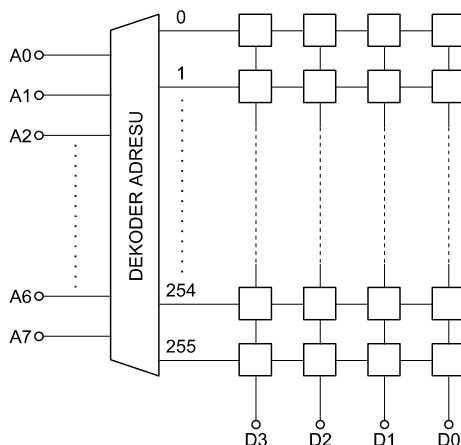


## Programowalne układy logiczne (PLD)

### Układy PLD – informacje ogólne

Pierwsze układy programowalne były pamięciami PROM. Pamięci ROM można używać do realizacji funkcji Boole'owskich jako generator funkcji logicznych.

Weźmy sobie pamięć ROM  $256 \times 4$  (256 komórek 4 bitowych). Pamięć taką organizuje się poprzez 8 linii adresowych („wchodzące do pamięci”) ( $2^8=256$ ) oraz 4 linie danych („wychodzące z pamięci”):



Linie adresowe są liniami wejściowymi matrycy. Te linie są wejściami do dekodera adresu. Dekoder jest układem kombinacyjnym, który na wejściu ma liczbę podaną w naturalnym kodzie binarnym, a wyjście działa na zasadzie „1 z n” (pobudza jedno ze swoich wyprowadzeń). Każda z linii pobudza jedno słowo 4-bitowe.

Linie poszczególnych bitów (biegnące pionowo) są liniami wyjściowymi. Bity pojawiające się na nich tworzą słowo wyjściowe odczytywane z pamięci.

Idea pracy pamięci ROM polega na podaniu adresu, który w dekodерze zostanie przetworzony w kod „1 z n”, pobudzając jedną z komórek pamięci, a zawartość komórek pojawia się na wyjściu pamięci.

Na linie wyjściowe można spojrzeć jako na 4 funkcje Boole'owskie 8 zmiennych. Zmienne funkcji podajemy na wejścia dekodera (wejścia adresowe pamięci), następnie odczytywany jest 1 bit odpowiadający danej kombinacji pobudzenia (8-bitowej kombinacji wejściowej) i ten bit jest wyprowadzany na któreś z wyjść.

Wyjście  $D_i$  jest funkcją sygnałów adresowych  $A_0 - A_7$ :

$$D_i = f(A_0, A_1, \dots, A_6, A_7)$$

Gdy programujemy pamięć, to wpisujemy w 256 komórek 1-bitowych zera i jedynki tak, jak występują one w tabeli prawdy.

Potem podajemy słowo wejściowe na wejścia adresowe, odczytywana jest wartość „0” lub „1” i pojawia się ona na konkretnym wyjściu. Programując pamięć ROM tabelą prawdy konkretnej funkcji uzyskujemy programowany generator funkcji Boole'owskiej.

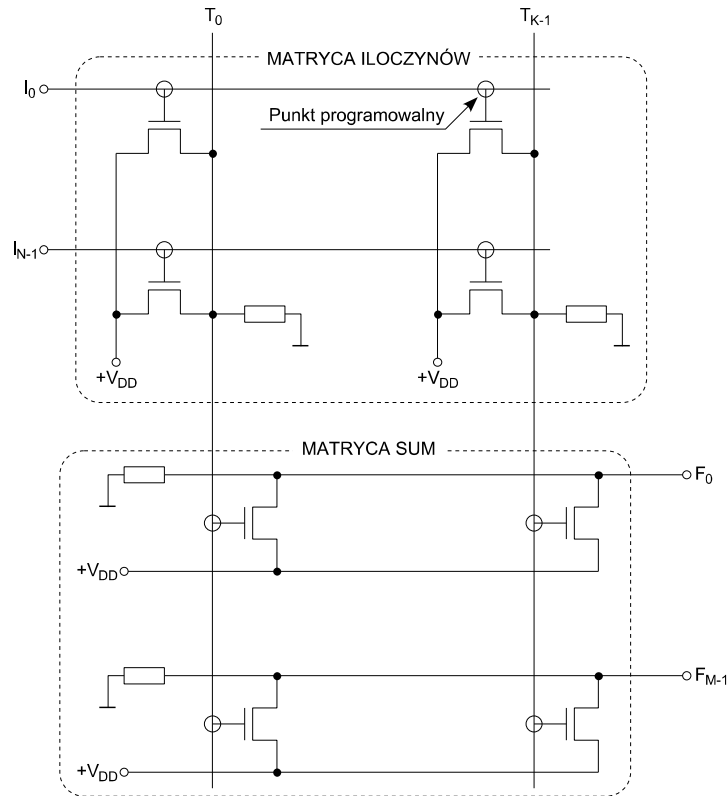
Pamięci ROM stworzone były by spełniać zadania pamięci stałej w systemach mikroprocesorowych, ale można było je wykorzystać jako pierwsze programowalne generatory funkcji.

Takie podejście ma wady. Mała elastyczność – na danym module pamięci zrealizujemy funkcje określonej liczby zmiennych (w naszym przypadku tylko 8 zmiennych). Jeżeli chcemy zrealizować funkcję 4 zmiennych (będzie miała 16 wartości), to na bity  $A_0 - A_3$  podajemy słowo wejściowe, a na pozostałe „0”. Blokuję to dostęp do pozostałych linii pamięci ROM, co zwyczajnie ogranicza pojemność wszystkich funkcji generowanych w danym module. Inną wadą jest wolna praca, szczególnie na początkach swojego istnienia (lata 70-te).

### Matryca programowalna PLD

Niedługo po pamięciach pojawiły się prawdziwe układy programowalne. Oparte były na matrycy programowalnej PLD. Koncepcja matrycy nie zmieniła się. Matryca pracuje na poziomie tranzystorów, dobrze się scala i można uzyskać duże gęstości upakowania. Dobrze przekłada się to na strukturę krzemową. Regularność i pow-

tarzalność położenia komórek, oraz połączeń ułatwia projektowanie struktury i pozwala na duże upakowanie.



Matryca składa się z dwóch części. Zaczynamy od wejść ( $I$ ).  $N$  oznacza ilość wejść matrycy. Linie wejściowe krzyżują się z liniami termów ( $T$ ). Termy są sygnałami wewnętrznymi matrycy.  $K$  oznacza ilość termów. Skrzyżowanie każdej linii wejściowej i linii termu opiera się na tranzystorze CMOS podłączonym między linią termu a linią napięcia zasilającego  $+V_{DD}$ . Ponadto każda linia termu jest poprzez rezystor połączona z masą. Bramka tranzystora jest zasilana przez linię wejściową poprzez punkt programowalny. Ten punkt może realizować połączenie, albo rozwarcie. W pierwszych rozwiązaniach punkty programowalne były realizowane w postaci bezpieczników, które przepalało się w procesie programowania. Można było w ten sposób zdecydować które z linii sygnałów wejściowych są dołączone do bramek tranzystorów.

W dolnej części matrycy koncepcja jest powielona z tą różnicą, że sygnałami wejściowymi są linie termów. Linie wyjściowe ( $F$ ) wychodzą na zewnątrz układu.  $M$  określa liczbę wyjść układu. Linia termu przez punkt programowalny steruje bramką tranzystora MOS. Tranzystor zapięty jest pomiędzy linią wyjściową a linią zasilającą  $+V_{DD}$ . Linia wyjściowa jest połączona poprzez rezystor z masą.

W analizie pracy tego układu zakładamy, że tranzystory MOS zastosowane w matrycy otwierają się „zerem” logicznym. Tranzystory są normalnie rozwarne. Do zastosowania w modelu nadają się najlepiej tranzystory MOS z kanałem typu P (PMOS) normalnie wyłączone. W praktyce stosuje się tranzystory z kanałem typu N z powodu większej ruchliwości elektronów.

Jak działa matryca? Stan linii termu jest ustalany w górnej części matrycy. Wybrane (podczas procesu programowania) przez nas sygnały wejściowe  $I$  są dołączone do tranzystorów. Jeżeli na danej linii  $I$  znalazło się „zero” logiczne, to tranzystor otwiera się i linia termu zwiera się z linią zasilającą  $+V_{DD}$  dostarczając na linię termu „jedynekę” logiczną.

Jeśli na wszystkich liniach wejściowych będą „jedyнки”, to żaden z tranzystorów się nie otworzy. Rezystor będzie ściągał napięcie termów do wartości zerowej.

Wartość  $i$ -tego termu będzie wynosiła „zero” logiczne wtedy i tylko wtedy gdy na wszystkich dołączonych wejściach będzie „jedyнка” logiczna. Jeżeli na którymkolwiek z wejść pojawi się „zero” logiczne, nastąpi otwarcie tranzystora (zwarcie linii termu z linią zasilającą  $+V_{DD}$ ) co powoduje ustawienie się „jedyнки” logicznej na linii termu.

Rezystor ma na celu zabezpieczyć układ przed zwarcie między linią zasilającą a masą układu. Jego wartość powinna być znacznie większa od rezystancji kanału otwartego tranzystora.

Górna oraz dolna część matrycy realizują funkcję NAND. Dla górnej części matrycy realizowana jest funkcja:

$$T_X = \text{NAND}(\alpha_{X,0} + I_0; \alpha_{X,1} + I_1; \dots; \alpha_{X,N-1} + I_{N-1})$$

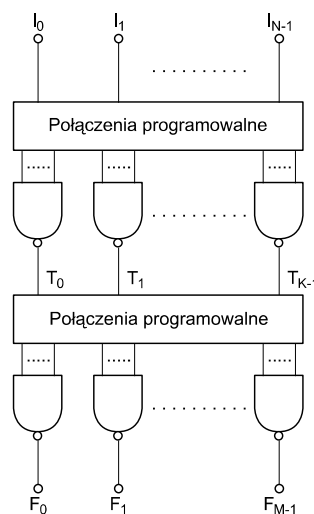
$$\alpha_{ij} = \begin{cases} 1 & \text{– gdy punkt progr. rozwarty} \\ 0 & \text{– gdy punkt progr. Zwarty} \end{cases}$$

Dla części dolnej:

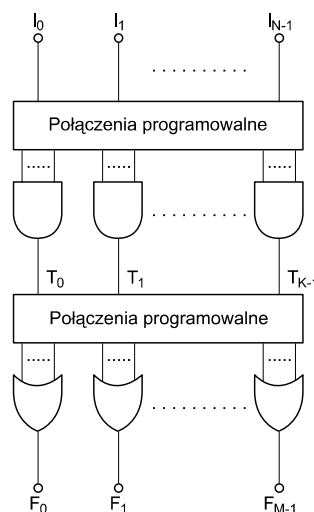
$$F_X = \text{NAND}(\beta_{X,0} + I_0; \beta_{X,1} + I_1; \dots; \beta_{X,K-1} + I_{K-1})$$

$$\beta_{ij} = \begin{cases} 1 & \text{– gdy punkt progr. rozwarty} \\ 0 & \text{– gdy punkt progr. zwarty} \end{cases}$$

Nie rysuje się schematów układów PLD na poziomie tranzystorów. Można zastosować uproszczenie:

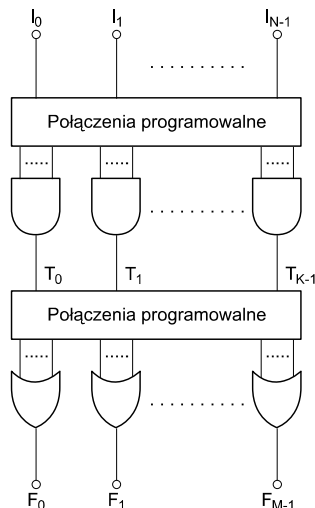


Z praw de Morgana dwuwarstwową strukturę NAND'ów można zastąpić strukturą AND oraz OR:

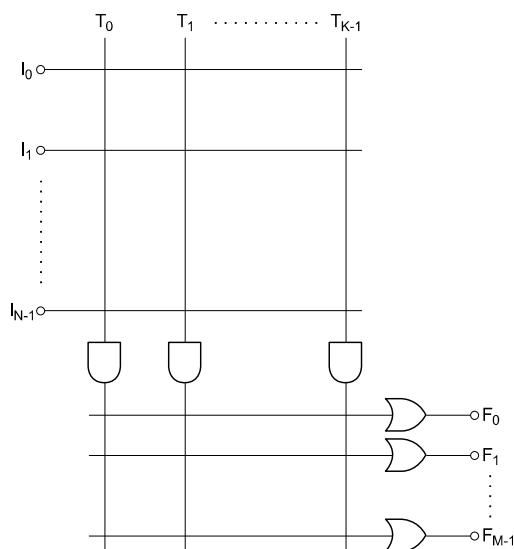


Z tego to powodu górną matrycę układu nazywamy matrycą iloczynów a dolną – matrycą sum.

Na poprzednim wykładzie zapoznaliśmy się ze schematem ideowym matrycy PLD. Narysowaliśmy jej uproszczony schemat w postaci:



Takich schematów matryc się nie rysuje. Sygnały wejściowe rysuje się w postaci linii poziomych, krzyżujących się z pionowymi liniami termów. Na liniach termów rysuje się symbole bramek AND by zaznaczyć, że to, co linia termu „zbiera” z matrycy programowalnej jest funkcją iloczynów. W matrycy dolnej rysuje się linie wyjściowe z symbolami OR. Linie wyjściowe krzyżują się z liniami termów. Linie wyjściowe „zbierają” funkcję sumy. Na skrzyżowaniach linii nie rysuje się punktów programowalnych, bo z góry wiadomo, że skrzyżowania realizują połączenia programowalne:



Standardowo górna matryca jest matrycą iloczynów, a dolna – matrycą sum. Początkowo produkowano układy PLD programowalne, gdzie obie matryce można było poddać procesowi programowania. Praktyka pokazała, że było to marnotrawstwem miejsca w układach scalonych. Poza tym programowanie obu matryc nie było konieczne. Układy PLD najbardziej rozpowszechnione mają tylko jedną matrycę programowalną.

Typ układu	PLA	PAL	PLE
Matryca AND	Programowalna	Programowalna	Stała
Matryca OR	Programowalna	Stała	Programowalna

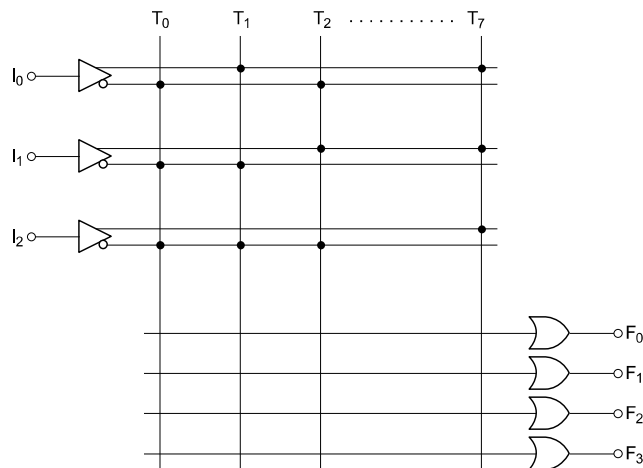
Układy **PLA** (Programmable Logic Array) były pierwszym chronologicznie układem PLD. Miały one obie matryce programowalne. Układy te nie zdobyły dużej popularności z powodu małej gęstości upakowania logiki (bezpieczniki zajmowały sporą powierzchnię układu). Poza tym parametry czasowe nie były zadowalające.

Znaleziono uproszczenie w budowie. Zrezygnowano z programowania matrycy sum. Decydować można tylko o tym jakie iloczyny zmiennych wejściowych mają być ze sobą wymnażane w poszczególnych termach. Wybrane termy są dołączone do konkretnych bramek sumy. Każda z bramek OR w dolnej matrycy ma swój zestaw iloczynów który jest do niej dołączony. Realizujemy funkcję suma iloczynów. To wystarcza do realizacji różnych funkcji kombinacyjnych. Są to układy **PAL** (Programmable Array Logic).

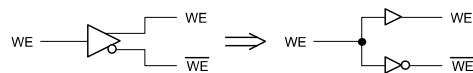
Możliwa jest sytuacja odwrotna (matryca iloczynów – stała, matryca sum – programowalna). W takim wypadku mamy do czynienia z układami **PLE** (Programmable Logic Element). Nazwy oczywiście nic ciekawego nie mówią o architekturze układu. Pełnią tylko rolę nazwy handlowej.

### Układy PLE

By omówić układ PLE posłużymy się układem z 3-ma wejściami i 4-ma wyjściami (i 8 linii termów):



Nowością jest fakt, że wszystkie sygnały wchodzące do matrycy, są w niej dostępne komplementarnie (sygnały wejściowe i ich negacje). Realizuje się to przy pomocy podwójnego bufora:

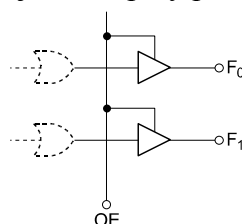


Pionowo „biegną” linie termów jest ich 8 ( $2^3$ ). Linie termów są dołączone do stałych sygnałów wejściowych. Matryca górna reprezentuje wszystkie możliwe kombinacje sygnałów wejściowych. Term  $T_0$  reprezentuje sygnał „000”, term  $T_1$  - „001” a term  $T_7$  - „111”. Dolna matryca jest programowalna.

Jakakolwiek kombinacja wejściowa „zapala” (uaktywnia) dokładnie jeden term. Na jednym z termów pojawia się „1”, a na pozostałych są zera. Patrząc na linię bitu wyjściowego (biegnącą w matrycy dolnej) możemy określić czy dany bit będzie „0” lub „1” w zależności od tego, czy odpowiedni punkt jest zaprogramowany (jest lub nie ma połączenia linii termu z linią wyjściową) i czy dana linia termu jest aktywna, czy nie. Jest to po prostu pamięć PROM o zawartości programowalnej w matrycy sum.

W matrycy iloczynów mamy realizowany szybki dekodery adresu. Układy PLE należy kojarzyć z szybkimi pamięciami programowalnymi.

Jeżeli dany układ ma być pamięcią, musi on posiadać „dodatek” w postaci buforów wyjściowych, umożliwiających podłączenie pamięci do magistral. Realizuje się to przy pomocy buforów 3-stanowych:



To „co wychodzi” z bramki sumy musi przejść przez bufor 3-stanowy, za którym znajduje się końcówka łącze-

niowa. Bufory są sterowane przeważnie jedną linią OE (Output Enable). Bufor 3-stanowy jest aktywny (na OE podano „1”) i na swoje wyjście przekazuje sygnał z wejścia. Gdy OE = „0” - bufor przechodzi w stan wysokiej impedancji (rozwarcie, wyjście „wisi” w powietrzu). Takie rozwiązanie jest niezbędne, gdy wyjście z układu jest dołączone do magistrali. Nie można zwierzać wyjść układów cyfrowych bezpośrednio, ponieważ może dojść do konfliktów.

Należy zapamiętać że komplementarność wejść i bufory 3-stanowe na wyjściach są typowymi rozwiązaniami w układach PLD.

## Układy PAL

W układach tych programowalna jest matryca górna. Układy te dzielimy na układy kombinacyjne, układy rejestrowe (z przerzutnikami) oraz układy z makrokomórkami programowalnymi.

## Układy PAL kombinacyjne

### Układ PAL 16L8

Układ umieszczony w obudowie z 20 wyprowadzeniami. Sygnały wejściowe podawane komplementarnie. Linie pionowe na schemacie reprezentują wejścia układu. Termy na schemacie biegną poziomo. Każda z bramek sum ma własny zestaw termów. Na wyjściach bramek OR znajdują się bufory 3-stanowe są sterowane sygnałami, obliczanymi w matrycy (można programować sygnał OE). Mamy tu sprzężenia zwrotne. Przydają się one wówczas, gdy trzeba stworzyć kilkupoziomowe (kaskadowe łączenie bramek OR) sumy iloczynów. Poza tym dodanie sprzężeń zwrotnych w układzie może sprawić, że układ stanie się układem sekwencyjnym.

Na schemacie układu widać, że bufory 3-stanowe są inwerterami czyli realizowana jest funkcja NOR z zaprogramowanych iloczynów.

Ogólna charakterystyka układu PAL 16L8: Układ ma 10 dedykowanych wejść, 8 dedykowanych wyjść. Nie wszystkie wyjścia są sobie równoważne. Spowodowane jest to obecnością (i nieobecnością) sprzężeń zwrotnych. W układzie 16L8 6 wyjść ma sprzężenia zwrotne. Na jednym wyjściu możemy zrealizować funkcję NOR maksymalnie 7 argumentów. 64 termy, 16 wejść (10 zewnętrznych + 6 sprzężeń zwrotnych). Rozmiar matrycy to ilość punktów programowalnych. Układ 16L8 ma rozmiar matrycy 32×64.

Wyprowadzenia ze sprzężeniami zwrotnymi są opisane jako wejścia/wyjścia. Konsekwencja wprowadzenia sprzężeń zwrotnych jest taka, że poprzez odpowiednie sterowanie buforem 3-stanowym uzyskujemy 3 tryby pracy 6-ciu wyjść I/O. Term sterujący sygnałem OE odpowiedniego bufora może być różnie zaprogramowany. Jeżeli cały czas na OE będzie „1”, wówczas końcówka będzie działała jako wyjście. Jeżeli OE = „0”, wówczas końcówka będzie działała jako dodatkowe wejście. Jeżeli sygnał na OE będzie się zmieniał, to możemy odpowiednią końcówkę traktować zamiennie jako wejście i wyjście. Dlatego też sprzężenie zwrotne zostało umieszczone przed buforem wyjściowym.

### Układ PAL 20L8

Obudowa z 24 wyprowadzeniami. 4 dodatkowe wyprowadzenia, to 4 dodatkowe wejścia. Układ ten posiada w sumie 14 wejść, 8 wyjść (w tym 6 ze sprzężeniami zwrotnymi). Matryca programowalna o wymiarach 40×64.

*[mały oftop: informacje o językach opisu sprzętu]*

Proste języki opisu sprzętu to PALASM i CUPL. Języki te są językami prymitywnymi. Bardziej zaawansowany był HDL. Pierwsze języki typu HDL opracowano na początku lat 80-tych. Języki te miały opisywać wszystko, co związane było z układami cyfrowymi. Nie są to języki programowania.

Przykładowe opisy układów kombinacyjnych w językach PALASM oraz CUPL:

PALASM	CUPL
Zdefiniowanie sygnałów wejściowych i wyjściowych PIN 16 Wy1 {COMB/REG}	Zdefiniowanie sygnałów wejściowych i wyjściowych PIN 16 = Wy1;
...	...
Numer wyprowadzeń bazują na fizycznych wyprowadzeniach układu. Piszemy „PIN <Nr wyprowadzenia> <nazwa sygnału>”. Opcjonalnie można podać tryb pracy pinu (COMB/REG). Kolejnym krokiem jest sekcja równań logicznych: EQUATION Wy1 = (...)	CUPL naśladuje nieco język C i każdą linię należy zakończyć średnikiem. Nagłówek „EQUATION” jest niepotrzebny i można przejść bezpośrednio do pisania równań logicznych: Wy1 = (...) By uzyskać dostęp do termu sterującego buforem trójstanowym, trzeba użyć zapisu: Wy2.OE = (...)
Trzeba uważać, by równanie logiczne dało się przełożyć na zasoby logiczne danego wyprowadzenia układu.	Obostrzenia w temacie złożoności równań logicznych są takie same jak opisane w

Jeżeli mamy wyjścia z opcją Output Enable, musimy użyć notacji z atrybutem TRST:

Wy. TRST = (...)

Powyższe równanie może być tylko pojedynczym iloczynem. Pasują tutaj równania takie jak „a\*b” (AND) lub „/(a+b)” (NOR). Równania w stylu „a+b” oraz „a\*b++c” nie będą akceptowane.

sekcji, dotyczącej języka PALASM.

*[wracamy do opisu kolejnych skalaków]*

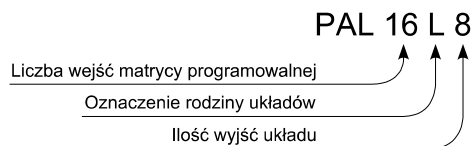
## Układy PAL rejestrowe

### **Układ PAL 16R8**

Obudowa z 20 wyprowadzeniami. Są 3 warianty układów 16RX: Układ 16R8 z 8-ma przerzutnikami oraz układy 16R6 oraz 16R4 odpowiednio z 6 oraz z 4 przerzutnikami. Układ zawiera 8 przerzutników synchronicznych. Rozmiar matrycy 32×64. Obecność przerzutników wymaga użycie sygnału zegarowego, który musi być jednocześnie podawany na wejścia wszystkich przerzutników. Układ posiada dedykowane wejście dla sygnału zegarowego. Bufory 3-stanowe są sterowane wspólnie poprzez sygnał pobrany z wyprowadzenia OE. Jest tu 8 dedykowanych wyjść (nie ma przełączania wejście/wyjście). Sprzężenie zwrotne jest brane z przerzutnika. Układ ma zatem 8 wejść „normalnych” (zewnętrznych) oraz 8 wejść wynikających ze sprzężeń zwrotnych. Do bramki OR jest podłączonych 8 termów.

W układach ze zmniejszoną liczbą przerzutników (16R6 oraz 16R4) bloki działające kombinacyjnie (pozbawione przerzutników) działają tak, jak bloki kombinacyjne w układzie 16L8: 7 termów steruje bramką OR, a 8-my term steruje buforem 3-stanowym. Sprzężenie zwrotne jest brane z bufora 3-stanowego i pin taki może pracować zarówno jako wyjście jak i wejście układu.

## Oznaczenia układów PAL



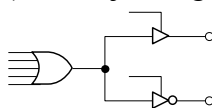
### Ilość wejść matrycy programowalnej

Jest to suma ilości wejść będących pinami wejściowymi w obudowie oraz ilości sprzężeń zwrotnych. W układzie 16L8 mieliśmy 10 wejść zewnętrznych oraz 6 sprzężeń zwrotnych.

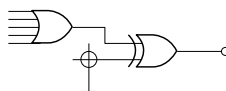
### Oznaczenie rodziny układów

**H, L** – Układy kombinacyjne z ustaloną polaryzacją wyjść (H – polaryzacja dodatnia, L – ujemna). W układach rodziny L funkcja sumy jest zanegowana przez bufor 3-stanowy.

