Projektowanie systemów cyfrowych w oparciu o układy programowalne

1 Język ABEL

Język ABEL (Advanced Boolean Expression Language) został opracowany przez firmę DATA I/O w pierwszej połowie lat osiemdziesiątych.

Opis projektowanego układu może być wprowadzony w języku ABEL w jednej z trzech postaci :

- równań logicznych
- tablicy prawdy
- grafu stanów

W programie w języku ABEL można wyróżnić cztery zasadnicze segmenty

- nagłówek
- segment deklaracji
- segment opisu układu
- segment symulacji

Poniżej zostaną omówione zasady tworzenia poszczególnych segmentów. We wszystkich opisach przyjęto następujące zasady :

- słowa kluczowe wytłuszczono (np. **equations**)
- w znaki <> ujęto parametry (np. <łańcuch>)
- w nawiasy [ ] ujęto części opcjonalne (np. [title ‘ ‘])

1.1 Nagłówek

Nagłówek programu w języku ABEL ma następującą postać :

```
module <nazwa> [flag 'parametr'[,'parametr 2'...]][title <łańcuch>]
```

Występująca w nagłówku nazwa nie musi być taka sama jak nazwa zbioru, w którym umieszczony jest program. Taka sama nazwa może natomiast pojawić się po słowie kluczowym end kończącym program w języku ABEL.

Poszczególne programy systemu ABEL są z reguły uruchamiane za pomocą systemu rozwijanych menu. Lecz mogą być one także uruchamiane oddzielnie. Wtedy parametry sterujące można podawać przy uruchamianiu każdego programu ale wygodniej jest umieścić

je w programie źródłowym. Umożliwia to słowo kluczowe flag, po którym należy podać odpowiednie parametry - każdy parametr musi być ujęty w apostrofy, kolejne parametry oddzielone są przecinkami. Spośród wielu możliwych parametrów omówiony zostanie tutaj tylko jeden, określa jaki rodzaj minimalizacji logicznej ma zostać przeprowadzony :

-r0 - nie należy przeprowadzać żadnej minimalizacji

-r1 - tylko prosta redukcja logiczna (parametr domyślny)

-r2 - minimalizacja za pomocą algorytmu PRESTO

-r3 - minimalizacja za pomocą algorytmu PRESTO PER PIN

Algorytm PRESTO daje najlepsze wyniki dla układów FPLA i FPLS. natomiast algorytm PRESTO PER PIN dla układów PAL. Dlatego użycie niewłaściwej procedury minimalizacji może spowodować, że projekt nie zmieści się w wybranym układzie PLD (mimo iż jest to możliwe przy zastosowaniu właściwej procedury. Opcjonalnie w nagłówku można także po słowie kluczowym title umieścić dowolny, tekst (nazwę projektu, krótki opis, datę lub inne informacje), który zostanie automatycznie dołączony do nagłówka zbioru JEDEC i tworzonej dokumentacji.

1.2 Segment deklaracji

Po nagłówku programu należy umieścić potrzebne deklaracje. Spośród wszystkich wymienionych poniżej deklaracji musi się pojawić jedynie deklaracja wyprowadzeń układu, pozostałe są opcjonalne. Każda linia programu musi być zakończona średnikiem.

deklaracja typu wykorzystywanego układu PLD

```
<nazwa> device '<typ układu>';
```

<nazwa> może być łańcuchem liter i cyfr rozpoczynającym się od liter. Pierwsze 8 znaków tego łańcucha zostanie użyte jako nazwa zbioru JEDEC wygenerowanego przez program.

<typ układu> musi być nazwą zbioru bibliotecznego (bez rozszerzenia) zawierającego informacje o danym układzie, np. układowi PAL22V10 odpowiada nazwa 'P22V10'.

deklaracje wyprowadzeń układu

```
<nazwa>[,<nazwa>...]pin [<nr>[,<nr 1>...]][istype '<typ>'] ;
```

<nazwa> jest nazwą symboliczna. która odpowiada wyprowadzeniu układu scalonego o numerze. W jednej linii można opisać więcej wyprowadzeń układu.

