Logika cyfrowa

Wykład 8: Układy sekwencyjne w SystemVerilogu

Marek Materzok

21 kwietnia 2021

Abstrakcje sekwencyjne

Terminologia

- Układy bez stanu nazywamy **kombinacyjnymi**.
- Układy ze stanem nazywamy **sekwencyjnymi**.
- Układy bez wspólnego zegara nazywamy **asynchronicznymi**.
- Układy ze wspólnym zegarem nazywamy **synchronicznymi**.

Powyżej „wspólny zegar” oznacza: jeden sygnał zegarowy wyzwalający wszystkie przerzutniki w układzie.

Dlaczego abstrakcje sekwencyjne?

Elementy stanowe symulowane na poziomie bramek są kłopotliwe:

- Konieczność przestrzegania czasów ustalania i podtrzymania
- Ryzyko wystąpienia oscylacji w symulatorze
- Nieefektywna symulacja
- Niemożliwa poprawna synteza na FPGA

Rozwiązanie: podniesienie poziomu abstrakcji (!)

* jeśli macie déjà vu z piątego wykładu to dobrze

Abstrakcyjne elementy stanowe

Abstrakcyjny przerzutnik jest „magiczny”:

- ma zerowy czas propagacji (w DigitalJS może być niezerowy)
- ma zerowy czas ustalania, podtrzymania, powrotu, odwołania
- nie ma stanów metastabilnych

Narzędzie do syntezy rozwiązuje problemy realnych przerzutników

...jeśli mu pozwolimy, **przestrzegając pewnych reguł**

Specyfikowanie elementów stanowych

- Specyfikacja proceduralna – za pomocą *kodu*
- Nowe rodzaje bloków `always`:
  - `always_latch` – wyzwalanie poziomem
  - `always_ff` – wyzwalanie zboczem (**preferowane!**)
- Modelowanie stanu za pomocą zmiennych
- Poprawne użycie ma inne ograniczenia niż `always_comb`

Specyfikowanie elementów stanowych

- Specyfikacja proceduralna – za pomocą *kodu*
- Nowe rodzaje bloków `always`:
  - `always_latch` – wyzwalanie poziomem
  - `always_ff` – wyzwalanie zboczem (**preferowane!**)
- Modelowanie stanu za pomocą zmiennych
- Poprawne użycie ma inne ograniczenia niż `always_comb`

PUŁAPKA: nieprawidłowy kod **może** nie być odrzucony w syntezie, lecz wygenerować układ o **niespodziewanym zachowaniu**!

Może też zsyntezować się do pożądanego układu, ale źle symulować się w symulatorach wysokiego poziomu (np. Icarus Verilog)!

Zatrzask

```
module d_latch(output logic q, input d, en);
    always_latch
        if (en) q = d;
endmodule
```

Uwaga: standard IEEE 1800 wymaga, aby porty wyjściowe, do których przypisuje się w blokach `always`, miały oznaczony typ (np. `logic`). Yosys to ignoruje, inne narzędzia mogą zwracać błąd.

Asynchroniczny przerzutnik SR

```
module sr_latch(output logic q, input s, r);
    always_latch begin
        if (s) q = 1;
        if (r) q = 0;
    end
endmodule
```

Uwaga: w DigitalJS nie symuluje się zgodnie z intencją ze względu na symulowane opóźnienia kombinacyjne!

Przerzutnik SR wyzwalany poziomem

```
module gated_sr_latch(output logic q, input s, r, en);
    always_latch if(en) begin
        if (s) q = 1;
        if (r) q = 0;
    end
endmodule
```

Uwaga: w DigitalJS nie symuluje się zgodnie z intencją ze względu na symulowane opóźnienia kombinacyjne!

Przerzutniki wyzwalane poziomem – uwagi

- Używać `always_latch` do zaznaczenia intencji modelowania przerzutników wyzwalanych *poziomem*
- Przerzutniki wyzwalane poziomem **nie mają wsparcia sprzętowego na wielu FPGA**!
- Układy wyzwalane poziomem sa narażone na trudne problemy związane z opóźnieniami w układzie – **unikać**!
- **Używać tylko w uzasadnionych sytuacjach**!

