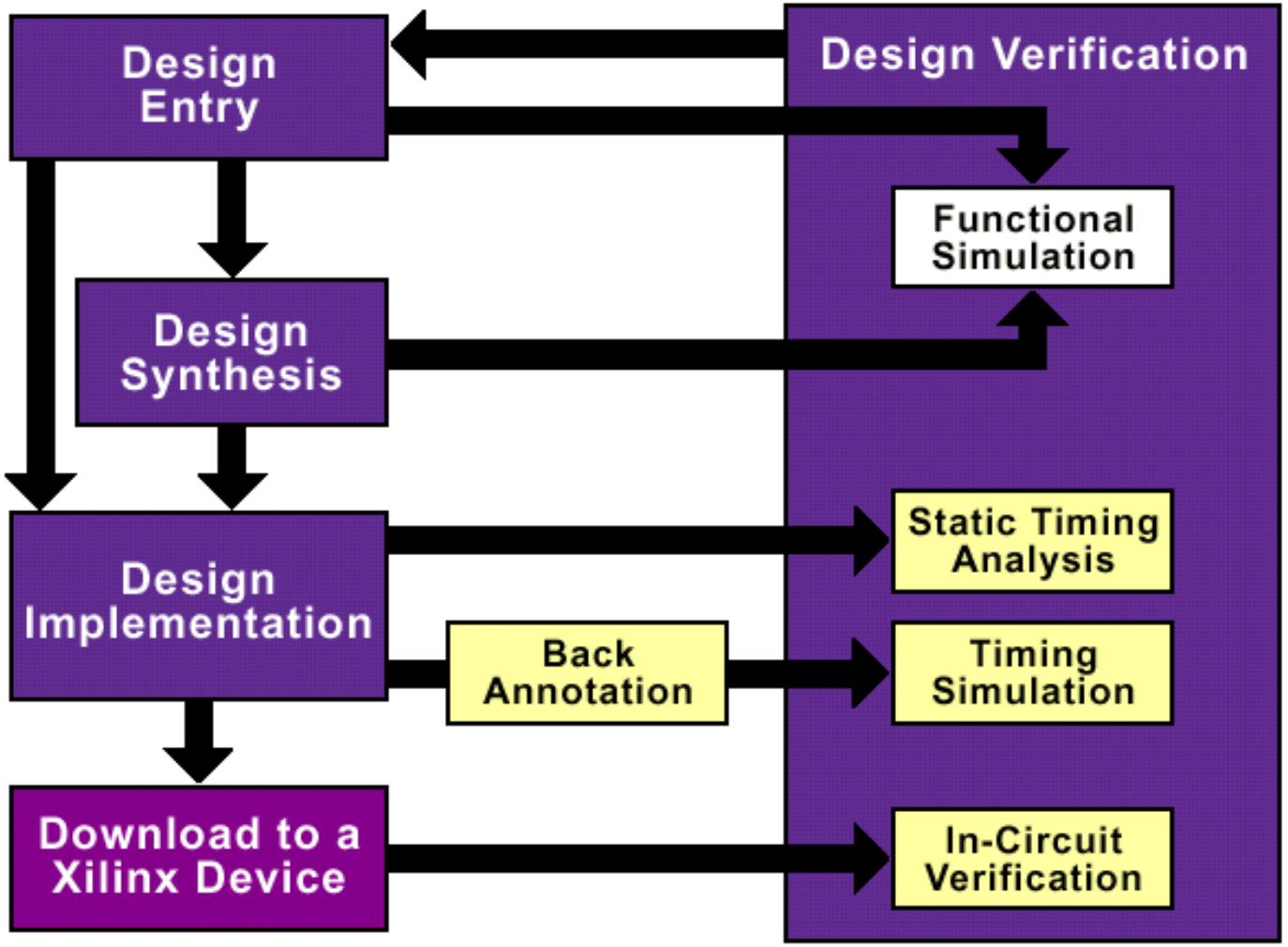
Synteza logiczna

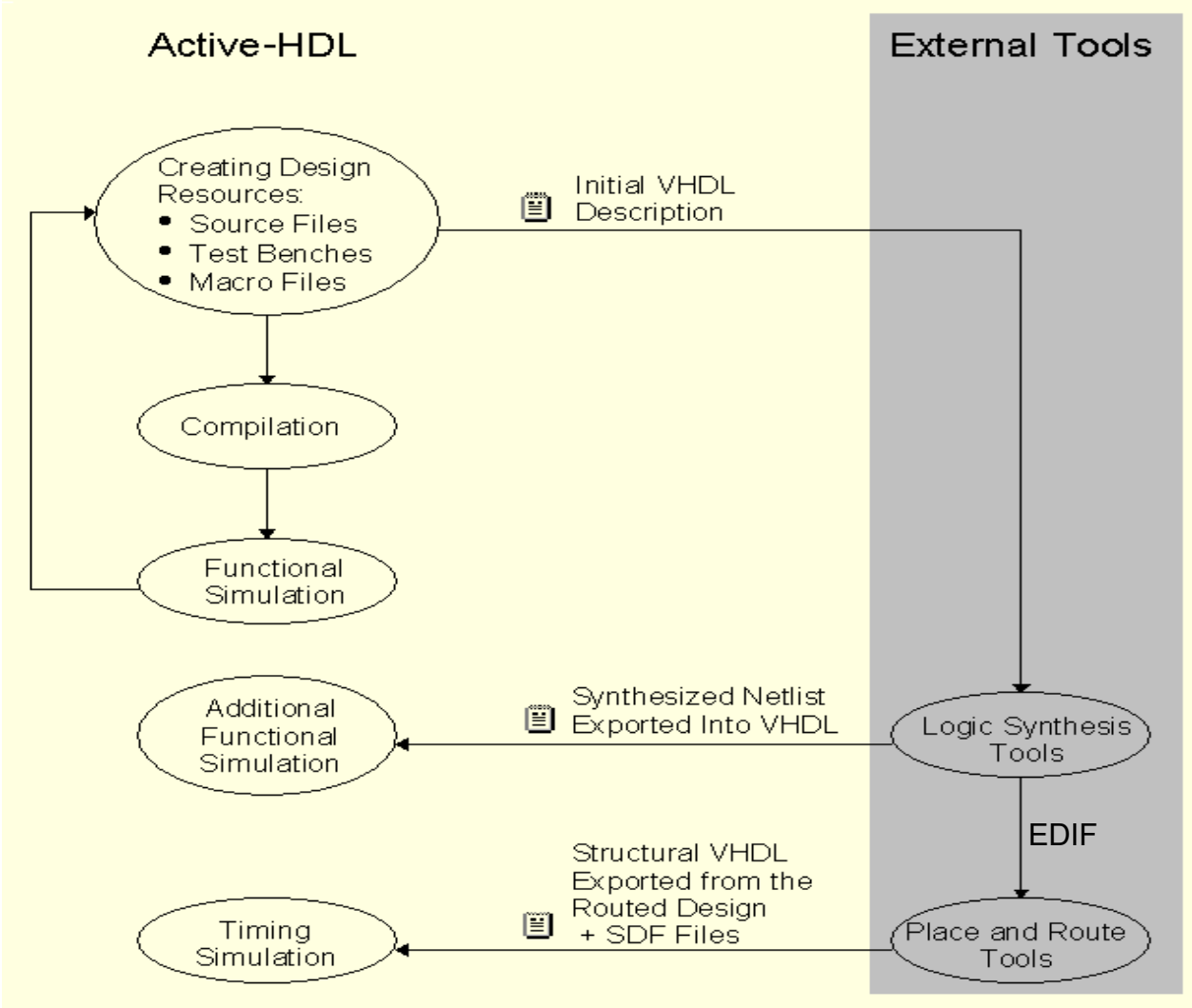
Program wykładu

- Wstęp do syntezy
- Synteza elementów podstawowych
- Sprzętowa reprezentacja obiektów
- Modelowanie układów kombinacyjnych i sekwencyjnych
- Synteza układów złożonych
- Konstrukcje niesyntezowalne

Synteza i implementacja
Projekt CPLD/FPGA Xilinx



Synteza i implementacja
Projekt w środowisku Active-HDL



Synteza i implementacja
Projekt w środowisku Active-HDL: *Design Flow*

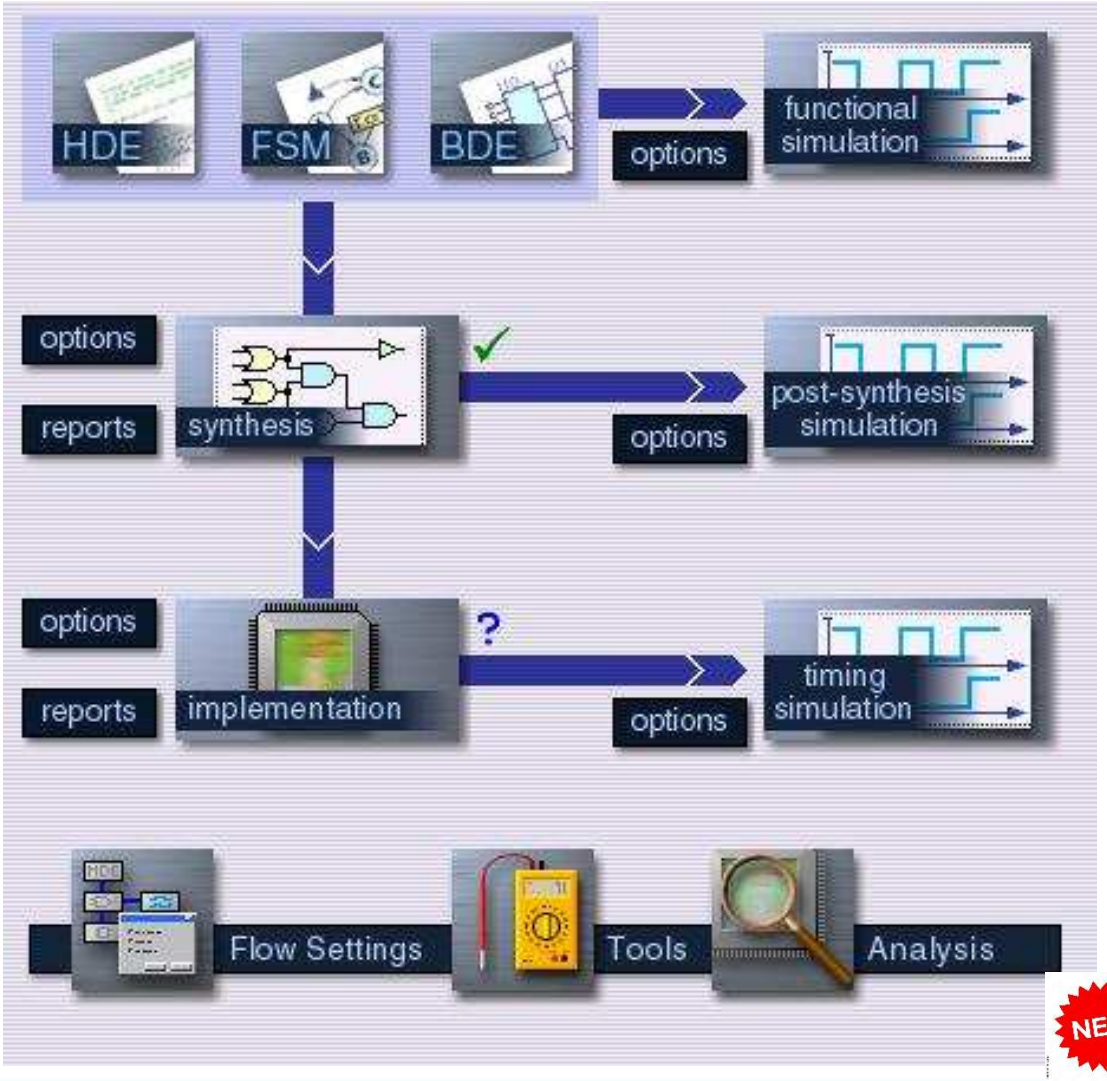
Zewnętrzne narzędzia do syntezy i implementacji

Graficzny *shell* po wybraniu opcji:

Tools/Preferences/ Environment/Flows/ Integrated Tools

oraz

View/Flow



Komponenty syntezowane, wywodzone i podstawiane

```
module synthesized_AND (A, B, X)
  always @(A, B)
    if ((A == 1) & (B == 1))
      X = 1;
    else                          // bramka AND syntezowana
      X = 0;


module inferred_AND (A, B, X)
  assign X = A & B;               // bramka AND wywiedziona
                                  // z biblioteki elementów
                                  // presyntezowanych


module instantiated_AND (A, B, X)
  AND2 GATE_INST                  // bramka AND podstawiona
  (.out(X), .i1(A), .i2(B));// z Verilog primitives
```

Komponenty wywodzone

Wywodzenie (*inferring*) komponentu polega na napisaniu ogólnego kodu HDL dla wykorzystania konkretnego, pożądanego zasobu, np. przerzutnika, pamięci czy mnożarki. Narzędzie do syntezy będzie wtedy próbowało wykorzystać zamierzony zasób na podstawie tego kodu HDL.