Niektóre układy PLD maja konfigurowalne makrokomórki wyjściowe (ang. macrocell), których konfigurację należy określić (np. układy firmy ALTERA). Można to zrobić przy

deklaracji wyprowadzeń używając słowa kluczowego **istype**, po którym należy podać odpowiednie słowa kluczowe (atrybuty) określające konfiguracji makrokomórki - jeżeli jest ich więcej należy je oddzielić przecinkami. Atrybuty określające konfigurację są następujące :

buffer - wyjście proste. Atrybut używany do konfiguracji wyjściowego inwertera

com - wyjście kombinacyjne. Jeżeli makrokomórka zawiera rejestr to zostanie on ominięty

invert - wyjście zanegowane. Atrybut używany do konfiguracji wyjściowego inwertera

reg - wyjście przez przerzutnik typu d. Typ przerzutnika jest określany przez użycie operatora ':=' lub rozszerzeń (.d, .t itd.) więc atrybut ten ma znaczenie tylko przy zastosowaniu diagramu stanów

reg_xx - wyjście przez przerzutnik typu xx (xx może przybierać wartości : d - przerzutnik typu d, t - przerzutnik typu t, jk - przerzutnik typu jk, sr - przerzutnik typu sr)

pos - wyjście proste

neg - wyjście zanegowane

xor - wyjście przez bramkę XOR

feed_or - sprzężenie zwrotne z wyjścia bramki OR. Atrybut istnieje aby zachować kompatybilność z poprzednimi wersjami. Zamiast niego można używać rozszerzeń .d, .t, .j itd. (np. out = (a & b) # out1.d)

feed_pin - sprzężenie zwrotne z wyprowadzenia układu. Atrybut istnieje aby zachować kompatybilność z poprzednimi wersjami. Zamiast niego należy używać rozszerzenia .pin

feed_reg - sprzężenie zwrotne z wyjścia rejestru. Atrybut istnieje aby zachować kompatybilność z poprzednimi wersjami. Zamiast niego można używać rozszerzenia .fb

deklaracje węzłów wewnętrznych

```
<nazwa>[,<nazwa1>...] node [<nr>[,<nr1>...]];
```

Niektóre - bardziej złożone - układy PLD posiadają pewne punkty, istotne dla działania układu, które nie są wyprowadzone na zewnątrz - są to tzw. węzły wewnętrzne. Dostęp do nich jest możliwy jeżeli zostaną zadeklarowane w programie - sposób deklaracji jest jak widać identyczny jak dla wyprowadzeń. Numery węzłów wewnętrznych dla poszczególnych układów PLD zawarte są w dokumentacji systemu ABEL.

konfiguracja makrokomórek wyjściowych

```
<nazwa>[,<nazwa1>...] istype '<typ>';
```

Makrokomórki wyjściowe można konfigurować przy deklaracji wyprowadzeń, o czym była mowa już wcześniej. Konfiguracja ta nie musi być jednak określona podczas deklaracji wyprowadzeń - można ją określić w osobnych liniach programu w sposób przedstawiony powyżej (nazwa jest nazwą przypisaną do określonego wyprowadzenia w deklaracji wyprowadzeń). Typy makrokomórek zostały opisane przy opisie deklaracji wyprowadzeń.

sterowanie makrokomórkami wyjściowymi

Pojedyncza makrokomórka wyjściowa jest zwykle połączona z matrycą iloczynów i sum więcej niż jedną linią. Linie sterujące makrokomórkami wyjściowymi oznaczane są następująco:

```
<nazwa>.<rozszerzenie>
```

