Architektury Komputerów - Zagadnienia

Filip Pasternak, Wojciech Kryściński

6 lutego 2014

TEORIA

Zagadnienie 1. *Magistrala*

Zespół linii oraz układów przełączających służących do przesyłania sygnałów między połączonymi urządzeniami w systemach mikroprocesorowych, złożony z trzech współdziałających szyn:

- **sterująca (kontrolna)** - połączenie między jednostką centralną i pamięcią oraz układem wejścia-wyjścia, które przenosi sygnały od mikroprocesora, określające jaki rodzaj operacji ma wykonać układ współpracujący (np. odczyt lub zapis pamięci)
- **adresowa (rdzeniowa)** - połączenie między jednostką centralną i pamięcią, które przenosi adres z/do miejsc, gdzie jednostka centralna chce czytać lub pisać. Liczba bitów szyny adresowej określa maksymalną wielkość pamięci, do jakiej procesor ma dostęp.
- **danych** - część magistrali odpowiedzialna za transmisję właściwych danych.

Jest elementem, który sprawia, że system komputerowy staje się określoną całością. Szerokość magistrali (liczba równoległych ścieżek szyny danych) określa, ile bitów może ona przesłać za jednym razem (w jednym takcie). Rozróżniane są 2 typy magistrali:

- jednokierunkowa (simplex) - dane przepływają tylko w jednym kierunku,
- dwukierunkowa (duplex)- dane mogą przepływać mogą w obu kierunkach,
  - dane mogą jednocześnie przepływać w obu kierunkach (full duplex),
  - dane w danym momencie mogą przepływać tylko w jednym kierunku (half duplex).

Zagadnienie 2. *Taktowanie*

Sposób sterowania pracą układów cyfrowych, polegający na dostarczeniu przez zegar sygnału elektrycznego (zwykle prostokątnego) o określonej częstotliwości. Układy takie wykonują jedną podstawową, jednostkową operację za każdym razem kiedy dotrze do nich impuls taktujący.

Zagadnienie 3. *Cykl maszynowy*

To cykl, podczas którego następuje wymiana danych między procesorem a pamięcią lub układem I/O: W każdym cyklu maszynowym następuje wysłanie:

- Adresu na magistralę adresową,
- Danych na magistralę danych,
- Sygnałów sterujących, informujących o rodzaju cyklu, na magistralę sterującą.

Jeden cykl maszynowy wykonywany jest w czasie jednego lub kilku taktów zegara.

Zagadnienie 4. *Cykl rozkazu*

(Przykładowe) składniki cyklu rozkazu:

A Wczytywanie rozkazu,

B Wczytywanie danych,

C Wykonanie rozkazu,

D Zapis wyniku.

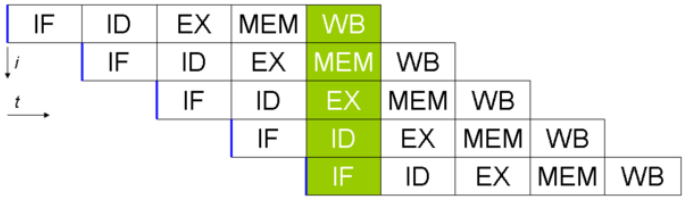
Każdy rozkaz wykonywany jest w kilku cyklach maszynowych. Przetwarzanie sekwencyjne (szeregowe):

$$A_1B_1C_1D_1A_2B_2C_2D_2... \rightarrow_{czas}$$

Zagadnienie 5. *Potokowość/Przetwarzanie potokowe*

Technika budowy procesorów polegająca na podziale logiki procesora odpowiedzialnej za proces wykonywania programu (instrukcji procesora) na specjalizowane grupy w taki sposób, aby każda z grup wykonywała część pracy związanej z wykonaniem rozkazu. Grupy te są połączone sekwencyjnie i wykonują pracę równocześnie, pobierając dane od poprzedniego elementu w sekwencji. W każdej z tych grup rozkaz jest na innym stadium wykonania. Można to porównać do taśmy produkcyjnej. W uproszczeniu, potok wykonania instrukcji procesora może wyglądać następująco:

1. Pobranie instrukcji z pamięci (instruction fetch - IF)
2. Zdekodowanie instrukcji (instruction decode - ID)
3. Wykonanie instrukcji (execute - EX)
4. Dostęp do pamięci (memory access - MEM)
5. Zapisanie wyników działania instrukcji (store; write back - WB)



Rysunek 1: Przetwarzanie potokowe.

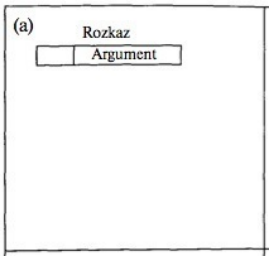
Zagadnienie 6. *Tryby adresowania*

Adresowanie natychmiastowe

Najprostsza forma adresowania w którym argument jest w rzeczywistości obecny w rozkazie.

$$ARGUMENT = A$$

Tryb ten może być stosowany do definiowania i używania stałych lub do ustalania początkowych wartości zmiennych. Liczba jest zwykle przechowywana w postaci uzupełnienia do dwóch; lewy bit pola argumentu jest używany jako bit znaku. gdy argument jest ładowany do rejestru danych, bit znaku jest rozszerzany na lewo w celu wypełnienia słowa danych. Zaletą adresowania natychmiastowego jest to, że żadne odniesienie do pamięci poza pobraniem rozkazu nie jest potrzebne do uzyskania argumentu, co pozwala zaoszczędzić jeden cykl pamięci głównej lub podręcznej w cyklu rozkazu. Wadą jest to, że rozmiar liczby jest ograniczony do rozmiaru pola adresowego, które w przypadku większości list rozkazów jest małe w porównaniu z długością słowa.

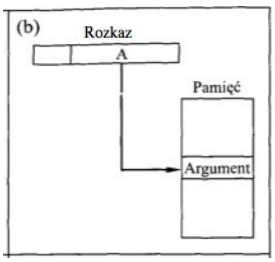


Rysunek 2: Adresowanie natychmiastowe

Adresowanie bezpośrednie
Pole adresowe zawiera efektywny adres argumentu:

$$EA = A$$

Metoda ta była powszechna we wcześniejszych generacjach komputerów i jest nadal spotykana w małych systemach komputerowych. Wymaga tylko jednego odniesienia do pamięci i nie wymaga żadnych obliczeń. Oczywistym ograniczeniem jest to, że umożliwia ona obsługę tylko ograniczonej przestrzeni adresowej.

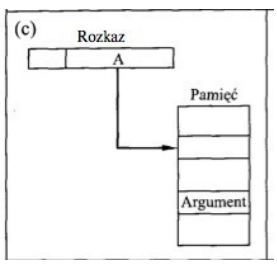


Rysunek 3: Adresowanie bezpośrednie

Adresowanie pośrednie
W przypadku adresowania pośredniego problemem, jest to, że długość pola adresowego jest zwykle mniejsza niż długość słowa, co ogranicza zakres adresów. W jednym z rozwiązań pole adresowe odnosi się do słowa w pamięci, które z kolei zawiera pełnej długości adres argumentu.

$$EA = (A)$$

(nawiasy oznaczają zawartość). Oczywistą zaletą tego rozwiązania jest to, że przy długości słowa N dostępna jest przestrzeń adresowa 2^N. Wadą jest to, że wykonanie rozkazu wymaga dwóch odniesień do pamięci w celu pobrania argumentu: jednego do otrzymania adresu, a drugiego do uzyskania samej wartości.



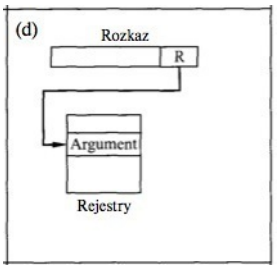
Rysunek 4: Pośrednie

Adresowanie bezpośrednie rejestrowe
Adresowanie bezpośrednie rejestrowe jest podobne do bezpośredniego. Jedyną różnicą jest to, że pole adresowe odnosi się do rejestru zamiast do adresu w pamięci głównej:

$$EA = R$$

Pole adresowe odnoszące się do rejestrów ma zwykle 3 lub 4 bity, wobec tego może się odwoływać do 8 lub 16 rejestrów roboczych.