**C** – Układy kombinacyjne z wyjściami komplementarnymi. Każda funkcja obliczona w danym bloku logicznym jest dostępna na dwóch wyprowadzeniach (afirmacja i negacja sygnału):



**P** – Układy kombinacyjne z programowalną polaryzacją wyjść. Wynik z bramki OR przechodzi przez bramkę XOR, której drugi argument pobierany jest z punktu programowalnego:



**R** – Układy rejestrowe.

**RA** – Układy rejestrowe tzw. asynchroniczne. Nazwa jest myląca, bo przerzutniki są synchroniczne. Asynchroniczne jest sterowanie otoczeniem przerzutnika.

**V** – Układy z makrokomórkami programowalnymi. Układy te posiadają konfigurowalne bloki, zwane makrokomórkami. Ich konfiguracja pozwala na podjęcie decyzji jaka funkcja (logiczna, rejestrowa, sterująca) będzie przez taką makrokomórkę realizowana.

### Ilość wyjść układu

W układach rejestrowych liczba ta opisuje ilość wyjść rejestrowych (ile jest przerzutników w układzie). W układach kombinacyjnych liczba ta wprost opisuje ilość wyjść.

## PALCE 20RA10

Układ zawiera 20 wejść do matrycy programowalnej (10 wejść zewnętrznych + 10 sprzężeń zwrotnych) oraz 10 wyjść zaopatrzonych w przerzutniki. Rozmiar matrycy programowalnej to 40×80 (8 termów w każdym bloku wyjściowym). Na każdy z przerzutników można załadować równolegle informację z pinów wyjściowych układu. Możliwe jest to dzięki sygnałowi ładującemu Preload (PL) pobieranemu z pinu 1 obudowy. Przydaje się to podczas testowania układu w systemie. Poprzez załadowanie danych do przerzutników, możemy testować układ z dowolnego, ustalonego przez nas stanu. Ładowanie realizuje się synchronicznie. Sygnały zegarowe są indywidualne dla każdego przerzutnika. Obliczane są z termów. Przerzutniki posiadają również wejścia AR (Asynchroniczny Reset) oraz AP (Asynchroniczny Preset). Pobierane są z matrycy jako termy (podobnie jak sygnały zegarowe). Otwieranie buforów 3-stanowych jest zrealizowane w sposób globalny (sygnał pobierany z pinu 13) lub indywidualnie z termów, które obliczają funkcję OE.

Ze schematu makrokomórki układu widać, że tylko 4 termy są podłączone do bramki OR. Reszta termów to termy sterujące (AR, AP, CLK, OE). Mamy pełną kontrolę nad każdym blokiem wyjściowym indywidualnie. Dołączenie multiplexera sterowanego iloczynem sygnałów AR i AP sprawia, że gdy oba sygnały będą miały stan wysoki (stan sprzeczny), przerzutnik jest pomijany. Funkcja obliczona w bramce OR omija przerzutnik i komórka pracuje w trybie pracy kombinacyjnej. Możemy w ten sposób przełączać tryby pracy bloków wyjściowych z kombinacyjnego na rejestrowy i na odwrót. W innych układach PLD ustawianie trybu pracy danej komórki



jest dokonywane podczas programowania układu (jednostrotnie) i nie jest możliwa zmiana trybu pracy komórki podczas jego normalnej pracy. Poza tym dodanie dodatkowego punktu programowalnego i bramki XOR daje nam możliwość konfiguracji polaryzacji (L/H) sygnału wyjściowego.

Podsumowując działanie makrokomórki mamy dużo opcji funkcjonalnych przerzutnika. Jest to możliwe kosztem tego, że tylko połowa termów w matrycy może być wykorzystywana do obliczania funkcji logicznych. To jest wadą układu. Inna wada, to opóźnienia sygnału CLK, który jest pobierany z matrycy AND.

## Układy PAL rodziny V

### **PALCE 22V10**

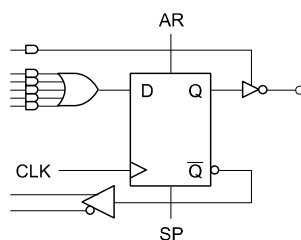
Układy te miały możliwości konfiguracyjne. Nie były one jednak tak „kosztowne” jak w poprzednio omawianym układzie z rodziny RA. Układ 22V10 ma 22 (12 zewnętrznych + 10 sprzężeń zwrotnych) sygnały wejściowe dochodzące do matrycy oraz 10 sygnałów wyjściowych. Rozmiar matrycy to 44×132. Termy w układzie są porozkładane nierównomiernie: skrajne wyprowadzenia mają 8 termów, a środkowe wyprowadzenia – 16. Ułożone jest to według reguły 8, 10, 12, 14, 16, 16, 14, 12, 10, 8. Obudowa układu ma 24-wyprowadzenia. Układ ma dodatkowe termy globalne. Skrajnie górny nazwany jest AR a skrajnie dolny to SP. Te dwa termy nazywane są węzłami globalnymi (wewnętrznymi).

Układ powiada 10 identycznych makrokomórek. 11 wejść jest ogólnego przeznaczenia. Sygnał zegarowy jest pobierany z dedykowanego wejścia (12). Można go również wykorzystywać jako jeden ze zwykłych sygnałów wejściowych (gdy używamy układu do działania tylko w trybie kombinacyjnym). Z każdej makrokomórki wychodzi sprzężenie zwrotne do matrycy. Możemy podczas konfigurowania układu wybrać czy sprzężenie ma pochodzić z przerzutnika, czy z pinu wyjściowego. Termy globalne (AR i SP) to sygnały Asynchroniczny Reset i Synchroniczny Preset. 10 pinów wyjściowych może pracować jako dodatkowe wejścia.

Każda makrokomórka posiada dwa punkty programowalne: S0 oraz S1. Daje to 4 tryby pracy każdej komórki. Punkty S0 oraz S1 są podłączone do wejść adresowych multiplexerów. Zaprogramowanie multiplexerów dokonane zostaje podczas programowania układu i nie ma możliwości przełączania multiplexera podczas normalnej pracy układu. W makrokomórkach układu 22V10 mamy dwa multiplexery: Multiplexer 4 na 1 decyduje o tym co jest obliczane jako funkcja na danym pinie. Multiplexer 2 na 1 decyduje o sprzężeniu zwrotnym (konfigurowany tylko punktem S1).

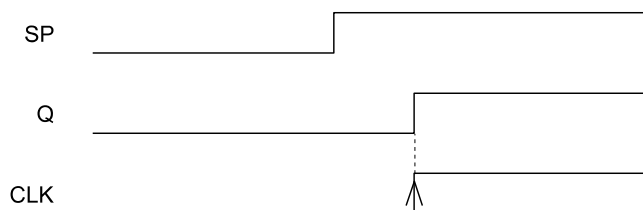
Punkt S1 decyduje o trybie pracy makrokomórki. Gdy S1=1 to mamy tryb kombinacyjny, a gdy S1=0 wówczas makrokomórka pracuje w trybie rejestrowym. S0 decyduje o polaryzacji sygnałów. S0=0 przekazuje na wyjście negację sygnału, a S0=1 – afirmację sygnału.

Gdy jakiś punkt programowalny ma wartość równą 0, wówczas mamy do czynienia z niezaprogramowanym punktem programowalnym realizującym zwarcie. Schemat makrokomórki dla konfiguracji S0=0 i S1=0 jest następujący:



Sprzężenie zwrotne dla trybów rejestrowych pobiera się z przerzutnika (sprzed bufora 3-stanowego). Podczas pracy kombinacyjnej układu sprzężenie zwrotne brane jest z bufora.

Term AR pozwala na asynchroniczne skasowanie zawartości przerzutnika. Natychmiast po pojawieniu się sygnału aktywnego na tym termie następuje skasowanie zawartości wszystkich przerzutników w układzie. Reakcja na pojawienie się aktywnego sygnału SP nastąpi dopiero przy pojawieniu się narastającego zbocza sygnału zegarowego:



Jak do tego „dostać się” w językach opisu?

PALASM	CUPL
Zdefiniowanie numeru wyprowadzenia i skojarzonego z nim sygnału z dodaniem atrybutu COMB (konfiguracja rejestrowa) lub REG (konfiguracja rejestrowa). Jest to opcjonalne. <b>PIN 16 NazwaWy {COMB/REG}</b> ... By dostać się do termów globalnych (AR, SP) należy zdefiniować węzeł wewnętrzny (globalny) <b>NODE 1 NazwaWezla</b> Trzeba wiedzieć ile jest węzłów wewnętrznych dostępnych w układzie. <b>EQUATIONS</b> Równania logiczne dla zdefiniowanych pinów wyjściowych. ... By dostać się do termów AR i SP trzeba użyć nazw kwalifikowanych: <b>NazwaWezla.SETF = ...</b> Równanie sygnału SP <b>NazwaWezla.RSTF = ...</b> Równanie sygnału AR. Równania powyższe muszą być równaniami jednego termu (tak jak te od funkcji OE z poprzedniego wykładu).	Zdefiniowanie sygnałów wejściowych i wyjściowych <b>PIN 14 = NazwaWy;</b> ... Nie podaje się atrybutów „COMB” oraz „REG”. Nie definiuje się również węzłów wewnętrznych. Do wszystkiego stosuje się nazwy kwalifikowane. Przykładowo: <b>NazwaWy.D = (...)</b> będzie równaniem wejścia przerzutnika. Gdybyśmy napisali samo „NazwaWy =” to dane wyjście pracowałoby kombinacyjnie. By obsługiwać termy SP i AR należy napisać: <b>NazwaWy.SP = ...</b> <b>NazwaWy.AR = ...</b> Równania powyższe muszą być równaniami jednego termu.  Jeżeli zdefiniujemy inne funkcje SP i AR dla każdego wyjścia, to kompilator wywali błąd. Definiowanie funkcji dla jednego wyprowadzenia, będzie obejmowało swoim działaniem pozostałe wyprowadzenia. Jeżeli chcielibyśmy definiować działanie funkcji AR i SP indywidualnie, to funkcje te musiałyby być identyczne.

## PALCE 16V8

Układy zaprojektowane w ten sposób, by pełniły rolę zamienników dla układów 16R8, 16L8 oraz 10H8. Standardowymi parametrami jest zamknięcie układu w obudowie z 20 wyprowadzeniami oraz matryca 32×64. Rozkład wyprowadzeń układu jest analogiczny jak w 16R8 lub 16L8.

Makrokomórka układu posiada dwa lokalne punkty programowalne SL0 i SL1. Dostępne są jeszcze dwa punkty globalne SG0 i SG1. Punkt SG0 decyduje tylko o działaniu skrajnych komórek. Jeżeli SG0=1 to wyprowadzenia 1 oraz 11 będą pełniły rolę wejść  $I_0$  oraz  $I_9$  (z układu 16L8). Gdy SG0=0 to wyprowadzenia te będą pracowały jako dedykowane wejścia CLK i OE (tak jak w 16R8).

SG1 służy do wyboru architektury. Jeżeli SG1=1 to mamy układ 16R8 lub 16L8. Gdy SG1=0 to układ będzie działał jak 10H8 (architektura nie omówiona na wykładzie). Możliwe jest to dzięki zastosowaniu multiplekserów w makrokomórce.

Gdy SL0=1 to makrokomórka będzie działała w trybie kombinacyjnym. Gdy SL0=0 to makrokomórka będzie działała w trybie rejestrowym. SL0=0 oznacza również sprzężenie zwrotne z rejestru, a SL0=1 oznacza sprzężenie zwrotne z pinu. Nie dotyczy to skrajnych komórek, gdzie sytuacja jest skomplikowana.

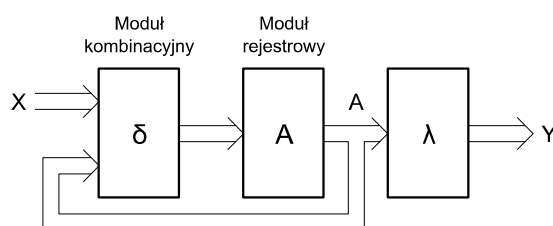
Układ ten może pracować w 3 trybach pracy. Tryb **REJESTROWY** (16R8). Wówczas w języku CUPL należy powołać się na układ G16V8MS. Tryb pracy **PROSTY** (10H8) pomijamy, bo ta rodzina układów nas nie interesowała. Ostatnim trybem pracy jest tryb **ZŁOŻONY** (16L8) gdzie powołujemy się na układ G16V8MA.

## Układy PLS

### PLSCE 105

Układ posiada dwie matryce programowalne. Posiada 16 zewnętrznych wyprowadzeń wejściowych oraz 6 sprzężeń zwrotnych (co daje w sumie 22 wejścia na matrycę AND). Termów jest 48. Przerzutniki znajdujące się w układzie są typu RS. Sygnał zegarowy jest pobierany z dedykowanego wejścia (pin 1). Przerzutników jest w sumie 14 (8 wyjściowych + 6 przerzutników wewnętrznych generujących sprzężenia zwrotne). Przerzutniki wewnętrzne pełnią rolę węzłów wewnętrznych (lub węzłów zagrzebanych). Rozmiar matrycy AND to  $45 \times 48$  ( $45 =$  po dwa sygnały z wejść + sygnał C) matryca OR ma rozmiar  $48 \times 29$  ( $29 =$  po dwa sygnały z przerzutników + sygnał C). Specjalny pin pozwala na podawanie sygnału Preset lub Output Enable (wyboru dokonuje się podczas programowania układu).

Struktura układów sekwencyjnych składa się z wektora przerzutników, które reprezentują zmienne stanów oraz bloku sprzężenia zwrotnego, który generuje pobudzenia dla bloku przerzutników. Przełączenie się automatu maszyny stanów polega na tym, że blok kombinacyjny generuje nowe pobudzenia sterujące przerzutnikami, które powodują przełączanie się ich przy kolejnych taktach sygnału zegarowego:



Wektor przerzutników oznacza się literą A. Powstaje tam sygnał stanu automatu A, który trafia do modułu obliczającego funkcję wyjść λ i poprzez sprzężenie zwrotne trafia do modułu kombinacyjnego δ generującego pobudzenia dla przerzutników na podstawie bieżącego stanu automatu oraz wektora wejściowego X. Tak wygląda schemat automatu Moore'a (część generująca sygnały wyjściowe λ nie zależy od sygnału wejściowego).

Wyżej opisany schemat ma swoje odbicie na strukturze układu. Sygnały trafiające do przerzutników są sygnałami RS. Zastosowanie przerzutników RS podyktowane jest tym, że nie jest potrzebna pętla sprzężenia zwrotnego by automat utrzymał swój stan. Wystarczy podać pobudzenie 00. Przy zastosowaniu przerzutników typu D do zachowania stanu przerzutnika wymagane jest sprzężenia zwrotne.

Więcej trzeba napisać o sygnale C (Complement). Pełni on rolę dopełnienia wszystkich warunków przełączających automat. Odpowiada on przejściom „else” w grafie automatu. Przejście takie jest aktywne gdy żaden z warunków przełączających automat nie będzie spełniony. Sygnał C stanie się aktywny gdy żaden z termów nie będzie aktywny. Dzieje się to dlatego, że sygnał C będzie zanegowaną sumą wybranych termów (wyboru dokonuje się podczas programowania).

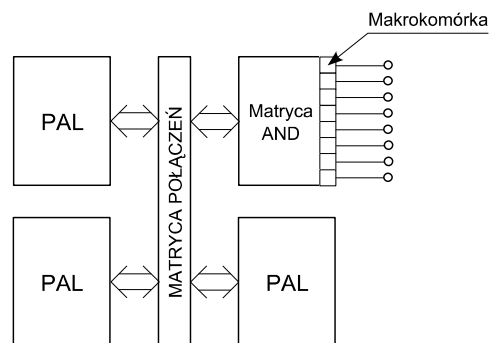
### Układy CPLD rodziny XC 9500XL

Układy CPLD pochodzą od największych układów PAL. Największe układy PAL posiadają w swojej budowie makrokomórki programowalne. Przykładowo układ PALCE 26V12. Posiadał programowalną matrycę AND o rozmiarze  $52 \times 150$ , miał 12 makrokomórek (podobnych do tych, które zastosowano w PAL 22V10). Obudowa z 28 wyprowadzeniami. W układzie tym pojawiły się dwie linie zegarowe.

Innym dużym układem był PALCE 29M16. Posiadał matrycę  $58 \times 188$ . Zamknięty w obudowie z 24 wyprowadzeniami. Zastosowano tu specyficzne rozwiązanie: było tylko 6 wyprowadzeń wejściowych oraz 16 makrokomórek, które były skojarzone z wyprowadzeniami wejściowo-wyjściowymi.

Istniała swego czasu układy MegaPAL, ale nie przyjęły się w zastosowaniach praktycznych, ponieważ ich działanie było wolne. Rozbudowana matryca wprowadzała dodatkowe opóźnienia (dzięki pojemnościom pasywnym).

Idea układów oparta była na tym, że w jednym układzie scalonym (w jednej strukturze) umieszczono kilka matryc PAL (zamiast jednej dużej) o ograniczonej wielkości (by nie tracić na szybkości działania tych matryc). Układ CPLD składa się z kilku matryc PAL połączonych ze sobą jedną globalną matrycą połączeniową:



Moduły PAL składają się z dwóch części: matrycy AND oraz makrokomórek (do makrokomórek są doprowadzone termity z matrycy AND). Matryca połączeń musi zapewniać komunikację między matrycami PAL oraz sprzężenia zwrotne między nimi.

Specyficzną rzeczą w układach CPLD jest brak dedykowanych wejść. Wszystkie wyprowadzenia są skojarzone z makrokomórkami. Mogą działać jako wejścia lub wyjścia, zależnie od sposobu zaprogramowania układu. Istnieją dedykowane wejścia zegarowe, kasujące lub ustawiające, jednak i one mogą być pobierane ze „zwykłych” wyprowadzeń dwukierunkowych.

### Rodzina XILINX XC 9500XL

Pierwszym kryterium podziału układów jest napięcie zasilania układów. Układy XC 9500 były zasilane napięciem 5V. Układu XC 9500XL są zasilane napięciem 3,3V a układu XC 9500XV – napięciem 2,5V. Obniżanie napięcia zasilania podyktowane było zmniejszeniem się strat mocy, które są proporcjonalne do kwadratu napięcia zasilającego. Zmniejszenie napięcia z 5V na 3,3V powoduje zredukowanie się strat mocy o około 40%. Redukcja napięcia z 5V do 2,5V redukuje straty mocy o 75%.

Rodzina układów XC 9500XL zasilana jest napięciem 3,3V. Blok wewnętrznej logiki i bloki wejść/wyjść są zasilane oddzielnymi napięciami. 3,3V zasilą logikę wewnętrzną. Bloki wejściowe i wyjściowe mogą być również zasilane napięciem 3,3V lub napięciem 2,5V by umożliwić współpracę układu z niskonapięciowymi układami cyfrowymi. Wejścia układów tolerują napięcia wejściowe o wartości 5V, co umożliwia zasilanie tych wejść wprost z układów TTL lub innych zasilanych napięciem 5V. Pamięć konfiguracji układu jest wykonana w technologii Flash (nieulotna, kasowalna elektrycznie). Konsekwencją tego jest możliwość konfiguracji układu, znajdującego się w systemie (ISP In-System Programming) bez wyciągania układu z systemu. Interfejsem konfiguracji układu jest port JTAG. Początkowo jego przeznaczeniem było testowanie układów, znajdujących się w systemie.

Rodzina XC 9500XL składa się z 4 układów. Różnią się one między sobą przede wszystkim ilością makrokomórek, która waha się od 36 do 288 makrokomórek. Każdy zestaw makrokomórek nazywa się blokiem funkcyjnym. W skład takiego bloku wchodzi 18 makrokomórek. Przyrost ilości bloków funkcyjnych jest eksponentialny: najmniejszy układ ma 2 bloki funkcyjne, następne mają 4, 8, a układ XC 95288XL ma ich 16. Jedna makrokomórka odpowiada jednemu rejestrowi (przerzutnikowi). Szybkość pracy kombinacyjnej układu  $T_{PD}=5\div6$  ns. Maksymalna częstotliwość pracy systemu waha się od 178 do 208 MHz.

Następna sprawa, to obudowy w których dostępny jest dany układ. Tabela w PDFie „ds054\_9500xlfamily.pdf” podaje nam ilość dostępnych wyprowadzeń wejścia/wyjścia. W najmniejszym układzie mamy 36 makrokomórek, więc powinno być 36 wyprowadzeń wejść/wyjść ale według tabeli są dostępne tylko 34 wyprowadzenia. Spowodowane jest to nietrywialnym zasilaniem. Układy mają szereg wyprowadzeń związanych z masą i napięciami zasilania. W układzie XC 9536XL mamy przykładowo 3 wyprowadzenia GND oraz 3 wyprowadzenia do podawania napięcia zasilającego. Dodatkowe 4 piny należą do interfejsu JTAG. Jeżeli jakaś makrokomórka nie jest podłączona do pinu, to nie tracimy jej. Ona pracuje, jej sygnał logiczny wraca do matrycy globalnej. Uczestniczy ona w projekcie i pełni rolę węzła zagrzebanego. Poza tym im więcej wyprowadzeń posiada obudowa układu, tym więcej wyprowadzeń jest przeznaczonych jako piny zasilania. Wszystkie wyprowadzenia masy muszą być z nią zwarte. Nie może zaistnieć takie zjawisko, że któreś z wyprowadzeń GND będzie „wisiało” niepodłączone.

Zajmiemy się teraz architekturą układu. Na schemacie po prawej stronie przedstawione są bloki funkcyjne. To są moduły PAL, występujące w liczbie od 2 do 16 w układzie. Wyprowadzenia zewnętrzne są połączone z blo-

kiem wejściowo-wyjściowym. Wejścia są standardowo dwukierunkowe. Dolne wejścia mają funkcje specjalne. Do każdego bloku funkcyjnego wchodzi 54 sygnały z globalnej matrycy połączeniowej. Nie ma bezpośredniego przejścia z któregośkolwiek z pinów. Sygnały wyjściowe (18 sygnałów) z makrokomórek kierowane są do boku we/wy i jednocześnie do globalnej matrycy połączeniowej.

Specyficznymi sygnałami są sygnały globalne. Sygnał GCK (Global Clock) jest podawany trzema liniami. Przerzutniki można zasiląć jednym z trzech sygnałów globalnych, co wystarcza do średnio zaawansowanych projektów. Sygnałem GSR (Global Set/Reset) sterowane są wszystkie przerzutniki, których sposób konfiguracji decyduje o tym, czy dany przerzutnik będzie ustawiany czy kasowany sygnałem GSR. Trzecim sygnałem jest GTS pełni rolę sygnału Output Enable. Mniejsze układy mają 2 a większe – 4 sygnały GTS. Jeżeli nie będziemy wykorzystywać sygnałów globalnych (nie korzystamy z zegara, czy buforów 3-stanowych), to piny te można wykorzystywać jako zwykłe wyprowadzenia wejścia/wyjścia.

Na górze schematu przedstawiono interfejs JTAG przeznaczony do konfigurowania i testowania układu. Nie korzysta się z niego podczas zwykłej pracy układu. JTAG składa się z 3 linii wejściowych i jednej wyjściowej.

### Blok funkcyjny i makrokomórka

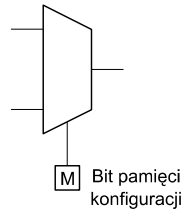
Blok funkcyjny można by zastąpić układem PAL 54V18 (gdyby coś takiego istniało). Wchodzi do niego 54 sygnały z matrycy globalnej. Sygnały te są dostępne komplementarnie. Wewnątrz bloku funkcyjnego nie ma wewnętrznych sprzężeń zwrotnych. W układach rodziny XC 9500XL wszystkie sprzężenia zwrotne biegną przez matrycę globalną, dzięki czemu mamy jednorodną strukturę układu. W bloku funkcyjnym jest 90 termów (5 termów na jedną makrokomórkę).

Istnieje coś takiego jak Product Term Allocator. Sprawia on, że mamy pewne ograniczone możliwości alokacji termów do makrokomórek. Standardowo do jednej makrokomórki jest przypisanych 5 termów. Układ PTA umożliwia przenoszenie nieużywanych termów z jednej makrokomórki do sąsiednich. Przydaje się to, ponieważ 5 termów na makrokomórkę jest zbyt małą ilością przy realizacji bardziej skomplikowanych funkcji. Jest to lepsze rozwiązanie niż to, zastosowane w PAL 22V10 gdzie termy były „przyszyte na stałe” do bramek OR w sposób nierównomierny.

Z każdej makrokomórki wychodzą dwa sygnały wyjściowe. Jest to właściwy sygnał logiczny oraz PTOE (Product Term Output Enable). Oprócz obok globalnego sygnału OE (GTS) istnieją lokalne sygnały obliczane w danej makrokomórce, które mogą sterować buforem 3-stanowym. Wyjściowy sygnał logiczny z makrokomórki trafia do bloku we/wy oraz równolegle do matrycy globalnej (w ten sposób realizuje się sprzężenie zwrotne). Istnieje możliwość przeprowadzenia syntezy sygnału zegarowego (obliczanie sygnału jako termu). Psuje to parametry czasowe układu, ale jest możliwe do realizacji.

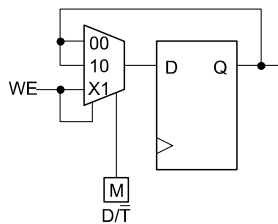
## Blok funkcyjny i makrokomórka c. d.

Wypadałoby powiedzieć coś o multiplekserach, które będą się teraz często pojawiały na schematach logicznych układów. Są one multiplekserami konfigurowalnymi przy użyciu bitu pamięci konfiguracyj. Multipleksery pojawiające się na schematach nie są przełączane podczas normalnej pracy układu. Konfiguruje się je tylko podczas programowania.



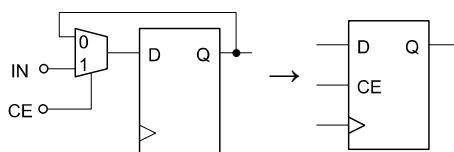
Na schemacie logicznym makrokomórki pojawiła się bramka XOR dzięki której realizuje się operację polaryzacji sygnału. Konfigurując multiplekser możemy zdecydować czy sygnał przechodzący przez bramkę XOR z Product Term Allocatora na wejście przerzutnika będzie negowany czy nie.

