遷移を表している。状態を表わす変数は整数型などでもよい。

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。

複数のプロセス (あるいはプロセス外の代入文) を記述した場合、各プロセスは並列に動作する。ポートやシグナルの参照はどのプロセスでもできるが、前述のように、一つのポートやシグナルへの代入は単一のプロセスに限られる。

```
-- counter4.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter4 is
  port (reset, clk, input: in std_logic;
        y: out std_logic_vector(3 downto 0));
end counter4;

architecture structure1 of counter4 is
  component counter2
    port (reset, clk, input: in std_logic;
          y0, y1: out std_logic);
  end component;
  for C1, C2: counter2 use entity work.counter2(behavior1);
  signal C2in: std_logic;
  signal y0, y1: std_logic;
begin
  C1: counter2 port map (reset => reset, clk => clk, input => input, y0
=> y0, y1 => y1);
  y(0) <= y0;
  y(1) <= y1;
  C2in <= y0 and y1;
  C2: counter2 port map (reset => reset, clk => clk, input => C2in, y0
=> y(2), y1 => y(3));
end structure1;
```

図 5: 構造記述を用いた 4 ビットカウンタ

2.7 コンポーネントと階層設計

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 つのエンティティを設計し上位のエンティティでそれを部品として用いるといった、階層設計を行なうのが普通である。VHDL では、コンポーネント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 5 に示す。前節の `counter2` を `C1`, `C2` として 2 つ用い、`C1` の出力が `"11"` になった時に `C2` をカウントアップする。

上位のアーキテクチャでは、中で用いるエンティティを `component` として宣言し、どの設計を用いるかを `for` 文で指定する。この部品の実体 `C1`, `C2` の結線関係は `port map` により指定する。

2.8 回路のテスト

回路の動作を確認するためには、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、コンポーネント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 6 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` をコンポーネントとして呼び出し、クロック等の入力を発生している。`reset`, `clk`, `input` には、`","` で区切られた値が `after` で指定された時刻にそれぞれ代入される。

```
-- test_counter2.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity test_counter2 is
end test_counter2;  -- no port

architecture test_structure of test_counter2 is
  component counter2
    port (reset, clk, input: in std_logic;
          y0, y1: out std_logic);
  end component;
  for count1: counter2 use entity work.counter2(behavior1);
  signal clk: std_logic := '0';
  signal reset, input, y0, y1: std_logic;
begin
  reset <= '1', '0' after 5 ns, '1' after 30 ns;
  clk   <= not clk after 10 ns;
  input <= '0', '1' after 50 ns;

  count1: counter2 port map (reset, clk, input, y0, y1);

end test_structure;
```

図 6: 2 ビットカウンタのテスト回路

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。

port map では、図 5 で書いたようなコンポーネントのポートの対応は省略した書き方になっているが、この記述では宣言された順番にそれぞれのシグナルが割当てられる。

このテストベンチで用いた after など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。

2.9 IEEE1164 ライブラリ

VHDL の文法としては基本的な構文が与えられているだけであり、通常用いる型や演算子などはライブラリとして提供される。

これまでの例でも、entity 宣言の前に IEEE 標準 (Std. 1164) ライブラリを呼び出して用いてきた。std_logic, std_logic_vector といった基本的な型も IEEE ライブラリで定義されており、ieee.std_logic_1164 を呼び出すことにより使用できる。ieee.std_logic_unsigned は、std_logic_vector などを符号無し 2 進数と見て扱うための加減算などの演算子が定義されている。他にも、実装するデバイス (特に FPGA など) に特有の機能に応じてライブラリが用意されていることがある。

図 7 に、ieee.std_logic_unsigned の演算子を用いた加算回路の記述を示す。論理合成後にこの記述がどのような加算回路となるかは、合成系に依存する。Synopsys Design Compiler では、回路全体の遅延時間などの制約に依存して、順次桁上げ加算器になるか、桁上げ先見加算器になるかが選択される。乗除算などの演算子は用意されていないこともあるが、用意されている場合でも、回路規模が大きくなるため注意する必要がある。

```
-- adder8.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adder8 is
  port (op1, op2 : in  std_logic_vector(7 downto 0);
        result   : out std_logic_vector(7 downto 0));
end adder8;

architecture behavior of adder8 is
begin
  result <= op1 + op2;
end behavior;
```

図 7: IEEE ライブラリを用いた 8 ビット加算器

3 論理シミュレーション

本実験では、Synopsys 社の論理シミュレータ VSS を使用する。回路を VHDL で記述し、以下の手順でシミュレーションを行なう。

環境設定 カレントディレクトリに設定ファイル `.synopsys_vss.setup` を用意する。

```
timebase = ns
work > default
default : ./work
```

1 行目はシミュレーションの時間の単位の設定、2, 3 行目は中間ファイルが置かれるディレクトリの指定である。2, 3 行目でこのように指定した場合、ディレクトリを作っておく必要がある (`mkdir work`)。

