Verilog. Czyli jak zaprojektować automat.

K. Swientek

AGH, WFiIS

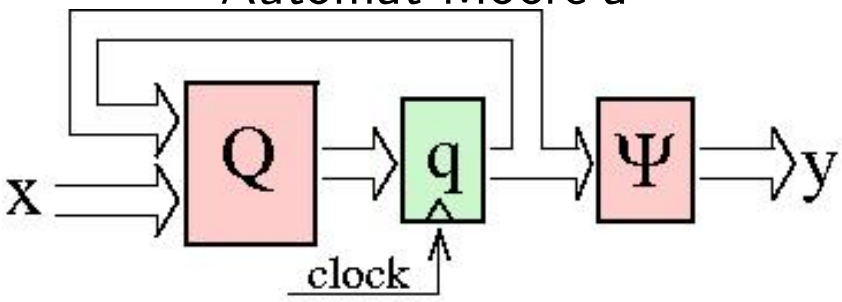
Październik 2008, Kraków

Spis treści

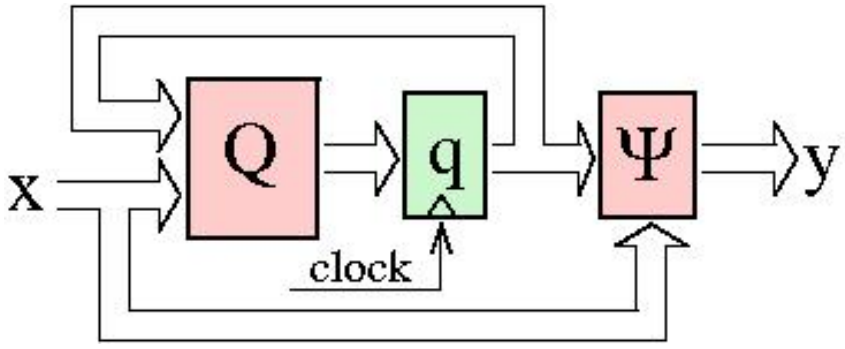
Spis treści

Automat albo maszyna skończona

Automat Moore'a



Automat Mealy'ego



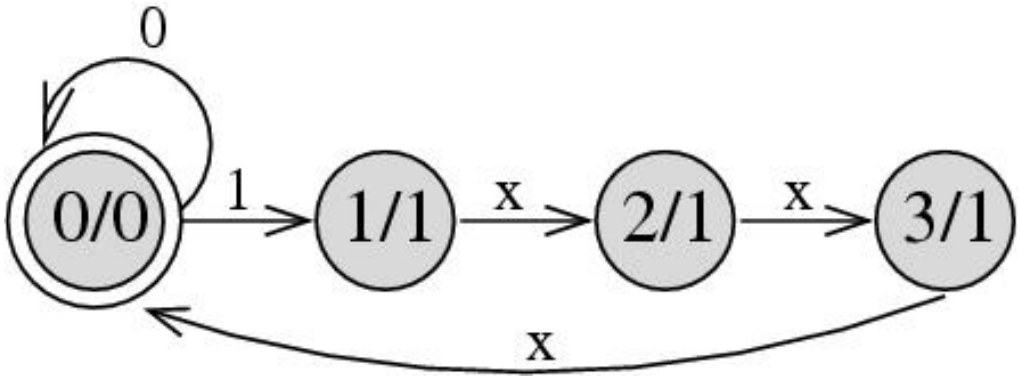
- Nazewnictwo: automat skończony, maszyna stanów skończonych (o skończonej liczbie stanów)
- Działanie:
  - dwie funkcje logiczne Q i Ψ plus element q pamiętający stan
  - $q_{n+1} = Q(q_n, x)$
  - $y = \Psi(q)$
- Automat Mealy'ego jest bardziej ogólny, ale może powodować problemy z zależnościami czasowymi jeśli jest kilka w szeregu
- Można przeprowadzić jeden automat w drugi
- W Verilogu będą trzy bloki odpowiadające q, Q i Ψ