Przerzutnik D wyzwalany zboczem

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

Przerzutnik T

```
module tff(output logic q, input t, clk);
    always_ff @(posedge clk)
        q <= q ^ t;
endmodule
```

Przerzutnik T z resetem synchronicznym

```
module tff(output logic q, input t, clk, nrst);
    always_ff @(posedge clk)
        if (!nrst) q <= 0;
        else q <= q ^ t;
endmodule
```

Przerzutnik T z resetem asynchronicznym

```
module tff(output logic q, input t, clk, nrst);
    always_ff @(posedge clk, negedge nrst)
        if (!nrst) q <= 0;
        else q <= q ^ t;
endmodule
```

Bloki `always_ff`, instrukcja oczekiwania

- Blok `always_ff` używa się do zaznaczenia intencji modelowania przerzutników wyzwalanych *zboczem*.
- Instrukcja `@()` oznacza uzależnienie wykonania operacji od wystąpienia zdarzenia:
  - zmiany stanu sygnału: `s`
  - zbocza narastającego: `posedge s`
  - zbocza opadającego: `negedge s`
- Typowo wykrywa się zbocze **zegara** i **resetu asynchronicznego**.
  Wykrywanie zboczy innych sygnałów lubi prowadzić do problemów związanych z opóźnieniami, glitchami itp. – **unikać**!

Przypisanie blokujące a nieblokujące

Przypisanie blokujące (=):

- oblicza wyrażenie po prawej i przypisuje je **od razu**
- zastosowanie: logika kombinacyjna (`always_comb`), zatrzaski (`always_latch`), zmienne tymczasowe

Przypisanie nieblokujące (<=):

- oblicza wyrażenie po prawej, ale przypisanie wartości jest **odroczone** do kolejnej „chwili czasu”
- przypisania nieblokujące są (w pewnym sensie) **współbieżne**
- zastosowanie: przerzutniki wyzwalane zboczem (`always_ff`), logika synchroniczna

Przypisanie blokujące a nieblokujące – przykład

Przypisanie blokujące – obydwa sygnały zyskują wartość b:

```
a = b; // teraz a == b
b = a; // nic nie zmienia
```

Przypisanie nieblokujące – wartości sygnałów zamieniają się miejscami:

```
a <= b; // w 'a' będzie 'b'
b <= a; // w 'b' będzie 'a'
```

Nieprawidłowe użycie przypisania blokującego

```
module two_dff(output logic q1, q2, input d, clk);
    always_ff @(posedge clk) q1 = d;
    always_ff @(posedge clk) q2 = q1;
endmodule
```

Jest możliwe, że q2 zaktualizuje się do d w jednym cyklu! (w symulacji wysokiego poziomu, bez syntezy)

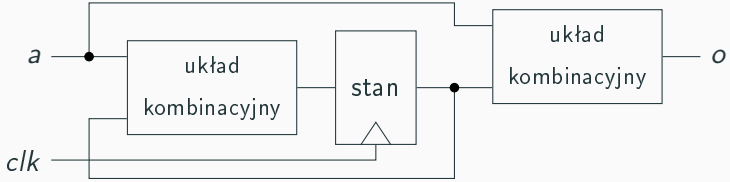
Poprawnie:

```
module two_dff(output logic q1, q2, input d, clk);
    always_ff @(posedge clk) q1 <= d;
    always_ff @(posedge clk) q2 <= q1;
endmodule
```

RTL – Register Transfer Level

Układy synchroniczne

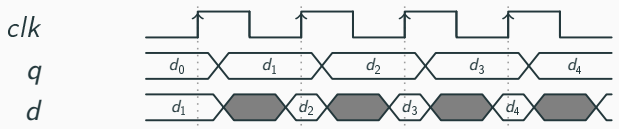
- Elementy stanowe: tylko przerzutniki wyzwalane zboczem
- Wszystkie przerzutniki wyzwalane jednym zegarem
- Przerzutniki połączone ze sobą, wejściami i wyjściami przez układy kombinacyjne



Dlaczego układy synchroniczne?

