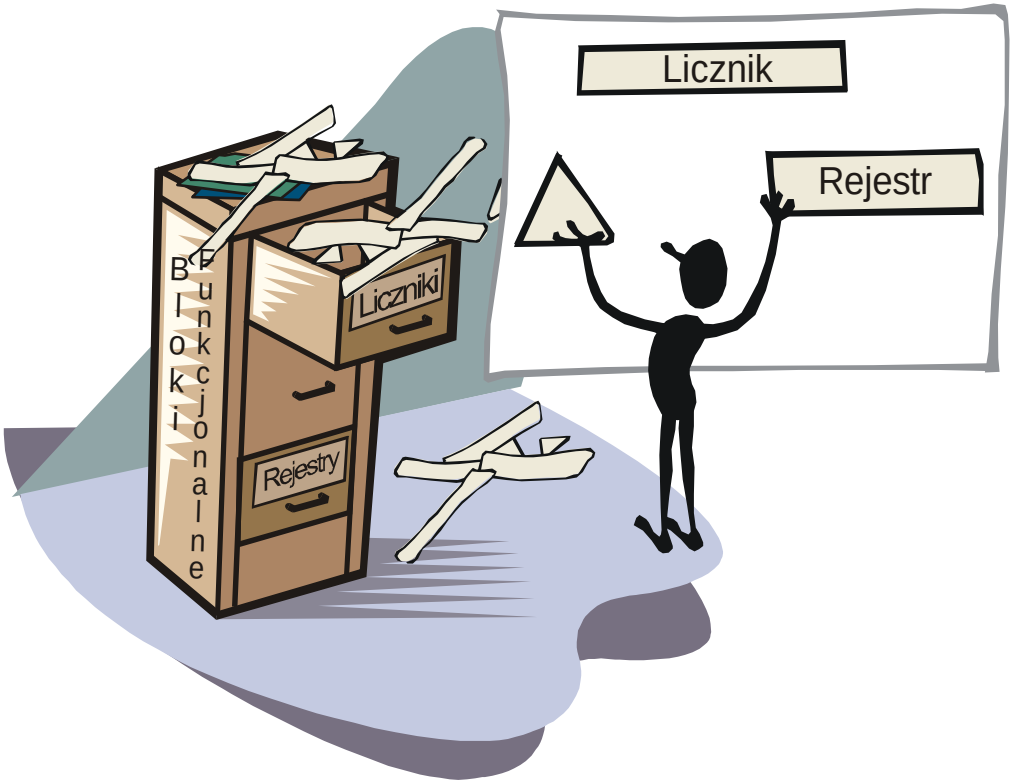
Język AHDL

Synteza strukturalna

dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Synteza strukturalna



1. Składamy układ z bloków funkcjonalnych

2. Odwzorowanie technologiczne

dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Komponent

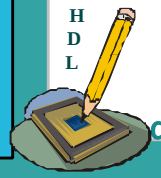
Wyboru odpowiedniej jednostki projektowej dokonuje się przez:

- deklarację komponentu (component declaration),
- a następnie jego podstawienie (component instantiation).

Deklaracja komponentu jest podobna do deklaracji jednostki projektowej – podaje interfejs komponentu.

Umieszcza się ją w przed sekcją SUBDESIGN.

```
FUNCTION __function_name(__input_name, MACHINE
   __state_machine_name)
   WITH (__parameter_name, __parameter_name)
   RETURNS (__output_name, __bidir_name, MACHINE
   __state_machine_name);
```

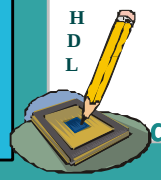


Podstawienie komponentu

Instrukcja podstawienia (łączenia) komponentu to współbieżna instrukcja określająca wzajemne połączenia sygnałów i podukładów zwanych komponentami.

Zawiera:

- nazwę komponentu (etykietę)
- typ komponentu
- mapę wyprowadzeń (*port map*), która pokazuje powiązania pomiędzy sygnałami aktualnymi, a wyprowadzeniami komponentu
- mapę parametrów ogólnych



Deklaracja komponentu

Komponent może być umieszczony w projekcie jako:

- Instrukcja podstawienia (konkretyzacja) w linii (*in-line reference*) – nie wymaga deklaracji w sekcji VARIABLE
  - Postawienie argumentów przez położenie

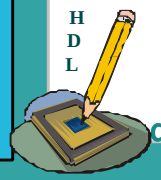
```
(x, y) = function_name(arg1, arg2)
WITH (parameter_name = value1, parameter_name = value2);
```

  - Podstawienie argumentów przez nazwy portów

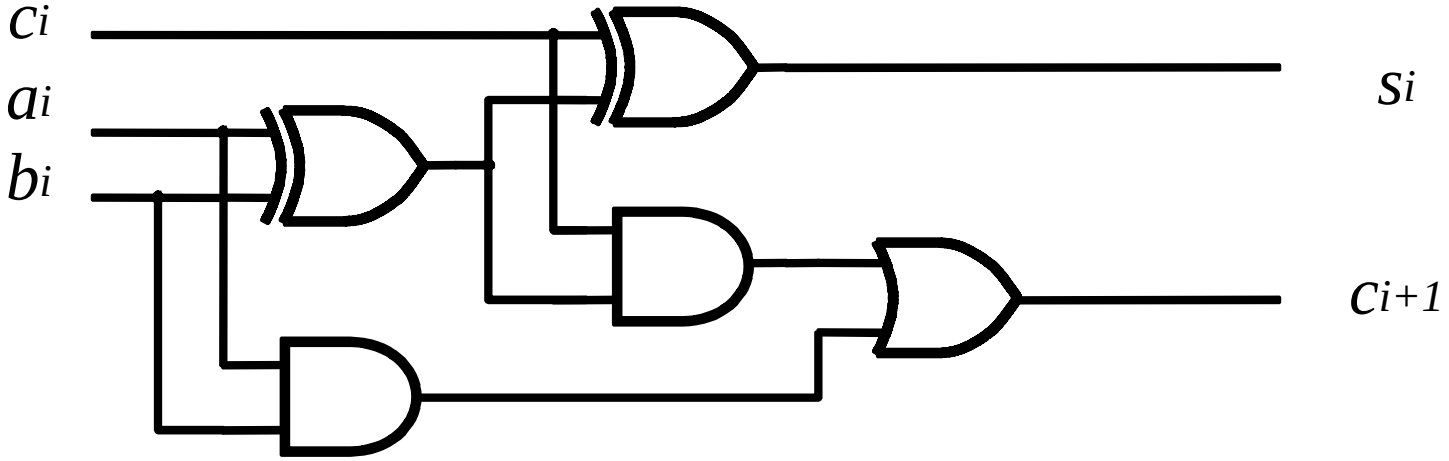
```
(x, y) = function_name (.port_name1 = arg1, .port_name1 =arg2)
   RETURNS (.port_name3, .port_name4);
```