Diagram stanów maszyny Moore'a

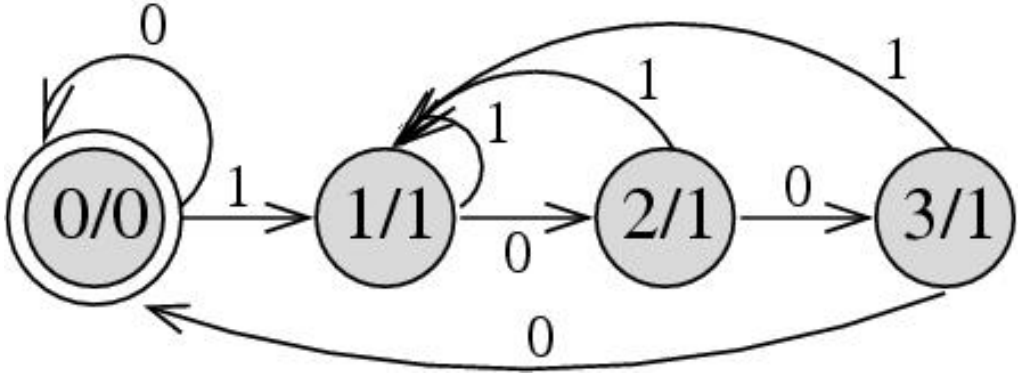
Przykład

Zbudujmy automat, który w odpowiedzi na jedynkę na wejściu wygeneruje ciąg 3 jedynek na wyjściu.

Bez przedłużania impulsu



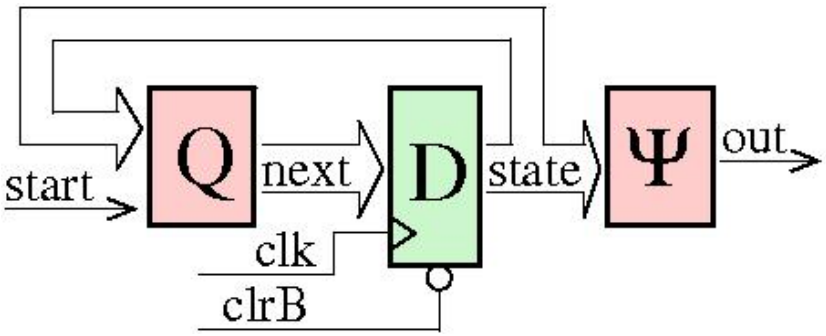
Z przedłużeniem impulsu

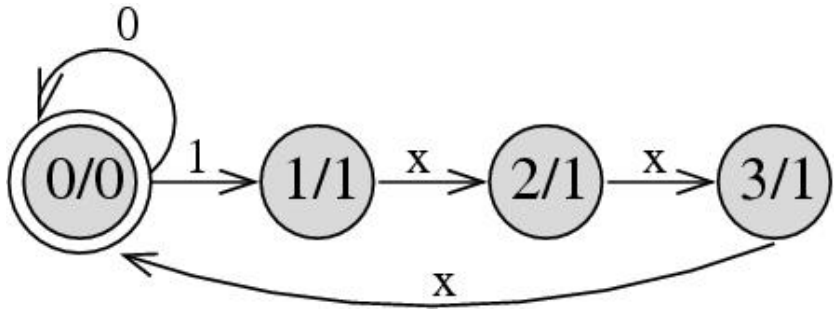


- Podwójne koło to start (reset)
- Zawartość koła: numer stanu maszyny i wartość wyjścia
- Liczby na strzałkach to warunki (wartość na wejściu) przejścia ze stanu do stanu

Do Veriloga przystąp!