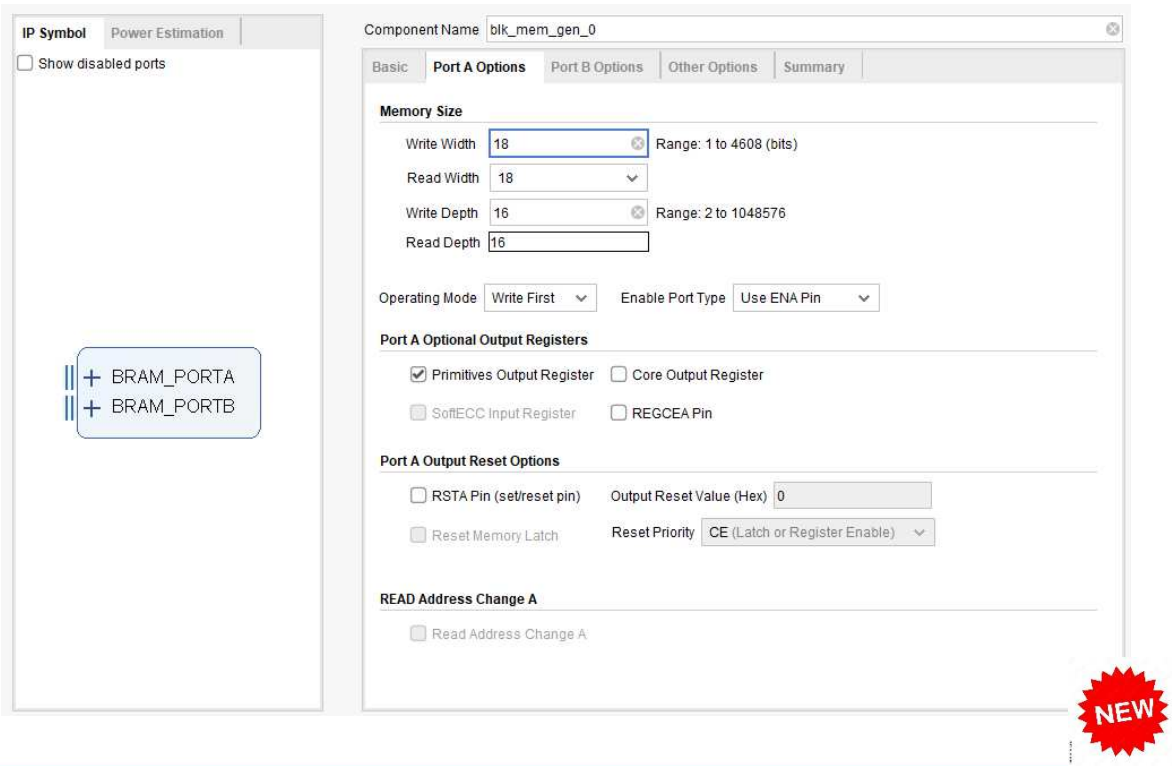
Główną zaletą wywodzenia jest przenośność kodu. Można sobie wyobrazić zbiór wielu komponentów napisanych w ten sposób, aby wspierać zarówno przenośność (aby łatwo korzystać z różnych dostawców czy producentów), jak i możliwość ponownego użycia – z racji bycia komponentami.

Główną wadą zaś jest to, że wywiedzenie nie zawsze jest możliwe. Bardziej skomplikowane zasoby, takie jak blok Ethernet MAC, nie będą możliwe do wywiedzenia przez narzędzie syntezy i muszą być tworzone poprzez podstawienie.

Komponenty podstawiane

Podstawienie komponentu to użycie wstępnie zdefiniowanego szablonu instancji, udostępnionego w dokumentacji dostawcy lub utworzonego za pomocą kreatora graficznego w celu zaimplementowania jakiegoś konkretnego zasobu, np. pamięci.

Główną zaletą stosowania instancji komponentów jest to, że otrzymuje się dokładnie tę pożądaną wersję zasobu. Wadą korzystania z instancji komponentów jest natomiast to, że jest się ograniczonym do pracy w ramach interfejsów zdefiniowanych przez dostawcę, co nie jest podejściem przenośnym.



Sprzętowa reprezentacja obiektów

Obiekt – sieć albo zmienna - może być reprezentowany jako:

- **przerzutnik**
  (element pamięciowy wyzwalany zboczem)
- **zatrzask**
  (element pamięciowy wyzwalany poziomem)
- **funktor**
  (element kombinacyjny)

Sprzętowa reprezentacja obiektów

```
always
@(posedge C)

// if (1)
      Q <= D;
// else
//    Q <= Q;
```

sygnał Q:
przerzutnik

```
always
@(G,D)

   if (G == 1)
      Q <= D;
// else
//    Q <= Q;
```

sygnał Q:
zatrzask

```
always
@(A,B,S)

   if (S == 0)
      Q = A;
   else
      Q = B;
```

sygnał Q:
multiplekser

Konstrukcje implikujące syntezę przerzutnika D

Przerzutnik – w sensitivity list tylko sygnały aktywne zboczem

```
always @ (posedge CLK)                          \\ sync CLR=1
  if (CLR) Q <= 0;
      else Q <= D;


always @ (posedge CLK, posedge CLR)    \\ async CLR=1
  if (CLR) Q <= 0;
      else Q <= D;


always @ (posedge CLK, negedge SET)    \\ async SET=0
  if (!SET) Q <= 1;
       else Q <= D;
```

Konstrukcje implikujące syntezę zatrzasku D

Zatrzask – w sensitivity list tylko sygnały aktywne poziomem

```
always @ (GATE, CLR, D)          \\ List complete
        if (CLR)   Q <= 0;
  else if (GATE) Q <= D;
```

Z tego zapisu synteza prawidłowo wyprowadzi zatrzask, a wyniki symulacji przed i po syntezie będą zgodne.

```
always @ (GATE, CLR)             \\ List not complete
        if (CLR)   Q <= 0;
  else if (GATE) Q <= D;
```

Z tego zapisu synteza prawidłowo wyprowadzi zatrzask, ale wyniki symulacji przed i po syntezie nie będą zgodne.

```
always @ (*)…                    \\ Default list
```

Z tego zapisu synteza prawidłowo wyprowadzi zatrzask, a wyniki symulacji przed i po syntezie będą zgodne.

