プロジェクト実験 P2 CPU の論理設計 岡本 吉央
平成 18 年 12 月 8 日 2 回目資料

1 本日行なうこと

1. イネーブル付き 12 進カウンタの設計 (サブテキスト 65 ページのリスト 3.2)
2. そのシミュレーション (サブテキスト 73 ページのリスト 3.6)
3. 中間レポートの作成

2 注意

- 経過報告はメーリングリスト (cpu@algo.ics.tut.ac.jp) へ送付すること．岡本個人宛にしないように．
- VHDL ファイルの拡張子は「.vhd」である．
- コマンド scs の引数は，行ないたいシミュレーションが記述されているコンフィグレーション名とする．
- 実験ツールがサイズの大きいファイルを生成している場合があり，それがファイルサーバへの負荷となるので，終了時に確認すること．例えば「ls␣-l」や「du」といったコマンドで確認できる．「core」や「*.ow」というファイルが発見された場合は削除すること．

3 ソフトウエアの検証とテスト (一般論)

- ソフトウエア検証は英語で「Verification & Validation」，略して「V&V」とよく呼ばれる．Verification と Validation の違いが何であるのか，共通の認識があるわけではなくて，人によって違う意味で用いたりしているので注意が必要である．
- 検証：ソフトウエアが要求されている品質を満たすことを確認する作業．
- 検証の対象：
  - 仕様の検証
    - 非形式的検証：よく「レビュー」(見直し，review) と呼ばれる．インスペクション (査閲，inspection) やウォークスルー (徒歩検査，walk-through) といった方法がある．
    - 形式的検証：仕様が (仕様記述言語などを用いて) 形式的に書かれている場合，論理・代数を用いて検証を行なうことができる．モデル検査 (model-checking) とも呼ばれる．
  - プログラムの検証：
    - 静的検証：プログラムを実行せずにその特性を解析する．コンパイラの行なう構文エラー検査，静的な型検査 (type checking) や，ホーア論理などを用いた正当性証明など．
    - 動的検証：プログラムを実行して行なう検証．

- テスト：プログラムを一定の条件下で実行し，結果を評価する．「一定の条件下で実行する」とはテストのためのデータを選択して実行環境を制御して実行するという意味である．
- 動的解析：例えば，プログラムの中に表明 (assertion，そこで成立すべき論理条件) を挿入してチェックする，など．他には，プロファイル解析や網羅度検査といったもの．

3.1 テストについて

テスト技術は検証の中でも古くから実践されてきたものであり，今日でも中心的な役割を担っている．

- テスト・プロセス：ソフトウエア・プロセスに対応させて，テストもプロセスを経て行なわれる．(ソフトウエア・プロセスについては次回．)
  - 単体テスト (unit testing)：プログラム単位としてのモジュールを，各モジュール仕様に対して独立にテストする．
  - 統合テスト (integration testing)：モジュールを結合し，ひとまとまりの処理を実行する単位にまとめてテストする．設計仕様に対するテストであると考えられる．
  - システム・テスト (system testing)：システム全体として要求仕様を満たしているかどうか判定するために行なうテスト．
  - 受入れテスト (acceptance testing)：システム運用前にユーザが実際に使用することで行なうテスト．開発段階に応じて，アルファテスト，ベータテストと更に分類される．
- テストにおける技術的な課題 (気を付けるべき事項)
  1. テストケースの選定 (テストケースは無限に存在するか，有限であっても膨大であり，全てを網羅することは実質的に不可能)
  2. テストの実行環境
  3. テスト結果の評価法

1. テストケースの選定によるテストの分類
   - 機能テスト (ブラックボックス・テスト)：プログラムの機能仕様のみを見て，入力空間と出力空間の特徴からテストケースを選定する．
     - 同値分割：入力空間をテストに対して同値な部分空間に分割して，各部分から 1 つのテストケースを選ぶ．
     - 限界値分析：同値分割と同様に入力空間を分割したとき，各部分空間の境界上やその近傍からテストケースを選ぶ．また，出力空間にも注目して，その境界上の結果を生むようなテストケースを選ぶ．
     - その他
   - 構造テスト (ホワイトボックス・テスト)：プログラムの内部構造まで見て，構造を出来る限り隈無く網羅するようにテストケースを選定する．
     - 文網羅度に基づく選定：「プログラムの実行文の中のどれだけが少なくとも 1 度は実行されたか」という指標 (文網羅度と呼ぶ) を考え，これが大きくなるようなテストケースを選ぶ．
     - 分岐網羅度に基づく選定：「プログラムの実行分岐のどれだけが少なくとも 1 度実行されたか」という指標 (分岐網羅度と呼ぶ) を考え，これが大きくなるようなテストケースを選ぶ．

  - その他

2. テストの実行環境に関して：実際に稼働しているシステムに開発中のソフトウエアを導入してテストを行なうと，稼働中のシステムだけではなくシステムの外部にまで悪影響を及ぼす可能性があるので，そうではなく，稼働中のシステムをモデル化した世界を作り (テストベッド，testbed，と呼ぶ)，その上でテストを行なうことが一般的である．