- Konkretyzację funkcji (*instance declaration*) – wymaga deklaracji w sekcji VARIABLE

```
function_instance_name    : function_name
   WITH (parameter_name1 = parameter_value1, parameter_name2 =
   parameter_value2);
```



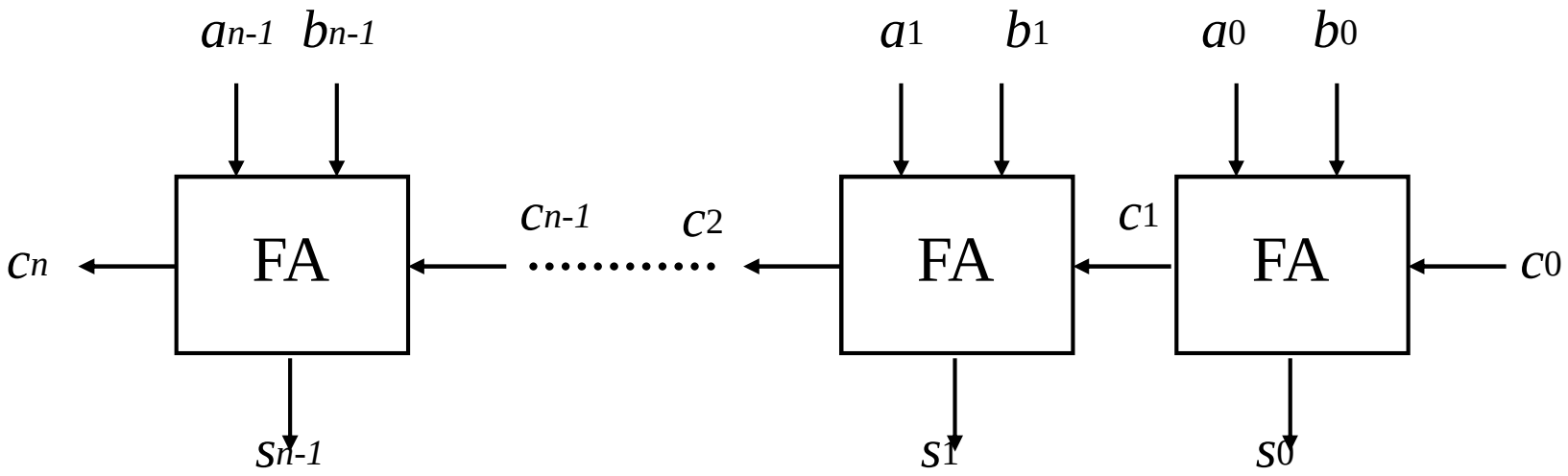
Sumator *full adder*



$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i b_i \vee c_i (a_i \oplus b_i) = a_i b_i \vee c_i (a_i \vee b_i)$$

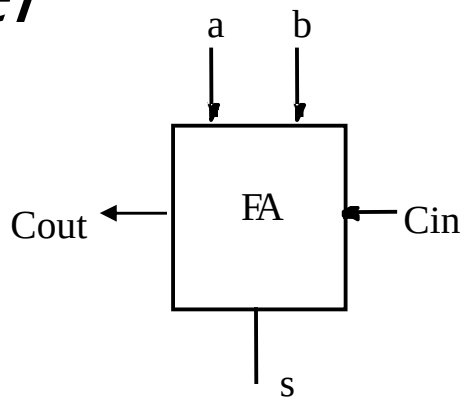
Sumator kaskadowy
Ripple carry adder

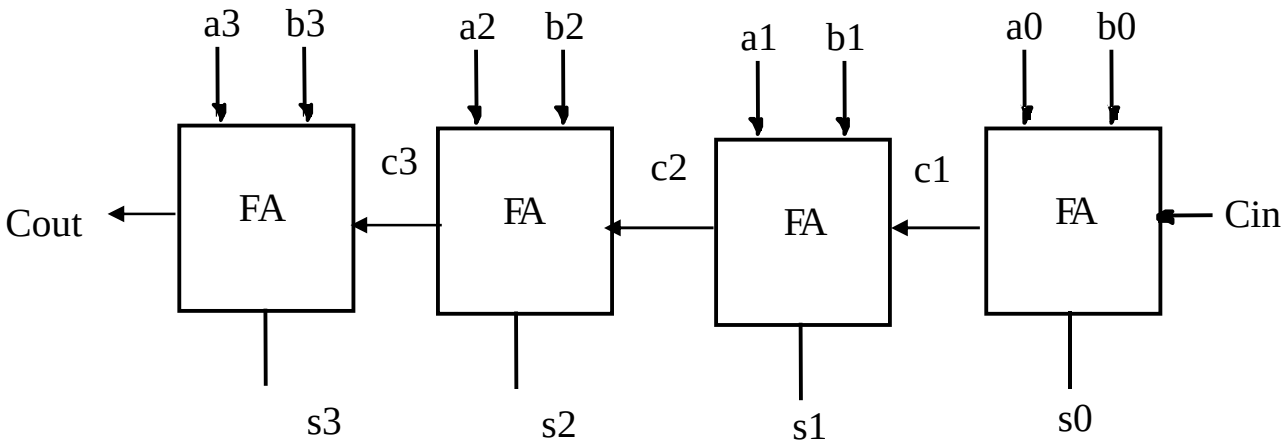


dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Przykład: zbudować *adder4*

- Sumator 4-bitowy złożony z bloków sumatorów 1-bitowych *fulladder*

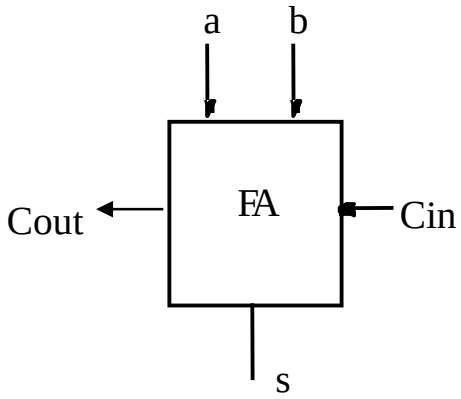


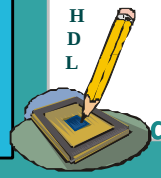


dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Sumator *fulladd* w AHDL

```
-- 1 bit full adder
SUBDESIGN fa
(
   a, b, cin   : INPUT;
   s, cout     : OUTPUT;
)

BEGIN
   s = a XOR b XOR cin;
   cout = a AND b OR a AND cin OR b AND cin;
END;
```