Automat Moore'a





Modułu w verilogu

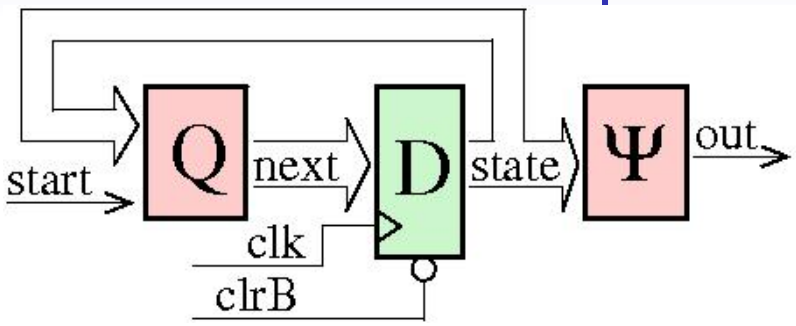
```
module imp3clk(out, clk, clrB, start);
output out;
input clk, clrB, start;
//inout ... ;

reg [0:1] state, next; //Rejestry
//wire ... ; //typ domyślny
   //Zawartość modułu
endmodule
```

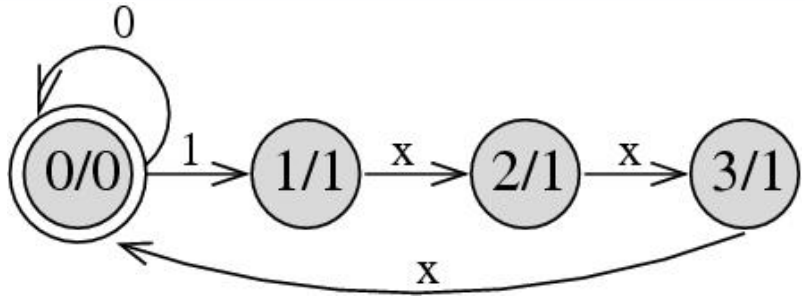
Funkcja Ψ

```
assign out = state[0] | state[1];

//Druga wersja
reg out; //Wymagane !!
always @(state) begin
  if (state==0) out = 1'b0; //1-bitowe
          else out = 1'b1; //liczby
  end
```

- `assign` to przypisanie ciągłe natomiast `@(...)` oznacza zdarzenie (*event*) na które blok ma zareagować. W tym wypadku zmiana wartości zmiennej `state`.

Opis w Verilogu c.d.





Funkcja *Q*

```
always @(state or start) begin
  case (state)
    2'b00: if (start==0) next = 2'b00;
           else next = 2'b01;
    2'b01: next = 2'b10; //Dwubitowe
    2'b10: next = 2'b11; //liczby
    2'b11: next = 2'b00; //binarne
  endcase
end
```

Blok D (element pamiętający)

```
always @(posedge clk or negedge clrB)
begin
  if (!clrB) state <= 2'b00;
    else state <= next;
end
```

- = to przypisanie blokujące
- <= to nieblokujące

- inne formaty liczb 6'h1a, 7'd2, 5 (domyślne 32 bity)
- Nigdy nie mieszmy ze sobą = i <= w jednym bloku `always`. Jeżeli scalimy *Q* z blokiem *D* zostaje tylko <=.

Przypisanie blokujące i nieblokujące

Przypisanie blokujące

```
module fbosc1(y1, y2, clk, clrB);
  output y1, y2;
  input clk, clrB;
  reg y1, y2;

always @(posedge clk or negedge clrB)
  if (!clrB) y1 = 0;
  else       y1 = y2;

always @(posedge clk or negedge clrB)
  if (!clrB) y2 = 1;
  else       y2 = y1;
endmodule
```

Przypisanie nieblokujące

```
module fbosc2(y1, y2, clk, clrB);
  output y1, y2;
  input clk, clrB;
  reg y1, y2;

always @(posedge clk or negedge clrB)
  if (!clrB) y1 <= 0;
  else       y1 <= y2;

always @(posedge clk or negedge clrB)
  if (!clrB) y2 <= 1;
  else       y2 <= y1;
endmodule
```

- W przypisaniu nieblokującym najpierw wyznaczana jest prawa strona wszystkich wyrażeń, a dopiero potem następuję podstawianie
- Dla przypisania blokującego kolejność wykonania bloków (powyżej) jest nieokreślona; dla nieblokującego symulacja zachowuje się jakby przypisania zachodziły równolegle. I o to chodzi!