Gdzie <nazwa> jest nazwą przypisaną do określonego wyprowadzenia w deklaracji wyprowadzeń. <rozszerzenie> może przyjmować następujące wartości (wartości dostępne wyznacza architektura zastosowanego układu PLD):

ap - asynchroniczne wejście ustawiające

ar - asynchroniczne wejście zerujące

ce - wejście zegarowe przerzutników wyzwalanych poziomem

clk - wejście zegarowe przerzutników wyzwalanych zboczem

d - wejście D przerzutnika typu D

fb - sprzężenie zwrotne z wyjścia przerzutnika o polaryzacji zgodnej z polaryzacja wyjścia (sygnał jest negowany jeśli to konieczne)

fc - wejście sterujące typem przerzutnika (np. możliwa jest zamiana przerzutnika JK na D podczas pracy układu). Wejście dostępne tylko w niektórych architekturach PLD

j - wejście J przerzutnika JK

k - wejście K przerzutnika JK

ld - wejście sterujące ładowaniem stanu przerzutnika. Wejście dostępne tylko w niektórych architekturach PLD

le - wejście latch-enable. Wejście dostępne tylko w niektórych architekturach PLD

oe - wejście sterujące trójstanowym buforem wyjściowym

pin - sprzężenie zwrotne z wyjścia układu

pr - wejście ustawiające (synchroniczne lub asynchroniczne)

q - sprzężenie zwrotne z wyjścia przerzutnika o polaryzacji zgodnej z polaryzacja wyjścia Q przerzutnika (sygnał jest negowany jeśli to konieczne)

r - wejście R przerzutnika SR

re - wejście zerujące (synchroniczne lub asynchroniczne)

s - wejście S przerzutnika SR

sp - synchroniczne wejście ustawiające

sr - synchroniczne wejście zerujące

t - wejście T przerzutnika T

deklaracja stałych

```
<nazwa> = <wartość >
```

W języku ABEL można używać zadeklarowanych wcześniej stałych, co często jest bardzo wygodne - w wyrażeniach można używać nazw symbolicznych zamiast konkretnych wartości. Można używać liczb dziesiętnych (bez dodatkowych oznaczeń lub poprzedzonych znakami ^d), ósemkowych (^o), szesnastkowych (^h), wektorów (ujętych w nawiasy []) oraz znaków ASCII (ujętych w apostrofy).

Np. liczbę 66 można zapisać w następujące sposoby :

66 = ^d66 = ^b1000010 = ^102 = ^h42 = [1,0,0,0,0,1,0] = 'B'

W języku ABEL istnieją również pewne stałe predefiniowane. przydatne przy tworzeniu wyrażeń i wektorów testowych. Są to :

.X. - stan nieokreślony

.Z. - stan wysokiej impedancji

.C. - dodatni impuls zegarowy (_--_)

.K. - ujemny impuls zegarowy (--_--)

.D. - tylne zbocze impulsu zegarowego (--__)

.U. - przednie zbocze impulsu zegarowego (__--)

.P. - ładowanie przerzutników

.SVn. - ustawienie napięcia równego n (n=2..9)

deklaracja zbiorów

```
<nazwa> = [ <nazwa1>, <nazwa2>, <nazwa3>, ...]
```

UWAGA: występujące tu nawiasy [ ] w tym przypadku nie oznaczają parametrów opcjonalnych, a są elementem składni języka.

<nazwa> - nazwa zbioru.

<nazwa1>, <nazwa2>, <nazwa3>, ... - nazwy wyprowadzeń lub stałe

Zbiór jest zestawem sygnałów lub stałych zachowującym się jak liczba n bitowa, gdzie n jest liczbą sygnałów zawartych w zbiorze. Operacje wykonywane na zbiorze są wykonywane na wszystkich jego elementach zgodnie z regułami języka. Do zbiorów można stosować

operatory logiczne (not, and, or, xor, xnor), arytmetyczne (+, -), relacyjne (==, !=, <, <=, >, >=) i przypisania (=, :=).

deklaracja makrodefinicji

```
<nazwa> macro [(<parametry>)] {<wyrażenie>};
```

Makrodefinicja przypisuje określonej nazwie pewne wyrażenie logiczne. Nazwy tej można następne używać w wyrażeniach opisujących projektowany układ. Makrodefinicja może mieć parametry - wtedy przy jej wywołaniu należy podać odpowiednie wielkości. Tworząc wyrażenie makrodefinicji parametry należy poprzedzić znakiem ?