Przerzutnik w makrokomórce można ominąć co umożliwia pracę makrokomórki w trybie kombinacyjnym lub rejestrowym. Przerzutnik jest konfigurowalny: może działać jako przerzutnik typu D lub typu T. Z przerzutnika typu D można w bardzo łatwy sposób zrobić przerzutnik T. Potrzebny jest do tego 3-wejściowy multiplekser i pętla sprzężenia zwrotnego:



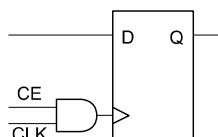
Podanie stanu wysokiego na bit sterujący sprawia, że przerzutnik będzie działał jako przerzutnik typu D. Podanie stanu niskiego zmienia tryb pracy na typ T. Podczas projektowania układu nie trzeba przejmować się sposobem wyboru przerzutnika. Tym zajmie się środowisko, w którym będziemy konfigurować układ.

Istotne jest wejście CE (Clock Enable) pozwalające na przełączenie się przerzutnika. Jeżeli to wejście jest nieaktywne, to przerzutnik podtrzymuje swój stan. Wejście CE realizuje się dzięki podłączeniu na wejście przerzutnika multipleksera. Multiplekser steruje się sygnałem CE:

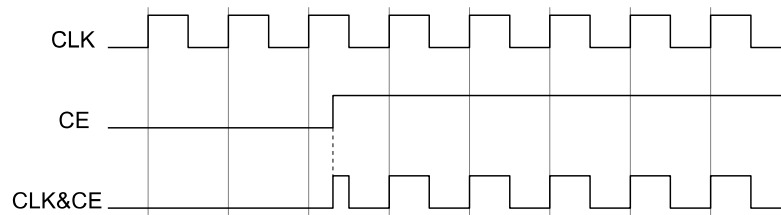


Podanie na wejście CE stanu niskiego sprawia, że przerzutnik będzie się przełączał z każdym narastającym zboczem zegarowym podtrzymując swój stan poprzedni. Takie rozwiązanie można zastosować tylko dla przerzutnika D.

Złym rozwiązaniem jest bramkowanie sygnału zegarowego:

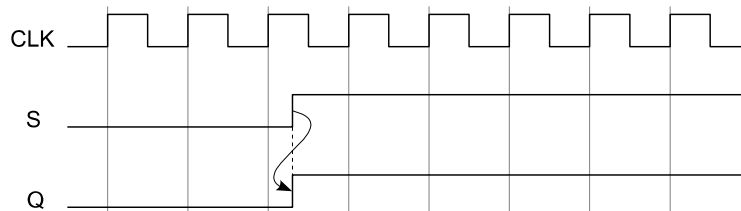


Należy zdawać sobie sprawę z tego, że każda manipulacja przy sygnale synchronizującym jest niebezpieczna. Idea synchronizacji pracy układu cyfrowego polega na tym, że momenty zbocza narastającego sygnału zegarowego są momentami przełączania się przerzutników w układzie. Jeżeli zastosowalibyśmy bramkowanie sygnału zegarowego, to może pojawić się problem w postaci narastającego zbocza zegarowego, które może pojawić się w nieodpowiednim momencie czasowym:

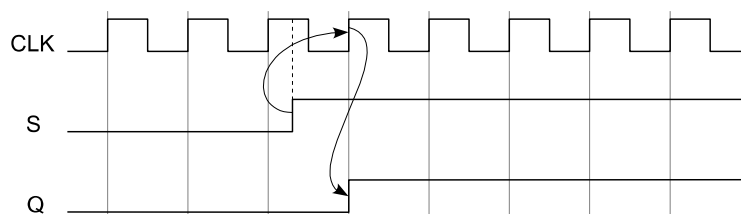


Narzędzia syntezy wyświetlają w takim wypadku komunikat ostrzegawczy „Gated Clock”. Zegary bramkowane mogą „popsuć” projekt, ponieważ bramkowanie sygnału zegarowego powoduje opóźnienia sygnału wynikające z czasów propagacji układów logicznych. Poza tym sygnałów zegarowych nie można bramkować.

Do każdej makrokomórki doprowadzone są sygnały asynchronicznego ustawiania i kasowania. Jeżeli sygnał Set działa w sposób asynchronicznie, to pojawienie się wartości aktywnej na nim wymusza natychmiastową zmianę stanu Q:



W przypadku pracy synchronicznej zmiana stanu Q nastąpi przy najbliższym zboczu zegarowym przełączającym przerzutnik:



Nazwy „Set” i „Reset” są tutaj używane niewłaściwie, ponieważ według reguły sygnały „Set” i „Reset” działają w sposób synchroniczny, a asynchronicznie działają sygnały „Preset” i „Clear”.

Gdy przyjrzymy się układowi połączeń, możemy stwierdzić że wszystkie sygnały sterujące (CLK, CE, S, R) mogą pochodzić z różnych źródeł. Mogą to być albo sygnały globalne (GCK – globalny zegarowy oraz GSR) lub lokalne w postaci obliczonych termów (PTCK – zegar, PTS – set, PTR – reset). Sygnał CE wyznacza się wyłącznie lokalnie; jest on produktem termu PTCE.

Z makrokomórki wychodzą dwa sygnały: sygnał logiczny kierowany równocześnie do bloku we/wy oraz do globalnej matrycy połączeniowej oraz sygnał PTOE (Product Term OE) sterujący buforem 3-stanowym w bloku we/wy.

### Alokator termów

Zwiększa on elastyczność konfiguracji układu. Pozwala nam na decydowanie ile linii termów może być podłączonych do jednej bramki OR.

Przeważnie wykorzystywany jest do zwiększania ilości termów podłączonych do jednej bramki OR. Idea pracy

alokatora jest taka, że nieużywane termy z sąsiednich makrokomórek są w obrębie alokatora sumowane i eksportowane w dół lub w górę i dołączane w makrokomórce docelowej do bramki OR poprzez dodatkową bramkę OR. Sumowanie takie daje efekt akumulacji termów podłączonych do jednej bramki OR. Alokator termów może działać w trzech trybach: import termów, eksport termów „do góry” oraz eksport „w dół”.

Zajmiemy się schematem alokatora termów. Symbolami prostokątów są oznaczone multiplexery, które decydują o tym czy dany sygnał dołączany będzie do wyjścia górnego czy dolnego. 3-wejściowa bramka OR na górze jest bramką zbiorczą, która łączy eksportowane termy z danej makrokomórki. Suma eksportowanych termów jest wysyłana w górę lub w dół. Suma ta składa się z trzech składników: termów które doszły do danej makrokomórki z góry, termów które doszły z dołu oraz termów obliczonych w danej makrokomórce. Każdy z 5-ciu termów obliczonych w danej makrokomórce może być przy pomocy multiplexera podłączony do 5-wejściowej bramki OR.

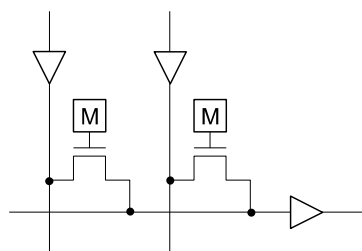
Importowanie termów może się odbywać z góry lub z dołu. Jeżeli importuje się jakieś termy, to nie dokłada się do nich termów pochodzących z danej makrokomórki. Termy importowane są sumowane ze sobą i wchodzi jako szóste wejście bramki OR (zaraz za szarą ramką).

Ze schematu widać, że sygnały sterujące są obliczane na poszczególnych termach. Term pierwszy może realizować funkcję Set lub Reset. Term drugi decyduje o polaryzacji sygnału wychodzącego z makrokomórki. Trzeci generuje sygnał PTC. Czwarty może realizować dwie funkcje: sygnału Reset lub Set. Ostatni term realizuje funkcję PTOE. Używanie termów jest ograniczone. Niemożliwa jest jednoczesna praca z pięcioma sygnałami sterującymi.

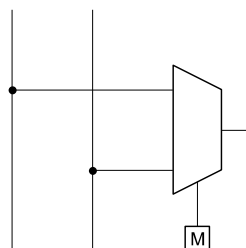
Importowanie i eksportowanie termów obarczone jest kosztami czasowymi. Opóźnienie termów importowanych  $t_{PTA}=0,7$  ns. Dodaje się go do parametru określającego czas propagacji sygnału  $t_{LOGI}=1,0$  ns. Term, który jest podłączony do właściwej makrokomórki wnosi opóźnienie 1 ns. Każdy stopień importu to dodatkowe 0,7 ns (na każdy stopień alokatora). Przejście przez dwa stopnie alokatora daje opóźnienie 1,4 ns. Maksymalny stopień opóźnienia dodatkowego opóźnienia wynosi  $8 \times t_{PTA}$ .

### Globalna matryca połączeń

Jest to niezbędny układ umożliwiający wzajemną komunikację poszczególnych bloków ze sobą. Wszystkie sygnały wejściowe i sprzężenia zwrotne przechodzą przez tą matrycę. Matryca ta musi działać szybko, by nie było mowy o wprowadzaniu dodatkowych opóźnień sygnałów. Pierwsze matryce połączeniowe zbudowane były w oparciu o punkty programowalne. Sygnały wejściowe mogły się znaleźć na liniach wyjściowych gdy punkty programowalne otwierały odpowiednie tranzystory:



To było wolne rozwiązanie, ponieważ miała na to wpływ rezystancja kanałowa tranzystora oraz pojemność paraszytnicza, które razem wprowadzały opóźnienie. Lepszym rozwiązaniem jest zastosowanie matrycy Fastconnect:

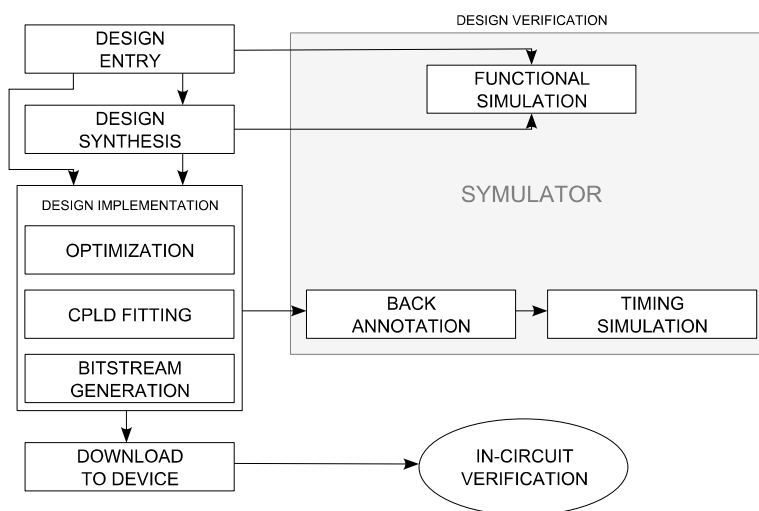


Działanie oparte jest tu na multiplexerach. Tutaj jeden punkt programowalny decydujący o tym do którego z wejść jest podłączone wyjście matrycy. Multiplexery w technologii CMOS są łatwe do implementacji. Rozwiązanie jest szybsze.



## Praca w środowisku Xilinx ISE

Poruszanie się w środowisku projektowania układów logicznych jest sporą sztuką z racji na złożony proces projektowy. Projektant układu cyfrowego powinien wiedzieć z jakimi etapami implementacji będzie miał do czynienia i co w poszczególnych etapach będzie się działo.



Zaczynamy od wprowadzenia opisu projektu (Design Entry). Można to porównać do pisania kodu programu w jakimś edytorze. Specyfikowanie projektu może się odbywać poprzez rysowanie schematu lub używanie języka opisu sprzętu. Kolejnym krokiem jest synteza projektu (Design Synthesis). W szczególności opis projektu za pomocą języków opisu sprzętu wymaga syntezy. Tekst opisu VHDL należy przełożyć (zsyntezować) na kategorię układów cyfrowych. W niektórych przypadkach można krok syntezy pominąć np. gdy opisujemy schemat wykorzystując elementy prymitywne (elementy, które nie wymagają syntezy, bo są w danej architekturze elementami podstawowymi).

Po syntezie przechodzimy do implementacji (Design Implementation) mającej na celu przygotowanie danych konfiguracyjnych, które zostaną przekazane do sprzętu i według tego strumienia będzie konfigurowany układ. W implementacji mamy krok optymalizacji (Optimalization) mający na celu uproszczenie wprowadzonego przez nas opisu. W układach CPLD jest jeszcze krok „Fitting” który polega na „wstawieniu” projektu do danej architektury układów CPLD. Na końcu mamy krok generacji strumienia bitowego, który posłuży do konfiguracji układu (Bitstream Generation). Powstaje gotowy strumień, który należy przesłać do urządzenia (Download to device).

Równolegle do tej ścieżki biegnie ścieżka symulacyjna. Mamy do czynienia z różnymi rodzajami symulacji. Symulacja funkcjonalna (Functional Simulation) przeprowadzana jest przed rozpoczęciem procesu implementacji projektu. Dokonywana jest symulacja naszego opisu układu przed lub po przeprowadzeniu procesu syntezy projektu. Symulacja ta nie uwzględnia zależności czasowych. Zależności czasowe stają się znane dopiero po przełożeniu projektu na daną architekturę układów. Wzbogacenie projektu o opis zależności czasowych oznaczony jest jako „Back Annotation”. Jest to uzupełnienie opisu funkcjonalnego o parametry czasowe, które są dostępne po implementacji fizycznej układu. Wtedy możemy przejść do procesu symulacji czasowej (Timing Simulation) która pokazuje, czy wszystkie zależności czasowe są spełnione. Po przesłaniu projektu do urządzenia możemy korzystać z różnych mechanizmów debugowania projektu pracującego w układzie (In-Circuit Verification). Potrzebne do tego jest użycie specjalnych próbników logicznych umożliwiających śledzenie wykonywania się projektu w praktyce.

W środowisku ISE całością zarządza Project Manager, czyli powłoka z interfejsem użytkownika, która uruchamia kolejne etapy syntezy i implementacji. Sprawia on najwięcej kłopotów, ponieważ nie jest to powłoka pewnie działająca. Oprogramowanie wykonujące syntezę to moduły które na podstawie plików wejściowych, zawierających schematy układów lub opis sprzętu generują pliki wyjściowe, które będą służyć do konfiguracji układu.

## Praca w środowisku ISE c. d.

### a) Schemat logiczny

Tworzenie schematu logicznego projektu skutkuje wygenerowaniem pliku z rozszerzeniem .SCH. Do tworzenia schematów używa się edytora ECS.

Symbole, z których możemy korzystać podczas opracowywania projektów, są opisane w „Libraries Guide”. Elementy dzieli się na:

- **PRIMITIVES** – elementy podstawowe. Są to takie elementy, których nie można rozłożyć na czynniki pierwsze. Z takich elementów składają się elementy bardziej złożone, nazywane makrami.
- **SOFT MACROS** – makra ogólne, składające się z elementów podstawowych. Takim przykładem może być przerzutnik typu JK, który składa się z przerzutnika D oraz towarzyszących mu bramek logicznych.
- **RELATIONARY MACROS** – są to makra zawierające informacje o rozmieszczeniu elementów składowych. Jest to ważne w układach FPGA gdzie mamy do czynienia z dwuwymiarową matrycą i tam ważne jest odpowiednie rozmieszczenie względem siebie elementów. Makra z informacją o rozmieszczeniu nazywane są „Relationary Placed Macros”. Przykładem mogą być sumatory kaskadowe, składające się z łańcucha pojedynczych sumatorów. W układach CPLD informacje o rozmieszczeniu nie są potrzebne.

Na schemacie logicznym umieszcza się porty we/wy („I/O Markers”). Jeżeli dany schemat reprezentuje element szczytowy projektu, to porty muszą być przypisane do konkretnych wyprowadzeń układu. W przypadku hierarchicznej struktury projektu (gdzie dany schemat jest tylko podschematem) porty we/wy nie muszą być opisywane.

### b) Opis w języku HDL (Hardware Description Language)

Obecnie używa się dwóch języków opisu sprzętu: Verilog i VHDL. My będziemy zajmować się tylko językiem VHDL, który był tworzony przez kilka firm naraz. Miał służyć do opisu układów wielkiej skali integracji. Język VHDL ma więcej możliwości opisu i przez to jest trudniejszy do opanowania.

Opisu można dokonywać przy pomocy zwykłego edytora tekstowego z zachowaniem odpowiedniej składni. Pliki z kodem mają rozszerzenie .VHD.

Dostępne są gotowe szablony, które należy wykorzystywać podczas opisywania układów w języku VHDL z przeznaczeniem do syntezy. Ścieżka dostępu to VHDL → Synthesis Constructs → Coding Examples. Szablony podają sposoby opisywania rejestrów, transkoderów, multiplekserów, liczników... Korzystanie z szablonów jest wskazane dla powodzenia procesu syntezy.

Opisy sprzętu w postaci rysowania schematów i kodów języka VHDL nie są rozdzielne. Można je mieszać i robi się to. Dla danego modułu utworzonego w języku VHDL można utworzyć symbol, pozwalający na umieszczenie modułu na schemacie.

### c) Maszyny stanów (Finite State Machines FSM)

Przeznaczone do opisu maszyn stanów, które mają realizować algorytm sekwencyjny, opisany przy pomocy grafu stanów.

W pakiecie istnieje edytor „StateCAD” służący do rysowania grafu automatu. Pliki generowane przez edytor mają rozszerzenie .DIA.

Opis zawarty w pliku .DIA jest automatycznie konwertowany na opis maszyny w języku VHDL. Później można opracować symbol maszyny, który można umieścić na schemacie logicznym układu.

Istotną rzeczą jest opis wyprowadzeń układu. Zależą one od obudowy w jakiej zamknięto układ. Wykorzystuje się tutaj plik z ograniczeniami projektowymi (User Constraints File) z rozszerzeniem .UCF. Trzeba go dodawać do każdego projektu. Definiuje się w nim przypisanie portów we/wy do wyprowadzeń obudowy:

```
NET „NazwaPortu” LOC = „Pn”;
```

„NazwaPortu” opisuje port we/wy występujący na schemacie szczytowym projektu. Format numeru wyprowadzenia „Pn” oznacza zapis „P20” lub „P8”. Numer wyprowadzenia zależy od typu obudowy. Jeżeli będziemy pracować z obudową PLCC, to będziemy używać oznaczenia „P05”. Jeżeli trafilibyśmy na bardziej skomplikowaną obudowę (BGA – wyprowadzenia rozmieszczone pod układem w postaci matrycy 2-wymiarowej) to wy-

przewodzenia opisuje się podobnie jak na planszy szachowej: „A0”, „B15”. Numer wyprowadzenia musi być brany ze schematu układu zamkniętego w konkretnej obudowie. Na laboratorium pracujemy ze scalakiem zamkniętym w obudowie PLCC-44. Jeżeli zabraknie pliku .UCF, nie zostanie wygenerowane żadne ostrzeżenie. Porty zostaną przypisane automatycznie przez narzędzie na zasadzie „jak leci”.

#### d) Symulacja

Praca będzie się odbywała z symulatorem „ModelSIM”. Jest to oprogramowanie zewnętrzne, ale integruje się z pakietem ISE. Możemy przeprowadzać dwa rodzaje symulacji układów:

- **FUNKCJONALNA (Behavioral)** – jest prostą symulacją zero-jedynkową. Nie są uwzględniane żadne opóźnienia sygnałów. Jest prostsza do analizy i szybciej wykonywana przez program. Symulacja czasowa może iść w godziny przy bardzo złożonych projektach. Symulacja funkcjonalna pracuje dokładnie na stworzonym przez nas opisie zanim został przeprowadzony proces implementacji. Umożliwia nam ona śledzenie sygnałów wewnętrznych projektu. Definiujemy pobudzenia dla portów wejściowych i zostają obliczone odpowiedzi dla portów wyjściowych.
- **CZASOWA (Post-Fit)** – Tu również można śledzić sygnały wewnętrzne, lecz są one już zsyntezowane przez narzędzie i mogą być one inne, niż przez nas ustalone, bo sygnały mogły zostać zoptymalizowane podczas procesu sklejania logiki.

Każda symulacja musi się rozpocząć od zdefiniowania pobudzeń czasowych na portach wejściowych. Definicje przechowywane są w pliku „Testbench” który może być tworzony edytorem graficznym (kształtowanie fal prostokątnych poprzez klikanie myszą) generującym pliki .TBW. Pobudzenia mogą być przechowywane w pliku VHDL, gdzie musimy definiować procesy podające pobudzenia na wejścia naszego projektu.

#### e) Konfiguracja układu

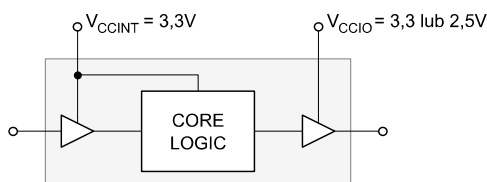
Jest to ostatni krok polegający na przesłaniu zaimplementowanego projektu do układu. Plik z danymi konfiguracyjnymi dla układów CPLD ma rozszerzenie .JED. Pliki konfiguracyjne dla scalaków FPGA mają rozszerzenie .BIT. Do przesyłania używa się programu „Impact” oraz kabla programującego dołączonego do portu LPT w komputerze. Dane konfiguracyjne są po stronie układu dostarczane przez interfejs JTAG.

*[Wracamy do omawiania architektury układów 9500]*

#### Bloki WE/WY

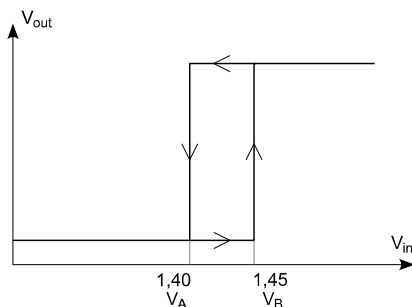
Każdy z bloków jest skojarzony z jedną makrokomórką by mógł wyprowadzać sygnał logiczny obliczony w danej makrokomórce. Sygnał wejściowy najpierw trafia do bloku we/wy, później do globalnej matrycy połączeń by w końcu trafić do makrokomórki.

Wspomnieć jeszcze należy o sposobie zasilania układów 9500XL, które jest podwójne:

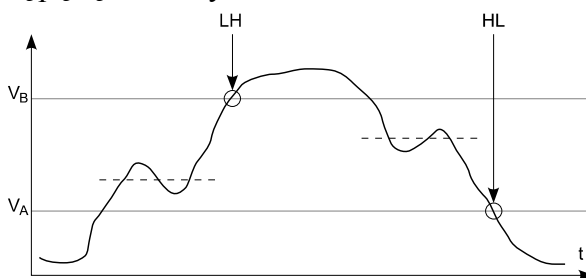


Logika wewnętrzna oraz bufor wejściowe są zasilane napięciem  $V_{CCINT}$  oraz bufor wyjściowe zasilane są napięciem  $V_{CCIO}$ .  $V_{CCINT}$  musi być zawsze równe 3,3V. Jeżeli chodzi o  $V_{CCIO}$ , to możemy podawać napięcie 3,3V lub 2,5V. Niższe napięcie przydaje się wówczas, gdy na wyjście układu 9500 podłączy się niskonapięciowe układy zasilane napięciem 2,5V.

Jeżeli dany pin pracuje jako wejście, to automatycznie zostaje wyłączona ścieżka wyjściowa w danym bloku. Sygnał przechodzi przez bufor wejściowy i trafia do matrycy Fastconnect. Napięcie 3,3V zasilające bufor wejściowy wynika ze stałego progu przełączania takiego bufora. Konstrukcja bufora jest tak pomyślana, by występowała tam mała pętla histerezy:



Napięcia określające zakres histerezy są ustalone na poziomie 1,40 oraz 1,45V. Gdy napięcie na pinie rośnie, to by na wyjściu pojawiła się „jedyńska” napięcie musi przekroczyć próg 1,45V. Gdy napięcie będzie spadało, „zero” logiczne pojawi się dopiero po przekroczeniu progu 1,40V. Takie formowanie sygnałów logicznych jest potrzebne. Ma to uodpornić piny wejściowe na fluktuacje napięcia wejściowego. W technologii TTL stosowało się przerzutniki Schmitta z szeroką pętlą histerezy.



Pętla przydaje się gdy sygnał wejściowy zmienia się w czasie. Szeroka pętla histerezy sprawi że odfiltrowane są wahania sygnału wejściowego; zapobiega ona wielokrotnym przełączaniom między zerem a jedynką w przypadku zboczy narastających wolno i z zakłóceniami. Gdybyśmy używali bramki z jednym progiem dyskryminacji (poziome linie przerywane), to każde przejście sygnału wejściowego przez próg powodowałoby przełączanie się bramki.

Ważną własnością jest tolerowanie napięć wejściowych do wartości 5V (zakres TTL). Sam układ pracuje z napięciem 3,3V. Podanie na wejście układu CMOS napięcia wyższego, niż napięcie zasilające powodowało spalanie układu. Poza tym konieczne było zastosowanie specjalnych procedur podłączania napięć zasilających. Układy 9500 są odporne na zjawisko „Power Sequencing” (kolejność podawania napięć zasilających). Układy CMOS bez zabezpieczeń wymagają ostrożnego podawania napięcia zasilającego.

Po stronie wejściowej istnieje zgodność z różnymi rodzinami układów. Najwyższe napięcia są generowane przez układy TTL i 5V układu CMOS. Można dołączać układy CMOS zasilane napięciami 3,3V oraz 2,5V. Z tego powodu wzięto się napięcie progowe na poziomie 1,45V.

### Bufor wyjściowy

Zasilanie bufora napięciem na poziomie 2,5V podyktowane jest sterowaniem przez bufor układów CMOS zasilanych napięciem 2,5V, które nie tolerują wysokich napięć na swoich wejściach. Układy 9500 stosuje się jako interfejs logiki niskonapięciowej. Jeżeli bufor będzie zasilany napięciem o wartości 3,3V, to mamy zapewnioną jego zgodność z układami TTL (i CMOS zasilanych napięciem 5V). Jeżeli  $V_{CCIO} = 2,5V$ , to pojawią się kłopoty ze współpracą z układami TTL i CMOS'ami zasilanymi 5V. Oczywiście zgodność z układami 2,5V jest zachowana.