Synteza a rodzaje przypisań

Całkiem źle

```
module pipe1(q3, clk, d);
  output [5:0] q3;
  input [5:0] d;
  input clk;
  reg [5:0] q3, q2, q1;

  always @(posedge clk)
  begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

Niby działa...

```
module pipe2(q3, clk, d);
  output [5:0] q3;
  input [5:0] d;
  input clk;
  reg [5:0] q3, q2, q1;

  always @(posedge clk)
  begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Zawsze dobrze

```
module pipe3(q3, clk, d);
  output [5:0] q3;
  input [5:0] d;
  input clk;
  reg [5:0] q3, q2, q1;

  always @(posedge clk)
  begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

- Wynik sekwencji przypisań blokujących zależy od kolejności
- Różna kolejność przypisań blokujących daje różne wyniki syntezy
- Dla przypisań nieblokujących wynik sekwencji jest zawsze ten sam

Pełny kod przykładów

Bez przedłużania

```
module imp3clk(out, clk, clrB, start);
output out;
input clk, clrB, start;
reg [0:1] state, next;

assign out = state[0] | state[1];

always @(posedge clk or negedge clrB)
  if (!clrB) state <= 2'b00;
    else state <= next;

always @(state or start) begin
  case (state)
    2'b00: if (start==0) next = 2'b00;
           else next = 2'b01;
    2'b01: next = 2'b10;
    2'b10: next = 2'b11;
    2'b11: next = 2'b00;
  endcase
end
endmodule
```

Z przedłużeniem

```
module imp3clkRep(out, clk, clrB, start);
output out;
input clk, clrB, start;
reg [0:1] state, next;

assign out = state[0] | state[1];

always @(posedge clk or negedge clrB)
  if (!clrB) state <= 2'b00;
    else state <= next;

always @(state or start) begin
  case (state)
    2'b00: if (start==0) next = 2'b00;
           else next = 2'b01;
    2'b01: if (start==0) next = 2'b10;
           else next = 2'b01;
    2'b10: if (start==0) next = 2'b11;
           else next = 2'b01;
    2'b11: if (start==0) next = 2'b00;
           else next = 2'b01;
  endcase
```

Parametry są super!

Licznik

```
//`define WIDTH 4
module parCounter(value, clk, clrB,
                  synchClr, pulse);
parameter WIDTH = 4;

output value;
input clk, clrB, synchClr, pulse;
reg [WIDTH-1:0] value;

always @(posedge clk or negedge clrB)
  if (!clrB) value <= 0;
  else if (synchClr)
    value <= 0;
  else if (pulse)
    value <= value + 1;
endmodule
```

Jak go użyć?

```
...
wire already100;
assign #4 already100 =
  (counterValue == 10'd200) ? 1'b1 : 1'b0;

parCounter count(.value(counterValue),
    .clk(clk), .clrB(clrB),
    .synchClr(start), .pulse(pulseEnd));
defparam count.width = 10;
...

//Druga wersja
parCounter #(10) count(counterValue,
    clk, clrB, start, pulseEnd);
...
```

- Unikanie `defparam` jest dobrym pomysłem
- Symbol # ma dwa różne znaczenia: wartość parametru lub opóźnienie

Wstawianie opóźnień

Opóźnienie inercyjne

```
assign #4 out = state[0] | state[1];