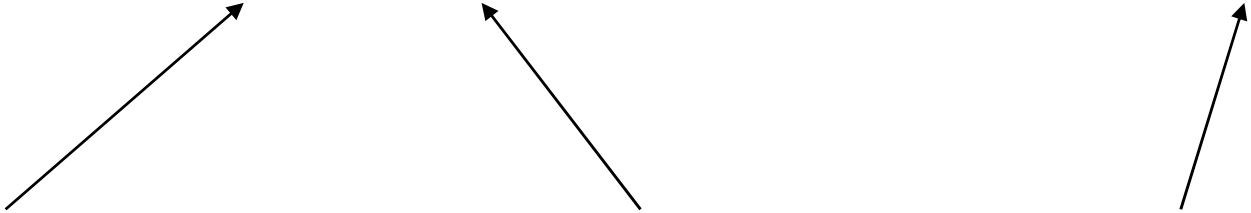




Plik nagłówkowy

- Plik nagłówkowy (prototyp funkcji) *fa.inc*

FUNCTION fa (a, b, cin) RETURNS (s, cout);

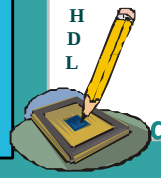


Nazwa funkcji jak nazwa główna pliku

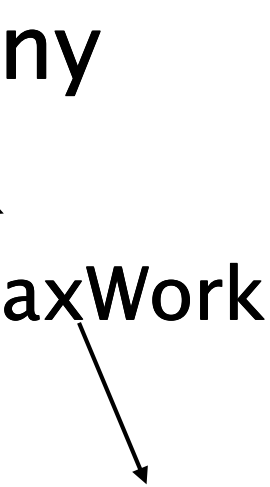
Lista argumentów wejściowych

Lista argumentów wyjściowych

- Można plik nagłówkowy wygenerować automatycznie
  - MP2: File>Create Default Include File
  - Q2: File>Create/Update>Create AHDL Include File for Current File

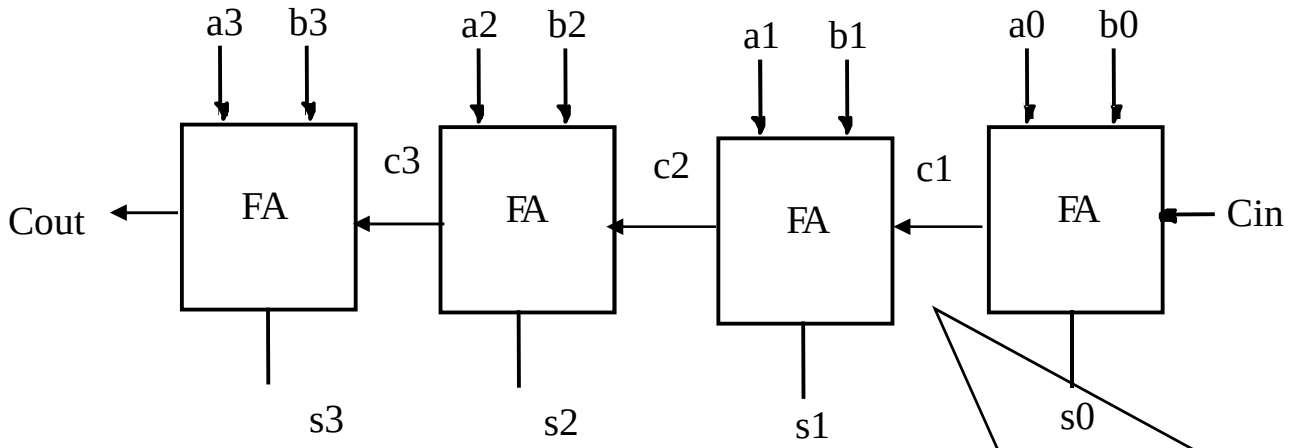


Organizacja plików specyfikacji *adder4*



dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Deklaracja jednostki *adder4*



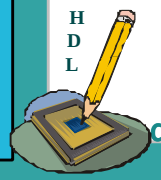

```
INCLUDE "fa.inc";
SUBDESIGN adder4
(
      a3, a2, a1, a0, cin : INPUT;
      b3, b2, b1, b0 : INPUT;
      s3, s2, s1, s0, cout : OUTPUT;
)
```

W deklaracji nie ma sygnałów wewnętrznych

Projekt *adder4*

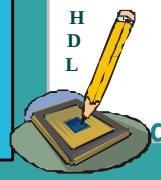
```
INCLUDE "fa.inc";  -- rozszerzenie opcjonalne
-- można zapisać INCLUDE "fa";

SUBDESIGN adder4
(
      a3, a2, a1, a0, cin : INPUT;
      b3, b2, b1, b0 : INPUT;
      s3, s2, s1, s0, cout : OUTPUT;
)

VARIABLE
      c3, c2, c1 : NODE;  -- deklaracja węzłów logicznych

BEGIN
      (s3, cout) = fa(a3, b3, c3);
      (s2, c3) = fa(a2, b2, c2);
      (s1, c2) = fa(a1, b1, c1);
      (s0, c1) = fa(a0, b0, cin);
END;
```

-- podstawienie funkcji
-- konkretyzacja (instancja)
-- argumenty przez położenie