Zaletą adresowania rejestrowego jest to, że w rozkazie potrzebne jest tylko niewielkie pole adresowe, oraz nie są wymagane odniesienia do pamięci. Czas dostępu do wewnętrznego rejestru procesora jest znacznie mniejszy niż czas dostępu do pamięci głównej.
Wadą adresowania jest bardzo ograniczona przestrzeń adresowa.

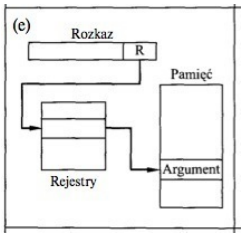


Rysunek 5: Bezpośrednie rejestrowe

Adresowanie pośrednie rejestrowe
Adresowanie pośrednie rejestrowe jest podobne do adresowania pośredniego. Różnicą jest to, czy pole adresowe odnosi się do lokacji w pamięci czy do rejestru. Wobec tego w przypadku pośredniego adresu rejestrowego:

$$EA = (R)$$

Zalety i ograniczeni pośredniego adresowania rejestrowego są w zasadzie takie same, jak adresowania pośredniego. W obu przypadkach ograniczenie przestrzeni adresowej (ograniczony zakres adresów) pola adresowego jest eliminowane w ten sposób, że pole to odnosi się do zawierającej adres lokacji o długości słowa. Ponadto, stosując pośrednie adresowanie rejestrowe, używa się o jedno od- niesienie do pamięci mniej niż przy adresowaniu pośrednim.



Rysunek 6: Pośrednie Rejestrowe

Zagadnienie 7. *Typy procesorów współczesnych.*

RISC (Reduced Instruction Set Computer) - architektura procesorów osiągających wysoką wydajność działania poprzez uproszczenie struktury procesora oraz jego rozkazów. Podstawowe cechy:

- Liczba rozkazów zredukowana do niezbędnego minimum (do kilkudziesięciu), podczas gdy w procesorach CISC sięga setek. Upraszcza to znacznie dekoder rozkazów,
- Redukcja trybów adresowania, dzięki czemu kody rozkazów są prostsze, bardziej zunifikowane, co dodatkowo upraszcza dekoder rozkazów,
- Ograniczenie komunikacji pomiędzy pamięcią a procesorem. Do przesyłania danych pomiędzy pamięcią a rejestrami służą dedykowane instrukcje,
- Zwiększenie liczby rejestrów, co ma wpływ na zmniejszenie liczby odwołań do pamięci.

W architekturze RISC generalnie wymaga się, aby rozkazy były na tyle proste, aby ich wykonanie było możliwe w jednym cyklu maszynowym. Cykl taki jest wyznaczony przez czas potrzebny do pobrania dwóch argumentów z odpowiednich rejestrów, wykonania operacji w ALU i umieszczenia wyniku w rejestrze. Podstawowym trybem adresowania jest adresowanie rejestrowe wskazujące bezpośrednio na rejestry argumentów i wyniku operacji.

CISC (Complex Instruction Set Computer) - architektura procesorów o następujących cechach:

- Występowanie złożonych, specjalistycznych rozkazów (instrukcji), które do wykonania wymagają od kilku do kilkunastu cykli zegara,
- Szeroka gama trybów adresowania,
- Rozkazy mogą operować bezpośrednio na pamięci (zamiast przesyłania wartości do rejestrów - RISC),
- Dekoder rozkazów jest skomplikowany.

Istotą architektury CISC jest to, iż pojedynczy rozkaz mikroprocesora wykonuje kilka operacji niskiego poziomu, np . pobranie z pamięci, operację arytmetyczną i zapis do pamięci.
Obecnie procesory Intel i AMD z punktu widzenia programisty są widziane jako CISC, ale ich rdzeń jest RISC-owy. Rozkazy CISC są rozbijane na mikrorozkazy, które są następnie wykonywane przez RISC-owy blok wykonawczy.

Zagadnienie 8. *Układ kombinacyjny*

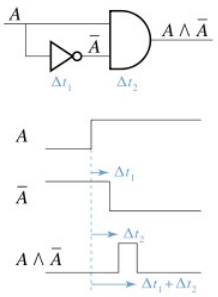
Układ (bramek logicznych) w którym wyjście zależy tylko od stanu wejść.

Zagadnienie 9. *Układ sekwencyjny*

Układ (bramek logicznych) w którym wyjście zależy od stanu wejść oraz poprzednich stanów.

Zagadnienie 10. *Zjawisko Hazardu*

Stan na wyjściu bramki cyfrowej nie zmienia się natychmiast po zmianie stanu wejść, lecz z pewnym opóźnieniem, zwanym czasem propagacji sygnału. Spowodowane jest to tym, iż elementy elektroniczne, tworzące bramkę cyfrową, pracują ze skończoną szybkością.
Hazardem nazywamy błędne stany na wyjściach układów cyfrowych, powstające w stanach przejściowych (przełączania).



Rysunek 7: Zjawisko Hazardu

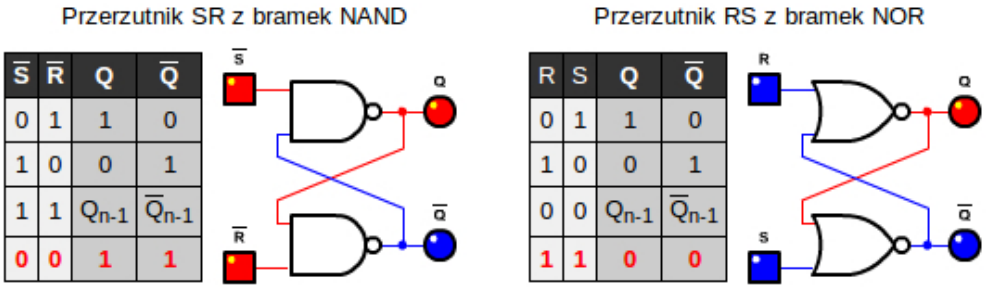
Zagadnienie 11. *Przerzutnik*

Jest układem cyfrowym wyposażonym w pamięć. W przypadku bramki cyfrowej stan jej wyjścia jest bezpośrednio uzależniony od stanów panujących na wejściach - opisuje to funkcja logiczna realizowana przez bramkę. W przerzutniku jest nieco inaczej - zapamiętuje on swój stan wewnętrzny. Stan ten może być zmieniony przez odpowiednie wysterowanie wejść.

Typowy przerzutnik jest układem cyfrowym posiadającym kilka wejść sterujących oraz dwa wyjścia komplementarne Q i $\bar{Q}$, na których panują zawsze przeciwne stany logiczne (z dokładnością do czasu propagacji sygnałów wewnątrz przerzutnika). Stan niski na wejściu S (Set - ustawianie) wymusza przejście wyjścia Q w stan 1. Z kolei stan niski na wejściu R (Reset - zerowanie) wymusza przejście wyjścia Q w stan 0. Stan wyjścia Q może się również zmieniać pod wpływem określonej kombinacji stanów wejść. Cechą charakterystyczną przerzutnika jest utrzymywanie ostatniego stanu logicznego na wyjściu Q po przejściu sygnałów sterujących w stan neutralny - jest to zatem element z pamięcią. Dzięki tej własności przerzutniki są powszechnie wykorzystywane do zapamiętywania stanów w układach cyfrowych (rejestry, liczniki, pamięci, układy sekwencyjne itp.).