// sygnał na wyjściu out zmieni się
// jeżeli wartość state pozostanie
// stabilna przynajmniej przez
// 4 jednostki czasu (zazwyczaj ns)
```

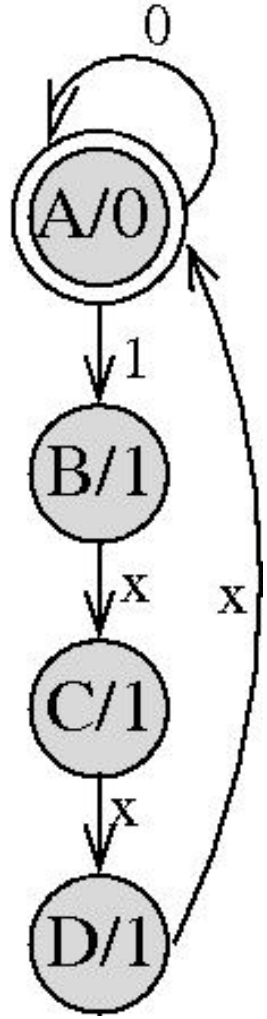
Opóźnienie transportowe

```
always @(a or b or c)
    {co, sum} <= #12 a + b + ci;
// sygnały na wyjściach co i sum
// zmieniają się zawsze 12 jednostek
// czasu po zmianie któregokolwiek
// z wejść a, b lub ci
```

- W Verilogu jest 5 rodzajów opóźnień
- Inne niż powyższe 2 są praktycznie bezużyteczne, nie modelują żadnej rzeczywistej sytuacji
- Model opóźnienia transportowego używa przypisania nieblokującego do opisu logiki kombinacyjnej! Niestety nie ma innego sposobu.
- Do modelowania opóźnień sygnałów zdecydowanie prostszy w użyciu jest VHDL

Kodowanie stanów

- Naturalne (ponumerowane stanu)
  - efektywne kodowanie stanów
  - minimalna liczba przerzutników
  - sporo logiki dodatkowej tworzącej funkcje Q i Ψ
  - przykład po prawej: (A,B,C,D)=('00','01','10','11')
- One-hot (zapalona jest tylko jedna jedynka)
  - rozrzutne kodowanie stanów
  - maksymalna liczba przerzutników
  - mało logiki dodatkowej, proste Ψ (tylko OR'y)
  - szybka maszyna
  - czyli (A,B,C,D)=('1000','0100','0010','0001')
- $y = q[1 : n]$ (brak Ψ)
  - tak kodujemy stany maszyny, żeby pewne wybrane bity reprezentowały stan wyjścia
  - liczba przerzutników pośrednia
  - brak glitch'y
  - jeżeli pierwszy bit reprezentuje stan wyjścia to: (A,B,C,D)=('000','100','101','111')

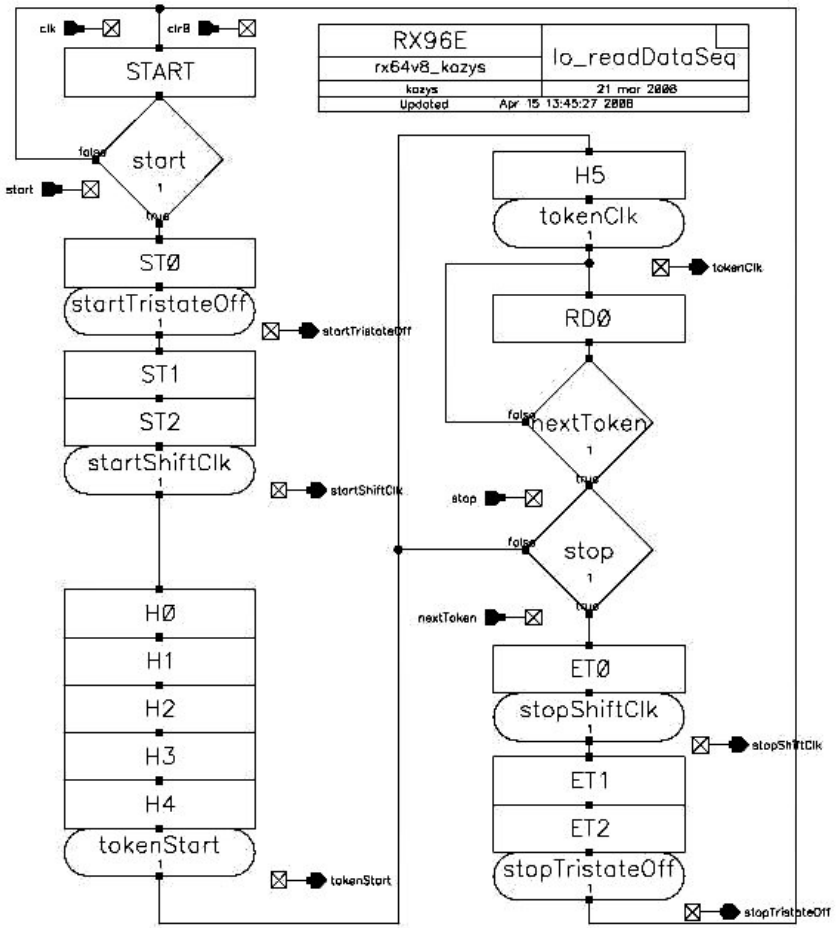


Pomoce przy projektowaniu FSM

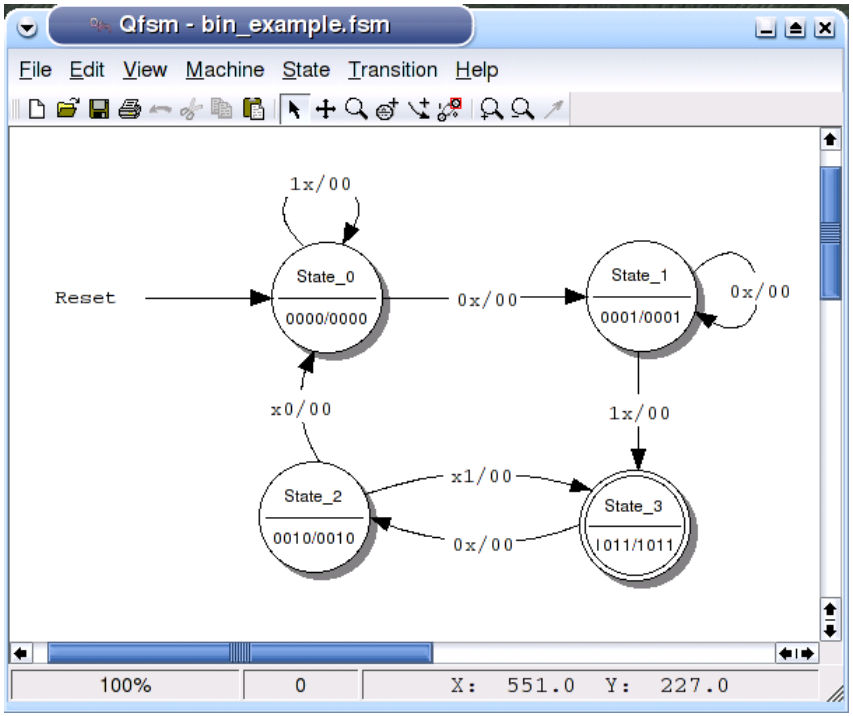
- Skrypt pod Cadence `stateTrans.il`