Sterowanie zdarzeniami

```
always @(posedge CLK, RST) // <= mixed event controls
  if (CLR)
    Q <= 0;
  else
    Q <= D;
```

[Synth 8-434] Mixed level sensitive and edge triggered event controls are not supported for synthesis.

Załóżmy, że chcemy stworzyć przerzutnik taktowany **zboczem narastającym** zegara (**CLK**), z **asynchronicznym** resetem (**RST**), aktywnym **stanem wysokim**. Zacznijmy więc budować pożądane zachowania. Pierwszym, jakie potrzebujemy, jest zmiana wyjścia (**Q**) adekwatnie do stanu wejścia (**D**), możliwa za każdym razem, kiedy wystąpi narastające zbocze zegara:

```
always @(posedge CLK)
  Q <= D;
```

Sterowanie zdarzeniami

Teraz dodajmy reset:

```
always @(posedge CLK)
  if (RST) Q <= 0;
  else     Q <= D;
```

Kiedy blok *always* jest wykonywany, co zachodzi, gdy spełniony jest warunek **posedge CLK**, to jeśli stan resetu jest wysoki, wyjście zostanie skasowane. W przeciwnym razie zostanie ono ustawione adekwatnie do stanu wejścia. Mamy więc zamodelowane zachowanie się resetu. Jednak dzieje się to tylko na narastającym zboczu zegara, więc jest to reset **synchroniczny**.

Aby uczynić go **asynchronicznym**, musimy wyzwolić wykonanie bloku *always* także wtedy, kiedy wystąpi zmiana stanu resetu. Mamy na to dwa sposoby: uwrażliwienie bloku *always* na stan lub na zbocze. Zobaczmy, jakie zachowanie się przerzutnika zostanie spowodowane przez uwrażliwienie na stan:

Sterowanie zdarzeniami

```
always @(posedge CLK or RST)
  if (RST) Q <= 0;
  else     Q <= D;
```

Gdy reset przyjmie wartość wysoką (narośnie), blok jest wykonywany bez względu na to, czy występuje zbocze zegara (a więc mamy zachowanie asynchroniczne !) Wyjście jest wtedy kasowane, ponieważ stan resetu jest wysoki.

Ale co się stanie, jeśli reset przyjmie wartość niską (opadnie)? Blok *always* zostanie wykonany ponownie ! Wyjście zostanie wtedy ustawione adekwatnie do wartości wejścia, ponieważ stan resetu jest niski i warunek nie jest spełniony ! To nie jest zachowanie, którego pożądamy - przerzutnik może zmienić swoje wyjście w odpowiedzi na wycofanie resetu !

Zobaczmy więc, jakie zachowanie się przerzutnika zostanie spowodowane przez uwrażliwienie na zbocze:

Sterowanie zdarzeniami

```
always @(posedge CLK or posedge RST)
  if (RST) Q <= 0;
  else     Q <= D;
```

Gdy reset przyjmie wartość wysoką (narośnie), blok jest wykonywany bez względu na to, czy występuje zbocze zegara (a więc tu też mamy zachowanie asynchroniczne !) Wyjście jest wtedy kasowane, ponieważ stan resetu jest wysoki.

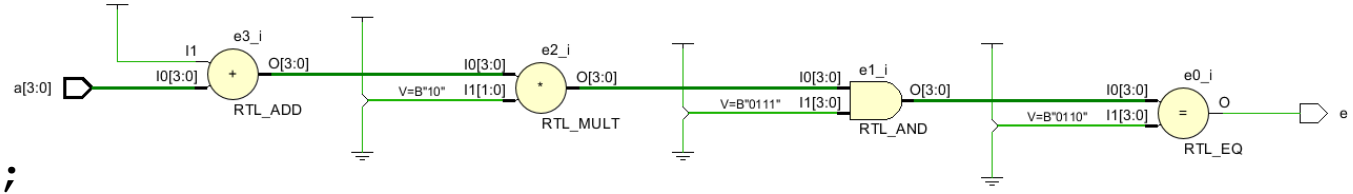
Ale co się stanie, jeśli reset przyjmie wartość niską (opadnie)? Absolutnie nic. Wyjście zachowa swoją wartość uzyskaną w wyniku resetu. Wycofanie resetu nie spowoduje już zmiany stanu wyjścia przerzutnika.

Tak więc aby uzyskać asynchroniczny reset, musimy uwrażliwić blok *always* zarówno na zbocze zegara, jak i na zbocze resetu.

Modelowanie układów kombinacyjnych i sekwencyjnych

```
always @ (*)
 begin // blokujące
  b = a + 1;
  c = b * 2;
  d = c & 7;
  e = (d == 6);
 end
```

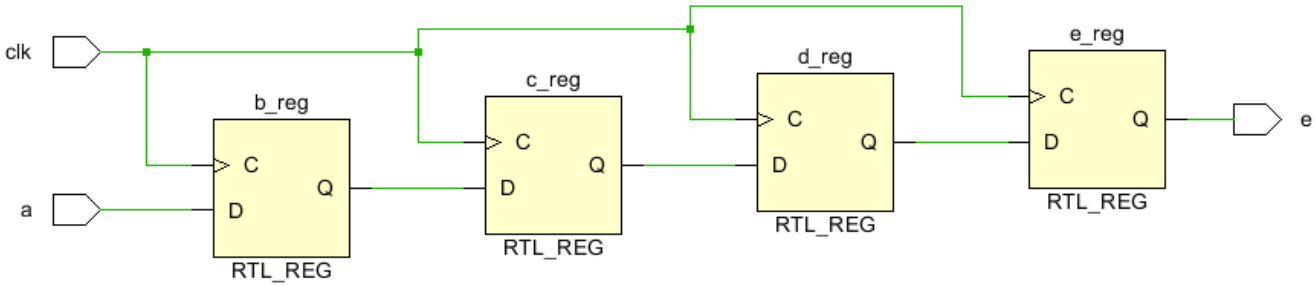
Modelowanie układów kombinacyjnych – tylko przypisania blokujące



Kaskada funkcji kombinacyjnych

```
always @ (posedge clk)
  begin // nieblokujące
    b <= a;
    c <= b;
    d <= c;
    e <= d;
  end
```