Zagadnienie 12. *Przerzutnik RS*

Przerzutnik RS (Reset Set - Zeruj, Ustaw) jest najprostszym rodzajem przerzutnika, który można zbudować z dwóch dwu wejściowych bramek NOR lub NAND. Przerzutnik powstaje dzięki sprzężeniu zwrotnemu wyjść z wejściami. Sprzężenie zwrotne powoduje, iż przerzutnik utrzymuje ostatni stan wyjść Qn-1 po przejściu stanów logicznych na wejściach w stan neutralny. Poniżej przedstawiamy symulację przerzutnika RS z bramek NAND i NOR.



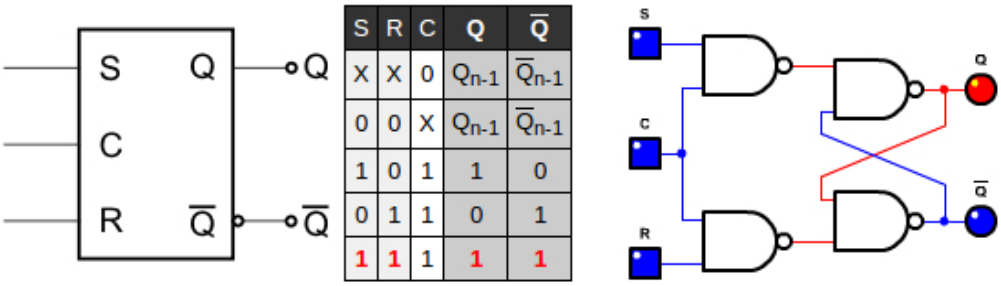
$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	1	1	0
1	0	0	1
1	1	Q_{n-1}	$\overline{Q}_{n-1}$
0	0	1	1

R	S	Q	$\overline{Q}$
0	1	1	0
1	0	0	1
0	0	Q_{n-1}	$\overline{Q}_{n-1}$
1	1	0	0

Rysunek 8: Budowa i stany przerzutnika RS

W przerzutniku SR zbudowanym z bramek NAND wejścia S i R są aktywne przy stanie 0. Stan 1 jest dla nich stanem neutralnym. Jeśli wejście S przejdzie w stan 0, to wymusi ono stan 1 na wyjściu Q. Przejście wejścia R w stan 0 wymusi stan 0 na wyjściu Q. Powrót wejść S i R do stanu neutralnego nie zmienia stanu logicznego wyjścia Q - przerzutnik zapamiętuje ustawiony stan logiczny. Jeśli oba wejścia S i R znajdą się w stanie niskim 0, będziemy mieli do czynienia ze stanem zabronionym - oba wyjścia Q i Q znajdą się w stanie wysokim 1. Powrót jednego z wejść S lub R do stanu neutralnego 1 wymusi odpowiedni stan przerzutnika. Problem jednakże pojawi się, jeśli oba wejścia S i R jednocześnie przejdą ze stanu 0 do stanu 1. W takim przypadku stan przerzutnika będzie zależał od wewnętrznych hazardów i wynik jest nieokreślony, tzn. na wyjściu Q może pojawić się zarówno stan 0 jak i stan 1 - nie da się przewidzieć, który z tych stanów ustali się w przerzutniku. (Analogicznie z przeciwnymi stanami wysokimi i niskimi działa przerzutnik na bramkach NOR - patrz tabelka).

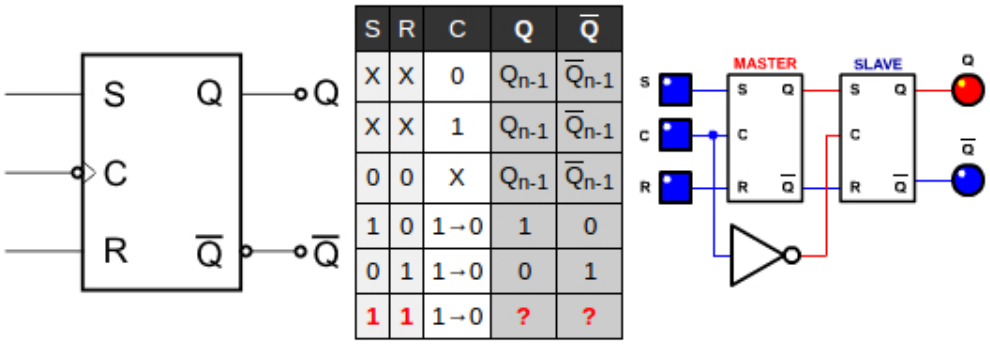
Synchroniczny przerzutnik RS Jednym ze sposobów zapobiegania hazardom w sieciach cyfrowych jest zastosowanie taktowania. Polega ona na tym, iż funkcje przełączające są sterowane dodatkowym sygnałem cyfrowym, zwanym sygnałem zegarowym lub taktem. Sygnał taktowania dociera jednocześnie do poszczególnych elementów sieci cyfrowej i umożliwia ich synchronizację - czyli jednoczesne, skoordynowane działanie. Poniżej prezentujemy synchroniczny przerzutnik RS wyposażony w dodatkowe wejście zegarowe C. Jeśli na wejściu zegarowym panuje stan niski, to wejścia R i S są odseparowane od przerzutnika. Zatem wszelkie zmiany ich stanów nie wpływają na stan wyjść przerzutnika. Pojawienie się stanu wysokiego na wejściu C odblokowuje wejścia R i S. Teraz stan wyjściowy przerzutnika może się zmieniać zgodnie z funkcjami sygnałów wejściowych R i S.



S	R	C	Q	$\overline{Q}$
X	X	0	Q_{n-1}	$\overline{Q}_{n-1}$
0	0	X	Q_{n-1}	$\overline{Q}_{n-1}$
1	0	1	1	0
0	1	1	0	1
1	1	1	1	1

Rysunek 9: Budowa i stany synchronicznego przerzutnika RS

Przerzutnik RS wyzwalany zboczem zegara Stan wyjść przerzutnika zmienia się zgodnie z jego definicją sygnałów sterujących tylko w krótkiej chwili, gdy sygnał zegarowy zmienia swój poziom, np. z 0 na 1 lub z 1 na 0. Unika się w ten sposób zakłóceń w pracy układów cyfrowych, które mogą wystąpić ze względu na hazardy pomiędzy zboczami sygnału zegarowego. Aby uzyskać taki sposób działania stosujemy dwa synchroniczne przerzutniki SR połączone następująco:



S	R	C	Q	$\overline{Q}$
X	X	0	Q_{n-1}	$\overline{Q}_{n-1}$
X	X	1	Q_{n-1}	$\overline{Q}_{n-1}$
0	0	X	Q_{n-1}	$\overline{Q}_{n-1}$
1	0	1→0	1	0
0	1	1→0	0	1
1	1	1→0	?	?

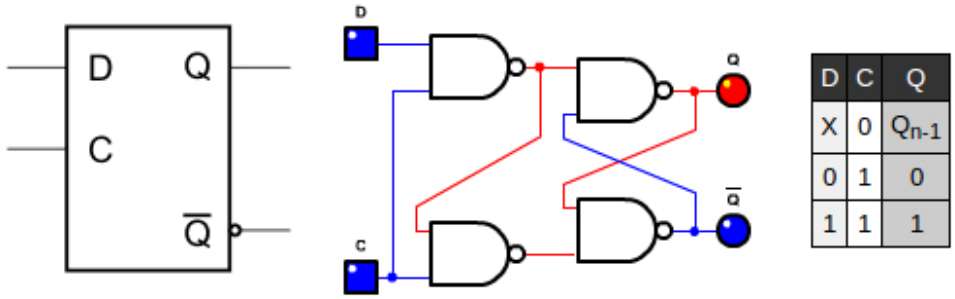
Rysunek 10: Budowa i stany przerzutnika RS wyzwalanego zboczem zegara.

Zadaniem przerzutnika Masterjest sterowanie drugim przerzutnikiem zwanym Slave. Do przerzutnika Slave dociera zanegowany sygnał zegarowy poprzez bramkę NOT. Dzięki temu w danym momencie zawsze aktywny jest tylko jeden z przerzutników.