Na przykład makrodefinicja realizująca funkcję NAND parametrów A i B będzie miała postać :

```
NAND macro ( A, B ) { !( ?A & ?B ) };
```

a wywołanie przypisujące wyjściu OUT funkcję NAND wejść X i Y

```
OUT = NAND ( X, Y )
```

Opisane powyżej deklaracje nie muszą wystąpić w ściśle określonej kolejności - z wyjątkiem deklaracji typu układu PLD, która musi pojawić się na początku.

W dowolnym miejscu programu w języku ABEL można umieścić komentarz, który rozpoczyna się znakiem cudzysłowu ("), a kończy wraz z końcem linii.

1.3 Segment opisu układu

Po segmencie deklaracji umieszcza się segment opisu układu. Jak już wcześniej wspomniano można posłużyć się trzema różnymi sposobami opisu - równaniami logicznymi, tablica prawdy lub grafem stanów - i w zależności od tego różna jest konstrukcja tego segmentu.

Segment opisu układu - równania logiczne

Segment zawierający równania logiczne rozpoczyna się słowem kluczowym **equations**, po którym zapisuje się równania dla wszystkich sygnałów wyjściowych i węzłów wewnętrznych. Ogólna postać równania jest następująca :

```
<wyjście> <operator przypisania> <wyrażenie logiczne>;
```

<wyjście> jest nazwa symboliczną przypisana w segmencie deklaracji do wyprowadzenia wyjściowego lub węzła wewnętrznego.

<operator przypisania> jest jednym z dwóch operatorów przypisania

= zwykły operator przypisania

:= przypisanie przy aktywnym zboczu sygnału zegarowego

Drugi z wymienionych operatorów służy do opisu układów sekwencyjnych za pomocą równań logicznych.

<wyrażenie logiczne> jest dowolnym wyrażeniem logicznym zbudowanym zgodnie z zasadami algebry Boole'a. Mogą tu wystąpić nazwy wyprowadzeń, stałe, makrodefinicje, warunki. Można używać nawiasów ( ), wielokrotnie zagnieżdżonych.

Wykorzystywane są następujące operatory logiczne :

! - NOT

& - AND

- OR

$ - XOR

!$ - XNOR

Operatory arytmetyczne :

-A negacja

A+B suma arytmetyczna

A-B różnica

A*B iloczyn arytmetyczny

A/B część całkowita dzielenia A przez B

A%B reszta z dzielenia A przez B

A<<B przesunięcie w lewo o B bitów

A>>B przesunięcie w prawo o B bitów

Warunki można tworzyć wykorzystując następujące operatory relacji :

!= - nie równy

== - równy

< - mniejszy

<= - mniejszy lub równy

=> - większy lub równy

> - większy

Równanie można też zapisać korzystając z instrukcji warunkowej:

```
when <warunek> then <równanie> [else <równanie>];
```

np.:

```
when b then a=c else a=d;
```

Segment opisu układu - tablica prawdy

Innym sposobem opisu układu jest tablica prawdy. Każdy układ kombinacyjny można opisać zarówno za pomocą równań logicznych, jak i tablicy prawdy. Który z tych sposobów jest w danym przypadku bardziej efektywny zależy od projektowanego układu.

Opis za pomocą tablicy prawdy rozpoczyna się od słowa kluczowego **truth_table**. Ogólna postać tablicy prawdy jest następująca :

```
truth_table ([ <sygnały_wejściowe> ] -> [ <sygnały_wyjściowe> ] )
             [ <wektor_wejściowy1> ] -> [ <wektor_wyjściowy1> ];
             [ <wektor_wejściowy2> ] -> [ <wektor_wyjściowy2> ];
                                        :
                                        :
```

UWAGA: występujące tu nawiasy [ ] w tym przypadku nie oznaczają parametrów opcjonalnych, a są elementem składni języka.

<sygnały_wejściowe> - są to wszystkie sygnały wejściowe układu wypisane kolejno (oddzielone przecinkami).

<sygnały_wyjściowe> - są to wszystkie sygnały wyjściowe układu i węzły wewnętrzne wypisane kolejno (oddzielone przecinkami).