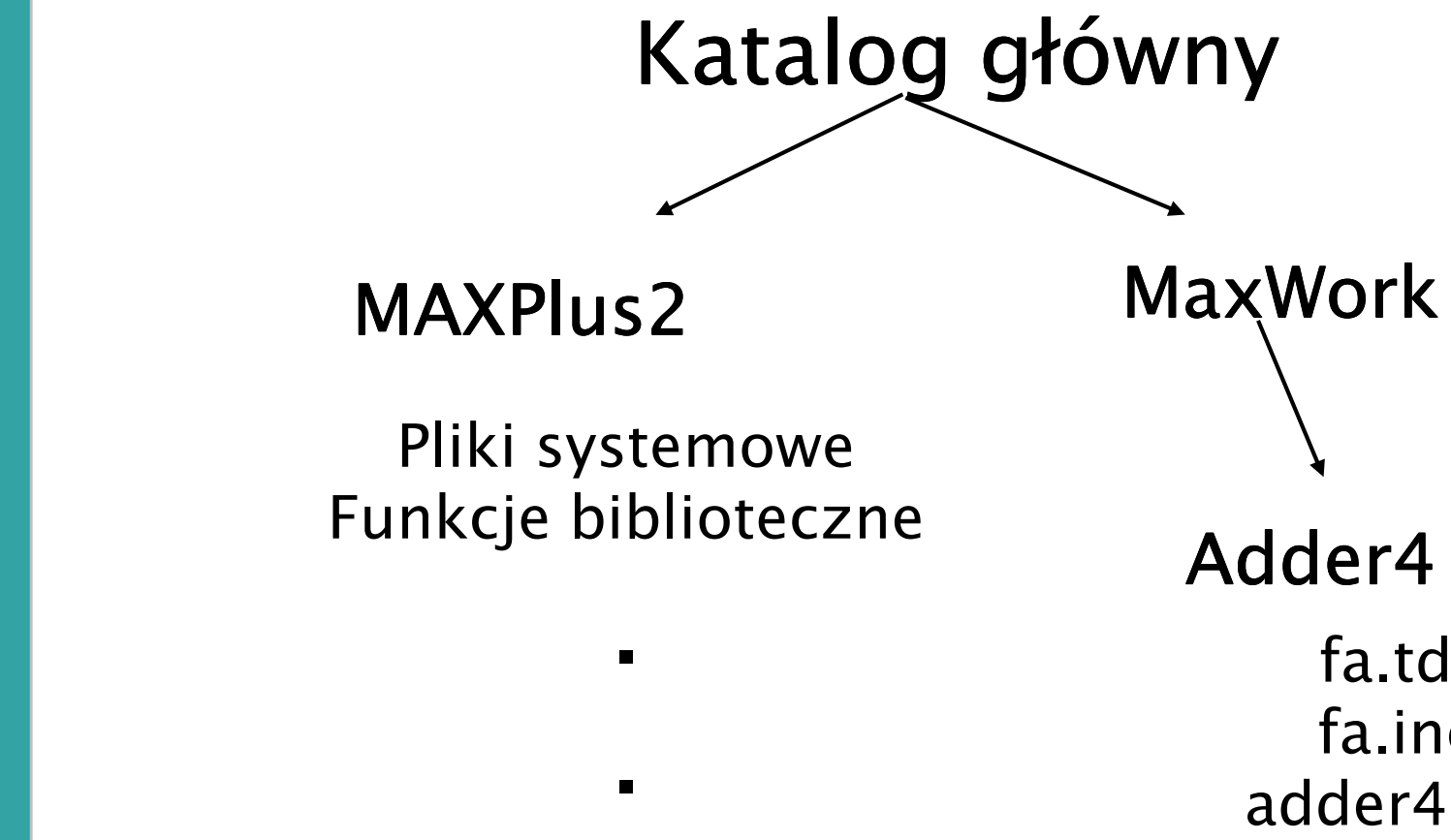


Projekt *adder4*

```
-- zamiast pliku nagłówkowego można podać deklaracje użytych funkcji
FUNCTION fa (a, b, cin) RETURNS (s, cout);

SUBDESIGN adder4
(
      a3, a2, a1, a0, cin : INPUT;
      b3, b2, b1, b0 : INPUT;
      s3, s2, s1, s0, cout : OUTPUT;
)

VARIABLE
      c3, c2, c1 : NODE;

BEGIN
      (s3, cout) = fa(a3, b3, c3);
      (s2, c3) = fa(a2, b2, c2);
      (s1, c2) = fa(a1, b1, c1);
      (s0, c1) = fa(a0, b0, cin);
END;
```

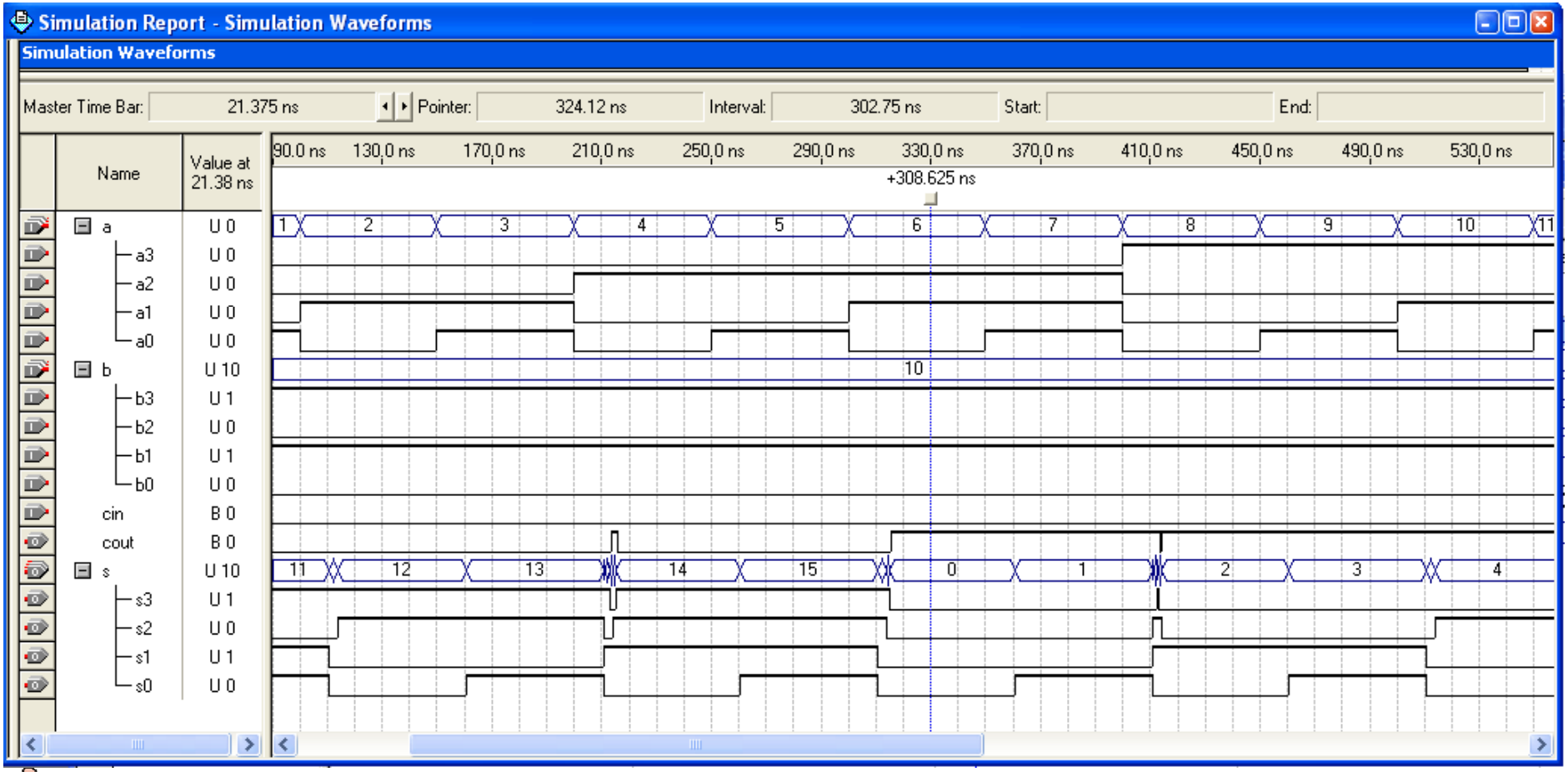


Organizacja plików specyfikacji *adder4*



dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Symulacja układu *adder4*



dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

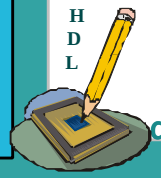
Podstawienie komponentu

- Podstawienie w linii
  - Argumenty przez położenie – ważna jest kolejność listy; jeżeli argument nieużywany to zostawiamy puste miejsce

```
(s3, cout) = fa(a3, b3, c3);
(s2, ) = fa(a2, , c2);
(c2, s1) = fa(a1, b1, c1); błędna  kolejność wyjść
(s0, c1) = fa(b0, a0, cin); błędna kolejność
  argumentów
```