Zagadnienie 13. *Przerzutnik D*

Przerzutnik D posiada wejście danych D (ang. Data), wejście zegarowe C oraz dwa komplementarne wyjścia Q i Q. Rozróżniamy dwa rodzaje przerzutników D, które różnią się sposobem pracy.

Przerzutnik D typu zatrzask (latch) W przerzutniku D latch stan wejścia D jest kopiowany na wyjście Q przy wysokim poziomie logicznym na wejściu C. Gdy poziom wejścia C zmieni się na niski, przerzutnik zapamiętuje ostatni stan wyjścia Q. Zmiany na wejściu informacyjnym D nie wpływają już na wyjście Q, które zostało źatrzaśnięteźmianą poziomu wejścia C.



D	C	Q
X	0	Q_{n-1}
0	1	0
1	1	1

Rysunek 11: Budowa i stany przerzutnika D typu zatrzask.

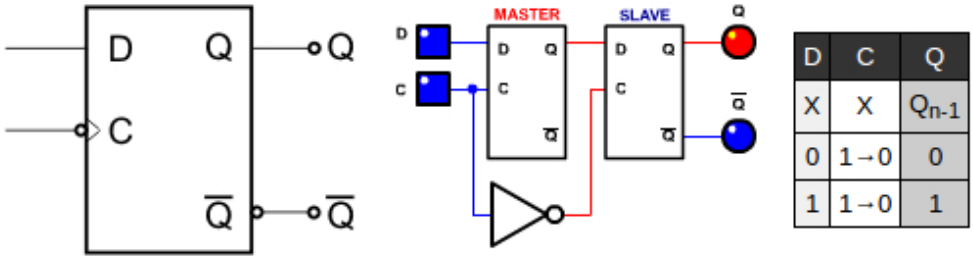
Ze schematu logicznego wynika, iż przerzutnik D Latch jest rozbudowanym przerzutnikiem SR. Rozbudowa polega na dodaniu dwóch bramek NAND sterujących wejściem ustawiającym S i zerującym R przerzutnika SR. Dzięki temu rozwiązaniu przerzutnik D nie posiada stanów zabronionych - nie dochodzi w nim do sytuacji, gdy oba wejścia S i R znajdują się jednocześnie w stanie niskim. Przerzutniki D Latch są zwykle stosowane w układach zapamiętujących stany logiczne (rejestry, pamięci, akumulatory, itp.).

Przerzutnik D wyzwalany zboczem zegara Drugi typ przerzutnika D jest wyzwalany zboczem (dodatnim lub ujemnym w zależności od rozwiązania) sygnału zegarowego. Oznacza to, iż przerzutnik zapamiętuje stan wejścia D tylko przy odpowiedniej zmianie poziomu logicznego na wejściu zegarowym C. Taki sposób pracy przerzutnika uzyskuje się łącząc dwa przerzutniki D typu zatrzask wg schematu Master-Slave.

Przerzutniki D wyzwalane zboczem są stosowane w układach licznikowych, rejestrach pamięciowych oraz w rejestrach przesuwnych.

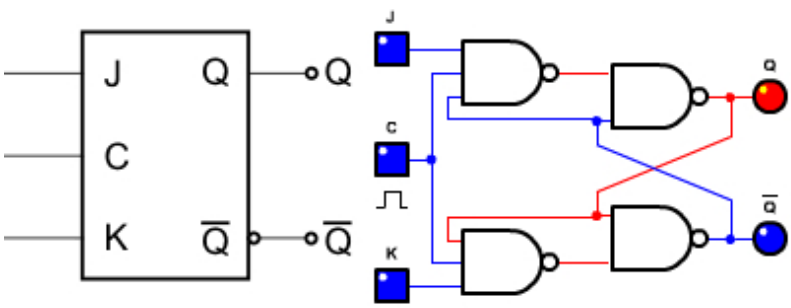
Zagadnienie 14. *Przerzutnik JK*

Przerzutnik J-K jest najpowszechniej stosowanym rodzajem przerzutnika cyfrowego z uwagi na swoją uniwersalność, która pozwala na łatwe zastosowanie w różnych układach cyfrowych. Przerzutnik posiada dwa wejścia sterujące J i K, jedno wejście zegarowe C oraz dwa komplementarne wyjścia Q i Q. Niektóre rozbudowane wersje tego przerzutnika posiadają dodatkowo dwa asynchroniczne wejścia PRESET (ustawia Q na 1) oraz CLEAR (ustawia Q na 0).



D	C	Q
X	X	Q_{n-1}
0	1→0	0
1	1→0	1

Rysunek 12: Budowa i stany przerzutnika D wyzwalanego zboczem.

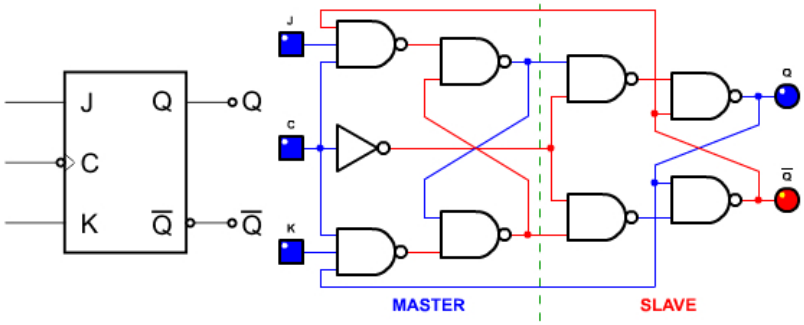


J	K	C	Q
X	X	0	Q_{n-1}
0	0	X	Q_{n-1}
1	0	1	1
0	1	1	0
1	1	⎍	$\overline{Q}_{n-1}$

Rysunek 13: Budowa i stany przerzutnika JK.

Przerzutnik J-K jest rozbudowanym przerzutnikiem S-R, do którego dodajemy dodatkowy człon z dwóch bramek sterujący sygnałami na wejściach S i R. Zadaniem tego członu jest uniemożliwienie wysterowania końcowego przerzutnika S-R sygnałami zabronionymi. Uzyskujemy to sprzęgając wejście S z wyjściem Q oraz wejście R z wyjściem Q. Ponieważ wyjścia Q i Q są komplementarne (o stanach przeciwnych), nigdy nie dojdzie do sytuacji, w której oba wejścia S i R znajdą się w stanie niskim.

Przerzutnik J-K Master/Slave Aby pozbyć się kłopotów z doborem czasu trwania impulsu zegarowego (ważne tylko dla J = K = 1), często stosuje się układ Master/Slave. Przerzutniki J-K Master/Slave są wyzwalane zboczem sygnału zegarowego, zatem nie wystąpią w nich problemy ze wzbudzaniem się układu.



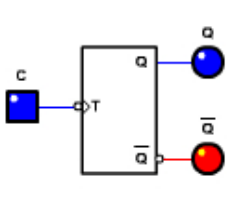
J	K	C	Q
X	X	0	Q_{n-1}
X	X	1	Q_{n-1}
0	0	X	Q_{n-1}
1	0	1→0	1
0	1	1→0	0
1	1	1→0	$\overline{Q}_{n-1}$

Rysunek 14: Budowa i stany przerzutnika J-K Master/Slave

Zagadnienie 15. *Przerzutnik T*

Przerzutnik T jest podstawowym elementem zliczającym. Przy każdym ujemnym (istnieją również rozwiązania przerzutnika T wyzwalane zboczem dodatnim) zboczu sygnału zegarowego zmienia stany wyjść na przeciwne.

Przerzutnik T nie jest produkowany w formie układu scalonego. Jednakże nie stanowi to żadnego problemu, gdyż można go w prosty sposób zbudować z przerzutnika D lub J-K. Przerzutnik T zbudowany z przerzutnika D jest wyzwalany zboczem dodatnim. Przerzutnik T zbudowany z przerzutnika J-K jest wyzwalany zboczem ujemnym.

