Programowanie układów FPGA

Wykład 3: modelowanie układów sekwencyjnych

Marek Materzok

22 października 2019

Modelowanie układów sekwencyjnych synchronicznych

Przypisanie nieblokujące

```
lhs <= expr;
```

- Wartość `expr` wyliczana od razu
- Przypisanie do rejestrów wykonywane na końcu kroku symulacji (ale przed monitorem)

Przykład – przerzutnik typu D

```
module d_flipflop(input clk, d, output reg q);
  always @(posedge clk) q <= d;
endmodule
```

W SystemVerilogu:

```
module d_flipflop(input clk, d, output reg q);
  always_ff @(posedge clk) q <= d;
endmodule
```

Przykład – przerzutnik typu D z resetem

```
module d_flipflop(input clk, rst, d, output reg q);
  always @(posedge clk or posedge rst)
    if (rst) q <= 1'b0;
    else q <= d;
endmodule
```

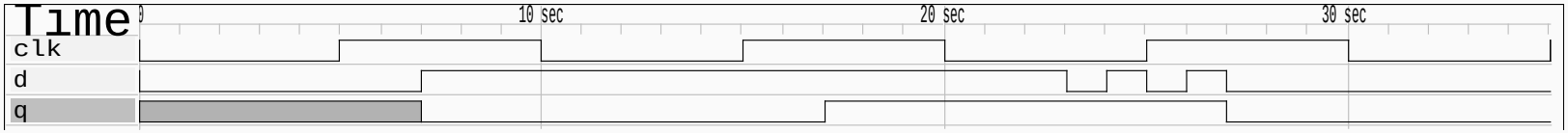
Inny przykład

```
module toggler(input clk, output reg v, w);
  initial v = 0;
  initial w = 1;
  always @(posedge clk) begin
    v <= w;
    w <= v;
  end
endmodule
```

Przerzutnik typu D z modelowanym opóźnieniem

```
module d_flipflop(input clk, d, output reg q);
  always @(posedge clk) q <= #2 d;
endmodule
```

Przebiegi




```
 0 clk=0 d=0 q=x
 5 clk=1 d=0 q=x
 7 clk=1 d=1 q=0
10 clk=0 d=1 q=0
15 clk=1 d=1 q=0
17 clk=1 d=1 q=1
20 clk=0 d=1 q=1
23 clk=0 d=0 q=1
24 clk=0 d=1 q=1
25 clk=1 d=0 q=1
26 clk=1 d=1 q=1
27 clk=1 d=0 q=0
30 clk=0 d=0 q=0
35 clk=1 d=0 q=0
```

Dlaczego przypisanie nieblokujące?

```
module bad_d_flipflop(input clk, d, output reg q);
  always @(posedge clk) q = d;
endmodule

module two_cycle_delay(input clk, d, output o);
  wire w;
  bad_d_flipflop d1(clk, w, o);
  bad_d_flipflop d2(clk, d, w);
endmodule
```

Testbench do złego przerzutnika

```
module testbench;
  reg clk, d;
  wire o;
  two_cycle_delay tcd(clk, d, o);
  always #1 clk = ~clk;
  initial begin
    clk = 0; d = 0;
    #4 d = 1;
    #4 d = 0;
    #2 d = 1;
    #10 $finish;
  end
  initial $monitor($time, " d=%d o=%d", d, o);
endmodule
```

Wynik

```
$ ./testbench
                0 d=0 o=x
                1 d=0 o=0 // za szybko!
                4 d=1 o=0
                5 d=1 o=1 // za szybko!
                8 d=0 o=1
                9 d=0 o=0 // za szybko!
               10 d=1 o=0
               13 d=1 o=1 // dobrze?!?
```

- Opóźnienie nie jest dwucyklowe!
- Ale czasami jest?!?

Wynik – dokładniej

```
$ ./testbench
```

```
 0 d=0 tcd.w=x o=x
 1 d=0 tcd.w=0 o=0
 4 d=1 tcd.w=0 o=0
 5 d=1 tcd.w=1 o=1
 8 d=0 tcd.w=1 o=1
 9 d=0 tcd.w=0 o=0
10 d=1 tcd.w=0 o=0
11 d=1 tcd.w=1 o=0
13 d=1 tcd.w=1 o=1
```

- W krokach 1, 5 i 8 wartość przechodzi przez oba przerzutniki w jednym cyklu
- Ale w 11 w dwa cykle

Inny wynik

```
$ ./testbench
                    0 d=0 tcd.w=x o=x
                    1 d=0 tcd.w=0 o=x
                    3 d=0 tcd.w=0 o=0 // dobrze
                    4 d=1 tcd.w=0 o=0
                    5 d=1 tcd.w=1 o=0
                    7 d=1 tcd.w=1 o=1 // dobrze
                    8 d=0 tcd.w=1 o=1
                    9 d=0 tcd.w=0 o=1
                   10 d=1 tcd.w=0 o=1
                   11 d=1 tcd.w=1 o=1 // gdzie 0?
```

- Niedeterministyczne zachowanie!