Pod takim nagłówkiem wypisuje się kolejno wartości wektorów wejściowych i odpowiadające im wartości wektorów wyjściowych. W wektorach tych można używać także stanów nieokreślonych (.X.).

Segment opisu układu - graf stanów

Układy sekwencyjne można opisywać za pomocą równań logicznych lub za pomocą odpowiednio zapisanego grafu stanów. Opis za pomocą grafu stanów rozpoczyna się słowem kluczowym **state_diagram**. Ogólna postać jest następująca:

```
state_diagram [ <wektor_stanu> ]
  state <stan1> :
         <równanie_wyjść>;
                  :
         <przejścia>;
  state <stan2> :
         <równanie_wyjść>;
                  :
         <przejścia>;
         :
         :
```

UWAGA: występujące tu nawiasy [ ] w tym przypadku nie oznaczają parametrów opcjonalnych, a są elementem składni języka.

<wektor_stanu> tworzą określone sygnały., które muszą tu zostać wypisane.

<stan> to kod stanu (wartość wektora stanu), który musi być wartością lub zdefiniowaną wcześniej stałą

<równanie_wyjść> określa wartość wyjść układu w danym stanie

Do definiowania przejść można posłużyć się jedna z trzech instrukcji

- skoku

```
goto <stan>;
```

- warunkowa

```
if <warunek> then <stan 1> else <stan 2>;
```

- wyboru

```
case  ( <warunek 1> ) :<stan 1>;
      ( <warunek 2> ) :<stan 2>;
                      :
                      :
      ( <warunek n> ) :<stan n>;
endcase;
```

Warunki tworzy się wykorzystując wymienione wcześniej operatory logiczne i relacji. We wszystkich powyższych konstrukcjach <stan> jest stanem docelowym, do którego nastąpi przejście, jeżeli spełniony jest warunek (w przypadku instrukcji **goto** przejście nastąpi oczywiście bezwarunkowo).

Wartości wyjść układu można określić dla danego przejścia wykorzystując instrukcję:

```
width
  <równanie_wyjść>;
        :
endwidth
```

Instrukcja ta może znajdować się wewnątrz instrukcji **if** lub **case**.

1.4 Segment symulacji

Jak wcześniej wspomniano w skład systemu ABEL wchodzi prosty symulator logiczny. Pozwala on na weryfikację opracowanego projektu za pomocą zamieszczonych w segmencie symulacji wektorów testowych. Segment ten może zostać pominięty - wtedy żadna symulacja nie zostanie przeprowadzona.

W czasie symulacji program wyświetla jedynie informację czy dany wektor był poprawny - (wyświetla wtedy znak '.'), czy też pojawił się błąd (wyświetla znak '*'). Po zakończonej symulacji można w zbiorze z rozszerzeniem 'sim' zobaczyć wszystkie wektory testowe, w których pojawiły się błędy, wraz z informacją jakich stanów oczekiwano, a jakie się pojawiły.

Segment symulacji rozpoczyna się słowem kluczowym **test_vectors**. Ogólna jego struktura jest następująca :

```
test_vectors ([ <sygnały wejściowe> ]  -> [ <sygnały wyjściowe> ] )
              [ <wektor wejściowy 1> ] -> [ <wektor wyjściowy 1> ];
              [ <wektor wejściowy 2> ] -> [ <wektor wyjściowy 2> ];
                                        :
```

:

UWAGA: występujące tu nawiasy [ ] w tym przypadku nie oznaczają parametrów opcjonalnych, a są elementem składni języka.

<sygnały wejściowe> to sygnały, których wartości będą wymuszane w czasie symulacji

<sygnały wyjściowe> to sygnały, których stan będzie sprawdzany podczas symulacji

Następnie występują kolejne wektory wymuszeń i oczekiwane odpowiedzi na te wektory. Przy tworzeniu wektorów testowych bardzo przydatne są opisane wcześniej stałe predefiniowane języka ABEL (.C., .K., .P., .SVn., .X., .Z.). Na szczególna uwagę zasługuje stała .P., która pozwala ustawić określony stan przerzutników na początku symulacji.