Bufor wyjściowy oferuje opcję obniżenia szybkości narastania zboczy sygnału (Slew rate). Regulacja szybkości jest tu dwustopniowa. Standardowo wyjście działa z szybkim narastaniem (opadaniem) zboczy. Zmiany prędkości narastania zboczy dokonać można przez użycie atrybutu SLEW w opisie portu wyjściowego. W pliku UCF będzie to wyglądało tak:

```
NET „Cośtam” LOC = „Coś” | FAST{SLOW};
```

Obniżanie szybkości przełączania stosuje się celem odfiltrowania impulsów prądowych powstających podczas jednoczesnego przełączania wielu wyjść naraz. Problem ten daje się we znaki w systemach magistralowych w czasie podawania adresu. Każde zbocze (przełączanie się bufora) generuje impuls prądowy. Dla pojedynczego

bufora przy szybkim zboczu może wystąpić impuls prądowy o wartości nawet 10mA. Jeżeli mamy kilkaset buforów i kilkadziesiąt będzie się przełączało naraz, to impuls prądowy jest już rzędu amperów. Poza tym musimy mieć zasilacze, które dadzą sobie radę z dostarczaniem odpowiedniej ilości energii oraz dobrze poprowadzone wyprowadzenia GND, ponieważ impuls prądowy może wywołać skok wartości potencjału masy, co może spowodować zakłócenia w pracy układu. Lepsze jest zatem wydłużenie czasu narastania zboczy by zmniejszyć impulsy prądowe. W układach FPGA regulacja prędkości narastania zbocza ma większą możliwość konfiguracji.

W bloku wyjściowym mamy różne źródła sygnału Output Enable. Może być on pobierany jako term z makrokomórki (PTOE), może to być również jeden z sygnałów GTS lub nawet wartość stała (0 lub 1).

Istnieje tu coś takiego jak User Programmable Ground, układ sprawiający że nieużywany pin może pełnić rolę dodatkowego wyprowadzenia masy GND. Przydaje się to wówczas gdy mamy wiele przełączających się wyjść i dodatkowa masa może tłumić w pewnym stopniu impulsy.

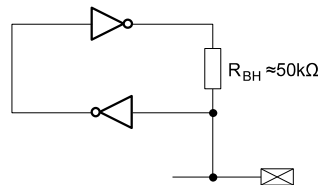
Bus Hold („Pin Keeper” w środowisku ISE). Jest to coś na kształt asynchronicznego przerzutnika, który ma za zadanie podtrzymywanie stanu sygnału na wyjściu.

Istnieje jeszcze układ Pull-Up (oparty na rezystorze  $R_{BH} \approx 50k\Omega$ ) który sprawia, że pin jest podłączony przez rezystor do jedynki logicznej. Ma to zapobiec samowolnemu przełączaniu się układu (nie sprzęga się z otaczającą logiką, nie łapie zakłóceń).

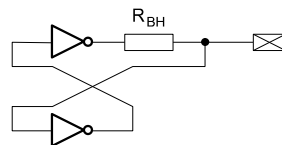
## Bloki WE/WY

Na poprzednim wykładzie układ Bus Hold („Pin Keeper”) był tylko wspomniany. Dzisiaj nadszedł moment, by przyjrzeć mu się bliżej. Przełącznik na schemacie tego układu wskazuje na dwa stany jego pracy. Jeżeli układ XC 9500XL jest skasowany, odbywa się proces programowania, odbywa się komunikacja przez interfejs JTAG lub jest inicjalizowane napięcie zasilania, to przełącznik jest w pozycji „0”. Wszystkie wyjścia są „wyciszone” rezystorami Pull-Up (jest uzyskiwana „słaba jedynka” logiczna na pinach). Podczas procesu włączania układu „słaba jedynka” jest utrzymywana na wyprowadzeniach do chwili, aż wartość napięcia zasilającego  $V_{CCINT}$  przekroczy poziom 2,5V.

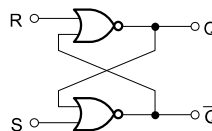
Jeżeli układ scalony pracuje normalnie, to przełącznik znajduje się w dolnej pozycji (zanegowany „PIN”). Wygląda to następująco:



Układ taki powinniśmy kojarzyć z asynchronicznym przerzutnikiem. Jest tu pętla sprzężenia zwrotnego identyczna z pętlami spotykanymi w przerzutnikach:



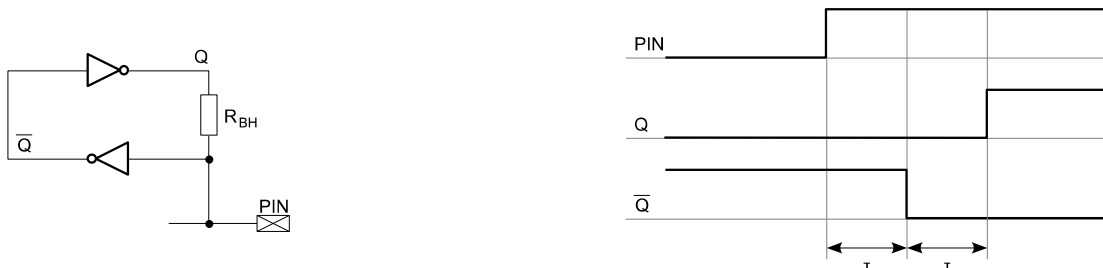
W tej postaci układ jest podobny do asynchronicznego przerzutnika RS. Normalny RS jest wykonany na dwuwejściowych bramkach:



W układzie Bus Hold nie mamy wejść sterujących R i S. Jest tylko pętla podtrzymująca stan przerzutnika. Może on przebywać w jednym z dwóch stanów bistabilnych. Przerzutnik wygląda „partyzancko”, bo nie ma wejść sterujących. Jak się to ustrojstwo przełącza? Dzięki umieszczeniu w nim rezystora. By prześledzić jego działanie, musimy na ten „przerzutnik” spojrzeć z „analogowej strony”.

Umieszczenie rezystora zapewnia przełączanie się przerzutnika od sygnału wyjściowego. Stan logiczny na pinie przełącza przerzutnik. Podtrzymuje on stan logiczny na linii zanim ta przejdzie w stan wysokiej impedancji. Stanie się to po zdjęciu sterownika, który steruje pinem. Układ wstawiono po to, by linia pracująca w systemie magistralowym w momencie przejścia nadajników w stany wysokiej impedancji zachowała swój stan i nie „wisiała w powietrzu”.

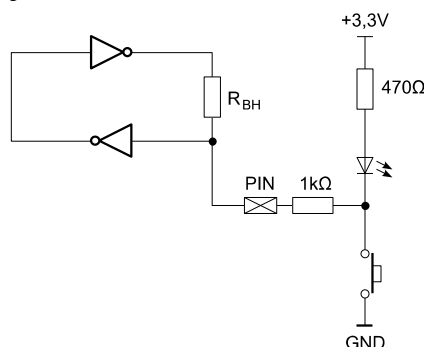
Przełączanie układu Bus Hold będziemy rozpatrywać na przykładzie podania zbocza narastającego na pin:



Zakładamy że przed przełączeniem stan  $Q$  był równy 0.  $\bar{Q}$  wynosi zatem 1. Przyjmujemy, że każda z bramek wnosi opóźnienie  $\tau$ . Gdy na  $PIN$  pojawi się „1”, to po opóźnieniu  $\tau$  zmieni się stan  $\bar{Q}$  (z wysokiego na niski). Gdy  $\bar{Q}$  zmienił swój stan, to stan  $Q$  zmieni się z 0 na 1 po czasie  $\tau$ . Gdy  $Q$  zmienił swój stan, przerzutnik znaj-

dzie się w stanie stabilnym. Zająłoby to odcinek czasu równy  $2\tau$ . Bez rezystora  $R_{BH}$  mielibyśmy konflikt w postaci zwarcia dwóch sterowników (nadajników). Rezystor  $R_{BH}$  zapewnia nam to, że może występować jakaś różnica napięć między węzłami, ale prąd przepływający między nimi będzie ograniczony. Gdy zasilamy układ napięciem 3,3V to przez rezystor może przepłynąć prąd o wartości 66μA. Jego wartość nie zaszkodzi układowi.

UWAGA: układy Bus Hold należy wyłączać!



Rezystor 1kΩ zabezpiecza przed bezpośrednim zwarciem wyprowadzenia do masy. Zwarcie wyprowadzenia z masą poskutkowałoby uszkodzeniem makrokomórki.

Kłopoty pojawiają się gdy wyprowadzenie będziemy wykorzystywali jako wejście. Naciśnięcie przycisku zestawu laboratoryjnego ma wymuszać „zero”. Naciśnięcie przycisku powoduje pojawienie się na pinie napięcia o wartości  $\frac{1}{51} \cdot 3,3 V$ . Jeżeli będziemy chcieli podać „jedynekę” na pin, to przełączenie nie wystąpi. Dzieje się tak przez obecność diody LED. Spadek napięcia na tej diodzie jest na poziomie około 2V. Gdy puścimy przycisk, to przełączenie do „jedynki” nie nastąpi z powodu spadku napięcia na diodzie i dzielniku napięcia. Gdyby diody LED nie było, to zwolnienie przycisku łączyłoby pin z „jedyneką” logiczną poprzez rezystancję 1,5kΩ (dokładniej  $\frac{50}{51,5} \cdot 3,3 V$ ). Daje to „dobrą” jedynkę. Dioda LED i spadek napięcia na niej sprawia, że potencjał na pinie ustali się na poziomie zakresu zabronionego i o przełączeniu się układu nie ma mowy. Zwarcie diody LED sprawiłoby, że układ Bus Hold od razu przełączyłby się do jedynki.

Wyłączanie układu Bus Hold jest ważne, ponieważ załączony układ zwyczajnie będzie przeszkadzał w pracy. Podawanie zera na pin będzie zachodziło bezproblemowo, natomiast będą kłopoty z podawaniem jedynki na wyprowadzenie układu. Wyłączania Bus Holda dokonuje się poprzez wejście w opcje implementacji układu (w środowisku ISE) i ustawieniu „Float” dla właściwości „I/O Pins”.

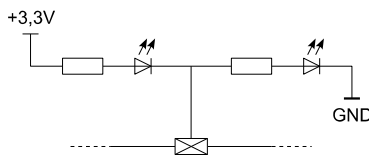
### Schemat logiczny zestawu laboratoryjnego

Umieszczono w nim jeden układ logiczny (9536XL) i 3 generatory sygnałów zegarowych (Rodzina 9500 posiada 3 wejścia zegarowe globalne i każdemu wejściu odpowiada jeden generator). Częstotliwość przebiegu uzyskiwanego z generatorów można zmieniać poprzez przełączanie kondensatorów przyłączonych do wyprowadzenia TRIGGER a masą układu 555. Przełączania dokonuje się przy użyciu dip switcha. Wyjście generatora steruje diodą LED, co ułatwia obserwację sygnału przy jego małej częstotliwości. Sygnał z generatorów nie jest podpięty na stałe do wyprowadzeń układu XC 9536XL. Dołączania dokonuje się przy pomocy zworek. Umieszczenie zworek umożliwia alternatywne podawanie sygnału zegarowego „z guzika”. Takie rozwiązanie nie jest zalecane, ponieważ w przyciskach istnieje zjawisko drgania styków. Jedno naciśnięcie przycisku może wygenerować szereg impulsów, które kilkakrotnie przełączą układ sekwencyjny sterowany ręcznie wygenerowanym sygnałem zegarowym.

W zestawie umieszczono dwie grupy układów we/wy, które mogą działać dwukierunkowo (przyciski i diody LED). Nie wszystkie wyprowadzenia układu 9536 są używane. Ten fragment schematu jest istotny, ponieważ na jego podstawie będzie trzeba zdefiniować w pliku UCF przypisania sygnałów do fizycznych wyprowadzeń układu. Na płytce przy diodach i przyciskach są wytrawione numery, lecz są to numery kolejnych makrokomórek w bloku funkcyjnym a nie wyprowadzeń układu.

Pamiętać trzeba, że naciśnięcie przycisku powoduje podane nie wyprowadzenie układu 9536 niskiego stanu logicznego. Diody będą zapalać się od podania stanu niskiego na wyprowadzenie. Wynika to z przesłanek histo-

rycznych. Diodę LED możemy podłączyć do wyprowadzenia układu cyfrowego na dwa sposoby:



Dioda po lewej stronie będzie się zapalać przy podaniu zera logicznego na pinie. Dioda prawa zapali się od podania jedynki na pinie. Czemu łączy się diody tak, by zapalały się od stanu niskiego na pinie? Ponieważ dioda zapalana „jedynką” jest zasilana wprost z układu scalonego (napiecie pobierane z  $V_{CCINT}$ ). Jeżeli mamy do zasilania kilkanaście diod naraz, to źródło  $V_{CCINT}$  będzie obciążone, a to może spowodować niepożądane wahania napięcia. Lepsze jest rozwiązanie z diodą zasilaną z zewnętrznego źródła. Nie musimy martwić się o stabilność napięcia  $V_{CCINT}$ .

Na płytce znajdują się dodatkowe 4 wyprowadzenia portu JTAG (komunikacja podczas programowania i testowania układu programowalnego). Nie mogą one być współdzielone ze „zwykłymi” wyprowadzeniami we/wy, ponieważ komunikacja przez JTAG odbywać się musi w sposób nadrzędny. Musi być ona dostępna podczas pracy układu.

### Inne cechy układu XC 9500 XL

Wspomnieć należy o możliwości konfiguracji makrokomórek do pracy w trybie obniżonego poboru mocy. Jest to konfiguracja indywidualna dla każdej makrokomórki. Wykorzystuje się atrybut:

`PWR_MODE = LOW|STD`

Atrybut należy dołączyć do przerzutnika, który umieścimy na schemacie. Przerzutnik zostanie umieszczony w którejś z makrokomórek i będzie ona pracowała w trybie obniżonego poboru mocy. Jeżeli będziemy wykorzystywać makrokomórkę do pracy kombinacyjnej, to atrybut „PWR\_MODE” dołączamy do bramki logicznej. Obniżenie poboru mocy spowalnia pracę układu. Gdy normalnie  $t_{LOGI}=1ns$ , to dla obniżonego poboru mocy czas zwiększa się do  $t_{LOGILP}=5ns$ . Oszacowanie zysku jest trudne, ale można w przybliżeniu określić, że wartość prądu pobieranego przez makrokomórkę pracującą w trybie obniżonego poboru mocy stanowi 50% prądu pobieranego podczas normalnej pracy. Rozwiązanie to stosuje się, gdy nie zależy nam na szybkości pracy układu. Częstotliwość pracy układu może być obniżona z 170MHz do 50MHz.

Inną ciekawostką jest ochrona konfiguracji układu. Jest to istotne zagadnienie w układach programowalnych. Przydaje się to wówczas, gdy chcemy chronić nasz projekt przed jego skopiowaniem. Układy z rodziny XC 9500XL mają pamięć konfiguracji opartą o technikę Flash. W układach FPGA jest problem, bo pamięć konfiguracji jest oparta o pamięć RAM. Strumień konfiguracyjny jest pobierany z zewnętrznej pamięci, a to umożliwia podsłuchiwanie strumienia danych. W układach FPGA zaszyty jest klucz, przy pomocy którego jest szyfrowana zawartość pliku konfiguracyjnego. Utrudnia to konfigurację układów, bo trzeba szyfrować pliki konfiguracyjne osobno dla każdego układu.

W układach PLD mamy dwa bity sterujące zabezpieczeniami. Bit „Read Security” zabezpiecza przed odczytem. Drugi bit („Write Security”) zabezpiecza przed omyłkowym nadpisaniem konfiguracji układu. Ustawienia tych bitów dokonuje się dzięki aplikacji programującej „Impact”. Ustawienie bitu „Write Security” nie oznacza, że układ jest stracony i nie będzie można go zaprogramować. Bit ten można wyzerować poprzez przesłanie odpowiedniego polecenia interfejsem JTAG. Oba bity są skutecznymi zabezpieczeniami na poziomie sprzętowym.

## Język VHDL

Język ten należy do kategorii języków opisu sprzętu (Hardware Description Language). Historia języka sięga roku 1980. Pojawiła się wtedy potrzeba stworzenia profesjonalnego języka opisu sprzętu. Pojawiły się wtedy języki do opisu układów PAL (Palasm, inne zależne od producentów sprzętu). Ministerstwo obrony USA zaczęło finansować wieloletni projekt związany z projektowaniem układów cyfrowych wielkiej skali integracji i pracujących z dużymi prędkościami VHSIC (Very High Speed Integrated Circuits). Celem projektu było zagospodarowanie możliwości, które stały się realne dzięki rozwojowi technologii.

Technologia osiągnęła taki poziom, że praktycznie każdy mógł skonfigurować sobie programowalny układ cyfrowy. Potrzebny był język, który umożliwiłby efektywną pracę kilku grupom ludzi nad projektem jakiegoś układu. Język musiał być bardziej zaawansowany niż Palasm czy CUPL. W ten sposób powstał VHDL. Prace

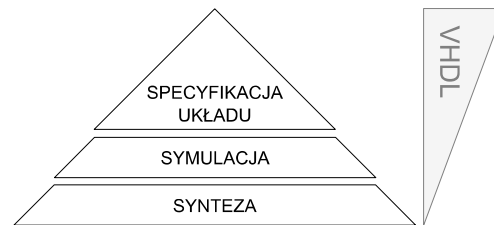


nad językiem były zakrojone na szeroką skalę, co sprawiło że konieczne było opracowanie standardu opisującego ten język. VHDL doczekał się opracowania IEEE 1076.

Język bazował na koncepcjach zaczerpniętych z języka ADA. Język ADA był również opracowany na zlecenie ministerstwa. ADA opierał się na koncepcjach, uważanych w tamtych czasach za nowatorskie, pozwalając na opisywanie procesów wykonywanych współbieżnie, a to jest ważne w językach opisu sprzętu.

Kolejne standardy opisujące język VHDL pojawiały się w latach 1993 oraz 2000. W 1993 ustanowiono finalną wersję składni języka. Rok 2000 przyniósł ze sobą drobne poprawki.

VHDL można wykorzystywać do różnych rzeczy:



Pierwszym zastosowaniem języka był opis specyfikacji układu. Trzeba było określić jakie są sygnały wejściowe i wyjściowe, z jakich modułów będzie składał się układ i co te moduły będą robić.

Kolejnym zastosowaniem była symulacja. Jeżeli mamy opis działania jakiegoś układu, to chcielibyśmy dokonać symulacji układu. Symulacja w pełni automatyczna była nieosiągalna do przełomu lat 80-tych i 90-tych, kiedy to powstały pierwsze softwareowe symulatory.

Obecnie nas i projektantów najbardziej interesuje synteza. Opisujemy jakiś układ po to, by uruchomić automatyczne narzędzie, które wygeneruje nam plik konfiguracyjny. Plik przesyłamy do układu programowalnego, który działa zgodnie z opisem.

Należy sobie zdawać sprawę z tego, że VHDL nie nadaje się „do wszystkiego”. Język ten został stworzony by umożliwić dokładny opis specyfikacji układu i potencjalnej symulacji. Łatwo jest napisać coś, co będzie składniowo poprawne, będzie się kiepsko symulowało i w ogóle nie będzie się syntezyowało. Sama znajomość języka nie wystarcza. Trzeba wiedzieć jak języka używać by opis się dobrze syntezyował lub symulował. Z tej to okazji skrót VHDL rozwijany był jako Very Hard Description Language. Pewnych rzeczy nie powinno się pisać w VHDL'u mimo że składniowo są one poprawne.

W pierwszej połowie lat 90-tych pojawiły się pierwsze automatyczne narzędzia syntezy. Rozróżniało się wtedy kod syntezywalny i niesyntezywalny. Każdy producent narzędzia syntezy specyfikował jakie kategorie opisu są dozwolone, a jakie nie są. Dzisiaj każdy producent chce być „jak najbardziej postępowy” wobec czego dziś jest coraz mniej konstrukcji niesyntezywalnych. Narzędzie będzie usiłowało zsyntezyować nawet najgłupszy zapis i uda mu się to. Efekt takiej syntezy będzie koszmarny pod względem zajętości zasobów i własności czasowych. Dzisiaj jest jeszcze gorzej niż dawniej, ponieważ jakość syntezywanego projektu może być nieciekawa.

## Język VHDL

### Jednostki i architektury

Poprzednio wspomnieliśmy o tym, że język VHDL pochodzi z początku lat 80-tych. Stworzony był by opisywać specyfikację układów cyfrowych, po części również symulację. Nie był tworzony z myślą o syntezie, do której będzie wykorzystywany na laboratorium i o której będziemy w głównej mierze mówić. Język VHDL pozwala na opis specyfikacji rozbudowanych projektów cyfrowych, które mogą być podzielone na moduły i z tego powodu podział projektu na elementy, czy też mniejsze jednostki jest podstawą. Wszystko co opisuje się w VHDL jest opisywane w postaci jednostki, z którą kojarzy się później kilka architektur.

```
entity HalfAdder is
  port ( A : in  BIT;
        B : in  BIT;
        S : out  BIT;
        C : out  BIT);
end entity HalfAdder;
```

Opis jednostki (entity) rozpoczyna się od listy portów (sygnałów wejściowych i wyjściowych lub interfejsu jednostki). Zaczynamy od słowa kluczowego „entity”, później wpisujemy identyfikator jednostki (HalfAdder), następnie słowo kluczowe „is”. Opis portów rozpoczyna się od słowa „port” po którym pojawia się lista opisanych portów. Każdy port ma unikalną nazwę, po dwukropku opisuje się tryb pracy portu. Porty przenoszą sygnały określonego typu. Tu jest użyty „BIT” (wbudowany w pakiecie standardowym). Tego typu nie używa się w opisach do syntezy, bo nie nadaje się on do opisywania sygnałów w układach rzeczywistych.

Definicji jednostki używa się tylko jeden raz. Do definicji jednostki możemy przypisać szereg architektur. Przydaje się to podczas specyfikacji i symulacji układów, natomiast nie nadaje się do syntezy. Jedną jednostkę można opisać na kilka sposobów.

Istnieją 3 poziomy opisy architektur. Będziemy omawiali poziomy w kierunku od najniższego do najwyższego.

#### Opis strukturalny

Najniżej w hierarchii znajduje się opis strukturalny, w którym opisujemy strukturę danego modułu, z jakich elementów i podzespołów się składa. Jest to po prostu lista instancji elementów niższego poziomu. Elementy pochodzą z biblioteki:

```
library UNISIM;
use UNISIM.VComponents.all;

. . .

architecture Structural of HalfAdder is
begin
  XOR_gate : XOR2 port map ( A, B, S );
  AND_gate : AND2 port map ( A, B, C );
end architecture Structural;
```

Opis architektury rozpoczynamy od słowa kluczowego „architecture”, następnie wpisujemy nazwę architektury i nazwę jednostki, której opis będzie dotyczył. Między „begin” a „end architecture” podajemy właściwy opis wewnętrznej organizacji modułu. Półsumator złożony jest z bramek AND oraz XOR. Bramki te są wzięte z biblioteki, co sprowadza na nas konieczność zastosowania klauzuli użycia biblioteki. W opisywanym przypadku korzystamy z biblioteki „UNISIM”. Następnie umieszczana jest klauzula użycia komponentów z tej biblioteki. Zapis „VComponents.all” oznacza że będziemy używać wszystkich komponentów. Zamiast „all” można by wypisać konkretne elementy (stosuje się to rzadko):

```
use UNISIM.VComponents.XOR2;
use UNISIM.VComponents.AND2;
```

Klauzula użycia biblioteki kojarzyć się nam będzie z dyrektywą „#include” z języka C++, gdzie zadeklarowany plik nagłówkowy jest ważny od momentu jego zadeklarowania do końca pliku z kodem. W przypadku VHDL zadeklarowana biblioteka dotyczy tylko najbliższej jednostki. Jeżeli w pliku mamy zdefiniowanych kilka jednostek, to każdą jednostkę musimy poprzedzić listą bibliotek, z których chcemy korzystać. Zamiast bibliotek można umieszczać instancje jednostek zdefiniowanych przez nas wcześniej. Tak powstaje hierarchiczna struktura projektu.

Tworzenie instancji jakiegoś elementu ma składnię:

```
XOR_gate : XOR2 port map ( A, B, S );
```

Etykieta przed dwukropkiem jest obowiązkowa. Jest to sensowne, ponieważ w procesie symulacji będziemy mogli odwoływać się do wstawianych elementów. Następnie użyta jest nazwa komponentu wstawionego z biblioteki (może to być jakaś jednostka wcześniej przez nas opracowana). Ostatnim elementem jest przypisanie portów. Jednostka „XOR2” zdefiniowana w bibliotece ma w swojej definicji listę trzech portów. My odwzorowujemy porty jednostki „HalfAdder” na porty opisane w definicji jednostki „XOR2”.

Identyfikatory w VHDL mogą się składać z liter, cyfr i znaku „\_”. Nazwa musi rozpoczynać się od litery, znak „\_” może znajdować się wewnątrz identyfikatora (nie może on znajdować się na początku ani na końcu). Język VHDL nie rozróżnia wielkości liter. Komentarze obejmują tylko jedną linię począwszy od dwóch myślników. Nie ma tu komentarzy blokowych.

Rysowanie schematu generuje opis na poziomie strukturalnym. Każdy plik .SCH jest automatycznie konwertowany na poziom strukturalny, którego opis znajduje się w pliku .VHF. Zaglądając do tego pliku możemy się dowiedzieć w jaki sposób schemat układu został przełożony na opis w kodzie VHDL.

### Opis przepływowy

Wyższym poziomem opisu jest opis przepływowy (Data Flow). Tutaj opisujemy sposób przenoszenia wartości logicznych pomiędzy poszczególnymi portami:

```
architecture Dataflow of HalfAdder is
begin
    S <= A xor B;
    C <= A and B;
end architecture Dataflow;
```

Zamiast wstawiania komponentów bibliotecznych piszemy że sygnał C jest funkcją AND portów wejściowych A i B. Operatory „and” i „xor” są wbudowanymi operatorami, które działają na typie BIT. Nasze porty również są typu BIT.

Opisując działanie naszego modułu na poziomie przepływowym nie musimy się powoływać na konkretne elementy. Narzędzie syntezy ma większe pole manewru. Może ono zoptymalizować wpisane przez nas równania pod kątem złożoności jak i architektury na której będzie implementowany układ.