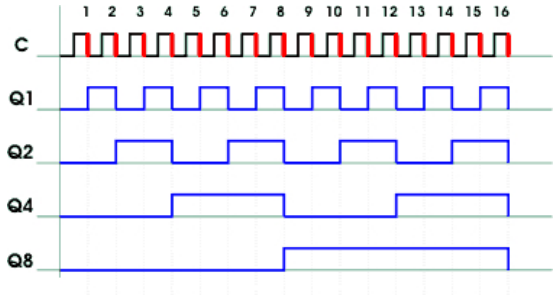


T	Q
0	Q_{n-1}
1	Q_{n-1}
1→0	$\overline{Q}_{n-1}$

Rysunek 15: Budowa i stany przerzutnika T.

Zagadnienie 16. *Licznik asynchroniczny*

Licznik asynchroniczny powstaje, gdy kilka przerzutników T połączymy szeregowo ze sobą tak, aby wyjście Q jednego przerzutnika łączyło się z wejściem T następnego. W liczniku asynchronicznym przerzutniki są sterowane wyjściami przerzutników poprzedzających. Powoduje to, iż stan licznika nie ustala się od razu, lecz kolejno na poszczególnych przerzutnikach z opóźnieniem równym czasowi propagacji sygnału w przerzutniku. Jeśli impulsy zegarowe mają dużą częstotliwość i ich okres jest porównywalny z czasem propagacji przerzutnika, to sygnały wyjściowe licznika mogą podawać złe wartości zliczonych impulsów - sygnał wyjściowy nie ma czasu na odpowiednie ustalenie się. Stan licznika odczytujemy z wyjść Q poszczególnych przerzutników.

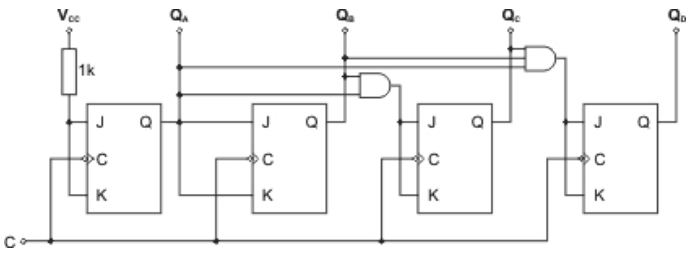


Rysunek 16: Symulacja 4-bitowego licznika asynchronicznego.

Kolorem czerwonym w sygnale zegarowym zaznaczono ujemne zbocza, które wyzwalają zmiany w pierwszym przerzutniku T. Ten z kolei steruje drugim przerzutnikiem T, itd. Po zliczeniu 16 impulsów licznik się zeruje.

Zagadnienie 17. *Licznik synchroniczny*

W takim liczniku przerzutniki zmieniają swój stan jednocześnie z taktem zegarowym. Licznik synchroniczny posiada sieć logiczną, która steruje odpowiednio wejściami przerzutników na podstawie stanów ich wyjść. Sygnał zegarowy doprowadzany jest do każdego przerzutnika, zatem zmiana stanów będzie odbywała się wg napływających taktów zegarowych.



Rysunek 17: Symulacja 4-bitowego licznika synchronicznego.

Wyjście Q_A zmienia swój stan na przeciwny po każdym impulsie zegarowym. Zatem do realizacji wyjścia Q_A wystarcza pojedynczy przerzutnik J-K MS pracujący jako przerzutnik T (oba wejścia J i K w stanie 1). Wyjście Q_B zmienia stan na przeciwny, gdy w poprzednim stanie wyjście Q_AQA znajdowało się w stanie 1. Do jego realizacji potrzeba pojedynczego przerzutnika J-K MS, którego wejścia J i K będą połączone z wyjściem Q_A pierwszego przerzutnika. Jeśli wyjście Q_A znajdzie się w stanie 1, to na oba wejścia J i K drugiego przerzutnika zostanie podany stan 1. Będzie on wtedy pracował jako przerzutnik T, zatem

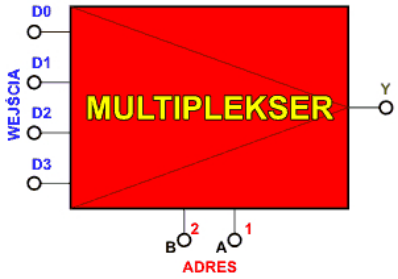
C	Wyjście			
	Q_D	Q_C	Q_B	Q_A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Rysunek 18: Tabela stanów 4-bitowego licznika synchronicznego.

przy kolejnym impulsie zegarowym zmieni swój stan na przeciwny. Jeśli wyjście Q_A przyjmie stan 0, to impuls zegarowy nie zmieni stanu wyjścia Q_B. Wyjście Q_C zmienia stan na przeciwny, gdy w poprzednim cyklu licznika wyjścia Q_A i Q_B jednocześnie były w stanie wysokim. Wejścia J i K trzeciego przerzutnika steruje się iloczynem logicznym stanów wyjść Q_A i Q_B. Wyjście Q_D zmienia stan na przeciwny, gdy w poprzednim cyklu wszystkie trzy wyjścia Q_A, Q_B i Q_C znajdowały się w stanie wysokim 1. Wejścia J i K czwartego przerzutnika steruje się zatem iloczynem logicznym tych wyjść i w ten sposób otrzymuje się 4 bitowy licznik synchroniczny.

Zagadnienie 18. *Multiplekser*

Multiplekser (selektor danych) jest układem cyfrowym posiadającym n wejść danych, jedno wyjście y oraz wejścia adresowe. Na wyjściu y pojawia się stan wejścia danych, którego numer podany został na wejścia adresowe. Przykładowy projekt dotyczy układu z czterema wejściami danych D0, D1, D2, D3, z jednym wyjściem Y oraz dwoma wejściami adresowymi A i B.



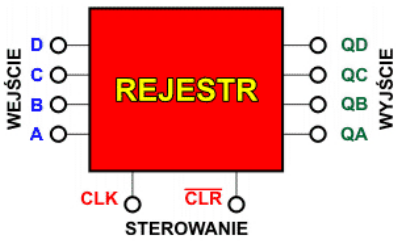
Rysunek 19: Ogólny schemat multipleksera o 4 bitach wejściowych i 2 bitach adresu.

B	A	D0	D1	D2	D3	Y
0	0	0	X	X	X	0
0	1	X	0	X	X	0
1	0	X	X	0	X	0
1	1	X	X	X	0	0
0	0	1	X	X	X	1
0	1	X	1	X	X	1
1	0	X	X	1	X	1
1	1	X	X	X	1	1

Rysunek 20: Tabela stanów wyjścia multipleksera w zależności od stanów wejść.

Zagadnienie 19. *Rejestr*

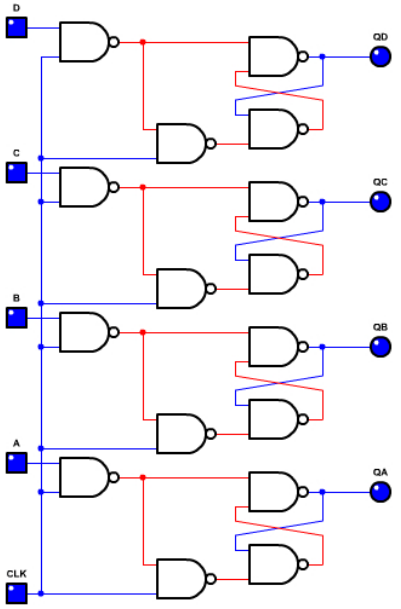
Rejestr jest układem cyfrowym służącym do zapamiętywania określonej porcji bitów danych. Rejestry stosuje się tam, gdzie występuje potrzeba chwilowego przechowania niewielkiej ilości informacji binarnej (np. wynik pewnej operacji arytmetycznej lub logicznej). Rejestry budowane są z przerzutników.