  1. rysowanie algorytmu maszyny (zamiast diagramu stanów) z użyciem bloczków z biblioteki `chart`
  2. ustawianie parametrów generacji kodu: typ resetu, kodowanie stanów, dopełnianie case'a itp.
  3. generacja kodu w Verilogu lub VHDLu
- Program QFSM (http://qfsm.sourceforge.net/)
  1. tworzenie nowej maszyny (liczba wejść i wyjść typu Moore i Mealy)
  2. rysowanie diagramu stanów w okienku QFSM
  3. testy integralności i symulacje działania
  4. generacja kodu w językach: AHDL, VHDL i Verilog HDL
  5. diagramy w eps i sgv, tabele stanów w Latex, HTML i txt
  6. brak opcji przy generowaniu kodu; OpenSource więc można dorobić
- Program brusey20 (http://www.2ub.org/brusey20/)
  1. rysowanie diagramu stanów w jednym z trzech formatów wektorowych: XFig, jfig lub WinFIG
  2. generacja kodu w VHDLu

Pomoce przy projektowaniu FSM c.d.

Diagram dla `stateTrans.il`



Screenshot z QFSM



Spis treści

Symulatory

- Icarus Verilog + GTKWave
  - Open Source
  - pełne wsparcie dla Verilog-1995
  - możliwość syntezy
- Verilog-XL + SimVision (Cadence)
  - zintegrowany z Cadence, można symulować używając schematu jako wejścia
  - wspiera tylko stary standard Veriloga-1995
  - właściwie już nie rozwijany
- NCVerilog + SimVision (Cadence)
  - zewnętrzne narzędzie
  - zintegrowany z SimVision w debugger
    - podgląd schematu i kodu
    - podglądanie wartości poszczególnych węzłów
    - symulacja krokowa
    - ustawianie pułapek w kodzie itp.
  - obsługuje Verilog-1995, większość Verilog-2001, System Verilog
  - można integrować kod w kilku językach opisu sprzętu: Verilog, VHDL, SystemC, ...

Symulujemy moduł