### Opis behawioralny

Inaczej nazywany jako opis proceduralny lub sekwencyjny. Opis taki nie ma nic wspólnego ze strukturą układu. Jest to opis najlepiej nadający się do symulacji. Tłumaczy on w jaki sposób komputer ma symulować pracę modułu. Nie ma żadnych danych o zastosowanych bramkach czy też przerzutnikach.

Wszystkie instrukcje, które są podane w architekturze, są wykonywane współbieżnie. Gdy chcemy cokolwiek opisywać sekwencyjnie, musimy użyć instrukcji procesu.

```
architecture Behavioral of HalfAdder is
begin
    process( A, B )
    begin
        -- Sum
        if A /= B then
            S <= '1';
        else
            S <= '0';
        end if;
        -- Carry
        if A = '1' and B = '1' then
            C <= '1';
        else
            C <= '0';
        end if;
    end process;
end architecture Behavioral;
```

Zaczynamy od słowa kluczowego „process” po którym następuje lista wrażliwości procesu, a między „begin” i „end process” mamy listę instrukcji programu sekwencyjnego.

Synteza z poziomu strukturalnego jest najprostsza. Wykonywana jest szybko, bo narzędzie syntezy „ma mało do roboty”. W przypadku opisu przepływowego narzędzie syntezy może dokonać optymalizacji układu i jego działanie będzie wolniejsze. W przypadku opisu behawioralnego wszystko zależy od narzędzia syntezy. Opisy behawioralne generują sporo zamieszania. Można się spotkać z opiniami, że opis behawioralny zwalnia projektanta od zajmowania się szczegółami implementacyjnymi. Narzędzia syntezy nie dają sobie rady z optymalnym realizowaniem opisów behawioralnych. Czasami narzędzie musi na podstawie opisu „zorientować się” o jaki układ nam chodzi, następnie dobrać odpowiedni sposób realizacji opisu przy pomocy zasobów sprzętowych by na końcu projekt zrealizować. Nie należy liczyć na to, że wynik syntezy będzie najlepszy z możliwych i będzie optymalny pod względem szybkości działania jak i zajętości zasobów.

Obok omówionych powyżej poziomów opisu istnieje również coś takiego jak:

### Poziom przesłań międzyrejstrowych

(Register Transfer Level). Jest to poziom, który jest specyficznym połączeniem opisów przepływowego i behawioralnego dla potrzeb syntezy. Opisy na tym poziomie będą używane przez nas do celów syntezy.

### Porty

Port może pracować w jednym z trybów pracy:

B : in BIT;

S : out BIT;

IN – port jest wówczas tylko do odczytu (wprowadzamy nim sygnał do układu)

OUT – port jest tylko do zapisu. Nie można go czytać wewnątrz architektury.

INOUT – praca dwukierunkowa.

BUFFER – to jest właściwy tryb wyjściowy z możliwością odczytu. Port jako taki jest wyjściowy a sygnał wystawiany na tym porcie można odczytywać wewnątrz jednostki.

Istnieje jeszcze tryb LINKAGE który został zgłoszony do usunięcia ze standardu. Nie należy go stosować. Był on używany jako łącznik podczas łączenia hierarchii modułów. Do takiego portu nie można było dokonywać zapisu ani odczytu.

W syntezie układów należy używać portów typu IN oraz OUT. Pozostałe porty mogą być syntezyzowane w sprzęcie przez narzędzia w dziwny sposób. Jeżeli będziemy chcieli użyć bufora 3-stanowego, to będziemy go musieli opisać go ręcznie. Możliwości pracy dwukierunkowej mogą być używane do specyfikacji i symulacji, ale nie do syntezy.

Jeżeli chcemy mieć port jednokierunkowy z możliwością odczytu, to należy korzystać z trybu BUFFER. Używanie trybu INOUT jest błędem. Poza tym port pracujący w trybie BUFFER może mieć tylko jedno przypisanie sygnału. Pojawia się problem z portami wyjściowymi, gdzie ich odczyt jest nielegalny:

```
entity AndNand is
  port ( A : in BIT;
         B : in BIT;
         C : in BIT;
         WY_And : out BIT;
         WY_Nand : out BIT);
end AndNand;
architecture DataflowBad of AndNand is
begin
  WY_And <= A and B and C;
  WY_Nand <= not WY_And;
end DataflowBad;
```

Mamy tutaj 2-wyjściową bramkę AND NAND. Można by to było w najprostszy sposób uzyskać poprzez obliczenie iloczynu trzech sygnałów i następnie na jego zanegowaniu. Problem w tym, że taki opis jest nielegalny, ponieważ niechcący napisaliśmy operację odczytu portu wyjściowego (podkreślona linia). Można to obejść w następujący sposób:

```
architecture DataflowOK of AndNand is
  signal Int_And : BIT;
begin
  Int_And <= A and B and C;
  WY_And <= Int_And;
  WY_Nand <= not Int_And;
end DataflowOK;
```

Polega to na zdefiniowaniu dodatkowego sygnału wewnętrznego (podkreślona linia) do którego przypisujemy wartość iloczynu. Następnie sygnał ten kopiujemy do portu wyjściowego (WY\_And) oraz po zanegowaniu do drugiego wyjścia (WY\_Nand).

Definiowanie sygnałów wewnętrznych polega na wstawieniu słowa „signal”, następnie podaniu nazwy sygnału i po dwukropku typu tego sygnału.

Instrukcję przypisania sygnału realizuje operator „<=”. W ten sposób można przypisywać wartości portom i sygnałom. Należy pamiętać o tym, że sygnały w przeciwieństwie do portów nie mają ściśle określonego trybu pracy. Z operacją przypisania związane jest tworzenie kolejki zdarzeń czasowych.

### Obiekty jawne VHDL

Do obiektów jawnych zalicza się SYGNAŁY, OPERATORY PRZYPISANIA oraz SPECJALNY MECHANIZM WARTOŚCIOWANIA. W układach cyfrowych często zdarza się, że jedna i ta sama linia jest sterowana

kilkoma sterownikami naraz. Jest to uwzględnione w mechanizmie wartościowania sygnałów. Każda instrukcja przypisania jest traktowana jako sterownik sygnału. Jeżeli dany sygnał jest przepisywany w wielu miejscach, to może mieć wiele sterowników. Wtedy jest potrzebne opracowanie sposobu według którego sygnał będzie ustalany. Realizuje się to przy pomocy FUNKCJI ROZSTRZYGAJĄCYCH. Z mechanizmem wartościowania są związane TRANSAKCJE (uwzględnienie momentu przypisania). Przypisanie sygnały nie musi być natychmiastowe. Jeżeli chcemy wiernie symulować pracę układu cyfrowego, to do każdego przypisania należy dodać pewne opóźnienie. Z każdym sygnałem jest tworzona lista sterowników i skojarzonych z nim transakcji. Podczas symulacji te transakcje są wykonywane.

Sygnały są tworem skomplikowanym, ale za to mogą w miarę wiernie naśladować pracę rzeczywistych sygnałów elektronicznych w układzie.

Obiektami jawnymi są również stałe (słowo kluczowe „constant”):

```
constant Pi : REAL := 3.14;
```

Istnieją też obiekty jawne w postaci zmiennych. Kod opisujący zmienną typu czasowego z przypisaną wartością początkową wygląda następująco:

```
variable SetupTime : TIME := 2.5ns
```

Mamy tu również przykład stałej typu fizycznego (nanosekundy) typu TIME. Zmienne nie są syntezowalne. Występują głównie w procesach. Kojarzyć je należy z opisami sekwencyjnymi. Ze zmiennymi nie ma takiego zamieszania jak z sygnałami. Nie dotyczą ich takie rzeczy jak wiele sterowników, funkcje rozstrzygające, transakcje...

Obiektami jawnymi są również PLIKI. Można je wykorzystywać do przechowywania pobudzeń układów i zapisywania ich odpowiedzi.

### Klauzula „generic”

Jest to sposób na parametryzowanie jednostek. Typowym zastosowaniem może być zdefiniowanie bufora z magistralami o różnej szerokości.

Opis pseudoformalny:

```
entity identifier is
  generic ( parameter_declarations ); -- optional
  port { port_declarations };         -- optional
  [ declarations ]                     -- optional
begin                                 \_ optional
  [ statements ]                       /
end entity identifier ;
```

Przed specyfikacją portów może wystąpić klauzula „generic” która określa parametry. Po klauzuli „port” mogą wystąpić dodatkowe deklaracje, gdzie można zdefiniować sygnały i zmienne, które będą globalne we wszystkich architekturach dla danej jednostki (rzadko się to stosuje). Po „begin” definiuje się część opisową jednostki. Przykład użycia klauzuli „generic”:

```
entity Buf is
  generic ( N : POSITIVE := 8;          -- data width
            Delay : DELAY_LENGTH := 2.5 ns );
  port ( Input  : in BIT_VECTOR( N-1 downto 0 );
        OE     : in BIT;
        Output : out BIT_VECTOR( N-1 downto 0 ) );
end entity Buf;
```

Tu mamy podane dwa parametry: N określa szerokość wektorów wejściowego i wyjściowego a parametr DELAY\_LENGTH będzie używany w modelu czasowym jako wartość opóźnienia wnoszona przez bufor. Jest to bufor 3-stanowy (ma zdefiniowane wejście OE). Widzimy tu sposób zdefiniowania wektorów. Typ wektorowy konkretyzuje się podając w nawiasach okrągłych zakres indeksów (u nas indeksy od N-1 do 0). Gdyby zaszała potrzeba numerowania z indeksem rosnącym, to zamiast „N-1 downto 0” należy napisać „0 to N-1”. W klauzuli „generic” są podane domyślne wartości parametrów.

### Pakiet „standard”

W kodzie źródłowym podanym przy okazji omawiania opisu strukturalnego była dołączona biblioteka UNISIM. Istnieje standardowy pakiet zdefiniowany od razu w języku VHDL, który jest domyślnie dołączany. Zawiera on definicje standardowych typów danych:

```
type INTEGER is range --usually typical INTEGER-- ;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type REAL is range --usually double precision f.p.-- ;
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is ( --256 characters-- );
type STRING is array (POSITIVE range <>) of CHARACTER;
type TIME is range --implementation defined-- ;
  units
    fs;          -- femtosecond
    ps = 1000 fs; -- picosecond
    ns = 1000 ps; -- nanosecond
    us = 1000 ns; -- microsecond
    ms = 1000 us; -- millisecond
    sec = 1000 ms; -- second
    min = 60 sec; -- minute
    hr = 60 min; -- hour
  end units;
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
```

6

Typ INTEGER jest tym, który znamy z innych języków programowania. W języku VHDL możemy tworzyć podtypy: z typu INTEGER zostały stworzone dwa podtypy (liczb naturalnych i dodatnich NATURAL i POSITIVE). Pojawia się tu również użycie atrybutu (przykładowo „INTEGER'HIGH”). Po lewej stronie apostrofu znajduje się coś, czego atrybut odczytujemy. Po prawej stronie – nazwa atrybutu. Jest to wyrażenie, które może zwracać różne rzeczy: mogą to być wartości, mogą to być podtypy.

## Pakiet „standard” c.d.

```

type REAL is range --usually double precision f.p.-- ;
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is ( --256 characters-- );
type STRING is array (POSITIVE range <>) of CHARACTER;
type TIME is range --implementation defined-- ;
  units
    fs;          -- femtosecond
    ps = 1000 fs; -- picosecond
    ns = 1000 ps; -- nanosecond
    us = 1000 ns; -- microsecond
    ms = 1000 us; -- millisecond
    sec = 1000 ms; -- second
    min = 60 sec; -- minute
    hr = 60 min; -- hour
  end units;
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;

```

Na poprzednim wykładzie zostały omówione typy liczb całkowitych do podtypu liczb dodatnich włącznie. Istnieje typ zmiennoprzecinkowy REAL. Obejmuje swoim zakresem liczby podwójnej precyzji. Typ boole'owski jest zdefiniowany jako typ wyliczeniowy (enumeracja). Typ znakowy CHARACTER również definiuje się jako wyliczeniowy. Stałe znakowe w języku VHDL zapisuje się w pojedynczych nawiasach.

STRING jest typem przechowującym napisy. Używa się tutaj wektora. Może on być wektorem z otwartym zakresem (w nawiasach ostrych nie ma konkretnej liczby określającej długość przechowywanej danej.). Typ TIME jest typem fizycznym, co oznacza że ma on zdefiniowane jednostki. Jednostką bazową dla TIME jest femtosekunda ( $10^{-15}$  sekundy). Typu tego używamy głównie do realizowania procesu symulacji działania układów cyfrowych. Ważne jest, by stosować odstęp między wartością a jednostką czasu. DELAY\_LENGTH jest podtypem czasowym.

## Typ BIT i napisy bitowe

Typ BIT jest typem wyliczeniowym:

```
type BIT is ('0', '1');
```

W VHDL'u nie ma fizycznego typu, który mógłby reprezentować stany cyfrowe. Symulacja opiera się na stałych znakowych. Wartości zmiennych typu BIT traktujemy jako wartości sygnałów logicznych.

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Z typem wektora bitowego spotkaliśmy się na poprzednim wykładzie. Istnieje on jako tablica, której rozmiar określony jest liczbą naturalną. Powyżej pokazana definicja wektora jest tzw. zakresem otwartym indeksu tablicy. Indeksy opisujące poszczególne bity wektora w VHDL mogą być ułożone w kolejności rosnącej lub malejącej. Zapis:

```
signal DataBus : BIT_VECTOR( 7 downto 0);
```

będzie oznaczał indeksowanie „w standardzie inżynierski”, czyli z malejącym indeksem. Zapis:

```
signal DataBus : BIT_VECTOR( 0 to 7);
```

oznacza indeksowanie z indeksem rosnącym.

Czym są napisy bitowe? Jest to specyficzny rodzaj napisów, które umieszcza się w cudzysłowach podwójnych. Napisy bitowe są specyficznym rodzajem napisów. Ich elementami mogą być wyłącznie zera i jedynki. Poniżej przedstawiono różne możliwości przypisania stałej bitowej do sygnału DataBus:

```

DataBus <= "10000000";
DataBus <= B"1000_0000";
DataBus <= X"80";

```

Kod powyżej reprezentuje napis bitowy z podaną podstawą liczbową. Jeżeli stosujemy zapis z podaniem podstawy, możemy korzystać ze znaków podkreślenia, które będą pełniły rolę separatorów przy oddzielaniu tetrad bitowych (by polepszyć widoczność). Należy uważać na to, że zapisem heksadecymalnym można przedstawiać napisy bitowe, których długość jest wielokrotnością czwórki.

Kolejne linijki są przykładami konstruowania agregatów. Tutaj podaje się wartości na poszczególnych pozycjach:

```

DataBus <= ( '1', '0', '0', '0', '0', '0', '0', '0' );
DataBus <= ( '1', others => '0' );

```

Zamiast wypisywać wszystkie pozycje bitowe osobno, można wypisać tylko pierwszą, użyć słowa kluczowego „others” i po nim wpisać wartość na pozostałych pozycjach. Agregaty przydają się gdy chcemy wyzerować jakiś wektor bitowy. Używamy do takiego celu zapisu:

```
DataBus <= ( others => '0' );
```

W agregacji można wskazywać pozycję w sposób jawny. Gdy chcemy na siódmym bicie wpisać jedynkę i na pozostałych bitach zero, posługujemy się kodem:

```
DataBus <= ( 7 => '1', others => '0' );
```

Jeżeli chcemy by na bitach 7 i 3 były jedynki, a zera na pozostałych pozycjach, można użyć składni:

```
DataBus <= (3|7 => '1', others => '0');
```

W wektorze bitowym możemy się odwoływać do jego fragmentu (przykładowo połowa bajtu):

```
HalfByte <= DataBus( 7 downto 4 );
```

Można się odwoływać również do pojedynczych elementów w wektorze bitowym:

```
MSB <= DataBus( 7 );
```

## Operatory

Większość z nich działa na wartościach numerycznych. Pogrupowane zostały według priorytetów wykonywania (od najwyższego do najniższego). Operatory w grupach mają jednakowy priorytet wykonywania. Łączność w języku VHDL jest zawsze lewostronna. Nie ma tutaj zwykłego operatora przypisania.

Listę otwierają operatory potęgowania, wyznaczania wartości bezwzględnej oraz wyznaczania negacji logicznej:

<b>**</b>	exponentiation,	numeric <b>**</b> integer,	result numeric
<b>abs</b>	absolute value,	<b>abs</b> numeric,	result numeric
<b>not</b>	complement,	<b>not</b> logic or boolean,	result same

Kolejne operatory opisują mnożenie oraz dzielenie. Polecenie „remainder” służy do wyznaczania reszty z dzielenia:

<b>*</b>	multiplication,	numeric <b>*</b> numeric,	result numeric
<b>/</b>	division,	numeric <b>/</b> numeric,	result numeric
<b>mod</b>	modulo,	integer <b>mod</b> integer,	result integer
<b>rem</b>	remainder,	integer <b>rem</b> integer,	result integer

Poniżej przedstawione zostały jednoargumentowe operatory zmiany znaku. Operatory można przeciążać.

<b>+</b>	unary plus,	<b>+</b> numeric,	result numeric
<b>-</b>	unary minus,	<b>-</b> numeric,	result numeric

Kolejnymi operatorami są dodawanie, odejmowanie oraz konkatencja:

<b>+</b>	addition,	numeric <b>+</b> numeric,	result numeric
<b>-</b>	subtraction,	numeric <b>-</b> numeric,	result numeric
<b>&amp;</b>	concatenation,	array or element,	result array

Ostatni operator jest użyteczny podczas przekształcania napisów bitowych lub innych obiektów będących wektorami. O konkatencji powiemy nieco później. Kolejne operatory służą do realizowania obrotów i przesunięć na macierzach lub wektorach:

<b>sll</b>	shift left logical,	log. array <b>sll</b> integer,	result same
<b>srl</b>	shift right log.,	log. array <b>srl</b> integer,	result same
<b>sla</b>	shift left arith.,	log. array <b>sla</b> integer,	result same
<b>sra</b>	shift right arith.,	log. array <b>sra</b> integer,	result same
<b>rol</b>	rotate left,	log. array <b>rol</b> integer,	result same
<b>ror</b>	rotate right,	log. array <b>ror</b> integer,	result same

Grupa poniżej reprezentuje operatory porównywania:

<b>=</b>	equality,	result boolean
<b>/=</b>	inequality,	result boolean
<b>&lt;</b>	less than,	result boolean
<b>&lt;=</b>	less than or equal,	result boolean
<b>&gt;</b>	greater than,	result boolean
<b>&gt;=</b>	greater than or equal,	result boolean

Należy zwrócić uwagę na operator równości, który składa się z jednego znaku równości. Przyzwyczajeni jesteśmy do faktu, że znak „=” w językach programowania reprezentował sobą przypisanie. W VHDL przypisanie realizuje się przez zapis „:=”.

Najniższy priorytet mają operatory logiczne:

<b>and</b>	logical and,	log. array or boolean,	result same
<b>or</b>	logical or,	log. array or boolean,	result same
<b>nand</b>	logical nand,	log. array or boolean,	result same
<b>nor</b>	logical nor,	log. array or boolean,	result same
<b>xor</b>	logical xor,	log. array or boolean,	result same
<b>xnor</b>	logical xnor,	log. array or boolean,	result same

Wszystkie mają ten sam priorytet. Nie obowiązują tutaj zasady rachunku logicznego, gdzie AND miał większy priorytet od OR. W języku VHDL nie można łączyć ze sobą różnych operatorów logicznych. Zapis:

```
C <= A and B and C;
```

nie będzie błędny, ponieważ operator iloczynu jest asocjacyjny. Można go kaskadowo łączyć. Nie można mie-



szać ze sobą różnych operatorów w jednym wyrażeniu logicznym. Kod:

```
D <= E and B or C;
```

będzie zapisem błędnym. Oczywiście jeżeli powyższy zapis przekształcony zostanie na:

```
D <= (E and B) or C;
```

to będzie on prawidłowy. Używanie wyrażeń złożonych nasuwa za sobą konieczność jawnego używania nawiasów. Prawdopodobnie ma to służyć jawnemu pokazywaniu priorytetów wykonywania poszczególnych operatorów logicznych.

VHDL jest rygorystyczny. Można ze sobą łączyć operatory AND, OR, XOR oraz XNOR. Nie można łączyć ze sobą NOR-ów i NAND-ów, ponieważ nie są one operatorami asocjacyjnymi.

Powiemy teraz więcej o operatorze konkatencji:

```
signal ASCII : BIT_VECTOR ( 7 downto 0);
```

```
signal Digit : BIT_VECTOR ( 3 downto 0);
```

Mamy dwa sygnały: 8-bitowy „ASCII” oraz 4-bitowy „Digit”. Aby dla danej cyfry dziesiętnej można było utworzyć kod ASCII, to należy dokleić na starszą tetradę trójkę, a na młodszą podać wprost zakodowaną cyfrę:

```
ASCII <= "0011" & Digit; -- X"3" & Digit;
```

Można powyższą operację opisać inaczej, odwołując się do pojedynczych bitów wektora:

```
ASCII <= X"3" & Digit ( 3 ) & Digit ( 2 ) &
        Digit ( 1 ) & Digit ( 0 );
```

Jeżeli mamy jakiś wektor i chcemy go przesunąć w prawo, to podstawiamy wektor mający na najstarszej pozycji zero a pozostałe bity bierzemy z wektora „ASCII”:

```
-- Shift right (this must be synchronous!):
ASCII <= '0' & ASCII( 7 downto 1 );
```

Przypisanie sygnału opisane powyżej musi być synchroniczne. Tak będziemy opisywać pracę rejestrów przesuwanych, które muszą działać synchronicznie, bo asynchroniczna praca takich układów byłaby bezsensowna.

Analogicznie do poprzedniego przykładu będzie działało przesuwanie w lewo:

```
-- Shift left:
ASCII <= ASCII( 6 downto 0 ) & '0';
```

## Pakiet „stdlogic”

Wspomniano wcześniej, że typ BIT jest dwuwartościowym typem abstrakcyjnym. Dla rzeczywistego sygnału logicznego dwa stany to za mało. Typ, który opisywałby właściwości sygnałów cyfrowych, został opisany w standardzie 1164. Pakiet dołączany jest automatycznie do każdego nowego modułu, opisywanego w VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

W tym pakiecie zdefiniowano typ wyliczeniowy 9-wartościowy, który będzie reprezentatywnie przedstawiać zachowanie się sygnałów w układach cyfrowych:

```
type STD_ULOGIC is ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                    );
```

Typ ten składa się ze stałych znakowych, co pozwala na łatwe stosowanie ich w napisach bitowych i wektorach. Synteza i symulacja układów cyfrowych opiera się na tym typie. Z punktu widzenia jest to świat, w którym analizuje się stałe znakowe. Nie ma żadnej zaszytej implementacji fizycznej tych stanów.

'U' – reprezentuje sygnał nie zainicjalizowany. Z takimi wartościami startuje symulacja. Wartości ich są nieznane.

'X', '0', '1' – grupa sygnałów „silnych” (silna wartość nieznana, silne/wymuszone zero i silna jedynka). Wartość nieznana pojawia się w sytuacji konfliktu sygnałów (gdy na jedną linię zostaną podane równocześnie stany '0' i '1').

'Z' – oznacza stan wysokiej impedancji. Nie ma czegoś takiego jak silna (lub słaba) impedancja.

'W', 'L', 'H' – grupa sygnałów o „słabych” wartościach. Pojawiają się one „w sąsiedztwie” rezystorów Pull-Up

(daje słabą jedynkę) oraz Pull-Down (daje słabe zero). Podanie tych sygnałów na jedną linię równocześnie wygeneruje stan 'W'.

'-' – to wartość, której interpretacja jest specyficzna. Nigdy nie jest ona generowana podczas pracy układu. Może być ona wpisana przez nas.

Omówiony zestaw 9-ciu stanów stał się standardem przemysłowym i na tym zestawie wszyscy pracują. Zdefiniowany został typ wektorowy:

```
type STD_ULOGIC_VECTOR is array ( NATURAL range <> ) of
STD_ULOGIC;
```

Nie pracuje się na typie BIT (może być on używany w specyficznych przypadkach). Korzysta się z ULOGIC. Typ ten pociągnął za sobą konieczność opracowania funkcji rozstrzygającej oraz tabeli zdefiniowanej w standardzie:

```
function resolved ( s : STD_ULOGIC_VECTOR ) return STD_ULOGIC;

constant resolution_table : stdLogic_table := (
--
--      U   X   0   1   Z   W   L   H   -   |   |
--
--      ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', -- | U |
--      ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', -- | X |
--      ( 'U', '0', '0', '0', '0', '0', '0', '0', '0', -- | 0 |
--      ( 'U', '1', '1', '1', '1', '1', '1', '1', '1', -- | 1 |
--      ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X', -- | Z |
--      ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X', -- | W |
--      ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X', -- | L |
--      ( 'U', 'X', '0', '1', 'H', 'W', 'H', 'W', 'X', -- | H |
--      ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', -- | - |
--
--      );
```

Nigdzie nie jest generowana wartość „Don't care”. Odpowiedziami dla silnych sygnałów są albo sygnały '0' i '1' lub 'X'. Tablica jest symetryczna względem przekątnej. Stan wysokiej impedancji powiela sygnał z drugiego sterownika (za wyjątkiem sygnału '-').

Typ STD\_LOGIC jest typem STD\_ULOGIC z dołączoną funkcją rozstrzygającą:

```
-----
-- *** industry standard logic type ***
-----
subtype STD_LOGIC is resolved STD_ULOGIC;
type STD_LOGIC_VECTOR is array ( NATURAL range <> ) of STD_LOGIC;
```

Tego typu należy używać przy opisie układów cyfrowych. Nie używa się BIT-u.