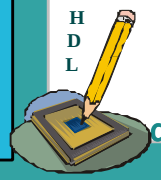
  - Argumenty przez nazwy portów – kolejność listy nieistotna

```
(s3, cout) = fa(.a=a3, .b=b3, .cin=c3);
(s2, c3) = fa(.c=c2, .a=a2, .b=b2);
(c2, s1) = fa(.a=a1, .b=b1, .c=c1) RETURNS (.c, .s);
(c1) = fa(.b=b0) RETURNS (.c);
```



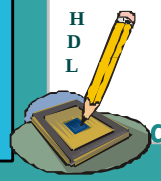
Podstawienie komponentu

- Konkretyzacja funkcji
  - Deklaracja kopii (instancji) w sekcji VARIABLE
    ```
    fa3, fa2, fa1, fa0 : fa;
    ```
  - Argumenty przez nazwy portów – kolejność listy nieistotna
    ```
    fa3.a = a3; -- przypisanie zmiennych do portów
    fa3.b = b3;
    fa3.cin = c3;
    s3 = fa3.s;
    cout = fa3.cout;

    fa2.(a,b,cin) = (a2, b2 , c2); -- przypisanie grupowe do
      portów
    (s2, c3) = fa2.(s, cout);

    fa1.(b,cin,a) = (b1, c1, a1); -- kolejność zmieniona
    (s1, c2) = fa1.(s, cout);

    fa.(a,b,cin) = (a0, b0, cin); -- przypisanie do nazwy funkcji
      zamiast do nazwy kopii
    (c1, s0) = fa0.(cout, s); -- kolejność zmieniona
    ```



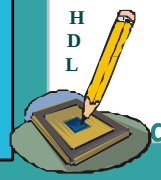
Projekt *adder4* – konkretyzacja w linii

```
INCLUDE "fa"; -- brak rozszerzenia .inc

SUBDESIGN adder4
(
      a3, a2, a1, a0, cin : INPUT;
      b3, b2, b1, b0 : INPUT;
      s3, s2, s1, s0, cout : OUTPUT;
)

VARIABLE
      c3, c2, c1 : NODE;

BEGIN
      (s3, cout) = fa(.a=a3, .b=b3, .c=c3);
      (s2, c3) = fa(.c=c2, .a=a2, .b=b2); -- kolejność argumentów
                                             nieistotna
      (c2, s1) = fa(.a=a1, .b=b1, .c=c1) RETURNS (.c, .s);     -- zmieniona
                                                kolejność wyjść
      (s0, c1) = fa(.a=a0, .b=b0, .c=cin);
END;
```



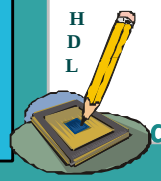
Projekt *adder4* – konkretyzacja w linii

```
INCLUDE "fa"; -- brak rozszerzenia .inc
SUBDESIGN adder4
(
      a3, a2, a1, a0, cin : INPUT;
      b3, b2, b1, b0 : INPUT;
      s3, s2, s1, s0, cout : OUTPUT;
)

VARIABLE
      c3, c2, c1 : NODE;
      fa3, fa2, fa1, fa0 : fa;

BEGIN
      (s3, cout) = fa(.a=a3, .b=b3, .c=c3);
      (s2, c3) = fa(.c=c2, .a=a2, .b=b2); -- kolejność argumentów
                                             nieistotna
      (c2, s1) = fa(.a=a1, .b=b1, .c=c1) RETURNS (.c, .s);     -- zmieniona
                                                                  kolejność wyjść
      (s0, c1) = fa(.a=a0, .b=b0, .c=cin);
END;
```



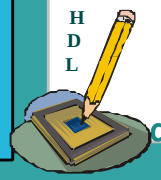
Projekt *adder4* – konkretyzacja

```
[...]
VARIABLE
        c3, c2, c1 : NODE;
        fa3, fa2, fa1, fa0 : fa; -- deklaracja
BEGIN
        fa3.a = a3; -- przypisanie zmiennych do portów
        fa3.b = b3;
        fa3.cin = c3;
        s3 = fa3.s;
        cout = fa3.cout;

        fa2.(a, b, cin) = (a2, b2, c2); -- przypisanie grupowe do portów
        (s2, c3) = fa2.(s, cout);

        fa1.(b, cin, a) = (b1, c1, a1); -- kolejność zmieniona
        (s1, c2) = fa1.(s, cout);

        fa0.(a, b, cin) = (a0, b0, cin);
        (c1, s0) = fa0.(cout, s); -- kolejność zmieniona
END
```



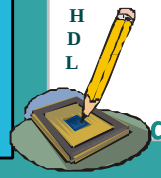
Projekt *adder4* – sygnały grupowe

```
INCLUDE "fa";

SUBDESIGN adder4
(
        a[3..0], cin, b[3..0] : INPUT;
        s[3..0], cout : OUTPUT;
)

VARIABLE
        c[3..1] : NODE;

BEGIN
        (s[3], cout) = fa(.a=a[3], .b=b[3], .c=c[3]);
        (s[2], c[3]) = fa(.c=c[2], .a=a[2], .b=b[2]);
        (c[2], s[1]) = fa(.a=a[1], .b=b[1], .c=c[1]) RETURNS (.c, .s);
        (s[0], c[1]) = fa(.a=a[0], .b=b[0], .c=cin);
END;
```



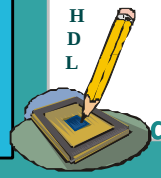
Projekt *adder4* – sygnały grupowe

```
[..]
VARIABLE
        c[3..1] : NODE;
        ifa[3..0] : fa; -- nie można zadeklarować fa[3..0]
                        bo nazwa kopii fa[x] musi być różna od nazwy funkcji fa
BEGIN
        ifa[3].(a,b,cin) = (a[3], b[3] , c[3]);
        (s[3], cout) = ifa[3].(s, cout);

        ifa[2].(a,b,cin) = (a[2], b[2] , c[2]);
        (s[2], c[3]) = ifa[2].(s, cout);

        ifa[1].(a,b,cin) = (a[1], b[1] , c[1]);
        (s[1], c[2]) = ifa[1].(s, cout);

        ifa[0].(a,b,cin) = (a[0], b[0] , cin);
        (s[0], c[1]) = ifa[0].(s, cout);
END;
```



