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

Zatrzask:

```
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;
```

Zatrzask jest niepełnym przerzutnikiem. Przerzutnik wyzwala się zboczem zegarowym – zatrzasku 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 zatrzasku 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 zatrzasku można dodać opcję asynchronicznego kasowania:

Zatrzask z asynchronicznym kasowaniem:

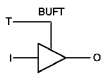
```
architecture archi of latches_2 is
begin
    process (CLR, D, G)
    begin
        if (CLR='1') then
            Q <= '0';
        elsif (G='1') then
            Q <= D;
        end if;
    end process;
end archi;
```

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
    O <= 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:

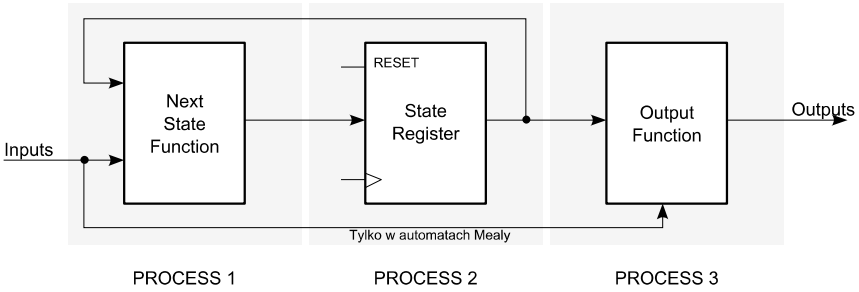
```
entity counters_6 is
    port(C, CLR, UP_DOWN : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_6;

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 powieźć (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 prezentuję 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 zatrzaskiwany wraz z kolejnym taktem 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 ;
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 <= s4;
            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-

żliwości procesu znajdują się sygnały kasowania oraz zegarowy.

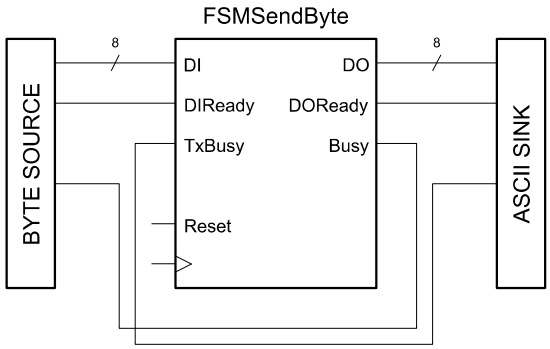
Next State Function opisana jest w `process2`. Można było tu umieścić zwykłe przypisania współbieżne, ale byłyby one bardzo długie i nieczytelne. Lepszym dla oka rozwiązaniem jest użycie procesu. Jest tam opis zachowania maszyny we wszystkich stanach. Najlepiej jest używać instrukcji wyboru `case`. Korzystanie 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 zatrzasku (który funkcjonalnie nam nie przeszkadza). Duża ilość takich zatrzasków 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 zbocza 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”:

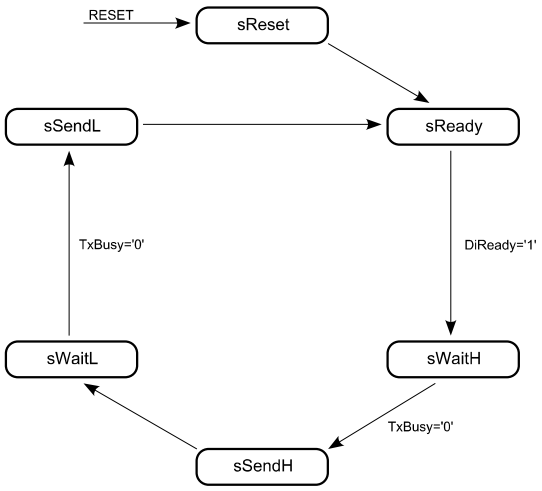


Sygnały tego układu to: sygnał zegarowy, sygnał kasujący oraz 8-bitową magistralę po której będą spływały dane wejściowe „DI”. Obecność nowego bajtu będzie sygnalizowana przez źródło sygnałem „DIReady”. Impuls jednotaktowy na tej linii będzie oznaczał obecność nowego bajtu do wysłania. Potrzebny będzie sygnał „TxBudy” bo musimy wiedzieć czy odbiornik jest gotowy czy nie. Magistralą „DO” będą przesyłane do odbiornika dane po przekodowaniu. Po linii „DOReady” 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 HalfByte : 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 sygnały „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 wygenerowanie impulsu „DOReady". 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: zatrzask 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 rejestry `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 sWaitH =>
        if TxBusy = '0' then
          NextState <= sSendH;
        end if;

      when sSendH =>
        NextState <= sWaitL;

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

      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          -- 0-15 => ASCII '0'-'F'
    DO <= X"30" when "0000",
          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;
  DORdy <= '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 „DOReady” 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 ta 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

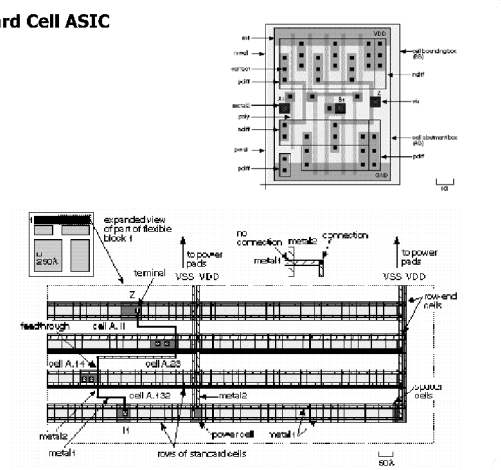
Jeżeli chodzi o układy programowane przez użytkownika to mamy do czynienia z:

- 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.

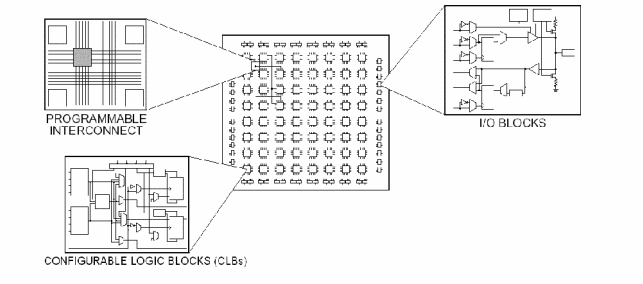
Standard Cell ASIC



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.

Można 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) $400 / day	2 $800	5 $2,000	5 $2,000
Hardware	$10,000	$10,000	$10,000
Software	$1,000	$20,000	$40,000
Design (10k gates) gates / day days $400 / day	500 20 $8,000	200 50 $20,000	200 50 $20,000
Production test design	0	5 days $2,000	5 days $2,000
NRE masks simulation test	0	$30,000 (3÷4) $10,000 $10,000 $10,000	$70,000 (15+) $50,000 $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^2
Utilization	60%	85%	100%	
Die size	1.67	0.59	0.40	cm^2
Die / wafer	88	248	365	
Defect density	1.10	0.90	1.00	def. / cm^2
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 wchodzą opłaty za sprzęt i oprogramowanie do projektowanie 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.