Cechy układów synchronicznych:

- Prosty model czasu: czas dyskretny
  - Chwile czasu liczone zboczami zegara
  - Stan w chwili $n+1$ zależy tylko od stanu w chwili n oraz wejść
- Odporność na glitche
  - Okres zegara > ścieżka krytyczna + czas ustalania
  - Zmiany sygnału pomiędzy zboczami zegara ignorowane



Obecnie praktycznie każdy większy układ cyfrowy jest układem synchronicznym!

Kod RTL – Register Transfer Level

- Styl specyfikacji sprzętu mechanicznie, jednoznacznie syntezowalny do schematu
- Logika kombinacyjna:
  - wyrażenia
  - bloki przypisania ciągłego `assign`
  - bloki `always_comb` wyzwalane poziomem
  - tylko przypisania blokujące `=`
  - przypisanie każdej przypisywanej zmiennej w każdej ścieżce kodu
- Logika sekwencyjna synchroniczna:
  - bloki `always_ff` wyzwalane zboczem zegarowym (i resetu)
  - reset asynchroniczny: `if` na początku bloku
  - tylko przypisania nieblokujące `<=`

Uwaga!

- Przypominam: pisanie kodu proceduralnego wygląda jak programowanie, ale nim NIE JEST.
- Każdy blok `always_ff` opisuje osobny obwód sekwencyjny.
- Każda zmienna przypisywana nieblokująco <= w bloku `always_ff` reprezentuje rejestr.
- Należy unikać pokusy opisywania całego układu jednym blokiem kodu!
- Należy wybierać użyty rodzaj bloku (`assign`, `always_comb`, `always_ff`) w zależności od potrzeb.
- Blok `always_ff` oblicza **nowy** stan z **poprzedniego**.
- Należy stosować **jak najmniej** stanu!

Przykłady układów synchronicznych w SystemVerilogu

Przykład: rejestr z resetem asynchronicznym

```
module reg16(
    output logic [15:0] q,
    input [15:0] d,
    input clk, nrst
);
    always_ff @(posedge clk, negedge nrst)
        if (!nrst) q <= 0;
        else q <= d;
endmodule
```

Przykład: rejestr z wejściem `enable`

```
module reg16_en(
    output logic [15:0] q,
    input [15:0] d,
    input clk, nrst, en
);
    always_ff @(posedge clk, negedge nrst)
        if (!nrst) q <= 0;
        else if (en) q <= d;
endmodule
```

Przykład: rejestr z multiplekserem

```
module muxreg16(
    output logic [15:0] q,
    input [15:0] d0, d1,
    input clk, sel
);
    always_ff @(posedge clk)
        if (sel) q <= d1;
        else q <= d0;
endmodule
```

Przykład: rejestr z multiplekserem, dwoma blokami

```
module muxreg16(
    output [15:0] q,
    input [15:0] d0, d1,
    input clk, sel
);
    logic [15:0] d;
    assign d = sel ? d1 : d0;
    always_ff @(posedge clk)
        q <= d;
endmodule
```

Przykład: rejestr przesuwny

```
module shift4(
    output logic [15:0] q,
    input i, clk, en
);
    always_ff @(posedge clk)
        if (en) q <= {i, q[15:1]};
endmodule
```

Przykład: licznik

```
module upcount(
    output logic [3:0] q,
    input clk, nrst, en
);
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else if (en) q <= q + 4'd1;
endmodule
```

Przykład: licznik z ładowaniem równoległym

```
module upcount_load(
    output logic [3:0] q,
    input [3:0] i,
    input clk, nrst, en, load
);
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else if (load) q <= i;
        else if (en) q <= q + 4'd1;
endmodule
```