Pozostała jeszcze kwestia przeciążonych operatorów logicznych. Rozwiązano to przy użyciu tablic, które opisują działanie operatorów:

```
constant and_table : stdLogic_table := (
--
--      |   U   X   0   1   Z   W   L   H   -   |
--      ( 'U', 'U', '0', 'U', 'U', 'U', 'U', 'U', 'U', -- | U |
--      ( 'U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X', -- | X |
--      ( 'U', '0', '0', '0', '0', '0', '0', '0', '0', -- | 0 |
--      ( 'U', '1', '0', '1', 'X', 'X', 'X', 'X', 'X', -- | 1 |
--      ( 'U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X', -- | Z |
--      ( 'U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X', -- | W |
--      ( 'U', '0', '0', '0', '0', '0', '0', '0', '0', -- | L |
--      ( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X', -- | H |
--      ( 'U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X', -- | - |
--
--      );

constant or_table : stdLogic_table := (
--
--      |   U   X   0   1   Z   W   L   H   -   |
--      ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', -- | U |
--      ( 'U', 'X', 'X', 'U', 'X', 'X', 'X', 'X', 'X', -- | X |
--      ( 'U', '0', '0', 'U', 'X', 'X', 'X', 'X', 'X', -- | 0 |
--      ( 'U', '1', '1', 'U', '1', '1', '1', '1', '1', -- | 1 |
--      ( 'U', 'X', 'X', 'U', 'X', 'X', 'X', 'X', 'X', -- | Z |
--      ( 'U', 'X', 'X', 'U', 'X', 'X', 'X', 'X', 'X', -- | W |
--      ( 'U', '0', '0', 'U', 'X', 'X', 'X', 'X', 'X', -- | L |
--      ( 'U', '1', '1', 'U', '1', '1', '1', '1', '1', -- | H |
--      ( 'U', 'X', 'X', 'U', 'X', 'X', 'X', 'X', 'X', -- | - |
--
--      );
```

## Instrukcje współbieżne

Są to instrukcje, które mogą wystąpić w ciele architektury (między słowami kluczowymi „begin” i „end” określającymi treść architektury). Ich kolejność jest nieistotna. Wystąpić mogą instrukcje:

- Współbieżne przypisanie sygnału („<=>”)
- Instancja komponentu
- Instrukcja generacji („generate”)
- Instrukcja procesu („process”)
- Instrukcja bloku (nie będzie omawiana)

- Współbieżne wywołanie procedury
- Współbieżne obliczenie asercji

Asercję używa się do przedstawiania warunków, które muszą być spełnione by program działał poprawnie. Gdy asercja nie jest spełniona, to wywoływana może być jakaś funkcja systemowa przerywająca działanie programowi wydrukowanie jakiegoś komunikatu. Asercje w VHDL są na bieżąco współbieżnie sprawdzane. Nie spełniona asercja w VHDL może generować ostrzeżenie, błąd lub przerwać działanie symulatora. Asercje nie są syntezywalne i służą tylko do debugowania modelu.

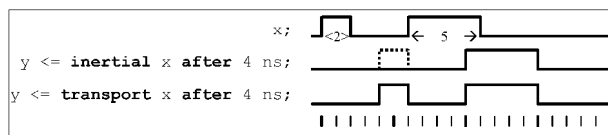
### Przypisanie sygnału

Podstawowa operacja, którą poznaliśmy na pierwszym wykładzie z VHDL-a. X oraz Y muszą być sygnałami. Mogą być one również portami:

```
y <= x;
y <= ( x1 and x2 ) or ( ( x3 nand x4 ) xor x5 );
```

UWAGA: Przypisanie zmiennej opisuje się inną instrukcją. Przypisania zmiennych nie można umieszczać w ciele architektury. Zmienne istnieją wewnątrz procesów.

W języku VHDL na potrzeby symulacji stworzono dwa modele przypisania sygnałów. Wiążą się one z sytuacją gdy przypisujemy sygnał z podaniem czasu opóźnienia (po klauzuli „after”):



Modele opóźnienia to model inercyjny i transportowy. W modelu **inercyjnym** tłumione są (nie są przypisywane) sygnały X, które są krótsze od czasu propagacji. W przypadku powyżej pokazanym przeniesiony zostanie tylko impuls o szerokości 5 ns. W modelu **transportowym** przenoszone są wszystkie impulsy bez względu na ich długość. Domyślnie stosowany jest model inercyjny z zerowym czasem opóźnienia:

```
-- Domyślnie:
y <= x;
y <= x after 4 ns; <=> y <= inertial x after 0 ns;
y <= x after 4 ns; <=> y <= inertial x after 4 ns;
```

Klauzula „after” jest niesyntezywalna. Odgrywa rolę tylko podczas symulacji projektu. Możliwe jest opisywanie wielokrotnej zmiany w reakcji na pojedynczą zmianę sygnału X:

```
-- Wielokrotna zmiana w reakcji na pojedynczą
-- zmianę sygnału:
y <= x after 4 ns, not x after 8 ns;
```

Przecinkami można oddzielać całą listę przypisań sygnałów z rosnącymi czasami opóźnień. Tego również się nie syntezyje. Oczywiście podczas symulacji się przydaje. Kod przedstawiony poniżej opisuje generację sygnału zegarowego:

```
-- W opisach sekwencyjnych (np. powtarzanych w pętlach):
Clk <= '1', '0' after ClkPeriod / 2;
..
```

„ClkPeriod” jest stałą typu DELAY\_LENGTHT.

### Przypisanie warunkowe

Często używane w opisie układów kombinacyjnych. Posłużymy się multiplexerem 4 na 1:

```
entity MUX_4 is
  port( A, B, C, D : in STD_LOGIC;
        Sel : in STD_LOGIC_VECTOR( 1 downto 0 );
        Y : out STD_LOGIC );
end MUX_4;
```

Ma on 4 wejścia (A, B, C, D), jedno wyjście (Y) oraz wejście selekcji będące 2-bitowym wektorem. Składnia przypisania warunkowego:

```
architecture Dataflow of MUX_4 is
begin
  Y <= A when Sel = "00" else
    B when Sel = "01" else
    C when Sel = "10" else
    D;
end Dataflow;
```

Stosując przypisanie warunkowe trzeba być ostrożnym. Przypisanie to nie żąda wyczerpującej listy warunków (może mieć miejsce sytuacja, że żaden z warunków nie będzie spełniony). Wektor „Sel” jest wektorem 2-ch sy-

gnałów o 9 wartościach. Wypisane zostały tylko 3 możliwości (z 81). Pozostałe 78 zostało skomasowane przy przypisaniu D. Narzędzie syntezy zinterpretuje prawidłowo układ jako multiplekser. Symulator działa inaczej i tak opisany multiplekser może dla 78 możliwości do wyjścia Y zostanie przypisane wejście D. Aby się przed tym zabezpieczyć, należy skorygować kod:

```

      . . .
      C when Sel = „10” else
      D when Sel = „11” else
      'X';

```

Nie należy tu używać 'U' ani '-!.

Jeżeli lista warunków jest niewyczerpująca możliwości, to zostanie wygenerowany przez narzędzie zatrask dla sygnału przypisywanego warunkowo. Gdy jeden z wypisanych warunków będzie spełniony, to zostanie przypisana nowa wartość sygnału. Gdy żaden z warunków nie zachodzi, to sygnał nie zmieni swojej wartości (zamykanie się zatrasku). Pojawianie się zatrasków projekcie bywa niebezpieczne, ponieważ burzy ono zależności czasowe. Dobrze jest pisać kompletne listy warunków w instrukcjach przypisywania warunkowego. Pojawienie się zatrasków generuje ostrzeżenia w raporcie syntezy.

Instrukcja przypisywania warunkowego z „when” „else” działa tak, że wyszukiwany jest pierwszy warunek, który jest spełniony (**ważna jest kolejność warunków**), wobec czego każdy warunek na pozostałych pozycjach zawiera sobie negację wszystkich warunków poprzednich. Jeżeli narzędzia syntezy nie dostrzegą w opisie „gotowca” i synteza przebiega na piechotę w postaci bramek logicznych, to pociąga to za sobą dużą zasobożerność projektu. Instrukcję tą należy stosować z rozwagą. Wady tej nie ma

## Przypisanie selektywne

Składnia polecenia jest inna:

```

architecture Dataflow2 of MUX_4 is
begin
    with Sel select
        Y <= A when "00",
           B when "01",
           C when "10",
           D when others;
end Dataflow2;

```

15

Instrukcja jest bezpieczniejsza, ponieważ jej składnia wymaga kompletnej listy warunków, które wzajemnie się wykluczają. Jeżeli lista warunków nie jest wyczerpująca, to należy używać słowa kluczowego „others”. Ponieważ warunki są rozłączne **nie jest istotna kolejność opcji** (za wyjątkiem „others”, która jeżeli występuje, to musi być ostatnia).

## Instancje komponentów

Mamy zdefiniowane jakieś jednostki i chcemy je wstawić we wnętrze innej architektury. W przykładzie użyjemy 2-wejściowe bramki XOR i AND i zmontujemy z nich półsumator. Najpierw potrzebować będziemy definicji jednostek („XOR\_2WE” oraz „AND\_2WE”). Mają one oprócz listy portów listę parametrów generycznych:

<pre>entity XOR_2WE is   generic( Tp : DELAY_LENGTH );   port ( i1, i2 : in STD_LOGIC;         O : out STD_LOGIC); end XOR_2WE; architecture A of XOR_2WE is begin   O &lt;= i1 xor i2 after Tp; end A;</pre>	<pre>entity AND_2WE is   generic( Tp : DELAY_LENGTH );   port (i1, i2 : in STD_LOGIC;         O : out STD_LOGIC); end AND_2WE; architecture A of AND_2WE is begin   O &lt;= i1 and i2 after Tp; end A;</pre>
---	--

Parametrem tym jest „Tp” (czas propagacji bramek). Architektura bramek składa się z jednej instrukcji współbieżnej (przypisanie z użyciem „after”). Pracujemy na typie STD\_LOGIC. Wykorzystując definicje jednostek, możemy ich użyć w półsumatorze:

```
entity HalfAdder is
  (...)
architecture Structural of HalfAdder is
  component XOR_2WE is
    generic( Tp : DELAY_LENGTH );
    port ( i1, i2 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
  component AND_2WE is
    generic( Tp : DELAY_LENGTH );
    port ( i1, i2 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
```

Na początku będziemy potrzebowali deklaracji komponentów. Po słowie kluczowym „architecture” wstawiamy definicje sygnałów oraz deklaracje komponentów rozpoczynające się od słowa kluczowego „component”. Później następuje lista argumentów oraz przypisanie portów. Tak zadeklarowane komponenty wstawiamy w treści architektury we współbieżnych instrukcjach instancji komponentów:

```
  generic map( 5 ns ) port map( A, B, S );
  AND_gate : AND_2WE generic map( 3 ns ) port map( O=>C, i1=>A, i2=>B );
end architecture Structural;
```

Każda instancja musi mieć nazwę („XOR\_gate : ” oraz „AND\_gate : ”). Nazwę tworzymy zgodnie z zasadami tworzenia identyfikatorów w VHDL-u. Nazwa wygląda tak, jak etykieta (posiada dwukropek). Po niej następuje nazwa komponentu z odwzorowaniem parametrów generycznych („generic map”) oraz portów („port map”). Powyżej przedstawiono dwa style odwzorowania. Przy instancji bramki XOR mamy styl pozycyjny (kolejność sygnałów na liście jest identyczna kolejności portów komponentu). Możemy przyporządkowywać według nazw (tak zrobiono przy instancji bramki AND). Druga możliwość pozwala na zmianę kolejności zapisu.

W instancjach niektórych komponentów może zająć konieczność nie dołączenia niektórych portów. Używa się słowa kluczowego „open”. Zdarza się to głównie przy portach wyjściowych. Zapisujemy to tak:

NazwaPortu => open;

Wejścia powinno się dołączać. Jeżeli na schemacie układu pojawiają się komponenty z portami wejściowymi nie połączonymi, to zostaną one przez środowisko Xilinx dołączone do masy układu (podanie '0' na wejście).

Parametry generyczne mogą mieć wartości domyślne. Jeżeli pominiemy parametr, który taką wartość ma, to zostanie ona automatycznie podana. Brak wartości domyślnej zmusza nas do jawnego podawania wartości parametru.

## Instrukcja generacji

Występuje ona w wariancie z pętlą i wariancie warunkowym. W wariancie z pętlą używa się słów kluczowych „for” oraz „generate”. W przykładzie posłużymy się sumatorem kaskadowym (zbudowany z 8-miu sumatorów pełnych). Układ będzie posiadał 8-bitowe porty wejściowe A oraz B, 8-bitowy port wyjściowy oraz odpowiednie wejścia i wyjścia przeniesień:

```
entity FullAdder is
  port ( A, B : in STD_LOGIC_VECTOR( 7 downto 0 );
        CI : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR( 7 downto 0 );
        CO : out STD_LOGIC);
end FullAdder;
```

Można by napisać w VHDL-u moduły sumatorów pełnych i powielić je. Zamiast tego posłużymy się pętlą. Potrzebny będzie nam sygnał wewnętrzny dla przeniesień (wektor „Cint”):

```

architecture Dataflow of FullAdder is
    signal Cint : STD_LOGIC_VECTOR( 8 downto 0 );
begin
    lb: for i in 0 to 7 generate
        S(i) <= A(i) xor B(i) xor Cint(i);
        Cint(i + 1) <= ( A(i) and B(i) ) or
            ( A(i) and Cint(i) ) or ( B(i) and Cint(i) );
    end generate;
    Cint( 0 ) <= CI;
    CO <= Cint( 8 );
end Dataflow;

```

Cała architektura będzie składała się z jednej instrukcji generacji i dwóch współbieżnych przypisań. Składnia funkcji generacji zaczyna się od słowa „for” następnie podaje się zakres indeksowania w generacji (coś w rodzaju pętli). Zakres będzie biegł w górę. Zmienna „i” jest zmienną iteracyjną. Tej zmiennej nie definiuje się w VHDL-u. Powołuje się ją ad hoc na potrzeby funkcji generacji. Jest to zmienna całkowitoliczbowa. Po słowie kluczowym „generate” piszemy blok instrukcji współbieżnych które zostaną powielone. Na końcu należy przypisać sygnały wejścia i wyjścia z łańcucha przeniesień:

```

        ( A(i) and Cint(i) ) or ( B(i) and Cint(i) );
    end generate;
    Cint( 0 ) <= CI;
    CO <= Cint( 8 );
end Dataflow;

```

Architektura taka składa się z trzech instrukcji: instrukcji generacji oraz dwóch instrukcji przypisania współbieżnego. Instrukcja generacji generuje 8 bloków składających się z instrukcji przypisania.

Druga wersja instrukcji generacji jest wersją warunkową:

```

label: if condition generate      -- label required
    block_declarative_items      \ optional
begin
    concurrent_statements
end generate label;

```

Jeśli warunek jest spełniony, to generowany jest blok instrukcji współbieżnych. Czasami zagnieżdża się tą instrukcją w iteracyjnej instrukcji generacji. Etykieta (poprzednio „lb”, teraz „label”) jest obowiązkowa. Jeżeli zapomnimy o etykietce, to kompilacja i synteza projektu nie powiedzie się.

## Opisy sekwencyjne

### Instrukcja procesu

Sama instrukcja procesu jest współbieżna. Hierarchia jest taka że mamy jednostkę, mamy architekturę, wewnątrz architektury umieszczamy instrukcję procesu, wewnątrz instrukcji procesu możemy umieszczać instrukcje sekwencyjne. Wszystkie instrukcje sekwencyjne muszą znajdować się jakimś procesie, nie można ich umieszczać na poziomie architektury. Na poziomie architektury można umieszczać tylko instrukcje współbieżne. Proces jest taką instrukcją współbieżną a w procesie może znajdować się ciąg instrukcji sekwencyjnych. Składnia procesu jest następująca:

```

[label:] process [ ( sensitivity_list ) ] [ is ]
    [ declarative_items ]
begin
    sequential_statements
end process [ label ];

```

Etykieta jest opcjonalna. Po słowie kluczowym „process” znajduje się lista wrażliwości (lista czułości) procesu. Jest to lista sygnałów, których zmiana wartości powoduje wykonanie procesu. Procesy są wyzwalane w chwili wykrycia zmiany wartości sygnału z listy wrażliwości. Lista ta jest opcjonalna. Gdy jej nie ma, to w procesie muszą jawnie wystąpić instrukcje „wait”. Można wstrzymywać wykonanie procesu dzięki poleceniom „wait” lecz jest to niezalecane dla procesu syntezy. Przydaje się to przy generowaniu pobudzeń w testbenchach.

Wewnątrz procesu można używać instrukcji sekwencyjnych. Są nimi:

- Instrukcja wstrzymania („wait”)
- Instrukcja asercji
- Instrukcja raportu (generuje komunikaty diagnostyczne podczas pracy symulatora)
- Instrukcja sekwencyjnego przypisania sygnału („<=”)
- Instrukcja przypisania zmiennej („:=”) Zmienne istnieją tylko wewnątrz procesów. Nie ma ich na poziomie

architektury. Wyjątkiem są zmienne dzielone [odpowiedniki zmiennych globalnych], ale są to obiekty niewidoczne dla syntezy)

- Instrukcja wywołania procedury (nie będzie omawiana)
- Instrukcja warunkowa „if”
- Instrukcja wyboru „case”
- Instrukcje „next”, „exit”, „return”
- Instrukcja pusta „null” (odpowiednik znaku średnika z C++ lub Javy)

### Instrukcja przypisania sygnału

Notacja jest taka sama jak przy instrukcji współbieżnego przypisania z tym wyjątkiem, że w procesach nie można używać przypisywania warunkowego i selektywnego. Można używać przypisania prostego z „after”. Przykładowo:

```
process (We1, We2) is
  begin
    Wy <= We1 and We2 after 4 ns;
  end;
```

Standard VHDL opisując pracę instrukcji współbieżnych przenosi wszystko do procesów. Każda prosta instrukcja przypisania jest przekładana na proces, który jest jawnie wyzwalany sygnałami występującymi po prawej stronie instrukcji przypisania.

Nielegalne są przypisania warunkowe oraz selektywne. Błędem byłoby:

```
process (We1, We2) is
  begin
    Wy <= We1 when We2 = '1'
      else '0';
  end;
```

### Instrukcja przypisania zmiennej

Najpierw musimy sobie stworzyć zmienną. Zmienne tworzymy na poziomie procesu:

```
process (A, B) is
  variable Var1, Var2:STD_LOGIC := 'u';
  begin
    . . .
    Var1 <= '0';
    . . .
    Var2 <= Var1 xor '1';
    . . .
  end;
```

Po słowie kluczowym „variable” występuje lista nazw zmiennych i po dwukropku nazwa typu zmiennych (tutaj STD\_LOGIC). Później można jeszcze przypisać wartości początkowe (w przykładzie przypisano 'u'). Zmienne tworzymy tak samo jak sygnały (w sensie zapisu) ale zmienne tworzy się **tylko** na poziomie procesów. Później można użyć jakichś instrukcji przypisania zmiennych (jest to natychmiastowe).

### Instrukcja „wait”

Proces wykonuje się sekwencyjnie i jego wykonanie można wstrzymać do określonego momentu czasowego:

```
wait for 10 ns;           -- timeout
wait until clk='1';       -- warunek logiczny
wait until A>B and S1 or S2;
wait on sig1, sig2;       -- lista wrażliwości
```

Wstrzymanie może na określony odcinek czasu („wait for”), albo tak długo aż zostanie spełniony warunek logiczny („wait until”), albo może być podana jawna lista wrażliwości. Wtedy wstrzymanie trwa do chwili zmiany wartości sygnału z listy wrażliwości („wait on”). Instrukcje „wait on” są syntezywalne i niektóre narzędzia pozwalają na ich stosowanie.


## Instrukcja warunkowa „if”

Składnia jest standardowa. Etykieta jest opcjonalna. Konieczne jest stosowanie zakończenia w postaci „endif”:

```
[ label: ] if condition1 then
    statements
elseif condition2 then \ optional
    statements
...
else \ optional
    statements
end if [ label ] ;
```

Poniżej po lewej stronie mamy opisany multiplekser 4 na 1 z jawnie wyróżnionymi 4-ma kombinacjami sterującymi i sprowadzający inne pobudzenia ('W' 'L' 'H' ...) do stanu nieznanego 'X'. Zapis pokazany po lewej jest równoważny procesowi opisanemu po prawej stronie:

```
architecture DF of MUX_4 is
begin
  Y <= A when Sel = "00" else
    B when Sel = "01" else
    C when Sel = "10" else
    D when Sel = "11" else
    'X';
end DF;
```




```
architecture DF_Eq of MUX_4 is
begin
  process ( Sel, A, B, C, D )
  begin
    if Sel = "00" then
      Y <- A;
    elsif Sel = "01" then
      Y <- B;
    elsif Sel = "10" then
      Y <- C;
    elsif Sel = "11" then
      Y <- D;
    else
      Y <= 'X';
    end if;
  end process;
end DF_Eq;
```

Proces składa się z kaskady warunków. Wygląda to podobnie jak przy instrukcji przypisania warunkowego. Na liście wrażliwości procesu będą występowały wszystkie sygnały, które znajdowały się po prawej stronie polecenia przypisania współbieżnego (sygnały wejściowe A, B, C, D oraz sygnał Sel). Warunki nie muszą być rozłączne. Spełnienie jakiegoś warunku implikuje to, że żaden inny nie jest spełniony.

*[Skok przez plot do slajdu 25]*

Znamy procesy, znamy listy wrażliwości, znamy instrukcje warunkowe, więc znamy wszystkie mechanizmy potrzebne do opisywania przerzutników. W VHDL nie ma czegoś takiego, jak deklaracja że dany sygnał jest sygnałem synchronicznym (przypisywanym np. do narastającego zbocza sygnału zegarowego). Opisy sygnałów synchronicznych wymagają specjalnych konstrukcji. Musi istnieć uzależnienie przypisania sygnału od zbocza narastającego sygnału zegarowego. Jak w VHDL-u opisać przerzutnik typu D? Przykład znajduje się po lewej stronie obrazka:

```
entity DFF is
port ( D : in STD_LOGIC;
      Clk : in STD_LOGIC;
      Q : out STD_LOGIC );
end DFF;
architecture RTL of DFF is
begin
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      Q <= D;
    end if;
  end process;
end architecture;
```



```
entity DFF is
port ( D : in STD_LOGIC;
      Clk : in STD_LOGIC;
      Q : out STD_LOGIC );
end DFF;
architecture Behavioral of DFF is
  signal Q_int;
begin
  Q <= Q_int;
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      Q_int <= D;
    end if;
  end process;
end architecture;
```

Będziemy potrzebowali procesu wyzwalanego zmianą sygnału „Clk”. Konieczna jest instrukcja warunkowa, testująca czy mamy do czynienia z narastającym zboczem zegarowym. Jest to złożone wyrażenie logiczne:

**if Clk'Event and Clk = '1' then . . .**

„Event” to atrybut mający wartość „TRUE” gdy sygnał po jego lewej stronie zmieni wartość w bieżącym cyklu symulacji. Drugi warunek jest konieczny do wykrycia zbocza narastającego. Jeżeli chcielibyśmy by przerzutnik reagował na zbocze opadające, to zamiast **and Clk = '1'** trzeba napisać **and Clk = '0'**. Działanie będzie poprawne gdy „Clk” będzie sygnałem typu BIT. W naszym przypadku „Clk” jest typem sygnału o 9 wartościach, ale nikt nie grzebie się w szczegółach, bo sygnały zegarowe z założenia są dwuwartościowe.

Proces jest wyzwalany tylko przy pomocy „Clk”. Zmiany sygnału na wejściu „D” nie powodują przypisania nowej wartości sygnału Q. Tylko zbocze sygnału zegarowego powoduje zmianę stanu przerzutnika. Dlatego na



liście wrażliwości nie ma innych sygnałów niż „Clk”. Jest to konieczne, by pokazać że uzależniamy się od sygnału zegarowego.

W drugim przykładzie mamy przedstawiony opis przerzutnika typu T (prawa strona):

```

entity TFF is
  port ( T : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC );
end TFF;

architecture RTL of TFF is
  signal Q_int : STD_LOGIC := '0';
begin
  Q <= Q_int;
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      if T = '1' then
        Q_int <= not Q_int;
      end if;
    end if;
  end process;
end architecture;
    
```

Różnica jest taka, że przerzutnik T zmienia swoją wartość tylko wtedy gdy na wejściu T jest stan '1'. Konieczne jest powielenie portu wyjściowego w postaci sygnału wewnętrznego i dlatego opis jest nieco dłuższy. Definiujemy sygnał wewnętrzny „Q\_int”, dodajemy instrukcję przypisania współbieżnego mówiącą o tym, że na porcie wyjściowym ma się znaleźć to samo, co na sygnale wewnętrznym „Q\_int”. Potem można dodać blok z warunkiem.

Ponieważ notacje `if Clk'Event and Clk = '1'` występują dość często, to w pakiecie STD\_LOGIC\_1164 zdefiniowano funkcje, które uwzględniają specyfikę logiki 9-wartościowej:

```

• Pakiet STD LOGIC 1164:
function rising_edge (signal s : STD_ULOGIC) return BOOLEAN;
function falling_edge (signal s : STD_ULOGIC) return BOOLEAN;
if Clk'Event and Clk = '1' then... => if rising_edge(Clk) then...
    
```

Należy z tych funkcji („rising\_edge” oraz „falling\_edge”) korzystać. Są one bezpieczniejsze w używaniu i ich używanie jest bardziej czytelne.

Opis przerzutników można rozszerzyć o sygnał Clock Enable. Proces zależy od o sygnału zegarowego (nie zależy od CE). Sygnał na CE sprawdzany jest po wykryciu zbocza narastającego sygnału zegarowego. Jeżeli na CE jest 0, to nic się nie dzieje (lewa strona obrazka):

```

entity DFF_E is
  port( D : in STD_LOGIC;
        CE : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC );
end DFF_E;

architecture RTL of DFF_E is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;

entity DFF_CE is
  port( D : in STD_LOGIC;
        Clr : in STD_LOGIC;
        CE : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC );
end DFF_CE;

architecture RTL of DFF_CE is
begin
  process ( Clk, Clr )
  begin
    if Clr = '1' then
      Q <= '0';
    elsif rising_edge(Clk) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
    
```

Weselej się robi gdy mamy do czynienia z asynchronicznymi sygnałami ustawiania lub kasowania. Po prawej stronie obrazka mamy kod przerzutnika z dodatkowym wejściem asynchronicznego kasowania. Na liście wrażliwości musi pojawić się sygnał „Clr”, bo jego wartość aktywna ma wyzerować natychmiast przerzutnik. Sygnał ten ma priorytet większy od „Clk”. Proces może wykonać się w trzech przypadkach (zmiana obu sygnałów, zmiana tylko „Clk” lub zmiana tylko „Clr”).