Modelowanie układów sekwencyjnych – tylko przypisania nieblokujące

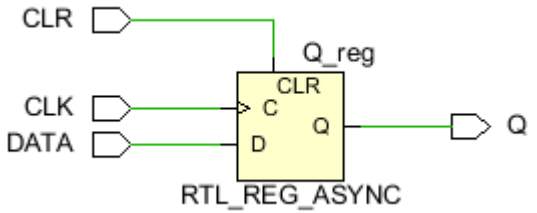


Przerzutniki pracujące synchronicznie – taktowane tym samym zegarem

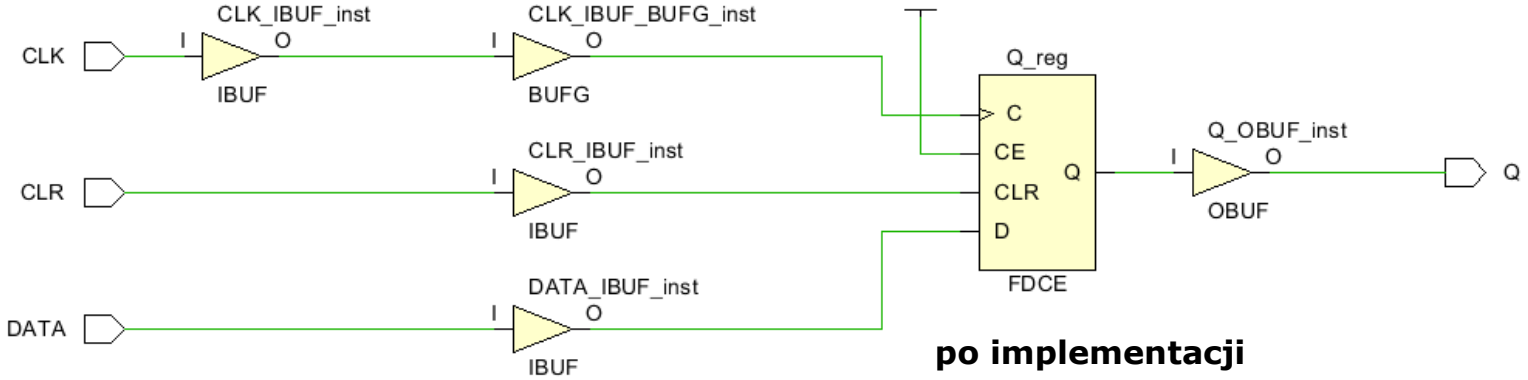
NEW

Synteza przypisań blokujących i nieblokujących

```
always @ (posedge CLK, posedge CLR)
  if (CLR)
    Q <= 0;   \\ nonblocking
  else
    Q <= D;   \\ nonblocking
```



po syntezie

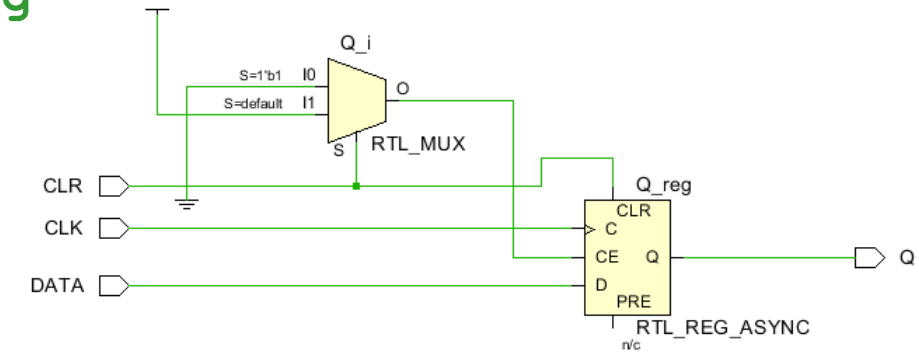


po implementacji

NEW

Synteza przypisań blokujących i nieblokujących

```
always @ (posedge CLK, posedge CLR)
  if (CLR)
    Q <= 0;   \\ nonblocking
  else
    Q = D;    \\ blocking
```

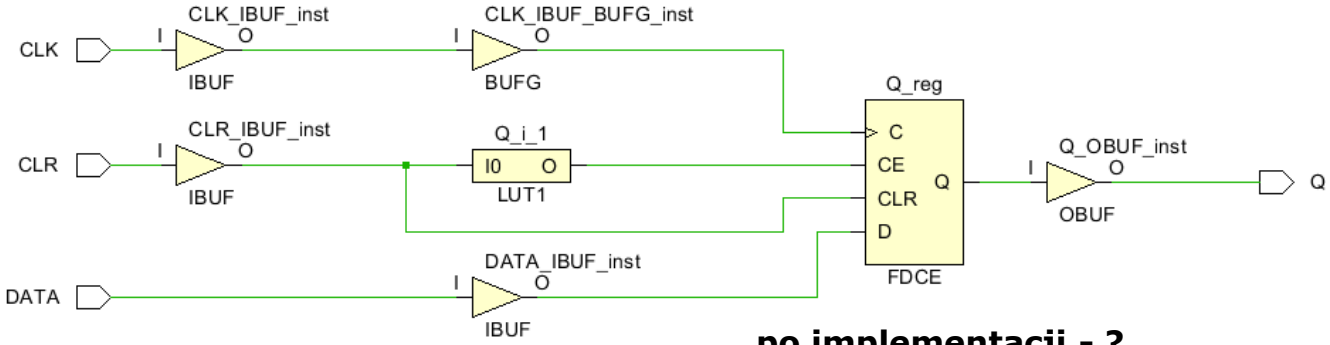


po syntezie - ?