```
‘timescale 1ns/1ps
//20 MHz; clk period in ns
‘define CLK 50

module test_imp3clk;
reg clrB = 1;
reg start = 0;

reg clk = 0;
always #((‘CLK)/2.0) clk=~clk;

initial begin
   $monitor("At time %t, clk =%b,
       start =%b, out =%b", $stime,
       clk, start, out);
   $dumpfile("vcd/imp3clk_test.vcd");
       //$shm_open(...)
   $dumpvars(0,test_imp3clk);
     //$shm_probe("AS")
   # (‘CLK) clrB = 0;
   # (‘CLK/2) clrB = 1;
end
```

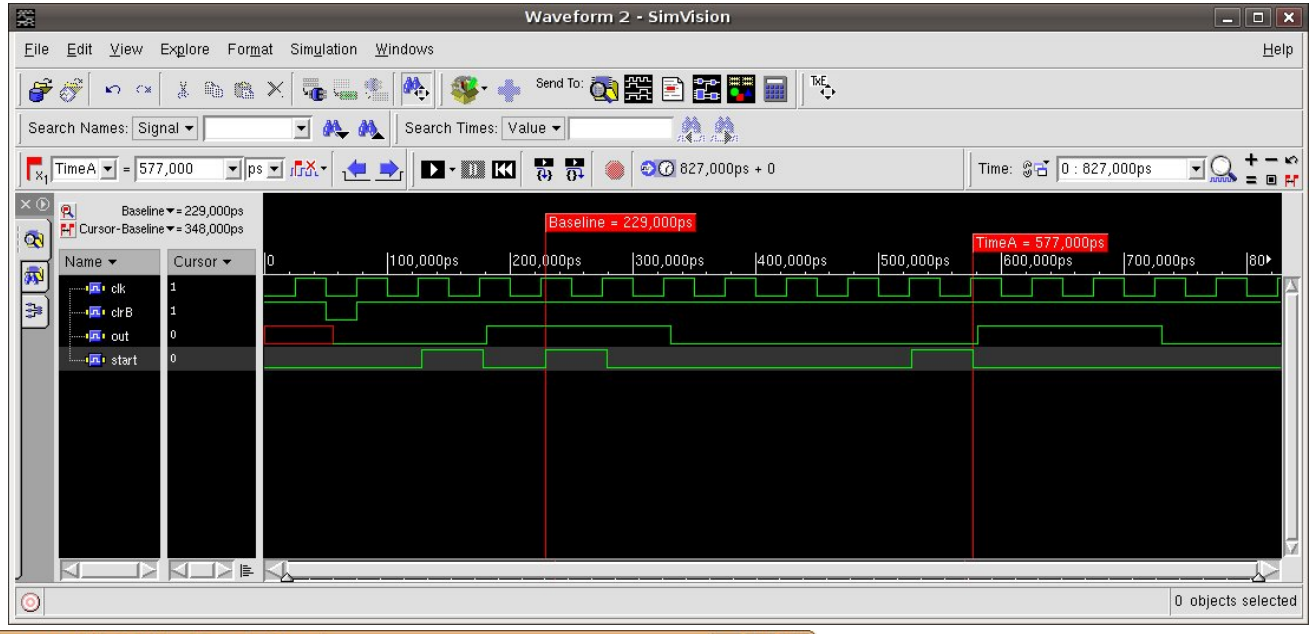
```
initial begin
   # (‘CLK*2+‘CLK/2+3 ) start = 1;
   # (‘CLK) start = 0;
   # (‘CLK+1) start = 1;
   # (‘CLK) start = 0;
   # (‘CLK*5-2) start = 1;
   # (‘CLK) start = 0;
   # (‘CLK*5) $finish;
end