1.5 Dyrektywy kompilatora

Dyrektywy rozpoczynają się od znaku '@'

@alternate

włącza rozpoznawanie przez kompilator alternatywnego zestawu operatorów.

@const <identyfikator> = <wyrażenie>

pozwala na deklarację stałych poza segmentem deklaracji. Jeżeli zostanie użyty <*identyfikator*> istniejącej stałej to deklarowana wartość zastąpi jej poprzednią wartość (np. @const xxx = xxx +3;)

@dcset

pozwala na wykorzystanie wartości nieokreślonych (Φ) podczas minimalizacji funkcji nie wpełni określonych

@exit

powoduje przerwanie przetwarzania pliku źródłowego i zgłoszenie błędu

@if <wyrażenie> <blok>

pozwala na włączanie fragmentów kodu źródłowego w zależności od wyniku <*wyrażenia*>. Jeżeli <*wyrażenie*> ma wartość true (różną od zera) to <*blok*> jest włączany do kodu źródłowego.

@irp <zmienna> (<argument> [,<argument>]...) <blok>

powtarza <*blok*> tyle razy ile argumentów zawierają nawiasy. Za każdym razem <*zmienna*> przyjmuje wartość kolejnego argumentu. Wartość zmiennej uzyskuje się poprzedzając jej identyfikator znakiem ?.

np.

```
@irp a ( 2, ^H0C, 1 )
```

```
{b=?a;
}
```

zostanie rozwinięte do:

```
b=2;
b=^H0C;
b=1;
```

@irpc <zmienna> (<argument>) <blok>

powtarza <*blok*> tyle razy ile znaków zawiera <*argument*>. Za każdym razem <*zmienna*> przyjmuje wartość kolejnego znaku argumentu. Wartość zmiennej uzyskuje się poprzedzając jej identyfikator znakiem ?.

np.

```
@irpc a ( Ala )
{b=?a;
}
```

zostanie rozwinięte do:

```
b=A;
b=l;
b=a;
```

@radix <wyrażenie>

zmienia domyślną podstawę systemu liczbowego. <*wyrażenie*> może przyjmować tylko wartości 2, 8, 10, 16.

@repeat <wyrażenie> <blok>

powtarza <*blok*> n razy gdzie n jest wartością <*wyrażenia*>

2 Przykłady

2.1 Dekoder BCD na 7seg

W przykładzie przedstawiono dekoder kodu BCD sterujący wyświetlaczem 7 - segmentowym o wspólnej anodzie. Dekoder posiada dodatkowe wejście ‘test’, służące do sprawdzania działania wyświetlacza.

2.1.1Opis za pomocą równań logicznych

```
”Nagłówek
module BCD27seg
title ‘Dekoder BCD sterujący wyświetlaczem 7seg’
”Segment deklaracji
  a,b,c,d,e,f,g pin istype 'com’;”Wyjścia
  x3,x2,x1,x0,test pin;          ”Wejścia
  bcd = [x3,x2,x1,x0];
  H,L,X =1,0,.X.
”Segment równań logicznych
equations
  a = !test & (!x3 & !x2 & !x1 & x0 # x2 & !x1 & !x0);
  b = !test & (x2 & !x1 & x0 # x2 & x1 & !x0);
```

```
  c = !test & (!x2 & x1 & !x0);
  d = !test & (x2 & x1 & x0 # !x3 & !x2 & !x1 & x0 # x2 & !x1 & !x0);
  e = !test & (x2 & !x1 # x0);
  f = !test & (!x2 & x1 # !x3 & !x2 & x0 # x1 & x0);
  g = !test & (!x3 & !x2 & !x1 # x2 & x1 & x0);
"Segment symulacji
test_vectors([test,bcd] -> [a,b,c,d,e,f,g])
             [0,0]      -> [0,0,0,0,0,0,1];
             [0,1]      -> [1,0,0,1,1,1,1];
             [0,2]      -> [0,0,1,0,0,1,0];
             [0,3]      -> [0,0,0,0,1,1,0];
             [0,4]      -> [1,0,0,1,1,0,0];
             [0,5]      -> [0,1,0,0,1,0,0];
             [0,6]      -> [0,1,0,0,0,0,0];
             [0,7]      -> [0,0,0,1,1,1,1];
             [0,8]      -> [0,0,0,0,0,0,0];
             [0,9]      -> [0,0,0,0,1,0,0];
             [1,.x.]    -> [0,0,0,0,0,0,0];
end
```