Cell Properties

Q_i_1

I0	O=!I0
0	1
1	0



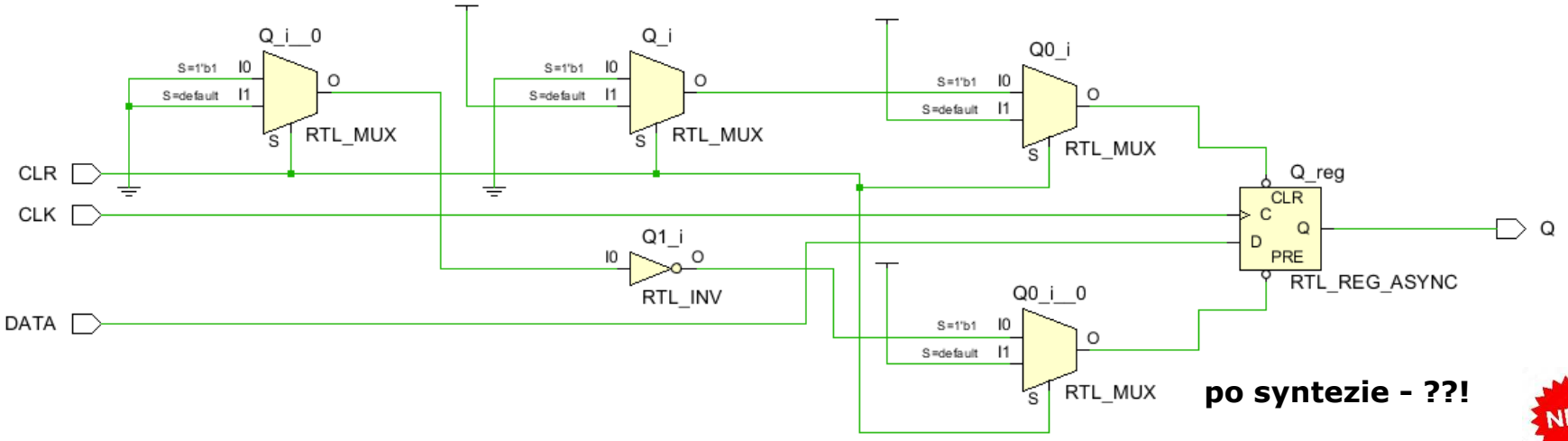
po implementacji - ?

NEW

Synteza przypisań blokujących i nieblokujących

```
always @ (posedge CLK, posedge CLR)
  if (CLR)
    Q = 0;   \\ blocking
  else
    Q <= D; \\ nonblocking
```

```
WARNING: [Synth 8-5788] Register Q_reg in module (…)
has both set and reset with same priority. This may
cause simulation mismatches. Consider rewriting code.
```

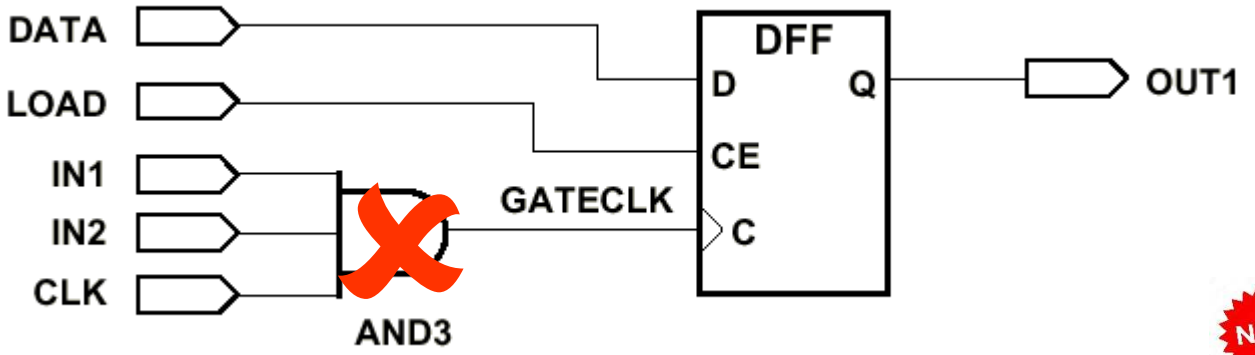


po syntezie - ??!

NEW

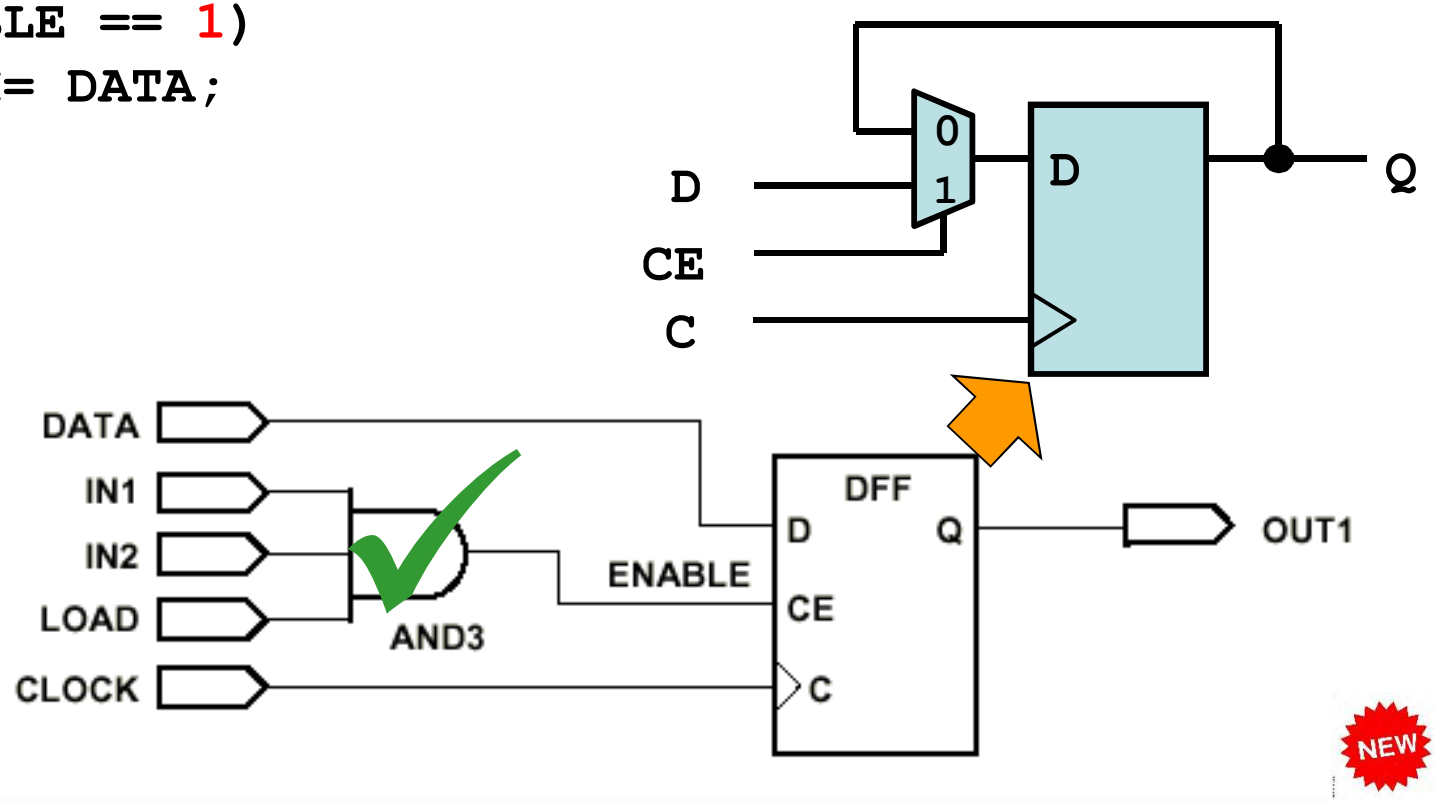
Przerzutnik z wejściem blokującym zegar

```
assign GATECLK = IN1 & IN2 & CLK;

always (posedge GATECLK )
  if (LOAD == 1)
    OUT1 <= DATA;
```