Przykład: licznik z ładowaniem równoległym

```
module upcount_load(
    output logic [3:0] q,
    input [3:0] i,
    input clk, nrst, en, load
);
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else if (load) q <= i;
        else if (en) q <= q + 4'd1;
endmodule
```

Przykład: licznik z ładowaniem równoległym, wersja 2

```
module upcount_load(
    output logic [3:0] q,
    input [3:0] i,
    input clk, nrst, en, load
);
    logic [3:0] d;
    assign d = load ? i : q + 4'd1;
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else if (load || en) q <= d;
endmodule
```

Przykład: licznik z ładowaniem równoległym, wersja 3

```
module upcount_load(
    output logic [3:0] q,
    input [3:0] i,
    input clk, nrst, en, load
);
    logic [3:0] d;
    assign d = load ? i : en ? q + 4'd1 : q;
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else q <= d;
endmodule
```

BŁĘDNY przykład: licznik kodów Graya

```
module bad_gray_counter(
    output logic [3:0] o,
    input clk, nrst
);
    logic [3:0] q;
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else begin
            q <= q + 4'd1;
            o <= q ^ (q >> 1);
        end
endmodule
```

BŁĘDNY przykład, wersja 2: licznik kodów Graya

```
module bad2_gray_counter(
    output logic [3:0] o,
    input clk, nrst
);
    logic [3:0] q;
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q = 0;
        else begin
            q = q + 4'd1;
            o = q ^ (q >> 1);
        end
endmodule
```

Przykład: licznik kodów Graya

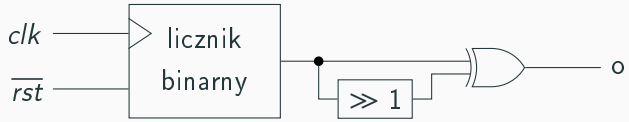
```
module gray_counter(
    output [3:0] o,
    input clk, nrst
);
    logic [3:0] q;
    always_ff @(posedge clk or negedge nrst)
        if (!nrst) q <= 0;
        else q <= q + 4'd1;
    assign o = q ^ (q >> 1);
endmodule
```

Porównanie – licznik kodów Graya

```
module compare_counters(
    output [3:0] o1, o2, o3,
    input clk, nrst
);
    gray_counter      c1(o1, clk, nrst);
    bad_gray_counter  c2(o2, clk, nrst);
    bad2_gray_counter c3(o3, clk, nrst);
endmodule
```

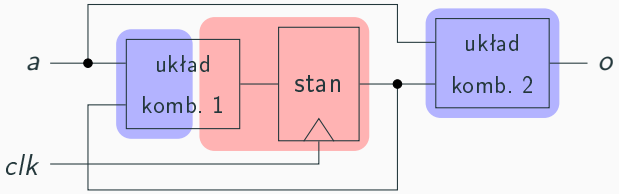
Co poszło nie tak?

Schemat licznika kodów Graya:



- Układ kombinacyjny liczący kod Graya podłączony do **wyjścia**.
- Zmienne przypisywane w `always_ff` odpowiadają **wyjściom przerzutników**.
- Wniosek – układów kombinacyjnych na wyjściu **nie można** wyrazić w bloku `always_ff`.
- Należy użyć wtedy `assign` lub `always_comb`.

Użycie bloków – przewodnik



- Stan – bloki `always_ff`
- Układ komb. 1 – bloki `assign/always_comb` *lub* w bloku `always_ff` powiązanego przerzutnika
- Układ komb. 2 – bloki `assign/always_comb`

Przykład: układ liczący silnię

Silnia – definicja

Silnia liczby naturalnej n – iloczyn wszystkich liczb naturalnych dodatnich nie większych niż n. Definicję można napisać w formie zwartej:

$$n! = \prod_{k=1}^{n} k$$

Albo w formie zależności rekurencyjnej:

$$n! = \begin{cases} 1 & \text{gdy } n = 0 \\ n \cdot (n-1)! & \text{w p.w.} \end{cases}$$