Rysunek 21: Ogólny schemat rejestru.

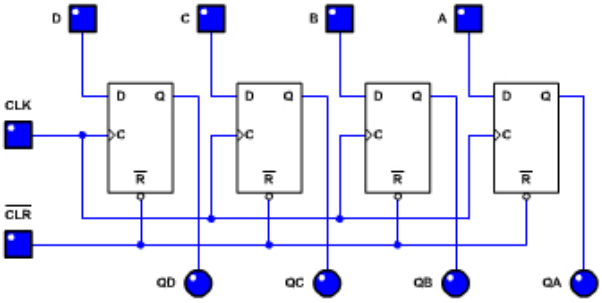
Sygnały wejściowe A, B, C i D podają informację do zapamiętania w rejestrze. Wejście CLK jest wejściem zapisującym informację z wejść A...D do rejestru. W zależności od typu zastosowanych przerzutników zapis może następować przy zmianie poziomu logicznego na wejściu CLK z 0 na 1 (zbocze dodatnie) lub z 1 na 0 (zbocze ujemne). Informacja przechowywane w rejestrze pojawia się na wyjściach QA, QB, QC i QD. Stan niski na wejściu CLR powoduje wyzerowanie wszystkich wyjść QA...QD rejestru.

Rejestr na bazie przerzutników D typu zatrzask.
Gdy sygnał zegarowy CLK przyjmuje poziom logiczny 1, odblokowane zostają bramki wejściowe i dane z wejść A...D przedostają się na wejścia przerzutników. Wejścia S i R przerzutników są zawsze wysterowane przeciwnymi sygnałami logicznymi, nie pojawi się zatem stan niedozwolony (S = 0 i R = 0). Przerzutnik SR ustawiają się zgodnie z sygnałami wejściowymi. Gdy sygnał zegarowy CLK powróci do stanu 0, na wejściach przerzutników pojawiają się sygnały neutralne i przerzutniki pamiętają ostatnio wprowadzony stan. W czasie trwania impulsu zegarowego stan wejść przenosi się na wyjścia. Jeśli dowolne wejście zmieni swój stan, to odpowiadające mu wyjście również się odpowiednio zmieni. Zatrzaśnięcie informacji następuje dopiero po przejściu sygnału zegarowego CLK w stan niski. W tym stanie zmiany wejść nie przenoszą się na wyjście - rejestr pamięta informację.



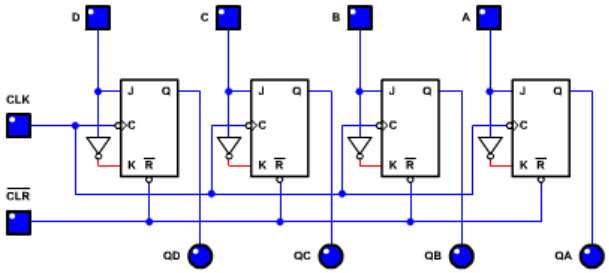
Rysunek 22: Schemat rejestru zbudowanego z bramek NAND tworzących przerzutniki RS pracujących jako D zatrzask.

Rejestr na bazie przerzutników D wzbudzanych zboczem. Jeśli do budowy rejestru zastosujemy przerzutniki D wzbudzane zboczem, to przepisanie danych wejściowych A...D do rejestru wystąpi przy narastającym zboczu sygnału zegarowego CLK. Przy ustalonym poziomie logicznym na wejściu CLK dane wejściowe A...D nie wpływają na pamiętaną przez rejestr informację.



Rysunek 23: Schemat rejestru zbudowanego z przerzutników D wzbudzanych zboczem.

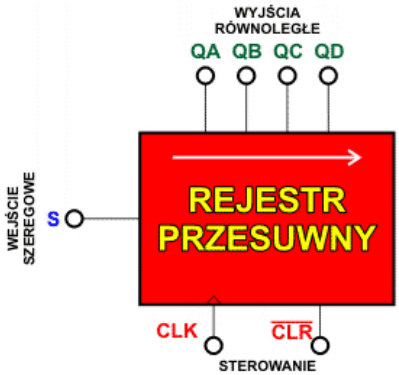
Rejestr na bazie przerzutników J-K. Budowa zwykłych rejestrów z przerzutników J-K jest oczywiście możliwa, lecz mniej korzystna od budowy rejestrów z przerzutników D, ponieważ musimy zastosować dodatkowe inwertory na wejściach K (przekształcenie przerzutnika J-K w przerzutnik D). Zalety przerzutników J-K ujawniają się przy bardziej zaawansowanych konstrukcjach. Wpis informacji z wejść A...D do przerzutników następuje przy ujemnym zboczu CLK.



Rysunek 24: Schemat rejestru zbudowanego z przerzutników J-K.

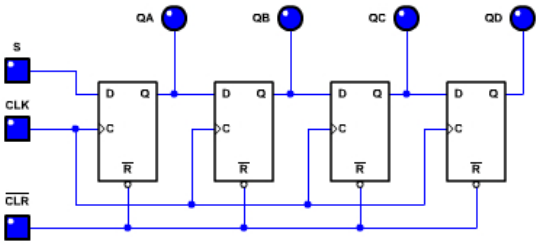
Zagadnienie 20. *Rejestr przesuwny*

Rejestr przesuwny jest rejestrem zbudowanym z przerzutników połączonych ze sobą w taki sposób, iż w takt impulsów zegarowych przechowywana informacja bitowa przemieszcza się (przesuwa) do kolejnych przerzutników. Rejestry przesuwne mogą być budowane z przerzutników synchronicznych D wzbudzanych zboczem (D-zatrzask nie nadają się do tego celu), RS lub JK.



Rysunek 25: Schemat ogólny rejestru przesuwnego.

Dane z wejścia S są wprowadzane do rejestru przy dodatnim zboczu sygnału zegarowego CLK (jeśli do konstrukcji rejestru zastosujemy przerzutniki J-K, to wpis może następować przy zboczu ujemnym). Jednocześnie dane z wyjść Q_A do Q_D zostają przesunięte o jeden bit w prawo. Na zwolnionym wyjściu Q_A pojawią się dane wprowadzone z wejścia S. Wejście sterujące CLR umożliwia wyzerowanie rejestru.



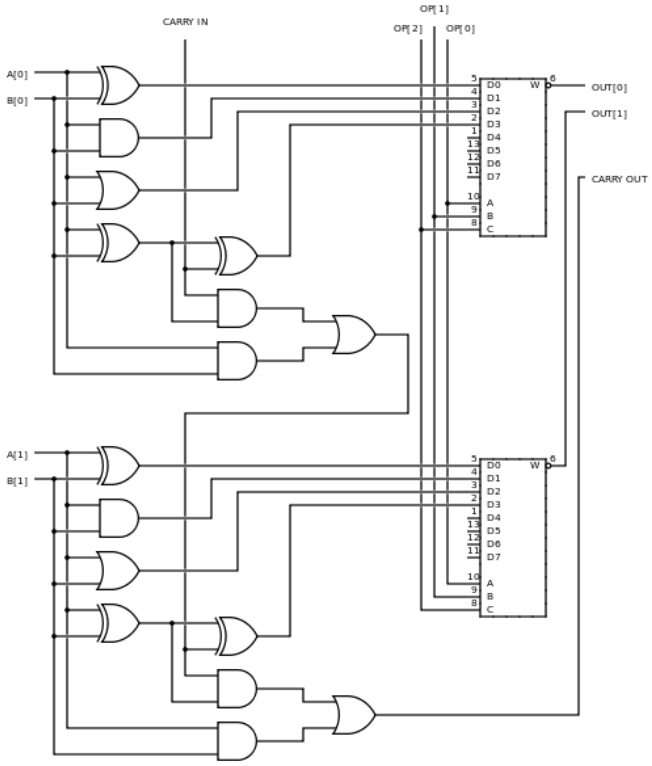
Rysunek 26: Schemat rejestru przesuwnego opartego na przerzutnikach D.