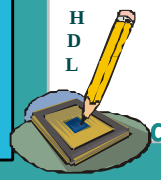
Projekt *adder4* – sygnały grupowe

```
[..]
VARIABLE
        c[3..1] : NODE;
        ifa[3..0] : fa;

BEGIN
        ifa[3].(a,b,cin) = (a[3], b[3] , c[3]);
        (s[3], cout) = ifa[3].(s, cout);

        -- podstawienie grupowe
        ifa[2..1].(a,b,cin) = (a[2], b[2] , c[2], a[1], b[1] , c[1] );
        (s[2], c[3], s[1], c[2] ) = ifa[2..1].(s, cout);

        ifa[0].(a,b,cin) = (a[0], b[0] , cin);
        (s[0], c[1]) = ifa[0].(s, cout);
END;
```



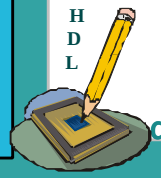
Generowanie kodu

- Niektóre układy cyfrowe mają wyraźnie regularną, powtarzalną budowę (pamięci, rejestry, układy iteracyjne)
- Instrukcja generate jest instrukcją współbieżną służącą do automatycznej generacji struktur regularnych, tworzonych na bazie struktury wzorcowej



Schematy generacji

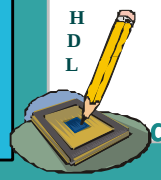
- Generowanie struktur replikowalnych może być dokonywane na dwa sposoby:
  - wg schematu for
  - wg schematu if
- Schemat for jest używany w przypadku struktur regularnych, które można zdefiniować za pomocą indeksu generacji w sposób jak w pętli for
- Schemat if jest używany, gdy istnieje nieregularność w strukturze, rzadko stosowany



Instrukcja *FOR GENERATE*

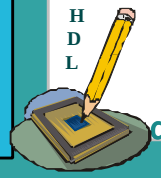
```
FOR zmienna IN zakres GENERATE
   instrukcja1;
   instrukcja2;
END GENERATE;
```

Zmienna nie ma realizacji sprzętowej, służy wyłącznie do indeksowania instrukcji.



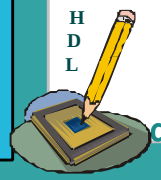
Projekt *adder4 – GENERATE*

```
[..]
VARIABLE
    c[3..1] : NODE;
    ifa[3..0] : fa;

BEGIN
    ifa[3].(a,b,cin) = (a[3], b[3] , c[3]);
    (s[3], cout) = ifa[3].(s, cout);

    FOR i IN 2 TO 1 GENERATE
     ifa[i].(a,b,cin) = (a[i], b[i] , c[i]);
     (s[i], c[i+1]) = ifa[i].(s, cout);
    END GENERATE;

    ifa[0].(a,b,cin) = (a[0], b[0] , cin);
    (s[0], c[1]) = ifa[0].(s, cout);
END;
```



Instrukcja *IF GENERATE*

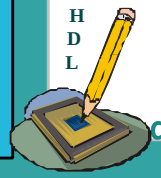
```
IF wyrazenie GENERATE
   instrukcja1;
   instrukcja2;
ELSE GENERATE
   instrukcja3;
   instrukcja4;
END GENERATE;
```



Przykład: *IF GENERATE*

- Parametr spipe decyduje o liczbie stopni potoku w sumatorze równoległym; jeżeli spipe=0 to realizacja kombinacyjna

```
-- sumator równoległy sekwencyjny
-- z potokiem, dołączony zegar do rejestrów
IF(spipe) GENERATE
   out[] = parallel_add (.data[][]=pipe0[][], .clock=clk)
        WITH (WIDTH=D, SIZE=B, WIDTHR=A, SHIFT=0,
   REPRESENTATION="SIGNED", PIPELINE=spipe);

-- sumator równoległy kombinacyjny
-- bez potoku, brak dołączonego zegara
ELSE GENERATE
   out[] = parallel_add (.data[][]=pipe0[][])
        WITH (WIDTH=D, SIZE=B, WIDTHR=A, SHIFT=0,
   REPRESENTATION="SIGNED", PIPELINE=spipe);
END GENERATE;
```



Parametry

- Parametry pozwalają pisać kod uniwersalny
  - Dłużej pisze się kod parametryzowany
  - Krócej przenosi się do innego projektu
- Definicja modułu z parametrem, umieszcza się przed sekcją SUBDESIGN

```
PARAMETER n=4;
SUBDESIGN (….)
```

- Domyślną wartość parametru można zmienić przez:
  - Wywołanie funkcji w linii

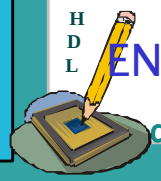
```
(x, y) = funkcja(arg1, arg2) WITH (parametr1 =
wartość1, parametr2 = ”string”);
```

  - Konkretyzację funkcji z parametrami

```
ifa : fa WITH (n = 4);
```

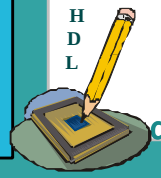
Projekt *adderN – PARAMETER*

```
INCLUDE "fa";
PARAMETER n=4;
SUBDESIGN addern
(
    a[n-1..0], cin, b[n-1..0] : INPUT;
    s[n-1..0], cout : OUTPUT;
)
VARIABLE
    c[n-1..1] : NODE;
    ifa[n-1..0] : fa;
BEGIN
    ifa[n-1].(a,b,cin) = (a[n-1], b[n-1] , c[n-1]);
    (s[n-1], cout) = ifa[n-1].(s, cout);

    FOR i IN n-2 TO 1 GENERATE
    ifa[i].(a,b,cin) = (a[i], b[i] , c[i]);
    (s[i], c[i+1]) = ifa[i].(s, cout);
    END GENERATE;

    ifa[0].(a,b,cin) = (a[0], b[0] , cin);
    (s[0], c[1]) = ifa[0].(s, cout);
END;
```



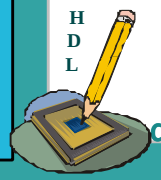
Projekt *adderN – PARAMETER*

```
PARAMETER n = 4;
-- można tez CONSTANT n = 4; ale nie jest widoczny na zewnątrz funkcji
SUBDESIGN addern
(
   a[n-1..0], b[n-1..0], cin : INPUT;
   c[n-1..0], cout : OUTPUT;
)
VARIABLE
   carry_out[n..1]   : NODE;
BEGIN
   carry_out[1] = cin;
   FOR i IN 0 TO n-1 GENERATE
      c[i] = a[i] $ b[i] $ carry_out[i];
      carry_out[i+1] = a[i] & b[i] # carry_out[i] & (a[i] $ b[i]);
    END GENERATE;
    cout = carry_out[n];
END;
```