3. テスト結果の評価：

   - 個々のテストケースの評価
     - 機能テストの評価：各テストケースに対して，予期した出力が得られているかどうかの確認．自動化ツールも多く開発されている．
     - 構造テストの評価：ツールを利用した網羅度の計測．UNIX 上の C プログラミングの場合，gcc に対して gcov という網羅度計測コマンドがある．
   - テスト工程全体の評価
     - テスト工程の進行にともなってエラーが逐次発見され修正されて，プログラムの信頼性は向上していくため，テストを適切な時点で打ち切るための基準を与えるという意味で重要
     - テスト期間による基準，工数による基準，テスト項目数・テストケース数による基準，網羅度による基準，発見エラー数による基準，発見エラー数の収束による基準，など．(後のものほど「科学的」な根拠がありそう)
     - 収束性の判断：テスト進行を時間軸，累積発見エラー数をもう一方の軸とし，エラー発見を確率過程と捉えた信頼性モデルの利用．例えば，非斉次ポアソン過程など．
   - テストの保存，管理：ソフトウエア保守において，変更後の再テスト (回帰テスト (regression testing) と呼ばれる) の際に変更前のソフトウエアに用いたテストと同じテストを用いる．テストケース，テストベッドだけではなく，テスト結果も保管する

★ テスト駆動開発 (test-driven development)：1990 年代後半から現れた考え方で，プログラム本体を設計する前に，まずテストケースを設計する開発法「テスト駆動開発」だけではなく，「エクストリームプログラミング (XP)」，「リファクタリング」，「アジャイルソフトウエア開発」といった検索語を Web サーチエンジンに問い合わせてみよ．テスト駆動開発はソフトウエア開発技法であり，テスト技法ではないので注意．

4　イネーブル付き 12 進カウンタ

サブテキストのリスト 3.2 (65 ページ)

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4
5  entity COUNT4EN is
6    port ( CLK, RESET, EN : in  std_logic;
7           COUNT : out std_logic_vector(3 downto 0)
8    );
9  end COUNT4EN;
10
11 architecture RTL of COUNT4EN is
12 signal COUNT_IN : std_logic_vector(3 downto 0);
13 begin
14   COUNT <= COUNT_IN;
15   process (CLK, RESET) begin
16     if(RESET='1') then
17       COUNT_IN <= "0000";
18     elsif(CLK'event and CLK='1') then
19       if(EN = '1') then
20         if(COUNT_IN = "1011") then
21             COUNT_IN <= "0000";
22         else
23             COUNT_IN <= COUNT_IN + '1';
24         end if;
25       end if;
26     end if;
27   end process;
28 end RTL;
29
30 configuration CFG_COUNT4EN of COUNT4EN is
31   for RTL
32   end for;
33 end;
```

部分的説明

- 7 行目：std_logic_vector(3 downto 0) は std_logic を 4 つ連ねたベクタである．サブテキスト 26 ページを参照．
- 15 行目：process の後の (CLK, RESET) は，このプロセス文の「センシティビティ・リスト」と呼んでいるもので，そこに並べられているものが変化したとき，このプロセスが実行される．サブテキスト 37 ページ参照．

- 18 行目：「elsif」である．「else if」ではないので注意．
- 18 行目：CLK'event とは信号 CLK が変化したとき真となる．よって，「CLK'event and CLK='1'」という条件は CLK が立ち上がったとき，そのときのみ，真になる．
- 20 行目：COUNT_IN は 12 行目で std_logic_vector(3 downto 0) と宣言されているので，この部分の代入により，COUNT_IN(3) が 1，COUNT_IN(2) が 0，COUNT_IN(1) が 1，COUNT_IN(0) が 1 となる．

5 イネーブル付き 12 進カウンタのシミュレーション記述

サブテキストのリスト 3.6 (73 ページ)

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4
5  entity TEST_COUNT4EN is
6  end TEST_COUNT4EN;
7
8  architecture SIM1 of TEST_COUNT4EN is
9  component COUNT4EN
10   port ( CLK, RESET, EN : in  std_logic;
11          COUNT : out std_logic_vector(3 downto 0)
12 );
13 end component;
14
15 constant CLK_CYCLE : Time := 20 ns;
16 signal CLK, INIT_RESET, EN : std_logic;
17 signal COUNT_OUT : std_logic_vector(3 downto 0);
18
19 begin
20   U0 : COUNT4EN port map ( CLK   => CLK, RESET => INIT_RESET,
21                            EN    => EN, COUNT => COUNT_OUT);
22   process begin
23     CLK <= '1';
24     wait for CLK_CYCLE/2;
25     CLK <= '0';
26     wait for CLK_CYCLE/2;
27   end process;
28
29   process begin
30     INIT_RESET <= '0'; EN <= '1';
31     wait for CLK_CYCLE/3;
32     INIT_RESET <= '1';
33     wait for CLK_CYCLE;
```

```
34      INIT_RESET <= '0';
35      wait for CLK_CYCLE*10;
36      EN <= '0';
37      wait for CLK_CYCLE*3;
38      EN <= '1';
39      wait;
40    end process;
41  end SIM1;
42
43  configuration CFG_TEST of TEST_COUNT4EN is
44    for SIM1
45    end for;
46  end CFG_TEST;
```