Zagadnienie 21. *ALU*

Jednostka arytmetyczno-logiczna to jedna z głównych części procesora, prowadząca proste operacje na liczbach całkowitych. Większość ALU potrafi dokonać następujących operacji:

1. operacje arytmetyczne na liczbach całkowitych (dodawanie, odejmowanie i czasami mnożenie oraz dzielenie, jednak te są bardziej kosztowne)
2. bitowe operacje logiczne (AND, NOT, OR, XOR)
3. operacje przesuwania bitowego (przesuwanie lub obracanie słowa o określoną liczbę bitów w lewo lub prawo, ze znakiem lub bez).
4. Przesuwanie może być rozumiane jako mnożenie przez 2 oraz dzielenie przez 2.

Jednostka ALU pobiera dane z rejestrów procesora. Wejściami ALU są dane na których się operuje (operandy) oraz algorytm z jednostki sterującej, który wskazuje którą operację należy wykonać. Wyjściem jest wynik obliczenia. Inne układy przesyłają dane pomiędzy tymi rejestrami a pamięcią. W wielu modelach, ALU generuje jako wejścia lub wyjścia zbiór kodów warunkowych z lub do rejestru statusowego. Kody te używane są do wskazywania takich procesów jak przeniesienie, przepełnienie, dzielenie przez 0 itp.



Rysunek 27: Prosta 2-bitowa jednostka arytmetyczno-logiczna, wykonująca operacje AND, OR, XOR oraz dodawania.

VHDL

Opis języka

Predefiniowane typy

- BOOLEAN - przyjmuje wartości TRUE lub FALSE,
- BIT - przyjmuje wartości ’1’ lub ’0’,
- BIT_VECTOR - ciąg wartości typu BIT np. ’00000000’, można użyć do modelowania magistral,
- CHARACTER - przyjmuje znaki alfanumeryczne,
- STRING - przyjmuje ciągi znaków,
- INTEGER - przyjmuje liczby całkowite
- REAL - przyjmuje liczby zmiennopozycyjne,

Logika wielowartościowa

Logika wielowartościowa posiada więcej typów niż tylko zero i jedynka logiczna. Pakiet *std_logic_1164* zawiera definicję typów *std_logic, std_logic_vector*, itd, które mogą przyjmować wartości

- ’U’ - wartość nigdy dotychczas nie została określona,
- ’X’ - wartość była znana ale aktualnie nie można podać jej konkretnej wartości,
- ’0’ - sygnał logicznego 0,
- ’1’ - sygnał logicznej 1,
- ’Z’ - stan wysokiej impedancji

Komponenty

ENTITY

Opis komponentów składowych projektu wykonywany jest za pomocą dyrektywy ENTITY opisującej sprzęg w strukturze hierarchicznej, bez definiowania zachowania. Jest odpowiednikiem symbolu na schemacie.

Blok ENTITY zawiera definicje sygnałów wejściowych i wyjściowych komponentu. Może również zawierać definicje parametrów i stałych.

```
entity identifier is
    generic ( generic_variable_declarations ) ; -- optional
    port ( input_and_output_variable_declarations ) ;
    [ declarations , see allowed list below ]   -- optional
begin                                          \__ optional
    [ statements , see allowed list below ]    /
end entity identifier ;
```

ARCHITECTURE

Zachowanie komponentu opisane jest w bloku ARCHITEKTURE. Po słowie ARCHTECTURE znajduje się zdefiniowana przez użytkownika nazwa. Blok ARCHTECTURE jest zawsze związany z blokiem ENTITY. Znaki '<=' są symbolami przypisania nowych wartości do sygnałów. Każdy blok ENTITY może posiadać kilka, różniących się nazwami, bloków ARCHTECTURE. Umożliwia to opisanie projektu na różnych poziomach abstrakcji.

```
architecture identifier of entity_name is
    [ declarations , see allowed list below ]
begin -- optional
    [ statements , see allowed list below ]
end architecture identifier ;
```

Instrukcje warunkowe, pętle

IF

```
if CONDITION then
   -- sequential statements
else
   -- sequential statements
end if;
```

SWITCH

```
case OBJECT is
   when VALUE_1 =>
      -- statements
   when VALUE_2 =>
      -- statements...
end case;
```

FOR

```
for I in 0 to 3 loop
    -- statements
end loop;
```

PRZYKŁADY

Bramka AND

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

entity AND_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end AND_ent;

architecture behav1 of AND_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='1') and (y='1')) then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;
```

Bramka OR

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

entity OR_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end OR_ent;

architecture OR_arch of OR_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='0') and (y='0')) then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;

end OR_arch;
```

Przerzutnik D

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_FF_VHDL is
    port
    (
        clk : in std_logic;

        rst : in std_logic;
        pre : in std_logic;
        ce  : in std_logic;

        d : in std_logic;
        q : out std_logic
    );
end entity D_FF_VHDL;

architecture Behavioral of D_FF_VHDL is
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            if (rst='1') then
                q <= '0';
            elsif (pre='1') then
                q <= '1';
            elsif (ce='1') then
                q <= d;
            end if;
        end if;
    end process;
end architecture Behavioral;
```

Przerzutnik JK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity JK_FF_VHDL is
   port( J,K: in  std_logic;
         Reset: in std_logic;
         Clock_enable: in std_logic;
         Clock: in std_logic;
         Output: out std_logic);
end JK_FF_VHDL;

architecture Behavioral of JK_FF_VHDL is
   signal temp: std_logic;
begin
   process (Clock)
   begin
      if (Clock'event and Clock='1') then
         if Reset='0' then
            temp <= '0';
         elsif Clock_enable ='0' then
            if (J='0' and K='0') then
               temp <= temp;
            elsif (J='0' and K='1') then
               temp <= '0';
            elsif (J='1' and K='0') then
               temp <= '1';
            elsif (J='1' and K='1') then
               temp <= not (temp);
            end if;
         end if;
      end if;
   end process;
   Output <= temp;
end Behavioral;
```

4-Bitowy Sumator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Adder is
    port
    (
        nibble1, nibble2 : in unsigned(3 downto 0);

        sum       : out unsigned(3 downto 0);
        carry_out : out std_logic
    );
end entity Adder;

architecture Behavioral of Adder is
    signal temp : unsigned(4 downto 0);
begin
    temp <= ("0" & nibble1) + nibble2;
    sum       <= temp(3 downto 0);
    carry_out <= temp(4);
end architecture Behavioral;
```

4-Bitowy ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU_VHDL is
    port
    (
        Nibble1, Nibble2 : in std_logic_vector(3 downto 0);
        Operation : in std_logic_vector(2 downto 0);
        Carry_Out : out std_logic;
        Flag : out std_logic;
        Result : out std_logic_vector(3 downto 0)
    );
end entity ALU_VHDL;

architecture Behavioral of ALU_VHDL is
    signal Temp: std_logic_vector(4 downto 0);
begin
    process(Nibble1, Nibble2, Operation, temp) is
    begin
        Flag <= '0';
        case Operation is
            when "000" => -- res = nib1 + nib2, flag = carry = overflow
                Temp   <= std_logic_vector((unsigned("0" & Nibble1) + unsigned(Ni
                Result <= temp(3 downto 0);
                Carry_Out   <= temp(4);
            when "001" => -- res = |nib1 - nib2|, flag = 1 iff nib2 > nib1
                if (Nibble1 >= Nibble2) then
                    Result <= std_logic_vector(unsigned(Nibble1) - unsigned(Nibble
                    Flag   <= '0';
                else
                    Result <= std_logic_vector(unsigned(Nibble2) - unsigned(Nibble
                    Flag   <= '1';
                end if;
            when "010" =>
                Result <= Nibble1 and Nibble2;
            when "011" =>
                Result <= Nibble1 or Nibble2;
            when "100" =>
                Result <= Nibble1 xor Nibble2;
            when "101" =>
                Result <= not Nibble1;
            when "110" =>
                Result <= not Nibble2;
            when others => -- res = nib1 + nib2 + 1, flag = 0
                Temp   <= std_logic_vector((unsigned("0" & Nibble1) + unsigned(no
                Result <= temp(3 downto 0);
                Flag   <= temp(4);
        end case;
    end process;
end architecture Behavioral;
```