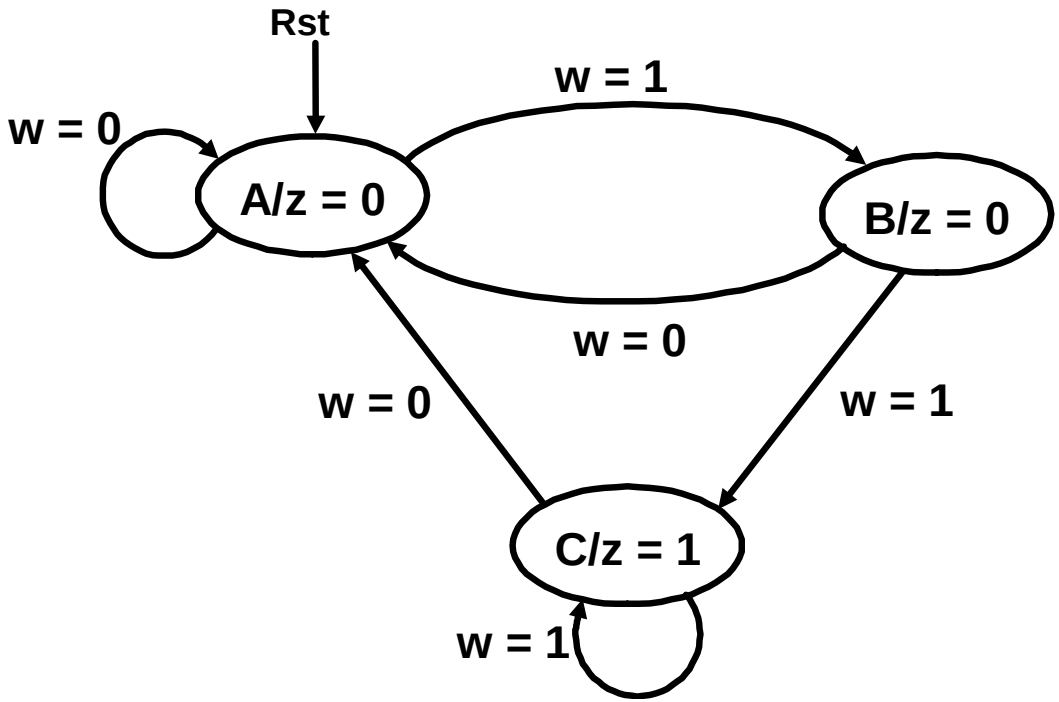
Automaty (jeszcze raz)

- Do realizacji automatu wystarczy podanie diagramu stanów i skonstruowanie tablicy przejść-wyjść
- Kompilator automatycznie dokonuje następujących operacji:
  - wyboru liczby bitów dla kodowania
  - doboru przerzutników D lub T
  - kodowania stanów
  - zastosowania syntezy logicznej do obliczenia funkcji wzbudzeń
- W celu wyspecyfikowania automatu wystarczy:
  - zadeklarować automat w sekcji VARIABLE
  - opisać równania kontrolujące pracę automatu (zegar, reset)
  - opisać przejścia między stanami za pomocą instrukcji: IF-ELSE, CASE lub TABLE



Przykład: automat Moore'a

- Automat Moore'a



s \ x	0	1	z
A	A	B	0
B	A	C	0
C	A	C	1

dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Automat – instrukcja *CASE*

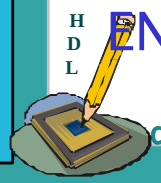
```
SUBDESIGN aut
(
    w, clk, reset : INPUT;
    z : OUTPUT;
)
VARIABLE
    aut : MACHINE WITH STATES (A, B, C);
BEGIN
    aut.clk = clk;
    aut.reset = reset;
    CASE aut IS
    WHEN A =>
        IF (w) THEN aut = B;
        ELSE aut = A; END IF;
    WHEN B =>
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    WHEN C =>
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    END CASE;
    z = (aut == C); -- wyjście zależy tylko od stanu C
END;
```

Pierwszy na liście to stan po włączeniu zasilania i po *reset*

Deklaracja abstrakcyjnej maszyny stanów

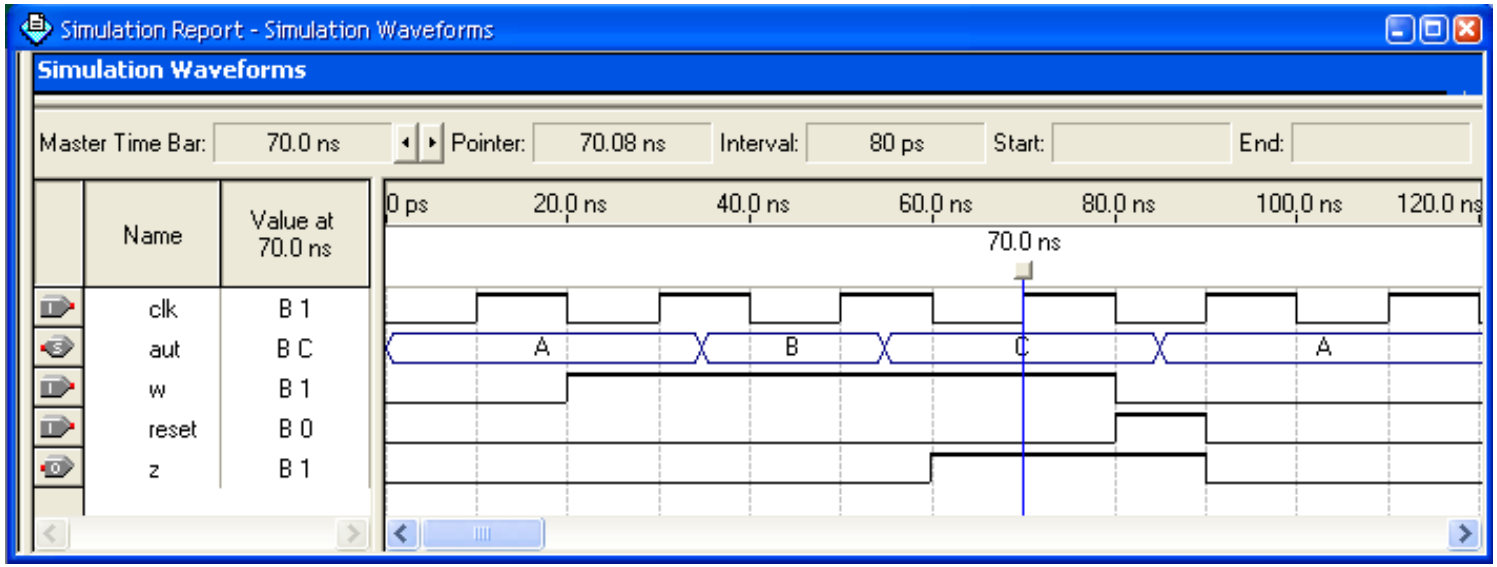
Opis przejść automatu

Opis wyjść automatu



Automat – symulacja

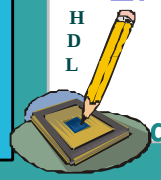
- Realizacja na trzech przerzutnikach przy automatycznym kodowaniu stanów; A=000, B=101, C=110



dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

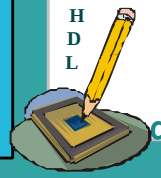
Automat – instrukcja *TABLE*

```
SUBDESIGN aut
(
    w, clk, reset : INPUT;
    z : OUTPUT;
)
VARIABLE
    aut : MACHINE WITH STATES (A, B, C);
BEGIN
    aut.clk = clk;
    aut.reset = reset;
    TABLE
        aut, w => aut, z; -- opis automatu Meale'go
        A, 0 => A, 0;
        A, 1 => B, 0;
        B, 0 => A, 0;
        B, 1 => C, 0;
        C, 0 => A, 0;
        C, 1 => C, 1;
    END TABLE;
-- z = (aut == C);  % Opis wyjścia w tabeli %

END;
```



Automat – instrukcja *IF*

```
SUBDESIGN aut
(
    w, clk, reset : INPUT;
    z : OUTPUT;
)
VARIABLE
    aut : MACHINE WITH STATES (A, B, C);
BEGIN
    aut.clk = clk;
    aut.reset = reset;
    IF (aut==A) THEN
        IF (w) THEN aut = B;
        ELSE aut = A; END IF;
    END IF;
    F (aut==B) THEN
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    END IF;
    IF (aut==C) THEN
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    END IF;
    z = (aut == C);
END;
```



Automat – kodowanie stanów

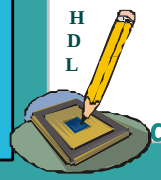
```
SUBDESIGN aut
(
    w, clk, reset : INPUT;
    z : OUTPUT;
)
VARIABLE
    aut : MACHINE OF BITS (q[1..0]) WITH STATES (A=0, B=1, C=2);
BEGIN
    aut.clk = clk;
    aut.reset = reset;
    CASE aut IS
    WHEN A =>
        IF (w) THEN aut = B;
        ELSE aut = A; END IF;
    WHEN B =>
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    WHEN C =>
        IF (w) THEN aut = C;
        ELSE aut = A; END IF;
    WHEN OTHERS => aut = A;
    END CASE;
    z = (aut == C); END;
```

Deklaracja przerzutników automatu

Kodowanie stanów

Realizacja na 2 przerzutnikach, zamiast 3 przy kodowaniu automatycznym (domyślnie: kodowanie one-hot dla układów FPGA)

Wychodzenie ze stanów zabronionych, zakodowano 3 z 4 stanów

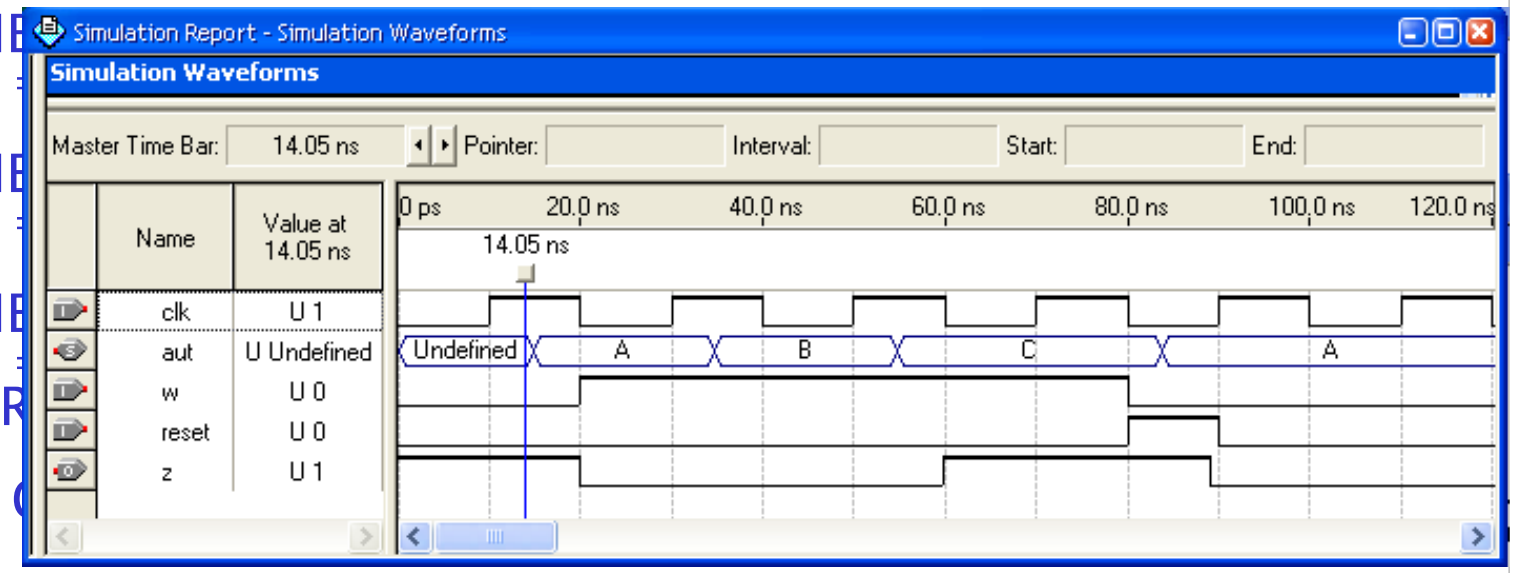


dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska

Automat – kodowanie stanów

```
SUBDESIGN aut
(
    w, clk, reset : INPUT;
    z : OUTPUT;
)
VARIABLE
    aut : MACHINE OF BITS (q[1..0]) WITH STATES (A=3, B=1, C=2);
BEGIN
    aut.clk = clk;
    aut.reset = reset;
    CASE aut IS
    WHEN A =>
        IF (w) THE
        ELSE aut =
    WHEN B =>
        IF (w) THE
        ELSE aut =
    WHEN C =>
        IF (w) THE
        ELSE aut =
    WHEN OTHER
    END CASE;
    z = (aut == (
```

Pierwszy stan zakodowany ≠ 0



dr inż. Paweł Tomaszewicz
Instytut Telekomunikacji
Politechnika Warszawska