Przerzutnik z wejściem blokującym zegar

```
assign ENABLE = IN1 & IN2 & LOAD;

always (posedge CLK)
  if (ENABLE == 1)
    OUT1 <= DATA;
```



Niezamierzona synteza zatrzasku

WARNING: [Synth 8-327] inferring latch for variable 'L_DAT_reg' - ???

Multiplexer 3:1

```
always (A, B, C, SEL)
        if (SEL == 2'b00)
          Y = A;
  else if (SEL == 2'b01)
          Y = B;
  else if (SEL == 2'b10)
          Y = C;
```

Problem:

implikacja zatrzasku na skutek niepełnej specyfikacji wartości (dla kombinacji **SEL = 2'b11** domyślnie przyjęta zostanie ostatnia wartość wyjścia, co spowoduje konieczność zastosowania elementu pamięciowego).

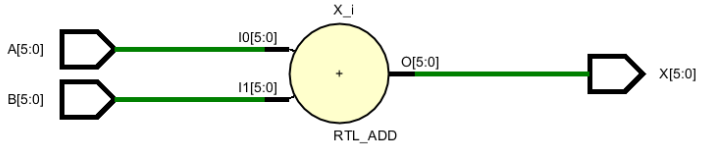
```
always (A, B, C, SEL)
        if (SEL == 2'b00)
          Y = A;
  else if (SEL == 2'b01)
          Y = B;
  else if (SEL == 2'b10)
          Y = C;
     else (SEL == 2'b11)
          Y <= 1'b0;
```

Rozwiązanie:

wyspecyfikowanie wszystkich możliwości (także w konstrukcji **case** – można tu użyć specyfikacji domyślnej przy pomocy klauzuli **default**).