Zdebugujmy to

```
module bad_d_flipflop#(whoiam = 0)
    (input clk, d, output reg q);
  always @(posedge clk) begin
    $display(whoiam, " triggered with %d", d);
    q = d;
  end
endmodule

module two_cycle_delay(input clk, d, output o);
  wire w;
  bad_d_flipflop #(1) d1(clk, w, o);
  bad_d_flipflop #(2) d2(clk, d, w);
endmodule
```

Wynik

```
$ ./testbench
                    0 d=0 tcd.w=x o=x
              2 triggered with 0
              1 triggered with 0
                    1 d=0 tcd.w=0 o=0
              1 triggered with 0
              2 triggered with 0
                    4 d=1 tcd.w=0 o=0
              2 triggered with 1
              1 triggered with 1
                    5 d=1 tcd.w=1 o=1
```

- Niedeterministyczna kolejność wykonania!

Wniosek

Przypisanie nieblokujące pozwala uszeregować działanie logiki kombinacyjnej i sekwencyjnej:

1. Pojawia się zbocze zegarowe
2. Wyzwalane są bloki logiki sekwencyjnej
    - Widoczne wartości z poprzedniego cyklu
3. Zachodzą zmiany rejestrów
4. Zmiany wyzwalają bloki logiki kombinacyjnej

Register Transfer Level – RTL

- Styl opisu układów sekwencyjnych
- Zapewnia poprawne modelowanie układów z rejestrami
- W ogólności: zależności postaci „gdy pojawi się zbocze zegara C, zastąp wartość rejestru R wynikiem obliczenia logiki kombinacyjnej zależnej od rejestrów R1 do Rn”
- W Verilogu:
  - Logika kombinacyjna jako wyrażenia, bloki przypisania ciągłego `assign`, bloki `always` wyzwalane poziomem zawierające przypisania blokujące =, przypisujące każdą przypisywaną zmienną w każdej ścieżce kodu
  - Logika sekwencyjna jako bloki `always` wyzwalane zboczem zegarowym (lub zboczem zegarowym i zboczem reset), zawierające przypisania nieblokujące <=

Semantyka Veriloga

Semantyka zdarzeniowa

- Opis: IEEE 1364-2005, sekcja 11
- Procesy
  - Elementy podstawowe, moduły, bloki `initial` i `always`, przypisania ciągłe `assign`, taski asynchroniczne, przypisania proceduralne
  - Procesy mogą mieć stan, reagują na zdarzenia aktualizacji
- Zdarzenia
  - *Zdarzenia aktualizacji* – zmiana wartości zmiennej, nazwane zdarzenia
  - *Zdarzenia ewaluacji* – wykonanie procesu (np. w odpowiedzi na zdarzenie aktualizacji)
- Zdarzenia są szeregowane przez kolejkę priorytetową
  - Zadania o równym priorytecie wykonują się w dowolnej kolejności!

Priorytetowa kolejka zdarzeń

1. *Zdarzenia aktywne*

   bieżący cykl symulacji, kolejność dowolna

2. *Zdarzenia nieaktywne*

   bieżący cykl symulacji, po zdarzeniach aktywnych

3. *Zdarzenia przypisań nieblokujących*

   generowane przez <=, z bieżącego lub dawnych cykli symulacji

4. *Zdarzenia monitoringu*

   od `$monitor` i `$strobe`

5. *Zdarzenia przyszłe*

   uporządkowane czasem symulacji

   - *Przyszłe zdarzenia nieaktywne*
   - *Przyszłe zdarzenia przypisań nieblokujących*

Obsługa kolejki zdarzeń

```
while(kolejka_niepusta) {
  if(brak_aktywnych) {
    if(sa_nieaktywne)        aktywuj_wszystkie_nieaktywne;
    else if (sa_przyp_niebl) aktywuj_wszystkie_przyp_niebl;
    else if (sa_monitory)    aktywuj_wszystkie_monitory;
    else {
      T = wybierz_nastepny_czas_sym;
      aktywuj_przyszle_dla(T);
    }
  }
  E = dowolne_aktywne;
  if(zdarzenie_aktualizacji(E)) {
    zaktualizuj_zmienna_dla(E);
    wznow_procesy_czekajace_na(E);
  } else { /* zdarzenie ewaluacji */
    wykonaj_kod_procesu_dla(E);
    dodaj_wygenerowane_zdarz_aktualizacji_dla(E);
  }
}
```

Co jest deterministyczne?

Kolejność wykonania w blokach `begin-end` deterministyczna

choć specyfikacja dopuszcza przeplatanie pracujących procesów:

```
initial begin
  a = 1;
  b = 2;
end
initial begin
  a = 2;
  b = 1;
end
```

Możliwe wyniki: a=1 b=2, a=2 b=1, ale też a=1 b=1, a=2 b=2

Co jest deterministyczne?