Układ liczący silnię – możliwe podejścia

- Układ kombinacyjny
  - Minus: układ szybko rośnie z maksymalnym akceptowanym n
  - Minus: długa ścieżka krytyczna
  - Plus: przekształca wprost wartość wyjściową w wynik

Układ liczący silnię – możliwe podejścia

- Układ kombinacyjny
  - Minus: układ szybko rośnie z maksymalnym akceptowanym n
  - Minus: długa ścieżka krytyczna
  - Plus: przekształca wprost wartość wyjściową w wynik
- Układ sekwencyjny
  - Plus: rozmiar układu zależny od liczby bitów obliczanych wartości
  - Plus: ogólna konstrukcja układu niezależna od rozmiarów wejść i wyjść
  - Minus: konieczność istnienia dodatkowych sygnałów sterujących

Układ liczący silnię – wejścia i wyjścia

Potrzebna jest metoda wprowadzania i wyprowadzania danych:

- Wejście `n` – liczba n w kodzie binarnym
- Wyjście `s` – liczba $n!$ w kodzie binarnym

Przyjmijmy wejście 4-bitowe. Wartość 15! zmieści się w 41 bitach.

Układ liczący silnię – wejścia i wyjścia

Potrzebna jest metoda wprowadzania i wyprowadzania danych:

- Wejście `n` – liczba n w kodzie binarnym
- Wyjście `s` – liczba $n!$ w kodzie binarnym

Przyjmijmy wejście 4-bitowe. Wartość 15! zmieści się w 41 bitach.

Potrzeba również kilku 1-bitowych sygnałów sterujących:

- Wejście zegarowe `clk` – wyzwala kolejne mnożenia
- Wejście startu `ini` – rozpoczyna obliczenia
- Wyjście statusu `fin` – sygnalizuje koniec obliczeń

Układ liczący silnię – stan

Jakich rejestrów potrzebujemy?

- Akumulator 41-bitowy `acc` – przechowuje wartość pośrednią i wynik końcowy
- Licznik 4-bitowy `cnt` – odlicza wartości do wymnożenia

Układ liczący silnię – stan

Jakich rejestrów potrzebujemy?

- Akumulator 41-bitowy `acc` – przechowuje wartość pośrednią i wynik końcowy
- Licznik 4-bitowy `cnt` – odlicza wartości do wymnożenia

Czy potrzeba innych rejestrów?

Układ liczący silnię – stan

Jakich rejestrów potrzebujemy?

- Akumulator 41-bitowy `acc` – przechowuje wartość pośrednią i wynik końcowy
- Licznik 4-bitowy `cnt` – odlicza wartości do wymnożenia

Czy potrzeba innych rejestrów?

Nie:

- Licznik może odliczać od n do 1 (lub 0)

  nie trzeba pamiętać n
- Koniec obliczeń gdy licznik 1 lub 0

  wyjście `fin` można generować kombinacyjnie

Układ liczący silnię – transfer rejestrów

Jak wartości rejestrów w chwili t zależą od wartości w chwili $t - 1$?

Układ liczący silnię – transfer rejestrów

Jak wartości rejestrów w chwili t zależą od wartości w chwili $t-1$?

Kiedy $\texttt{ini}_{t-1} = 1$:

- $\texttt{acc}_t \leftarrow 1$
- $\texttt{cnt}_t \leftarrow \texttt{n}_{t-1}$

Układ liczący silnię – transfer rejestrów

Jak wartości rejestrów w chwili t zależą od wartości w chwili $t-1$?

Kiedy $\texttt{ini}_{t-1} = 1$:

- $\texttt{acc}_t \leftarrow 1$
- $\texttt{cnt}_t \leftarrow \texttt{n}_{t-1}$

Kiedy $\texttt{ini}_{t-1} = 0$ i $\texttt{cnt}_{t-1} > 1$:

- $\texttt{acc}_t \leftarrow \texttt{acc}_{t-1} \cdot \texttt{cnt}_{t-1}$
- $\texttt{cnt}_t \leftarrow \texttt{cnt}_{t-1} - 1$

Układ liczący silnię – transfer rejestrów

Jak wartości rejestrów w chwili t zależą od wartości w chwili $t-1$?