Nieco inaczej sprawy się mają gdy mamy do czynienia z synchronicznym sygnałem resetującym. Na liście wrażliwości procesu nie ma sygnału „Reset”. Sygnał „Reset” jest próbkowany tylko w momentach wystąpienia zbocza narastającego sygnału zegarowego. Tutaj „Reset” ma większy priorytet od wejścia „D”. Gdy „Reset” ma wartość aktywną, to następuje wyzerowanie sygnału „Q”. Jeżeli „Reset” jest nieaktywny, to sprawdza się jaką wartość ma sygnał „D”:

```
entity DFF_RE is
  port( D   : in  STD_LOGIC;
        Rst : in  STD_LOGIC;
        CE  : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Q   : out STD_LOGIC );
end DFF_RE;

architecture RTL of DFF_RE is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if Rst = '1' then
        Q <= '0';
      elsif CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```

Umieszczanie w treści architektury różnych konstrukcji odpowiada rysowaniu na schemacie różnych fragmentów projektu cyfrowego. Jeżeli mamy do stworzenia projekt zawierający przerzutnik z podłączoną na jego wejście logiką kombinacyjną.

W architekturze tworzymy sygnały wewnętrzne:

architecture ...

```
...
signal D1 : STD_LOGIC;
```

Sygnały kombinacyjne nie muszą mieć podanych wartości początkowych. W opisie należy dodać port wyjściowy:

```
signal Q1 : STD_LOGIC := '0';
```

**Sygnałom synchronicznym należy przypisywać wartość początkową.** Jeżeli nie przypiszemy wartości początkowej sygnałowi „Q1”, to będzie on miał wartość 'U', a to może nieźle namieszać w symulacji. Na syntezę nie będzie miało to żadnego wpływu.

Później opisuje się bloki z których składa się architektura. Przykładowo zaczynamy od opisu części kombinacyjnej:

```
begin
D1 <= (We1 and We2) or ( ... );
```

Niżej można przypisać jakąś wartość do „Q1”. Możemy używać przypisań różnego rodzaju. Po załatwieniu części kombinacyjnej umieszczamy w treści architektury procesy odpowiedzialne za działanie przerzutnika:

```
process (clk) is
begin
...
end process;
```

Proces będzie wyzwalany od sygnały „clk”. Tak jak poprzednio umieszczamy odpowiednie opisy zależnie od rodzaju przerzutnika który chcemy umieścić. Narzędzie XST potrafi rozpoznać rejestry, liczniki, różnego rodzaju układy akumulacyjne. W „XST User Guide” w rozdziale 2 znajdują się wskazówki jak należy w VHDL opisywać standardowe bloki tak, by narzędzie XST sobie poradziło z poprawną synteza.

*[A teraz wielki powrót do slajdu 21]*

## Instrukcja wyboru „case”

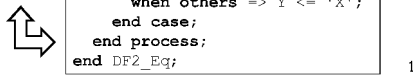
```
[ label: ] case expression is
  when choice1 =>
    statements
  when choice2 => \_ opt.
    statements    /
  ...
  when others => \_ opt. if all choices
    statements  / covered
end case [ label ] ;
```

Instrukcja przypomina tą, znaną z C++. Obliczana jest wartość wyrażenia, która jest porównywana z opcjami wyboru „choice1”, „choice2”. **Opcje wyboru muszą być rozłączne.** Muszą być kompletne. Występuje opcja

„others” zawierająca pozostałe możliwości. Notacja „=>” pełni tylko rolę składniową i nie jest jakkolwiek instrukcją przypisania. Nie ma czegoś takiego jak „wyskakiwanie” z opcji („break” znany z C++ i Javy), ponieważ po wykonaniu się danej opcji wychodzimy z instrukcji „case”.

Instrukcja „case” jest równoważna selektywnemu przypisywaniu:

<pre>architecture DF2 of MUX is begin   with Sel select     Y &lt;= A when "00",     B   when "01",     C   when "10",     D   when "11",     'X' when others; end DF2;</pre>	<pre>architecture DF2_Eq of MUX_4 is begin   process ( Sel, A, B, C, D )   begin     case Sel is       when "00" =&gt; Y &lt;= A;       when "01" =&gt; Y &lt;= B;       when "10" =&gt; Y &lt;= C;       when "11" =&gt; Y &lt;= D;       when others =&gt; Y &lt;= 'X';     end case;   end process; end DF2_Eq;</pre>
---	--



Jeśli mamy multiplekser 4 na 1 opisany funkcją przypisania selektywnego, to jest on z definicji równy procesowi w którym występuje jedna instrukcja „case”. Lista wrażliwości zawiera wszystkie sygnały występujące z prawej strony wyrażenia przypisania oraz sygnał selekcji.

Czasami zamiast „others” należy używać instrukcji „null” (jeżeli nie chcemy nic robić).

Dziś zakończymy omawianie zagadnień związanych z opisem sekwencyjnym. Należy wspomnieć o tym, że w języku VHDL każdą jednostkę opisujemy przy pomocy architektury. Zamieszczone tam instrukcje są wykonywane współbieżnie. Instrukcjami takimi są instrukcje przypisania, generacji oraz instrukcje procesu. Wewnątrz procesów zawarte są instrukcje wykonywane w sposób sekwencyjny. Mówiliśmy o tym, że procesy mogą mieć jawnie podaną listę wrażliwości (lista sygnałów, których zmiana powoduje wykonanie procesu) lub jawnie podane instrukcje „wait” wstrzymujące działanie procesu „aż coś się stanie”. Poruszono również temat instrukcji warunkowych. Odpowiednio użyte instrukcje warunkowe odpowiadają przypisaniu selektywnemu. Należy pamiętać o tym że według standardu każda instrukcja przypisania współbieżnego (występująca w architekturze) jest zamieniana na proces wyzwalany wszystkimi sygnałami występującymi po prawej stronie instrukcji przypisania współbieżnego.

Dzisiaj omówione zostaną kolejne instrukcje wykorzystywane w opisie sekwencyjnym.

### Pętle (Iteracje)

W języku VHDL są trzy instrukcje iteracyjne:

```
[ label: ] loop
    statements  -- use exit to abort
end loop [ label ] ;

[ label: ] for variable in range loop
    statements
end loop [ label ] ;

[ label: ] while condition loop
    statements
end loop [ label ] ;
```

Pierwsza pętla jest pętlą nieskończoną. Zestaw instrukcji między „loop” oraz „end loop” będzie wykonywany w nieskończoność. Z pętli tej musimy „wydostać się” w sposób jawny używając instrukcji „exit”.

Składnia drugiej pętli przypomina tą z poznanych już języków programowania. Różnica polega na zdefiniowaniu „na poczekaniu” zmiennej iterującej „variable” oraz zakresu zmian tej zmiennej. Definicja zakresu w postaci 0 to 7 spowoduje wzrost wartości zmiennej z każdym „przejściem” pętli. Zapis 7 downto 0 będzie oznaczał odliczanie od 7 w dół.

Ostatnia pętla jest pętlą „while” ze sprawdzaniem warunku na początku. Jeżeli chcielibyśmy używać pętli ze sprawdzaniem warunku na końcu, musielibyśmy użyć pętli pierwszej z jawnym użyciem instrukcji wyjścia w przypadku gdy jakiś warunek zostanie spełniony.

Etykiety w pętlach są opcjonalne.

### Instrukcje „next” i „exit”

Instrukcja „next” odpowiada poleceniu „continue” z języka C. Oznacza przerwanie bieżącej iteracji i przejście do następnej. Instrukcja może mieć etykietę informującą o pętli, której ta operacja dotyczy (potrzebne gdy pętle są zagnieżdżone):

```
next;
next outer_loop;
next when A>B;
next this_loop when C=D or A>B;
```

Odpowiednikiem „break” z języka C jest „exit”:

```
exit;
exit outer_loop;
exit when A>B;
exit this_loop when C=D or A>B;
```

Dotyczą jej te same uwagi co „next”. Można wskazać która pętla zostanie przerwana. Spowoduje to wyjście poza pętlę i przejście do następnej instrukcji sekwencyjnej. Przerwanie można uzależnić od spełnienia jakiegoś warunku.

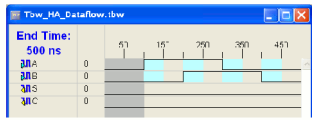
Posługiwanie się opisami sekwencyjnymi wiąże się z pewnym ryzykiem. Czasami takie opisy źle lub wcale się nie syntezują. By tego uniknąć, trzeba używać specyficznych konstrukcji które poprawnie zostaną rozpoznane przez oprogramowanie XST. Nie należy projektować układów w ten sposób, że w architekturze znajduje się instrukcja procesu, w której się „wszystko dzieje”. To może działać poprawnie w symulacji, ale synteza może być niewykonywalna.

Instrukcji „return” (powrotu z funkcji/procedury) nie będziemy omawiać. Może się ona przydać podczas tworzenia testbenchy. O instrukcjach sprawdzania asercji i debugowania też nie będziemy mówić.

## Jednostki symulacji

Opisy sekwencyjne występują w jednostkach symulacji. Języka VHDL używa się do opisywania środowiska symulacji. Jak to wygląda?

### ISE Testbench



```
entity HA_Testbench is
end HA_Testbench;
architecture TB_Arch of HA_Testbench is
    file RESULTS: (...);
    component HalfAdder
        port (
            A : in STD_LOGIC;
            B : in STD_LOGIC;
            S : out STD_LOGIC;
            C : out STD_LOGIC
        );
    end component;
    signal A : STD_LOGIC := '0';
    signal B : STD_LOGIC := '0';
    signal S : STD_LOGIC := '0';
    signal C : STD_LOGIC := '0';
```

```
begin
    UUT : HalfAdder
        port map ( A => A,
                  B => B,
                  S => S,
                  C => C );

    process
    begin
        -- Current TIME: 100ns
        wait for 100 ns;
        A <= '1';
        -- Current TIME: 200ns
        wait for 100 ns;
        B <= '1';
        -- Current TIME: 300ns
        wait for 100 ns;
        A <= '0';
        -- Current TIME: 400ns
        wait for 100 ns;
        B <= '0';
        -----
        wait for 100 ns;
        assert (...); -- NOT in v.9.2!
    end process;
end TB_Arch;
```

Będziemy posługiwać się układem półsumatora. Układ ma 2 wejścia A i B oraz dwa wyjścia. Zwykle testbench generujemy rysując wykresy poprzez klikanie myszą. Wszystko dalej dzieje się na poziomie języka VHDL. To, co widzi ModelSIM wygląda tak, że jednostki dla symulacji nie mają zdefiniowanych wejść i wyjść. Definiowana jest jednostka HA\_Testbench. Jednostka ta ma jedną architekturę w której dzieje się wszystko. Można zdefiniować plik służący do wyprowadzania rezultatów (`file RESULTS`) ale nie będziemy z tego korzystać. Najbardziej interesuje nas definicja komponentu. Powielone są tutaj porty, które są typu `STD_LOGIC`. Następna sekcja to definicja sygnałów, które nazywają się identycznie jak porty. W przypadku automatycznej generacji modelu sygnały te będą miały przypisane wartości początkowe w postaci zer.

Po prawej stronie slajdu znajduje się treść architektury. Składa się ona z dwóch instrukcji współbieżnych: instrukcji instancji komponentu oraz instrukcji procesu. Komponent przez nas symulowany jest wstawiany pod nazwą UUT (Unit Under Test). Będziemy się do tej nazwy odwoływać w ModelSIM-ie podczas poszukiwań sygnałów wewnętrznych lub portów symulowanego modułu. W instrukcji procesu zawarte są instrukcje odpowiedzialne za generowanie pobudzeń (przypisywanie wartości do portów wejściowych).

Struktura procesu jest trywialna. Nie posiada on listy wrażliwości. Jego działanie jest wstrzymywane w sposób jawny poprzez użycie instrukcji „wait”. Odpowiednie odcinki czasowe uzyskuje się dzięki `wait for 100 ns`. Po każdym wstrzymaniu jest wykonywane przypisanie.

Na końcu procesu wstawiało się instrukcję asercji, której zadaniem było zatrzymanie symulacji wyświetlenie komunikatu „This is not an error”. W starszych wersjach ISE było to stosowane zawsze. Od wersji 9 skasowano asercję. ModelSIM zawsze wykonuje symulację o długości 1µs po czym się zatrzymuje. Trzeba jawnie wydać polecenie „run” oraz wartość określającą jak długo będzie trwała symulacja.

Testbenche można pisać samodzielnie w języku VHDL. Wówczas wskazuje się jednostkę, która będzie symulowana. Spowoduje to wygenerowanie odpowiedniego szablonu z komponentem, sygnałami, instrukcją instancji na słowie `begin` (rozpoczynającym opis procesu) kończąc. Tak wygenerowany plik można modyfikować. Można dopisać instrukcję generującą sygnał zegarowy:

```
architecture
...
begin
    UUT:(...);
...
    Clk <= not Clk after 10 ns;
```

Da nam to sygnał zegarowy o częstotliwości 50 MHz o nieskończonej długości. Gdybyśmy chcieli dodać jeszcze jeden sygnał zegarowy, którego przebieg będzie przesunięty w fazie o 3 ns od poprzedniego, to potrzebne

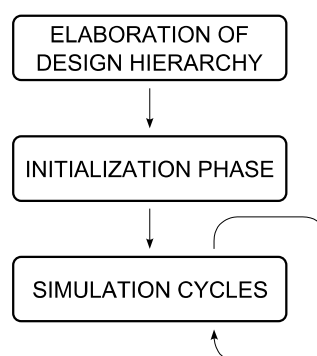
będzie stworzenie procesu:

```
process
begin
    wait for 3 ns;
    petla:
    Clk2 <= not Clk2;
    wait for 10 ns;
    end petla;
end process;
```

Opisywanie długich testbenchy przy pomocy języka VHDL sprawdza się szczególnie wtedy, gdy mamy do czynienia z „długimi” plikami pobudzeń.

## Model symulacji VHDL

Standard VHDL opisuje sposób symulacji opisu. Standard nic nie wspomina o syntezie i działaniu projektu w układzie po przeprowadzeniu procesu syntezy. Standard określa się terminami:



Opracowanie hierarchii projektu (Elaboration of design hierarchy) jest etapem wstępnym. Kolejnym krokiem jest etap inicjalizacji (Initialization phase). Na końcu mamy cykle symulacji (Simulation cycles) które się zapętłają (symulator działa w nieskończoność).

OPRACOWANIE HIERARCHII polega na kompilacji opisu modelu i stworzeniu kodu, który będzie nadawał się do symulacji. Każdy kompilator przekłada kod VHDL na własny, wewnętrzny model wykonywalny. Wykonywanie kodu oznacza symulację. Generowanie kodu do symulacji wiąże się z ustaleniem sterowników wszystkich sygnałów. Efektem symulacji jest określenie w jaki sposób będą zmieniały się wartości sygnałów i portów. Sterowniki są tworzone przez instrukcje przypisania sygnału. Gdy jeden sygnał ma wiele sterowników, używa się funkcji rozstrzygającej. Każdy sterownik jest tak naprawdę pamiętany w postaci listy transakcji. Transakcja to para „wartość/czas” (co oznacza: przypisz „wartość” po takim „czasie”). Jest to przewidywana fala prostokątna skojarzona z danym sterownikiem.

Musimy wprowadzić pojęcia ogólne:

**Sygnał aktywny** to sygnał którego sterownik ma transakcję w danym cyklu symulacji (w danym cyklu symulacji zostało wykonane przypisanie z danego sterownika do sygnału). To czy sygnał jest aktywny, czy nie można rozpoznać dzięki atrybutowi ACTIVE. Ma on wartość „prawda” gdy w danym cyklu symulacji sygnał jest aktywny (miał transakcję)

**Zdarzenie** to zmiana wartości sygnału w danym cyklu. Słowo „zmiana” jest ważne, ponieważ nie mówimy o zmianie wtedy, gdy do sygnału została przypisana jego poprzednia wartość. Zdarzenia wykrywa się przy użyciu atrybutu EVENT (zwraca „prawdę” gdy w danym cyklu symulacji sygnał zmienił swoją wartość). Atrybut ten spotkaliśmy na poprzednich zajęciach.

Będziemy posługiwać się oznaczeniami:  $T_C$  oznacza bieżący czas cyklu symulacji. Każdy symulator startuje od wartości 0, a potem ta wartość rośnie. Oznaczeniem  $T_N$  będziemy opisywać czas cyklu następnego.

INICJALIZACJA to przypisanie wartości początkowych wszystkim sygnałom. W omawianym wcześniej modelu mieliśmy jawne przypisane wartości początkowych. Wartości tych sygnałów będą propagować przez porty

wejściowe do wstawionej tam jednostki. Dalej propagacja obejmie sygnały wewnętrzne. W języku VHDL nie ma sygnałów niezainicjowanych. Sygnały mają albo wartości podane w sposób jawny (operator przypisania) albo są to domyślnie przypisane wartości „skrajnie lewe w danym typie sygnału”. W przypadku wartości liczbowych będzie to najmniejsza wartość z zakresu. W przypadku typów wyliczeniowych bierze się pierwszą pozycję. W typie STD\_ULOGIC pierwszą wartością jest 'U'. Należy pamiętać o tym, że sygnały, które nie zostaną przez nas jawnie zainicjowane będą miały przypisaną domyślnie wartość 'U'.

Faza inicjalizacji ma jeszcze jeden ważny element. Pozwala na przepropagowanie się wartości początkowych przez procesy. Wykonywane są jednokrotnie wszystkie procesy do momentu wstrzymania (gdy trafi na instrukcję „wait” lub na koniec procesu). Przed wejściem w pętlę symulacyjną pod wartość  $T_N$  podstawia się najbliższy moment transakcji (przeglądane są wszystkie listy transakcji i wybierany jest najbliższy moment wystąpienia transakcji) lub najbliższy moment wznowienia procesu (**wait for coŝtam**).

CYKL SYMULACJI składa się z następujących operacji:

- Podczas bieżącej symulacji jest podstawiany  $T_N$
- Aktualizacja sygnałów aktywnych w danym cyklu (konsekwencją będzie pojawienie się zdarzeń)
- Wykonanie procesów wyzwolonych zdarzeniami (procesy z listami wrażliwości) lub wstrzymanych do  $T_C$  instrukcją **wait**
- Podstawienie najbliższego momentu transakcji/wznowienia procesu pod  $T_N$

Mówi się, że gdy w ostatnim kroku wyznaczony  $T_N = T_C$  (następne zdarzenie jest dla tej samej chwili czasowej jak bieżący cykl symulacji), to następny cykl będzie tzw. „cyklem  $\Delta$ ”. Cykle  $\Delta$ , to takie cykle, w których czas nie posuwa się do przodu. Efekt jest taki, że symulator pętli się w jednej i tej samej chwili czasowej. Cykle  $\Delta$  opisują propagację zmian w układzie cyfrowym. Bierze się to z rozdzielania aktualizacji wartości sygnałów i wykonywania procesów. Nie ma współbieżności w wywoływaniu instrukcji przypisania i aktualizowaniu wartości sygnałów. W standardzie powiedziane jest że instrukcja przypisania sygnału nie modyfikuje jego wartości, tylko listę transakcji tego sygnału. Zapis **Coŝtam <= '0'** oznacza nieformalnie **Coŝtam <= '0' after  $\Delta$** . Należy więc pamiętać o tym, że przypisanie wartości jakiegoś sygnału będzie widoczne dopiero w następnym cyklu symulacji. Zmienne nie mają takich własności, ponieważ są inaczej traktowane. Poza tym na zmiennych nie da się opisać pracy układu cyfrowego. Podsumowując:

Każdy cykl symulacji składa się z dwutaktu. Najpierw aktualizuje się sygnały a potem wykonuje procesy. Oba kroki są rozdzielone. Jest to zgodne z rzeczywistym działaniem układów cyfrowych. Sygnały potrzebują czasu na propagację.

Zajmiemy się teraz przykładem. Przedstawione są instrukcje przypisania współbieżnego (na poziomie architektury):

#### Cykle symulacji - przykład

```
signal S1, S2 : STD_LOGIC;
(...)
S1 <= A and B;  -- A, B = porty in
S2 <= B xor S1;
```

Zakładamy że wartości początkowe sygnałów były następujące:

Czas	A	B	S1	S2
(...)	'1'	'1'	'1'	'0'

W chwili 10 ns port wejściowy A zmienił swoją wartość na '0':

Czas	A	B	S1	S2
(...)	'1'	'1'	'1'	'0'
10 ns	'0'	'1'	'1'	'0'

A'Event + transakcja dla S1

Co będzie działo się z instrukcjami przypisania współbieżnego? Będziemy na nie patrzeć jak na procesy. Zmiana na porcie A oznacza aktualizację wartości A w pierwszym etapie cyklu (dla 10 ns) i zostało wygenerowane zdarzenie (sygnał A będzie miał zdarzenie). Dalej zostanie wyzwolony proces przypisujący do sygnału S1. Zostanie obliczona wartość ('0') dla S1 i zostanie ona umieszczona na liście transakcji (również z czasem 10 ns). Właściwe przypisanie nastąpi dopiero w następnym cyklu (będzie to cykl  $\Delta$ ):

Czas	A	B	S1	S2	
(...)	'1'	'1'	'1'	'0'	
10 ns	'0'	'1'	'1'	'0'	A'Event + transakcja dla S1
10 ns + Δ	'0'	'1'	'0'	'0'	S1'Event + transakcja dla S2
10 ns + 2Δ	'0'	'1'	'0'	'1'	S2'Event

W momencie „10 ns + Δ” zostanie zdjęta transakcja z listy dla S1 i sygnał ten zostanie zaktualizowany. Zostanie dla niego wygenerowane zdarzenie, które wyzwoi przypisanie dla S2. Proces ten wygeneruje transakcję dla sygnału S2. Ponieważ w procedurze nie ma żadnego opóźnienia, to transakcja dostanie czas 10 ns. Symulator będzie cały czas trwał w chwili 10 ns. Kolejny cykl będzie również cyklem Δ. Wówczas zostanie zdjęta z listy i wykonana transakcja dla sygnału S2. Jeżeli zmiana S2 wywoływałaby kolejne instrukcje przypisania, to mielibyśmy kolejne instrukcje przypisania i kolejne cykle Δ. W przykładzie mieliśmy 3 cykle potrzebne do opisanie chwili 10 ns.

Należy pamiętać o tym że w VHDL-u nie ma natychmiastowych przypisań sygnałów. Istnieje opóźnienie Δ. Przypisanie (fizyczna zmiana sygnału po lewej stronie instrukcji przypisania) nastąpi dopiero w następnym cyklu.

Co się stanie jeżeli instrukcje współbieżne zamienimy na instrukcje sekwencyjne, umieszczone w procesie, który wyzwalany jest zmianami sygnałów A, B oraz S1?

```
signal S1, S2 : STD_LOGIC;
(...)
process (A, B, S1)
begin
  S1 <= A and B;
  S2 <= B xor S1;
end process;
```

Czy coś się zmieni? W kwestii cykli symulacji nic się nie zmieni. Analiza powyższego zapisu przebiegałaby identycznie jak poprzedniego. Gdyby wewnątrz procesu umieścić instrukcje testujące wartość sygnału S1, to wyniki testów będą zwracały poprzednią wartość tego sygnału. Nie powinno się mieszać wewnątrz jednego procesu przypisań i odczytów jednego i tego samego sygnału. Najlepiej będzie rozdzielać procesy na osobne kawałki, które pozwolą nad wszystkim zapanować. Najlepiej poświęcać osobne procesy do przypisywania wartości osobnych sygnałów. Podobnie z odczytywaniem.

*[Tymczasowy powrót do tematu układów synchronicznych]*

Na poprzednich zajęciach omówione zostały sposoby opisu układów synchronicznych. Pozostał jeszcze do opisanie układ rejestru przesuwającego SIPO (Serial In Parallel Out):

```
entity SReg8b is
  port ( Din : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR( 7 downto 0 ) );
end SReg8b;
architecture RTL of SReg8b is
  signal iQ : STD_LOGIC_VECTOR( 7 downto 0 );
begin
  Q <= iQ;
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      iQ( 7 downto 0 ) <= iQ( 6 downto 0 ) & Din;
    end if;
  end process;
end architecture;
```

Konieczne jest pracowanie na kopii wewnętrznej portu wyjściowego (odczyt portu wyjściowego jest postępowaniem błędnym). Zasada działania rejestru przesuwającego oparta jest na instrukcji przypisania:

```
iQ( 7 downto 0 ) <= iQ( 6 downto 0 ) & Din;
```

Z taką konstrukcją spotkaliśmy się przy omawianiu operatora konkatencji. Instrukcja ta musi być wykonywana synchronicznie. Objęta jest warunkiem testującym zbocze narastające zegara. Dalej bierzemy 6 młodszych bitów wektora „Q”. Z prawej strony doklejamy wartość „Din”. Tak powstanie przesunięcie w lewo. Przesuwanie w prawo wymagałoby modyfikacji kodu:

```
iQ( 7 downto 0 ) <= Din & iQ( 7 downto 0 );
```



Do poprzednio omówionych sposobów opisu różnych bloków funkcjonalnych układów cyfrowych dołożyć należy kolejne.

Zatrask:

```
entity latches_1 is
  port(G, D : in std_logic;
        Q : out std_logic);
end latches_1;

architecture archi of latches_1 is
begin
  process (G, D)
  begin
    if (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

Zatrask jest niepełnym przerzutnikiem. Przerzutnik wyzwala się zboczem zegarowym – zatrasku nie. Opis działania jest podobny do opisu działania przerzutnika. Działa tutaj proces wyzwalany zmianą sygnałów G i D. Nie ma w nim warunku testującego „czy sygnał G miał zdarzenie” (G'EVENT), bo oznaczałoby to że przypisanie odbywałoby się tylko w momencie zmiany sygnału G z '0' na '1'. W przypadku zatrasku przypisanie będzie wykonywane wtedy, gdy sygnał bramkujący G=1. Na liście wrażliwości znalazł się sygnał D, ponieważ każda zmiana tego sygnału musi być przeniesiona na wyjście Q.