2.1.2Opis za pomocą tablicy prawdy

W przykładzie przedstawiono opis identycznego kodera jak powyżej za pomocą tablicy prawdy. Ponieważ nagłówek, segment deklaracji i symulacji nie ulęgają zmianie, więc przedstawiono jedynie segment opisu układu. Porównując przykłady 2.1.1 i 2.1.2 można zauważyć, że w przypadku układu kodera opis za pomocą równań logicznych jest nieco krótszy,, za to opis za pomocą tablicy prawdy jest znacznie prostszy,.

```
"Segment równań logicznych
equations
@dcset
truth_table([test,bcd]  -> [a,b,c,d,e,f,g])
             [0,0]      -> [0,0,0,0,0,0,1];
             [0,1]      -> [1,0,0,1,1,1,1];
             [0,2]      -> [0,0,1,0,0,1,0];
             [0,3]      -> [0,0,0,0,1,1,0];
             [0,4]      -> [1,0,0,1,1,0,0];
             [0,5]      -> [0,1,0,0,1,0,0];
             [0,6]      -> [0,1,0,0,0,0,0];
             [0,7]      -> [0,0,0,1,1,1,1];
             [0,8]      -> [0,0,0,0,0,0,0];
             [0,9]      -> [0,0,0,0,1,0,0];
             [1,.x.]    -> [0,0,0,0,0,0,0];
```

2.2 Licznik

W przykładzie przedstawiono dwukierunkowy licznik liczący w kodzie BCD. Licznik posiada wyjścia trójstanowe i asynchroniczne wejście zerujące.

2.2.1Opis za pomocą równań logicznych

```
module licznik
  q3,q2,q1,q0 pin istype 'reg';                          "wyjścia licznika
  dir pin;                                    "wejście sterujące kierunkiem
  clear pin;                                             "wejście zerujące
  oe pin;                          "wejście sterujące trójstanowymi buforami
  clk pin;
  count = [q3,q2,q1,q0];
equations
```

```
  count.oe=!oe;
  count.clk=clk;
  count.ar=!clear;
  when dir then
    count:=(count<9) & (count+1)
  else
    when (count==0) then
      count:=[1,0,0,1]
    else
      count:=count-1;

test_vectors( [clk, clear, dir,oe] -> count )
              [0,   0,     0,  0]  ->  0;
              [0,   1,     0,  1]  ->  .z.;
@const i=1;
@repeat 9 {
              [.c., 1,     1 , 0]  ->  i;
@const i=i+1;
}
              [1,   0,     0,  0]  ->  0;
@const i=9;
@repeat 10 {
              [.c., 1,     0,  0]  ->i;
@const i=i-1;
}
end
```

2.2.2Opis za pomocą grafu stanów

W poniżej przedstawiono opis identycznego jak w poprzednim przykładzie licznika za pomocą grafu stanów. Ponieważ segmenty deklaracji i symulacji są identyczne, więc zostały pominięte.

```
equations
  count.oe=!oe;
  count.clk=clk;
  count.ar=!clear;
  state_diagram (count)
  state 0:
    if dir then 1
    else 9;
  state 1:
    if dir then 2
    else 0;
  state 2:
    if dir then 3
    else 1;
  state 3:
    if dir then 4
    else 2;
  state 4:
    if dir then 5
    else 3;
  state 5:
    if dir then 6
    else 4;
  state 6:
    if dir then 7
    else 5;
  state 7:
    if dir then 8
```

```
  else 6;
state 8:
  if dir then 9
  else 7;
state 9:
  if dir then 0
  else 8;
```