アナライズ `vhdlan [ファイル名]` で VHDL 記述ファイルをシミュレーションのための中間ファイルに変換する。

シミュレータ起動 `vhdldbx` で起動する。シミュレーション対象の名前を引数として書いてもよい。`vhdldbx` の内部で `vhdlsim` が起動される。`vhdlsim` を直接起動する場合はシミュレーション対象を指定する必要がある。後の操作は `dbx` デバッガなどと同様と考えれば理解しやすい。

トレースするシグナルの指定 最下行のコマンド入力から `ls`, `cd` などの通常のファイル操作コマンドと同様の方法で設計の内部のシグナル名などの空間を表示できる。`trace [シグナル名]` で波形を表示するシグナルを指定する。`trace *'sig` とすると全てのシグナルがトレースされる。

シミュレーション実行 `run [時間]` でシミュレーションを実行する。単位は `.synopsys_vss.setup` で指定した `timebase` である。途中で止める場合は `Intr.` ボタンをクリックする。

詳しくは、`sold` でオンラインドキュメンテーションを読むこと。

4 論理合成

本実験では、Synopsys 社の論理合成系 Design Compiler を使用する。論理合成は以下の手順で行なう。

環境設定 カレントディレクトリに設定ファイル .synopsys_dc.setup を用意する。

```
Designer = "ktakagi"
Company = "Nagoya-U"
target_library = { lsi_10k.db }
link_library = { lsi_10k.db }
symbol_library = { generic.sdb lsi_10k.sdb }
```

1, 2 行目は自分の名前の指定、3〜5 行目は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のセットの機能と特性が記述されたものである。

デザインアナライザの起動 design_analyzer で起動する。Design Compiler のユーザインタフェースは design_analyzer の他に コマンドライン入出力の dc_shell がある。design_analyzer から、Setup -> Command Window... で dc_shell のインタフェースを呼び出すこともできる。以下では design_analyzer の操作の後に dc_shell でのコマンドを併記する。

設計の読み込み File -> Read... (read -f vhdl [ファイル名]) でソースファイルを指定する。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。

制約条件の指定 遅延時間や面積 (回路規模) の制約を与える。

論理最適化 Tools -> Design Optimization... (compile) で最適化を行なう。遅延時間、面積を表示し評価する。

ネットリストの出力 File -> Save as... (write -f vhdl [出力ファイル名] -hierarchy) 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 でオンラインドキュメンテーションを読むこと。

5 FPGA を用いた回路の実現

5.1 Altera UP1 ボード

Altera UP1 ボードは MAX7000 と FLEX10K の 2 個の FPGA を搭載した実験用ボードである。入出力は MAX7000, FLEX10K それぞれ独立である。左の MAX7000 は 16 個の DIP スイッチと 2 個のプッシュスイッチを入力、16 個の個別の LED と 2 桁の 7 セグメント LED を出力として利用できる。右の FLEX10K は 8 個の DIP スイッチと 2 個のプッシュスイッチを入力、2 桁の 7 セグメント LED を出力とし、このボード単体では少ないが、この他にマウス、キーボードを接続できる PS/2 コネクタとディスプレイを接続できる VGA コネクタが接続されている。

5.2 Altera MAX+PlusII

Altera MAX+PlusII は Altera FPGA 用の開発ソフトウェアである。設計した回路は、MAX+PlusII を用いて以下の手順で UP1 ボードにダウンロードし動作させる。

MAX+PlusII の起動 max2win で起動する。

プロジェクトのオープン `File -> Project -> Name...` (あるいは、白いファイルに階層構造の絵が描かれているアイコン) でプロジェクト (設計ファイル) を選択するウィンドウが表示されるので、選択する。

最上位デザインのオープン、設計入力 `File -> Hierarchy Project Top` (階層構造のトップに矢印が付いているアイコン) でファイルを開く。

論理合成、マッピング `Max+plus II -> Compiler` (工場のアイコン) で論理合成と FPGA マッピングを行なう。エラー、警告が出たら `Locate` でソース上の該当個所を表示できる。

FPGA ダウンロード `Programmer` (青、水色、赤のケーブルのアイコン) でプログラマを起動する。`Configure` を押すと FPGA へのダウンロードが開始する。

6 演習課題

以下の設計課題について、シミュレーション、論理合成、FPGA 上での動作を行なう。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、サイズがどのように変わるか観測する。

6.1 練習課題

図 7 の 8 bit 加算器の設計を用いて、シミュレーション、論理合成、FPGA 実装の各段階の操作を修得する。

6.2 基本課題

1. パリティジェネレータ
   入力 (serial or parallel) の 8bit 毎にパリティビットを付加して出力 (serial) する。

2. start/stop, lap 付きストップウォッチ
   内部クロック周波数を `constant` で定義する。クロックを内部で分周し、秒のみを 2 桁の BCD コード (計 8 bit) で出力する。

6.3 応用課題

1. 16bit ALU
   16bit 値を 2 個と、演算の種類 (加算、減算、論理演算) を指定する制御信号を入力とし、16bit 値とステータスフラグを出力とする。

2. 16bit CPU
   IF, ID, EX, MEM, WB の 5 ステージからなる汎用プロセッサで、入出力は通常のプロセッサと同様とする。