Kiedy $\texttt{ini}_{t-1} = 1$:

- $\texttt{acc}_t \leftarrow 1$
- $\texttt{cnt}_t \leftarrow \texttt{n}_{t-1}$

Kiedy $\texttt{ini}_{t-1} = 0$ i $\texttt{cnt}_{t-1} > 1$:

- $\texttt{acc}_t \leftarrow \texttt{acc}_{t-1} \cdot \texttt{cnt}_{t-1}$
- $\texttt{cnt}_t \leftarrow \texttt{cnt}_{t-1} - 1$

W przeciwnym wypadku wartości rejestrów nie zmieniają się.

Układ liczący silnię – transfer rejestrów

Jak wartości rejestrów w chwili t zależą od wartości w chwili $t-1$?

Kiedy $\texttt{ini} = 1$:

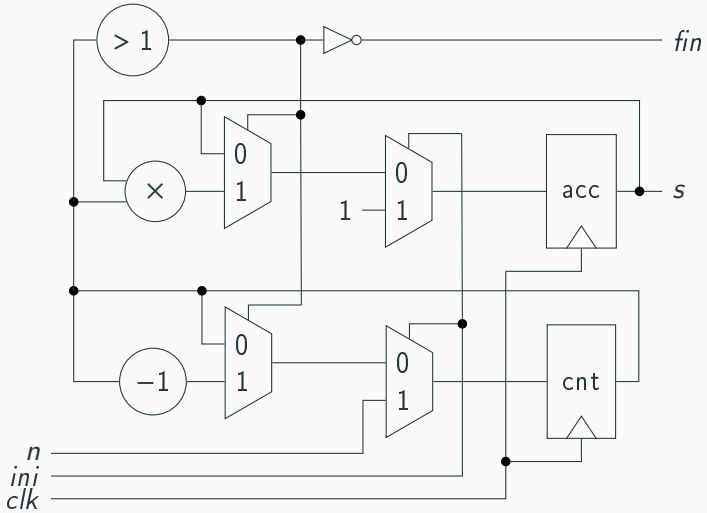
- $\texttt{acc} \leftarrow 1$
- $\texttt{cnt} \leftarrow \texttt{n}$

Kiedy $\texttt{ini} = 0$ i $\texttt{cnt} > 1$:

- $\texttt{acc} \leftarrow \texttt{acc} \cdot \texttt{cnt}$
- $\texttt{cnt} \leftarrow \texttt{cnt} - 1$

W przeciwnym wypadku wartości rejestrów nie zmieniają się.

Układ liczący silnię – schemat



Układ liczący silnię – SystemVerilog, wersja 1

```
module silnia(
  input [3:0] n,
  input ini, clk,
  output [40:0] s,
  output fin
);
  logic cmp;
  logic [40:0] acc, newacc;
  logic [3:0] cnt, newcnt;
  assign cmp = cnt > 4'b1;
  assign fin = !cmp;
  assign s = acc;
  assign newacc =
    ini ? 41'b1 :
    cmp ? acc * cnt : acc;
  assign newcnt =
    ini ? n :
    cmp ? cnt - 4'b1: cnt;
  always_ff @(posedge clk)
    acc <= newacc;
  always_ff @(posedge clk)
    cnt <= newcnt;
endmodule
```

Układ liczący silnię – SystemVerilog, wersja 2

```
module silnia(
  input [3:0] n,
  input ini, clk,
  output [40:0] s,
  output fin
);
  logic cmp;
  logic [40:0] acc;
  logic [3:0] cnt;
  assign cmp = cnt > 4'b1;
  assign fin = !cmp;
  assign s = acc;
  always_ff @(posedge clk)
    if (ini) begin
      acc <= 41'b1;
      cnt <= n;
    end else if (cmp) begin
      acc <= acc * cnt;
      cnt <= cnt - 4'b1;
    end
endmodule
```