Operatory arytmetyczne – suma

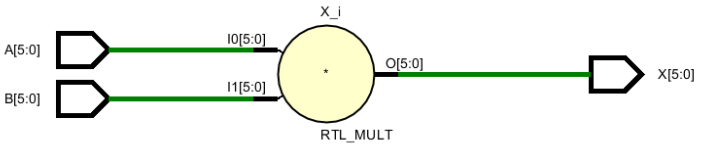
```
assign X = A + B; //[5:0]
```



NEW

Operatory arytmetyczne – iloczyn

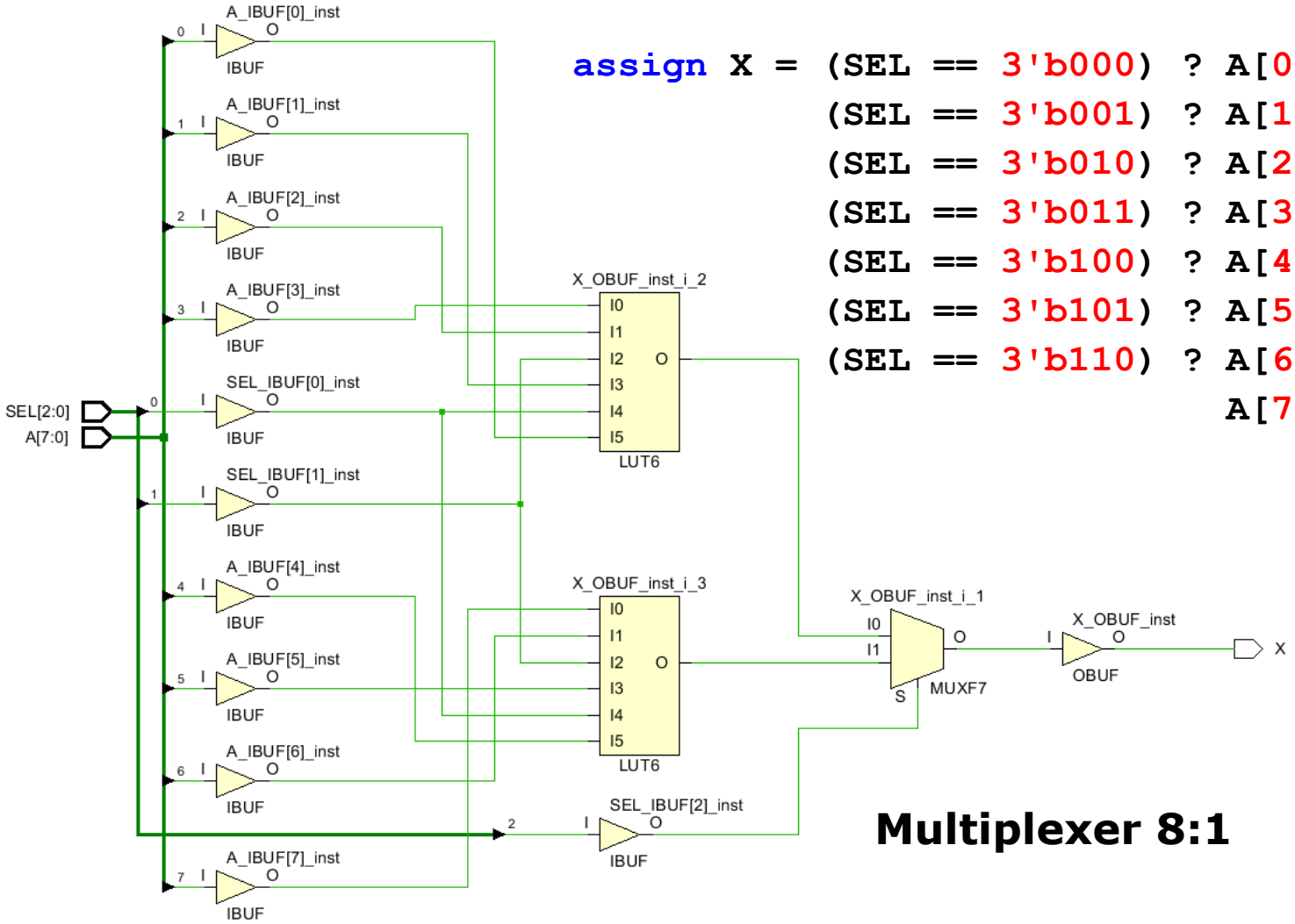
```
assign X = A * B; //[15:0]
```



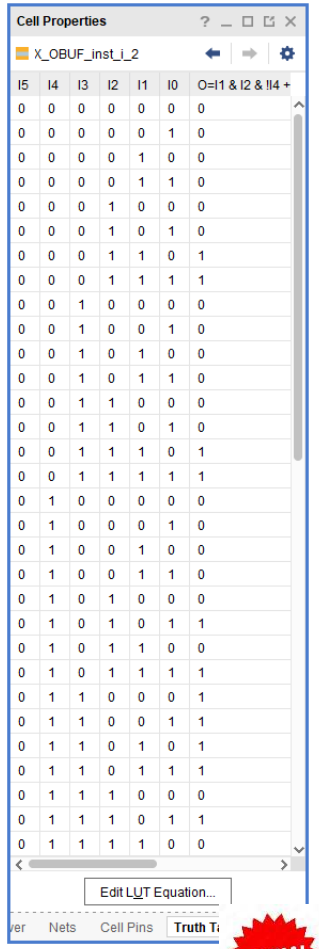
NEW

Instrukcja warunkowa `assign`

```
assign X = (SEL == 3'b000) ? A[0] :
           (SEL == 3'b001) ? A[1] :
           (SEL == 3'b010) ? A[2] :
           (SEL == 3'b011) ? A[3] :
           (SEL == 3'b100) ? A[4] :
           (SEL == 3'b101) ? A[5] :
           (SEL == 3'b110) ? A[6] :
                             A[7] ;
```

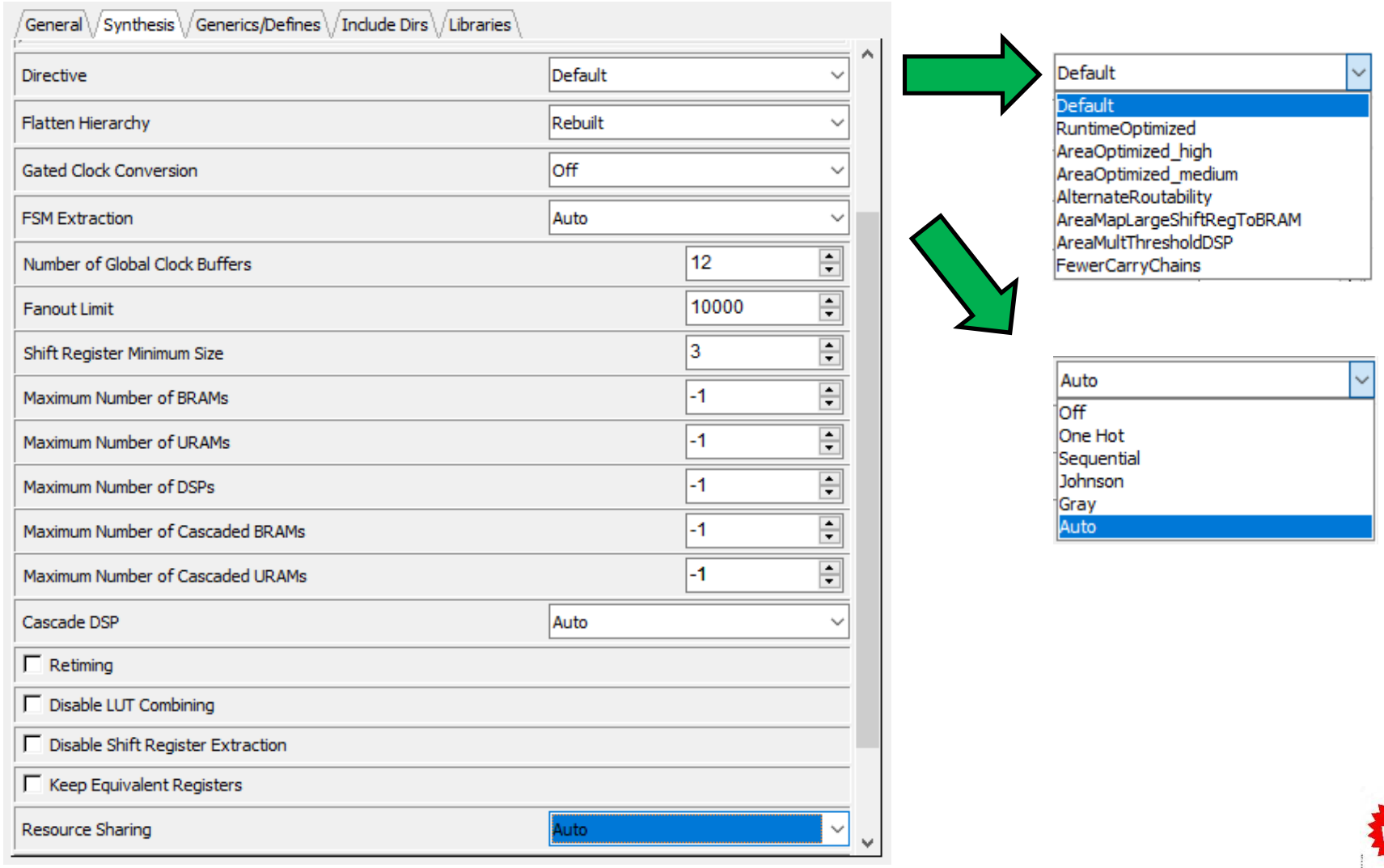


Multiplexer 8:1



NEW

Synteza
Vivado Synthesis Options



Algorytmy kodowania stanów

- ***Auto***
  Najlepszy, dostosowany do automatu algorytm kodowania.

- ***One-Hot***
  Zawsze włączony tylko jeden przerzutnik. Odpowiedni dla FPGA (duża liczba przerzutników) oraz optymalizacji prędkości i poboru mocy.

- ***Sequential***
  Identyfikuje długie ścieżki i nadaje w nich kolejne wartości binarne.

- ***Gray***
  Zawsze tylko jedna zmienna zmienia swoją wartość. Właściwy dla automatów o długich ścieżkach bez odgałęzień. Minimalizuje hazardy i szpilki.

- ***Johnson***
  Podobnie jak Gray.

Ciąg dalszy nastąpi…