Operation	Result	Flag	Description
000	Nibble1 + Nibble2	Carry = Overflow	Addition
001	| Nibble1 - Nibble2 |	1 if Nibble2 > Nibble1, 0 otherwise	Test / diff
010	Nibble1 AND Nibble2	0	Bitwise AND
011	Nibble1 OR Nibble2	0	Bitwise OR
100	Nibble1 XOR Nibble2	0	Bitwise XOR
101	15 - Nibble1	0	Bitwise inverse of Nibble1
110	15 - Nibble2	0	Bitwise inverse of Nibble2
111	Nibble1 + Nibble2 + 1	Carry = Overflow	Addition

Rysunek 28

Multiplekser

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multiplexer_VHDL is
    port
    (
        a, b, c, d, e, f, g, h : in std_logic;
        Sel : in std_logic_vector(2 downto 0);

        Output : out std_logic
    );
end entity Multiplexer_VHDL;

architecture Behavioral of Multiplexer_VHDL is
begin
    process (a, b, c, d, e, f, g, h, Sel) is
    begin
        case Sel is
            when "000"  => Output <= a;
            when "001"  => Output <= b;
            when "010"  => Output <= c;
            when "011"  => Output <= d;
            when "100"  => Output <= e;
            when "101"  => Output <= f;
            when "110"  => Output <= g;
            when others => Output <= h;
        end case;
    end process;
end architecture Behavioral;
```

Klawiatura PS/2

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
ENTITY KeyboardCtrl IS PORT (
   Reset: IN STD_LOGIC;
   KeyboardClock: IN STD_LOGIC;
   KeyboardData: IN STD_LOGIC;
   keycode: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END KeyboardCtrl ;
ARCHITECTURE Behavioral OF KeyboardCtrl IS
  TYPE state_type IS (
s_start,s_d0,s_d1,s_d2,s_d3,s_d4,s_d5,s_d6,s_d7,s_parity,s_stop);
SIGNAL state: state_type;
BEGIN
  FSM: PROCESS(KeyboardClock, Reset)
  BEGIN
    IF (Reset = '1') THEN
     state <= s_start;
    -- this FSM is driven by the keyboard clock signal
    ELSIF (KeyboardClock'EVENT AND KeyboardClock = '1') THEN
     CASE state is
     WHEN s_start =>
        state <= s_d0;
     WHEN s_d0 =>
        state <= s_d1;
     WHEN s_d1 =>
        state <= s_d2;
     WHEN s_d2 =>
        state <= s_d3;
     WHEN s_d3 =>
        state <= s_d4;
     WHEN s_d4 =>
        state <= s_d5;
     WHEN s_d5 =>
        state <= s_d6;
     WHEN s_d6 =>
        state <= s_d7;
     WHEN s_d7 =>
        state <= s_parity;
     WHEN s_parity =>
        state <= s_stop;
     WHEN s_stop =>
        state <= s_start;
       WHEN OTHERS =>
   END CASE;
 END IF;
END PROCESS;

output_logic: PROCESS (state)
BEGIN
CASE state IS
WHEN s_d0 =>
 keycode(0) <= KeyboardData; -- read in data bit 0 from the keyboard
```

```
WHEN s_d1 =>
  keycode(1) <= KeyboardData; -- read in data bit 1 from the keyboard
WHEN s_d2 =>
  keycode(2) <= KeyboardData; -- read in data bit 2 from the keyboard
WHEN s_d3 =>
  keycode(3) <= KeyboardData; -- read in data bit 3 from the keyboard
WHEN s_d4 =>
  keycode(4) <= KeyboardData; -- read in data bit 4 from the keyboard
WHEN s_d5 =>
  keycode(5) <= KeyboardData; -- read in data bit 5 from the keyboard
WHEN s_d6 =>
  keycode(6) <= KeyboardData; -- read in data bit 6 from the keyboard
WHEN s_d7 =>
  keycode(7) <= KeyboardData; -- read in data bit 7 from the keyboard

WHEN OTHERS =>
    END CASE;
  END PROCESS;
END Behavioral;
```

Karta graficzna

```
entity kartaGraficzna is
        port(   clk : in std_logic;
                        reset : in std_logic;
                        RGB : out std_logic_vector(2 downto 0);
                        hSync : out std_logic;
                        vSync : out std_logic;
                );

-- Zegar taktuje z czestotliwoscia 40 MHz
-- Okres taktu to 1/40MHz czyli 25ns.
-- Calosc ramki poziomej to CALY czas A, tj. 26.4 us.
-- Calosc ramki pionowej to CALY czas 0, tj 16.597 ms.
-- Czas synchronizacji poziomej to 3.2us i pionowej: 0.106ms
-- Po nich nastepuja krotkie okresy przerwy w nadawaniu odpowiednio Q i C
-- Nastepnie leci sygnal wyswietlania przez czasy D i R,
-- oraz ostatecznie dwie przerwy E i S koncza OKRES nadawania ramki
-- odpowiednio poziomej i pionowej

architecture arch1 of kartaGraficzna is
        signal poz : std_logic_vector(9 downto 0);
        signal pion : std_logic_vector(9 downto 0);
begin
        process(clk, reset)
                if reset = '1' then
                        poz <= (others => '0');
                        pion <= (others => '0');
                end if;
                if clk'event and clk = '0' then
                        poz <= poz + 1;
                        if poz = '1056' then
                        -- Chcemy uzyskac obliczenie calego czasu
                        -- w  ktorym operujemy tj. cale A
                        -- Dzielimy czas A 26400/25 = 1056
                                pion <= pion + 1
                                poz <= (others => '0');
                                if pion = '628' then
                                -- dzielnikiem nie jest 25ns tylko czas A bo
                                -- wartosc zwieksza sie po przebiegu calego c
                                -- Chcemy uzyskac caly czas 0. Dzielimy 0/A
                                -- 16579/26.4 = 628.
                                        pion <= (others => '0');
                                end if;
                        end if;
                end if;
        end process;

        -- W karcie z wykladu sygnaly synchronizacji szly PO wyswietleniu ram
        -- Tutaj jest odwrotnie najpierw synchronizacja (B i P) pozniej reszt
        -- Dzielimy odpowiednio czasy
        -- P/25ns -> 3200 / 25 = 128 oraz P/A -> 106/26.4 = 4

        hSync <= '0' when poz >= '0' and poz <= '128';
```

```
                else '1';
        vSync <= '1' when pion >= '0' and pion <= '4';
                else '0';

                -- zeby wysterowac sygnal na dany kolor i trafic
                -- idealnie w srodek ekranu musimy wiedziec
                -- kiedy rzucac kolor czerwony, a kiedy zielony.
                -- Zajmujemy sie sygnalem poziomym, bo pas ma byc pionowy
                --(tzn na kazdym wierszu poziomym kreska 10px)
                -- Czekamy odpowiednia ilosc czas, tj. czas wysterowania B,
                -- Czas kiedy nic sie nie dzieje C i chwile czasu D
                -- dokladnie 395 (zeby powstala po srodku)
                -- Jak dodamy czasy (C + D)/25ns = 216 i dodamy ilosc
                -- pikseli 395 od ktorej trzeba zaczac kolorowac na czerwono
                -- Otrzymamy 611. 10px czyli do 622 wlacznie.

        RGB <= "100" when pion >= 611 and pion <= 621;
          else "010";
end architecture;
```