Do zatrasku można dodać opcję asynchronicznego kasowania:

Zatrask z asynchronicznym kasowaniem:

```
architecture archi of latches_2 is
begin
  process (CLR, D, G)
  begin
    if (CLR='1') then
      Q <= '0';
    elsif (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

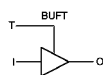
Na liście wrażliwości procesu znajdują się sygnały kasowania CLR, wejściowy G oraz bramkujący G. Najpierw sprawdzany jest warunek testujący aktywność sygnału CLR. Gdy jest on aktywny, na wyjście zostaje przypisane '0'. Niższy priorytet od sygnału CLR ma sygnał bramkujący i jeżeli G=1 to na wyjście układu zostanie przypisana wartość znajdująca się na wejściu układu.

Innym układem, z którego często się korzysta w układach cyfrowych jest bufor 3-stanowy:

Bufor trójstanowy:

```
entity three_st_2 is
  port(T : in std_logic;
        I : in std_logic;
        O : out std_logic);
end three_st_2;

architecture archi of three_st_2 is
begin
  C <= I when (T='0') else 'Z';
end archi;
```



Jest tu tylko jedna instrukcja przypisania na poziomie architektury. Jeżeli T=0 to na wyjście jest przypisywany sygnał z wejścia. W przeciwnym wypadku wyjście przechodzi w stan wysokiej impedancji (przypisanie wartości 'Z').

Innymi układami są liczniki. Opisuje się je bardzo łatwo dzięki przeciążonym operatorom dodawania. Nie musimy się zagłębiać w strukturę logiczną licznika. Ważne jest użycie biblioteki `ieee.std_logic_unsigned.all`. Konieczne jest używanie sygnału wewnętrznego tmp na którym w głównej mierze pracuje licznik. Układ jest asynchronicznie kasowany, co spowodowało zamieszczenie na liście wrażliwości procesu (obok sygnału zegarowego).

Taki sposób opisywania sprawdza się w przypadku liczników „zwykłych”. Jeżeli chcemy opracować liczniki

pracujące w kodzie niestandardowym, to trzeba to zrobić na maszynie stanów lub wprost na przerzutnikach.

Licznik z asynchronicznym kasowaniem:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_1 is
  port(C, CLR : in std_logic;
        Q : out std_logic_vector(3 downto 0));
end counters_1;

architecture archi of counters_1 is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (C, CLR)
  begin
    if (CLR='1') then
      tmp <= "0000";
    elsif (C'event and C='1') then
      tmp <= tmp + 1;
    end if;
  end process;

  Q <= tmp;
```

Inną wersją licznika może być układ z ładowaniem:

*JS UC 1*

Licznik ładowalny:

```
entity counters_3 is
  port(C, ALOAD : in std_logic;
        D : in std_logic_vector(3 downto 0);
        Q : out std_logic_vector(3 downto 0));
end counters_3;

architecture archi of counters_3 is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (C, ALOAD, D)
  begin
    if (ALOAD='1') then
      tmp <= D;
    elsif (C'event and C='1') then
      tmp <= tmp + 1;
    end if;
  end process;

  Q <= tmp;

end archi;
```

Nie ma on wejścia kasującego. Ładowanie odbywa się asynchronicznie. Gdy sygnał ALOAD jest aktywny, to do stanu licznika należy skopiować stan wejść. Działanie jest oparte na procesie, na którego liście wrażliwości są umieszczone wszystkie sygnały wejściowe układu. Inkrementowanie licznika następuje z każdym zboczem narastającym sygnału C.

Na koniec pozostał do omówienia licznik rewersyjny:

Licznik rewersyjny:

```
entity counters_6 is
  port(C, CLR, UP_DOWN : in std_logic;
        Q : out std_logic_vector(3 downto 0));
end counters_6;

architecture archi of counters_6 is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (C, CLR)
  begin
    if (CLR='1') then
      tmp <= "0000";
    elsif (C'event and C='1') then
      if (UP_DOWN='1') then
        tmp <= tmp + 1;
      else
        tmp <= tmp - 1;
      end if;
    end if;
  end process;

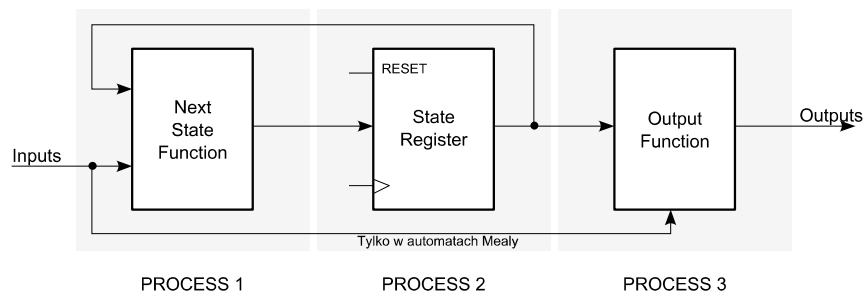
  Q <= tmp;
```

Ma asynchroniczne kasowanie. Aktywny sygnał CLR wymusza wyzerowanie stanu licznika. Jeżeli sygnał CLR jest nieaktywny, to z każdym zboczem narastającym zegara następuje dodawanie (sygnał UP\_DOWN='1') lub odejmowanie (UP\_DOWN='0') jedynki od aktualnego stanu licznika.

Aby proces syntezy się powiódł musimy podążać za szablonami. Jeżeli odchylimy się od tej ścieżki, to synteza może się nie powieść (mimo że zapis był poprawny i symulacja behawioralna pokazała prawidłowe działanie). Poza tym jakość narzędzia syntezy, z którego korzystamy jest nieciekawa. Dobre narzędzia syntezy niestety kosztują.

## FSM – Maszyny stanów

Najczęściej układ cyfrowy nie jest układem kombinacyjnym, realizującym funkcje logiczne. Częściej zachodzi potrzeba realizowania przez układ ściśle określonego algorytmu i od tego są FSM. Algorytm jest rozbity na szereg kroków, które mogą być rozłożone na operacje elementarne lub przedstawiają sobą uzależnienia czasowe.



Powyższy rysunek prezentuje istotę funkcjonowania niemal wszystkich układów cyfrowych.

Maszyna stanów musi być opisana w sposób umożliwiający narzędziu syntezy jej rozpoznanie. Robi się to poprzez opis na wysokim poziomie abstrakcji. Nie musimy przejmować się sposobem kodowania stanów, funkcjami wzbudzeń przerzutników. Narzędzie syntezy potrafi rozpoznać maszynę stanów, określić ilość jej stanów, ich kodowanie. Kodowanie stanów lepiej jest pozostawić narzędziu syntezy, które może stosować jakieś optymalizacje lub uproszczenia kodowania.

Stan maszyny przechowywany jest w rejestrze (State Register). Znajduje się tam wektor przerzutników reprezentujący stan układu. Jako osobną część kombinacyjną wydziela się część obliczającą stan następny (Next State Function). Sygnał z NSF wchodzi do wektora przerzutników i jest w nim zatraskiwany wraz z kolejnym taktiem zegara. Sygnał ten będzie nazywany później jako `next_state`. Z wektora przerzutników „wychodzi” sygnał `state`, który trafia na wejście bloku kombinacyjnego NSF. Pętla sprzężenia zwrotnego jest istotą przełączania się maszyny stanów. W bloku „Output Function” obliczana jest wartość funkcji wyjścia.

W przypadku automatu Mealy'ego „dochodzi” jeszcze połączenie bloku NSF z OF (bo w nim funkcja wyjściowa zależy od sygnału wejściowego).

Te trzy bloki opisuje się w postaci trzech rozłącznych procesów. Oczywiście jeżeli ktoś się uprze, będzie mógł zamieścić wszystko w jednym procesie, ale grozi to pojawieniem się bałaganu.

```
library ieee;
use IEEE.std_logic_1164.all;
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
          outp : OUT std_logic);
end entity;

architecture beh1 of fsm_3 is
    type state_type is (s1,s2,s3,s4);
    signal state, next_state, state_type : state_type;
begin
    process1: process (clk,reset)
    begin
        if (reset = '1') then
            state <= s1;
        elsif (clk = '1' and clk'event) then
            state <= next_state;
        end if;
    end process process1;

    process2: process (state, x1)
    begin
        next_state <= state;
        case state is
            when s1 => if x1='1' then
                        next_state <= s2;
                    else
                        next_state <= s3;
                    end if;
            when s2 => next_state <= s3;
            when s3 => next_state <= s4;
            when s4 => next_state <= s1;
        end case;
    end process process2;

    process3: process (state)
    begin
        case state is
            when s1 => outp <= '1';
            when s2 => outp <= '1';
            when s3 => outp <= '0';
            when s4 => outp <= '0';
        end case;
    end process process3;
end beh1;
```

Slajd powyżej przedstawia opis układu, zaczerpnięty z „XST Userguide”. Na przykładzie „maszynki, która nie wiadomo co robi” widzimy podstawowe cechy opisu maszyn stanów.

Układ ma 1-bitowe wejście i wyjście. W maszynie stanów zawsze warto zawrzeć sygnał „Reset”, który przyda się podczas uruchamiania projektu do „zapanowania” nad nim.

W architekturze widać definicję abstrakcyjnego typu przechowującego stan maszyny (`type state_type`). Stany definiuje się symbolicznie. Nazwy są nazwami własnymi i mogą być określane dowolnie. Układ ma 4 stany. Ważna jest definicja sygnałów `state` oraz `next_state`, które zostaną później zamienione na wektor. Następnymi elementami są procesy.

Process1 opisuje rejestr przechowujący stan układu. Pracuje on na sygnałach `state` i `next_state`. Na liście wra-

zliwości procesu znajdują się sygnały kasowania oraz zegarowy.

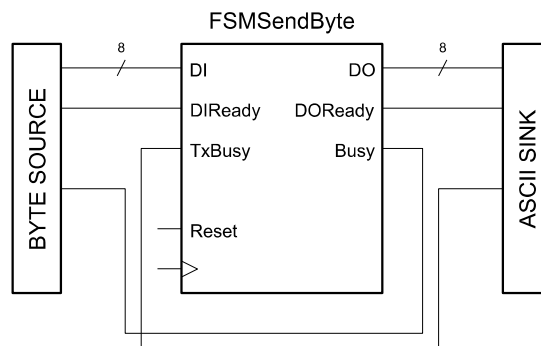
Next State Function opisana jest w process2. Można było tu umieścić zwykłe przypisania współbieżne, ale byłyby one bardzo długie i nieczytelne. Lepszym dla oka rozwiązaniem jest użycie procesu. Jest tam opis zachowania maszyny we wszystkich stanach. Najlepiej jest używać instrukcji wyboru case. Korzystanie z niej wymaga użycia wyczerpujących i rozłącznych opcji. Zmusza nas to do opisywania wszystkich stanów występujących w maszynie. Ważna jest linijka:

```
next_state <= state;
```

Jest to przypisanie domyślne, którego użycie jest zalecane. Domyślnie maszyna ma pozostać w stanie bieżącym (dzieje się to gdy nie jest spełniony żaden warunek mówiący o przełączeniu się automatu). Instrukcji tej mogło by nie być, ponieważ na przykładzie są podane warunki przełączenia dla każdego stanu. Jeżeli będziemy pisać bardziej złożoną maszynę stanów, to może się zdarzyć że nie skończymy listy tych warunków. Może to spowodować wygenerowanie zatrasku (który funkcjonalnie nam nie przeszkadza). Duża ilość takich zatrasków może zburzyć zależności czasowe i zaimplementowany projekt może nie działać.

Czas na process3 realizujący funkcję wyjściową. Działa tu funkcja kombinacyjna. Nie ma żadnego uzależnienia od zbrocza zegarowego. Zamiast procesu trzeciego można by napisać instrukcję przypisania współbieżnego i dla przykładowej konfiguracji było by to proste.

Ostatnim przykładem będzie układ, mający swoje zastosowanie w praktyce. Poruszymy problem transkodowania informacji między sobą. W projekcie będzie występował moduł będący źródłem bajtów (Byte Source). Strumień bajtów z tego urządzenia będziemy chcieli przekodować na znaki ASCII i wysłać do jakiegoś innego urządzenia (ASCII Sink). Konieczne będzie użycie transkodera „FSMSendByte”:



Sygnały tego układu to: sygnał zegarowy, sygnał kasujący oraz 8-bitową magistralę po której będą sływały dane wejściowe „DI”. Obecność nowego bajtu będzie sygnalizowana przez źródło sygnałem „DIRdy”. Impuls jednotaktowy na tej linii będzie oznaczał obecność nowego bajtu do wysłania. Potrzebny będzie sygnał „TxBusy” bo musimy wiedzieć czy odbiornik jest gotowy czy nie. Magistralą „DO” będą przesyłane do odbiornika dane po przekodowaniu. Po linii „DORdy” będą przekazywane impulsy do odbiornika, sygnalizujące obecność nowego bajtu do odebrania. Do źródła musi być wysyłany sygnał „Busy” informujący o zajętości układu FSM. Protokół zakłada że wszystkie moduły będą działać poprawnie. Nie ma tutaj handshakingu:

#### Przykład:

Moduł transkodujący otrzymany bajt do dwóch znaków ASCII

```
entity FSM_SendByte is
  port ( Clk : in STD_LOGIC;
        Reset : in STD_LOGIC;
        DI : in STD_LOGIC_VECTOR (7 downto 0);
        DIRdy : in STD_LOGIC;
        TxBusy : in STD_LOGIC;
        DO : out STD_LOGIC_VECTOR (7 downto 0);
        DORdy : out STD_LOGIC;
        Busy : out STD_LOGIC );
end FSM_SendByte;

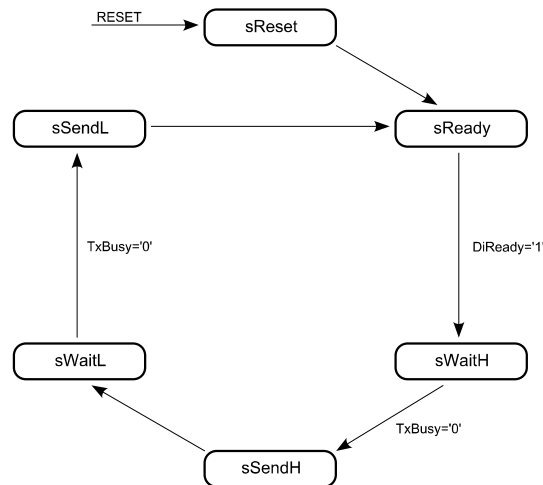
architecture RTL of FSM_SendByte is

  type state_Type is ( sReset, sReady, sWaitH, sSendH,
                      sWaitL, sSendL );

  signal State, NextState : state_Type;
  signal regDI : STD_LOGIC_VECTOR (7 downto 0);
  signal Ha_fByte : STD_LOGIC_VECTOR (3 downto 0);

begin
  (...)
```

Jeżeli chodzi o architekturę, to będzie ona się składać z trzech procesów. Musimy opracować jeszcze graf maszyny stanów. Graf jest (podobno) trywialny:



Potrzebny jest stan „sReset” do którego wchodzimy po podaniu sygnału resetującego. Gdy sygnał ten zniknie, przechodzimy do stanu „sReady”. Z tego stanu przechodzimy do „sWaitH” po pojawieniu się aktywnego sygnału „DIReady”. Stan „sWaitH” oznacza oczekiwanie na wysłanie starszej cyfry ASCII. Przejście do „sSendH” możliwe jest gdy odbiornik nie jest zajęty (TxBusy=0). W stanie tym wysyłamy starszą część kodu ASCII. Po wysłaniu zostanie wygenerowany impuls „DIReady”. Później czekamy na gotowość odbiornika do odebrania starszej cyfry kodu ASCII („sWaitL”). Do „sSendL” przejdziemy gdy odbiornik będzie gotowy na odbiór i to wszystko.

Wracamy do slajdu z kodem. Oprócz deklaracji portów w architekturze modułu zdefiniowane są sygnały wewnętrzne (State, NextState). Są tam również sygnały pomocnicze: regDI służy do zatrzaśnięcia w momencie startu stan magistrali „DI”. Dzięki temu po wystartowaniu maszyny stanów nadajnik bajtów będzie mógł zmienić zawartość magistrali „DI” i nie będzie musiał jej utrzymywać przez cały cykl pracy. To była część deklaracyjna. Przechodzimy dalej:

```

(...)

-- Input register (with CE)
regDI <= DI when rising_edge( Clk ) and State = sReady;

-- HalfByte selection
HalfByte <= regDI( 7 downto 4 ) when State = sSendH or
              State = sSendL
              else regDI( 3 downto 0 );

-- State register (with asynchronous reset)
process ( Clk, Reset )
begin
    if Reset = '1' then
        State <= sReset;
    elsif rising_edge( Clk ) then
        State <= NextState;
    end if;
end process;

(...)
    
```

Po pierwsze: zatrask do którego wprowadzane są dane z magistrali wejściowej. Przypisanie następuje w chwili pojawienia się narastającego zbocza sygnału zegarowego.

Sygnał obliczający półbajt działa jak multiplekser. Do tego sygnału dołączana jest starsza lub młodsza część rejestru regDI. Do wpisywania starszej połówki przełączamy się przy przejściu w stan „sSendH” lub „sSendL”, bo w tych stanach nadajnik ASCII będzie wysyłał starszy półbajt.

Kolejnym elementem jest rejestr stanu, zrealizowany w postaci procesu wyzwalanego sygnałem kasującym i narastającym zboczem sygnału zegarowego.

W opisie konieczne jest zawarcie informacji o przełączaniu się maszyny stanów:

```
(...)

-- Next state decoding
process ( State, DIRdy, TxBusy )
begin

    NextState <- State; -- default

    case State is
    when sReset =>
        NextState <= sReady;

    when sReady =>
        if DIRdy = '1' then
            NextState <= sWaitH;
        end if;

        when sWaitL =>
            if TxBusy = '0' then
                NextState <=
                    sSendL;
            end if;

        when sWaitH =>
            if TxBusy = '0' then
                NextState <= sSendH;
            end if;

        when sSendH ->
            NextState <- sWaitL;

        when sSendL ->
            NextState <- sReady;

    end case;
end process;

(...)
```

Ze stanu „Reset” natychmiast przechodzimy do „Ready” (natychmiast po zdjęciu sygnału kasującego). Kolejne przejścia odpowiadają odpowiednim krawędziom na grafie układu. Ostatnim elementem jest funkcja generująca sygnały wyjściowe:

```
(...)

-- Outputs
with HalfByte select
DO <= X"30" when "0000", -- 0-15 => ASCII '0'-'F'
      X"31" when "0001",
      X"32" when "0010",
      X"33" when "0011",
      X"34" when "0100",
      X"35" when "0101",
      X"36" when "0110",
      X"37" when "0111",
      X"38" when "1000",
      X"39" when "1001",
      X"41" when "1010",
      X"42" when "1011",
      X"43" when "1100",
      X"44" when "1101",
      X"45" when "1110",
      X"46" when others;
DIRdy <= '1' when State = sSendH or State = sSendL
        else '0';
Busy <- '1' when State /= sReady
        else '0';

end RTL;
```

Ogranicza się to do operacji przypisywania warunkowego. Sygnał „DORdy” będzie miał wartość aktywną, gdy układ znajdzie się w stanach „sSendH” lub „sSendL”. Do sygnału „Busy” przypisuje się wartość '1' wtedy, gdy znajduje się on we wszystkich stanach poza „sReady”.

Kodowanie stanów w maszynach FSM jest realizowane na zasadzie „1 z n”. Upraszcza to logikę obsługującą generowanie sygnałów zależnych od stanu układu.

Ostatnim zagadnieniem, jakie uda się nam omówić są:

## Układy ASIC

Wracamy do części wykładu dotyczącej programowalnych układów logicznych. Skrót ASIC rozwija się jako „Application Specified IC”. Są to układy w których użytkownik ma jakiś wpływ na ich strukturę logiczną. Wyróżnia się dwie główne kategorie układów:

- Układy programowalne maską – modyfikowane przez użytkownika przed ich fizycznym wykonaniem.
- Układy programowalne po wyprodukowaniu – programowalne przez użytkownika.

Programowanie maską wynika z procesu technologicznego produkcji układu scalonego. Maski określają geometrię poszczególnych warstw struktury półprzewodnikowej. Projektant (użytkownik) przygotowuje maski, które będą używane w kolejnych krokach produkcji układów. Są to układy, które są zaprogramowane przed ich fizycznym wykonaniem. Do producenta wysyła się projekty masek. Po wyprodukowaniu układu nie można już przeprogramować. Używa się tu technologii:

- Full Custom (układy z pełnym cyklem produkcyjnym) – tutaj wpływ użytkownika na projekt jest pełny w całym cyklu produkcyjnym.
- Semi Custom – tu są różne warianty wykonania:

- ◆ Standard Cells – układ stworzony z komórek standardowych.
- ◆ Gate Array – matryca bramek

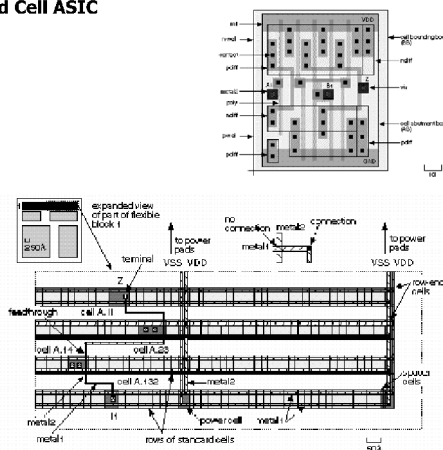
Jeżeli chodzi o układy programowane przez użytkownika to mamy do czynienia z:

- Układy PLD – PAL, PLA, PLE ...
- Układy CPLD – omawiana przez nas rodzina 9500
- Układy FPGA – najbardziej rozbudowane.

Układy programowalne maską pozwalają nam na ingerencję w strukturę układu przed jego wyprodukowaniem. Możemy wysłać do producenta maski układu zaprojektowanego przez nas na wszystkich warstwach (Full Custom) lub tylko części. Inaczej przebiega sprawa z układami programowalnymi przez użytkownika.

Układy z pełnym cyklem projektowym wykorzystywane są tylko tam gdzie nie ma gotowych rozwiązań, gdy zastosowanie układu ma miejsce w ekstremalnych warunkach. Technologia ta jest kosztowna. Oprogramowanie do projektowania układu musimy kupić u producenta, procedura projektowania jest długotrwała. Poza tym nikt nam nie daje gwarancji że nasz projekt będzie działał. Na wsparcie od producenta nie ma co liczyć. Wsad układów obejmuje serie układów rzędu dziesiątek tysięcy sztuk. Aby nieco ułatwić projektowanie używa się biblioteki standardowych komórek. Są to komórki przygotowane z gwarancją działania, odpowiednio rozmieszczone. Układ składa się z gotowych komórek.

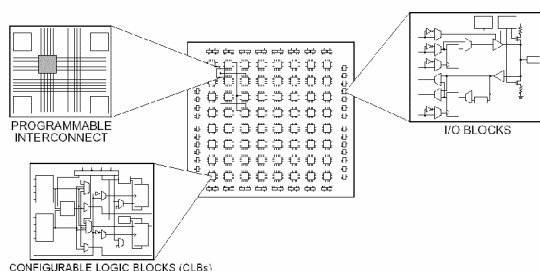
### Standard Cell ASIC



Nie ma tu dowolności w rozmieszczeniu. Są one ułożone w wierszach, mają one standardową wysokość. Szerokość może być zmienna. Największym problemem przy projektowaniu jest prowadzenie połączeń. By było to nieco ułatwione – są pozostawione specjalne kanały na ich prowadzenie.

Gdy korzystamy z matryc bramek, otrzymujemy gotowy „prefabrykat” w postaci gotowej matrycy bramek. Producenci mają katalogi matryc z zestawieniami „które są do czego”. My musimy tylko postarać się o projekt połączeń. Zaletą jest szybkość projektowania. Nie musimy projektować wszystkich warstw (tylko połączenia), produkcja jest szybsza i mamy większą gwarancję, że układ będzie działał poprawnie. Układy takie są tańsze od poprzednio opisanych.

## Field Programmable Gate Arrays (FPGA)



W układach FPGA mamy gotową matrycę bloków funkcyjnych (odpowiedników makrokomórek), mamy połączenia które użytkownik może sam programować.



Mozna jeszcze porównać technologie pod kątem kosztów. Koszt całkowity (koszt projektu) jest sumą kosztu stałego (przygotowania projektu) oraz kosztu jednostkowego układu pomnożonego przez ilość zamawianych scalaków.

#### Koszty stałe

	FPGA	G.A.	S.C.
Training (days)	2	5	5
\$400 / day	\$800	\$2,000	\$2,000
Hardware	\$10,000	\$10,000	\$10,000
Software	\$1,000	\$20,000	\$40,000
Design (10k gates)	500	200	200
gates / day	20	50	50
days	\$8,000	\$20,000	\$20,000
\$400 / day			
Production test design	0	5 days \$2,000	5 days \$2,000
NRE		\$30,000 (3+4)	\$70,000 (15+)
masks	0	\$10,000	\$50,000
simulation		\$10,000	\$10,000
test		\$10,000	\$10,000
Second source (projekt „awaryjny”)	(5days) \$2,000	(5days) \$2,000	(5days) \$2,000
Total	\$21,800	\$86,000	\$146,000

Koszty jednostkowe układów układają się odwrotnie. Koszt jednostkowy scalaka FPGA jest kilka razy większy niż koszt układu SC:

#### Koszty jednostkowe

	FPGA	G.A.	S.C.	
Wafer size	6	6	6	inches
Wafer cost	1,400	1,300	1,500	\$
Design size	10k	10k	10k	gates
Density	10k	20k	25k	g / cm <sup>2</sup>
Utilization	60%	85%	100%	
Die size	1.67	0.59	0.40	cm <sup>2</sup>
Die / wafer	88	248	365	
Defect density	1.10	0.90	1.00	def. / cm <sup>2</sup>
Yield	65%	72%	80%	
Die cost	25	7	5	\$
Profit margin	60%	45%	50%	
Price / gate	0.39	0.10	0.08	cents
Part cost	39	10	8	\$

Jeżeli mamy niewielką ilość sztuk, to duży koszt jednostkowy układu FPGA nas „nie boli”, bo mamy niskie koszty stałe. Jeżeli chcemy mieć więcej układów, to lepiej pozwolić sobie na większe koszty projektu układów SC które zwrócą się przez niskie koszty jednostkowe.

W koszty stałe wchodzi opłaty za sprzęt i oprogramowanie do projektowania układów. Dla układów SC oprogramowanie jest droższe. Konieczny jest trening w obsłudze software'u. Później musimy zapłacić za maski, testowanie układu i jego symulację.

Przy małych ilościach najbardziej opłacają się układy FPGA, przy średnich – GA. Jeżeli chcemy zaszaleć z ilością – wybieramy układy SC.