部分的説明

- 15 行目：定数宣言には constant を用いる．サブテキスト 81 ページ参照．ここではクロック周期を 20 ナノ秒に設定している．
- 20 ~ 21 行目：ここではコンポーネントの信号の対応を明示的に書いている．
- 31 行目：クロック周期の整数倍だけ待ってから信号を変化させると，クロックの立ち上がりと信号の変化が同時に起こるので，どのような動作をするのか想定できなくなる．そのため，ここではクロック周期の 1/3 倍だけ待つことでこのような問題を回避している．

6　波形のプリントアウト方法

波形の出ている窓で，一番左上のメニューの [file] → [print] を選択する．

1. [Begin Time] と [End Time] でプリントアウトする波形の部分の開始時刻と終了時刻をそれぞれ定める．デフォルトでは，窓に表示されている部分に対応する時刻が入っている．
2. [Print Options] は [Standard Sheet Size] として適当なものを選択．通常は [A4] である．[Rotate Output] がデフォルトで選択されているが，これによって横向きにプリントアウトされる (はず)．
3. ファイルに出力する場合は，[Print Only To File] を選び，ファイル名を指定．これは PostScript 形式のファイルになるので，通常は拡張子を「.ps」とする．LaTeX 文書に取り込む場合などは，EPS 形式にしておくと都合がよいので，[Encapsulated Postscript] も選択する．この場合は拡張子を「.eps」に変更する．
4. プリンタに直接出力する場合は，[Print Command] として
   `lpr␣-Pedups01`
   `lpr␣-Pedups02`
   `lpr␣-Pedups03`
   `lpr␣-Pedups04`
   のいずれかを入力．(オプション P の後に来ているものはプリンタ名であり，これらは計算機室のプリンタであるので注意すること．)
5. 最後に [OK] をクリックする．

7　中間レポート

- 提出〆切：12 月 15 日 (金) 実験時間中に回収
- 場所：計算機室
- 内容：サブテキストのリスト 3.2 (イネーブル付き 12 進カウンタ) の動作確認をリスト 3.6 のシミュレーションにより行なう．
- 様式：A4 用紙にて，本文は 5 枚程度．表紙も付けること．また，2 つのプログラムリストと出力波形も添付すること．
- 注意
  1. 実際に，動作の確認を行なうためには何が必要かということを念頭に置いてシミュレーションを行なうこと．
  2. 少なくとも以下のことはレポートで述べる必要がある．
     (a) (設計したい) イネーブル付き 12 進カウンタとはどのような振舞いをするカウンタか．また，それを踏まえたリスト 3.2 の説明．
     (b) 何を検証すれば動作が確認できたと言えるか．また，それを踏まえたリスト 3.6 の説明．
     (c) シミュレーションの結果，正しく動作していることをどのように確認できたか．

8　付録：同値分割，限界値分析に基づくテストケース選定の例

次の C プログラムを考える．これは入力した数 n が閉区間 $[20, 30]$ に属しているかどうか判定するプログラムである．

```
1  #include <stdio.h>
2  int main() {
3    int n;
4    printf("Enter n: ");
5    scanf("%d", &n);
6    if(n < 20 || n > 30) {
7      printf("%d doesn't lie in the interval [20, 30]\n", n);
8    } else {
9      printf("%d lies in the interval [20, 30]\n", n);
10   }
11   return 0;
12 }
```

この場合，同値分割として次のようなものが考えられる．

- $n < 20$ となる数 n．仕様から，これらに対してプログラムは「いいえ」を返さなければならない．
- $20 \leq n \leq 30$ となる数 n．仕様から，これらに対してプログラムは「はい」を返さなければならない．
- $n > 30$ となる数 n．仕様から，これらに対してプログラムは「いいえ」を返さなければならない．

各同値クラスから 1 つずつ数を選択して，例えば「10」「25」「42」として，これらをテストケースとする．これが同値分割に基づくテストケース選択の例である．

限界値分析に基づく場合，閉区間 $[20, 30]$ のちょうど端の部分で処理が分かれるので，その分かれ目をテストケースとする．この場合は，左端に対応する「19」と「20」，右端に対応する「30」と「31」の 4 つが選択されるテストケースとなる．これが限界値分析に基づくテストケース選択の例である．

いずれの場合においても (特に同値分割に基づく方法においては)，仕様を満たさない入力 (上の例の場合は'A' を入力することなど) もテストケースに含めることが一般的である．同値分割の場合，仕様を満たさない入力を 1 つの同値クラスとして扱い，それを無効同値クラスと呼んでいる．一方，仕様を満たす入力全体を有効同値クラスと呼んでいる．無効同値クラスからテストケースを選択して，実際テストすることで，ソフトウエアが想定外の入力を受け取ったときの振る舞いを見ることができ，システムの頑健性を観察できる．仕様を満たさない入力が渡されることを予め想定したり，その他の部分においてもあり得ないと思われることに対処できるようなプログラムを設計するスタイルを「防御的プログラミング」(defensive programming) と呼んでいる．