Kolejność wykonania przypisań nieblokujących deterministyczna:

```
begin
  a <= 0;
  a <= 1;
end
```

Mimo, że przypisania są odraczane, ostatecznie są wykonywane w kolejności – na końcu a=1

Inny przykład niedeterminizmu

```
assign p = q;
initial begin
  q = 1;
  #1 q = 0;
  $display(p);
end
```

- Możliwe zarówno 0 i 1
- Jeśli opóźnić `$display` (nawet #0), na pewno 0
- Jeśli zamiast `$display` dać `$strobe`, na pewno 0

Semantyka przypisania ciągłego

```
assign lhs = expr;
```

- Proces uśpiony na zmiennych z `expr`
- Dodatkowo wykonywany w czasie 0 dla inicjalizacji
- Po obudzeniu dodaje zdarzenia aktualizacji do kolejki aktywnych
- Chyba że zawiera opóźnienie, wtedy zdarzenie trafia do kolejki przyszłych, wcześniejsze zdarzenia w kolejce są usuwane

Semantyka przypisania blokującego

```
lhs = expr;
```

- Oblicza `expr` natychmiast
- Jeśli jest opóźnienie, proces jest usypiany
- Po wznowieniu (lub natychmiast, przy braku opóźnienia) wykonywane jest przypisanie
- Dodaje do kolejki aktywnych zdarzenie aktualizacji `lhs`

Semantyka przypisania nieblokującego

```
lhs <= expr;
```

- Oblicza `expr` natychmiast
- Dodaje zdarzenie przypisania nieblokującego do bieżącego czasu (jeśli brak opóźnienia) lub przyszłego (z opóźnieniem)

Semantyka połączeń modułów

```
modul inst(expr1, expr2, expr3);
```

- Dla portów wejściowych równoważne z

  ```
  assign port = expr;
  ```

- Dla portów wyjściowych równoważne z

  ```
  assign expr = port;
  ```

  (wyrażenie musi być poprawną lewą stroną)

Syntezowalny podzbiór Veriloga

Etapy syntezy

- Generowanie netlisty

  konstrukcje wysokopoziomowe są zamieniane na logikę cyfrową: rejestry, multipleksery, sumatory, bramki...
- Optymalizacja netlisty

  pod kątem powierzchni, opóźnień, zużywanej energii
- Technology mapping

  dopasowanie bloków realizowalnych w danej technologii produkcyjnej lub danym układzie FPGA
- Fitting (place & route)

  fizyczne rozmieszczenie bloków na chipie lub FPGA
- Analiza timingu

  maksymalne opóźnienie, częstotliwość, setup/hold, metastabilność

Syntezowalność

- Brak dobrej definicji, **zależy od narzędzia**!
- Możliwe rozbieżności między semantyką symulacji a zachowaniem układu!
- Niesyntezowalne konstrukcje często **ignorowane** i nie wywołują błędów!

Styl RTL

- Logika kombinacyjna:
  - wyrażenia
  - bloki przypisania ciągłego `assign`
  - bloki `always` wyzwalane poziomem
  - tylko przypisania blokujące =
  - przypisanie każdej przypisywanej zmiennej w każdej ścieżce kodu
- Logika sekwencyjna:
  - bloki `always` wyzwalane zboczem zegarowym (i sygnałem reset)
  - tylko przypisania nieblokujące <=

Niesyntezowalne – opóźnienia

- Opóźnienia w Verilogu używa się do *modelowania* i *testbenchów*
- Rzeczywiste opóźnienia zależą od użytej technologii i fizycznego ułożenia
- Narzędzia do syntezy obliczają maksymalne opóźnienia w docelowym układzie

Niesyntezowalne – pętle

- W ogólności pętle są niesyntezowalne
- Jeśli liczba iteracji znana w czasie kompilacji – syntezowalne przez rozwinięcie pętli
- Pętle generujące syntezowalne

```
integer k;
always @(posedge clk) begin
  buffer[0] <= sig;
  for (k = 0; k < N-1; k = k+1)
    buffer[k+1] <= buffer[k];
end
```

Niesyntezowalne – bloki initial

- Bloki initial w ogólności niesyntezowalne
- Należy dodać obsługę sygnału resetującego

```
module parity(input clk, rst, en, in, output reg out);
  always @(posedge clk or posedge rst)
    if (rst) r <= 0;
    else if (en) r <= r ^ in;
endmodule
```

Niesyntezowalne – przypisanie przez wiele procesów

- Każdy `reg` powinien być przypisywany przez dokładnie jeden blok `always`
- Każdy `wire` powinien być przypisywany dokładnie raz (przez `assign` lub instancję)

```
always @(posedge clk)
  r <= e1;
always @(posedge rst)
  r <= 0;
```

Niesyntezowalne – bity x

- Bity wartości nieznanej `x` niesyntezowalne – brak fizycznego odpowiednika
- Stan wysokiej impedancji `z` obsługiwany tylko dla portów I/O
- Porównywanie do `x` i `z` niesyntezowalne
- Ale `casez` syntezowalny (z zastrzeżeniami)

```
n = 0; v = 1;
casez(a) // a nie zawiera x ani z!
  2'b1?: n = 0;
  2'b?1: n = 1;
  default: v = 0;
endcase
```