wire out;
imp3clk testMod(out, clk, clrB, start);

endmodule // test_imp3clk
```

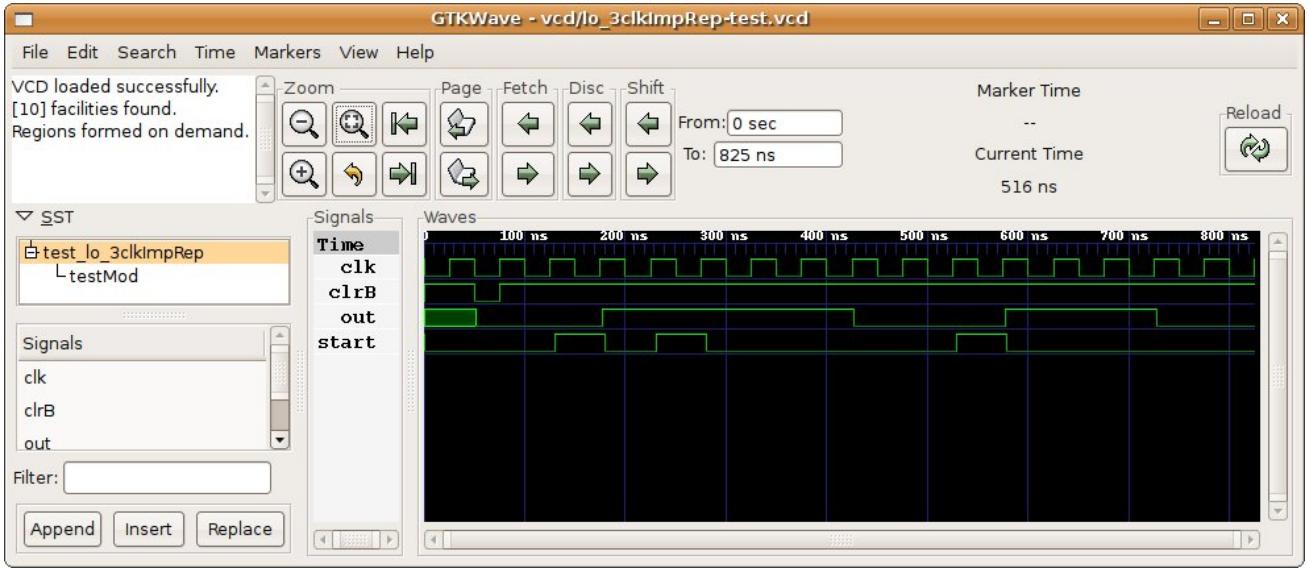
Uwagi

- W trybie z GUI (nclaunch) zapisuje się samo
- Można dołożyć opóźnienia fizyczne — SDF

Wyniki symulacji

SimVision
`imp3clk`





GTKWave
`imp3clkRep`

A co dalej?

Dalej jest ...

A co dalej?

Dalej jest ...

1. Synteza

A co dalej?

Dalej jest ...

1. Synteza
2. Implementacja na krzemie w formie ASIC'a lub w FPGA

A co dalej?

Dalej jest ...

1. Synteza
2. Implementacja na krzemie w formie ASIC'a lub w FPGA

Dziękuję za uwagę!