Układy programowalne, część 3

Zgodnie z zapowiedzią z marcowego wydania EP, przechodzimy do przedstawienia podstaw języka CUPL, za pomocą którego już wkrótce będziemy opisywać własne projekty. W tej części artykułu przedstawiamy operatory, działania i funkcje dostępne w CUPL-u i część poleceń preprocesora, za pomocą których można sterować pracą kompilatora.

Pomimo swojej długiej historii CUPL (*Universal Compiler for Programmable Logic*, na rynku dostępny od ok. 1983 roku) jest typowym językiem opisu sprzętu (HDL - *Hardware Description Language*), w związku z czym ma niewiele wspólnego z typowymi językami programowania. Powoduje to m.in. taki skutek, że nieprawdziwe staje się twierdzenie, dość często spotykane wśród wytrawnych programistów, że ich wcześniej nabyte umiejętności znacznie uproszczą im pracę z układami PLD.

Czemu? Otóż pisząc „klasyczny" program, programista określa kolejne kroki wykonywania zadania, natomiast opisując sprzęt, projektant opisuje (można to zrobić na wiele sposobów, z których część jest dostępna w CUPL-u) jego zachowanie (tzw. opis behawioralny) lub budowę (tzw. opis strukturalny). „Program" napisany w języku HDL przekłada się więc na budowę układu, a nie kolejność wykonywania czynności przez mikrokontroler o ustalonej architekturze.

Ze względu na swoją specyfikę, CUPL umożliwia przede wszystkim opis strukturalny na relatywnie niskim poziomie abstrakcji. Dlatego właśnie CUPL-a warto stosować do implementacji projektów w niewielkich układach PLD.

CUPL i historia

Amerykańska firma Logical Devices opracowała CUPL-a w roku 1983 jako uniwersalny język HDL drugiej generacji. Szybko zdobył on uznanie i przez wiele lat nie miał - poza ABEL-em - liczącej się konkurencji. Ponieważ nie był przez producenta rozwijany, dość szybko się zestarzał i stopniowo tracił popularność. W 1995 roku prawa do CUPL-a zakupił Protel (kompilator jest wbudowywany w Protela 99SE i DXP), a od 1996 roku windowsową wersję CUPL-a bezpłatnie udostępnia Atmel.

Abstrakcja w HDL

Według słownika języka polskiego *abstrakcja* oznacza "pojęcie nierzeczywiste lub pozostające w luźnym związku z rzeczywistością (...)". W praktyce projektowej *abstrakcja* oznacza możliwość opisania sposobu działania projektowanego układu w sposób wygodny i czytelny dla projektanta, bez konieczności zagłębiania się w tajniki implementacji projektu w strukturze PLD.

Podstawy języka CUPL

CUPL jest językiem wyposażonym w szereg mechanizmów zwiększających wygodę opisywania sprzętu. „Zwiększających" przede wszystkim w stosunku do ówczesnych konkurentów jak np. PALASM lub Opal (były to kompilatory HDL na poziomie mikroprocesorowych asemblerów), lecz ich przejrzystość docenią także współcześni projektanci.

Zarezerwowane słowa i symbole

Kompilator CUPL rozpoznaje 37 słów kluczowych oraz 23 symbole, które nie mogą być wykorzystywane jako nazwy zmiennych, węzłów, wejść i wyjść. W **tab. 4** zestawiono zastrzeżone słowa, w **tab. 5** zastrzeżone symbole. Kompilator nie jest „czuły" na to, czy słowa kluczowe pisane są małymi, czy też dużymi literami, w związku z czym zapisy: *Node*, *NODE*, *noDE* itp. są traktowane równorzędnie.

Tab. 4. Słowa zastrzeżone w języku CUPL

APPEND	ASSEMBLY	ASSY	COMPANY	CONDITION
DATE	DEFAULT	DESIGNER	DEVICE	ELSE
FIELD	FLD	FORMAT	FUNCTION	FUSE
GROUP	IF	JUMP	LOC	LOCATION
MACRO	MIN	NAME	NODE	OUT
PARTNO	PIN	PINNNODE	PRESENT	REV
REVISION	SEQUENCE	SEQUENCED	SEQUENCEJK	SEQUENCERS
SEQUENCET	TABLE			

Liczby

Kompilator CUPL-a operuje na liczbach 32-bitowych, które mogą być zapisane w jednym z czterech kodów: binarnym, ósemkowym, dziesiętnym lub szesnastkowym. Twórcy CUPL-a przyjęli, że numeryczne oznaczenia wyprowadzeń i indeksy zmiennych są zapisywane w kodzie dziesiętnym, a pozostałe liczby w kodzie szesnastkowym. Jeżeli takie założenie

Tab. 5. Symbole zastrzeżone w języku CUPL

&	#	(	)	-
*	+	[	]	/
:	.	..	/*	*/
;	,	!	«	=
@	$	^		

Tab. 6. Przedrostki stosowane do oznaczania liczb zapisanych w różnych systemach kodowania

Kod liczbowy	Baza	Przedrostek
Binarny	2	'b', 'B'
Ósemkowy	8	'o', 'O'
Dziesiętny	10	'd', 'D'
Szesnastkowy	16	'h', 'H'

Tab. 7. Przykładowe wyniki zastępowania cyfr znakami 'X'

Liczba	Wartości wynikowe
'b'0x	00 lub 01
'B'11x0	1100 lub 1110
'D'9X	90...99
'h'Bxx	B00...BFF
'b'X101	0101 lub 1101

odpowiada projektantowi, to system kodowania liczb nie musi być w żaden sposób oznaczany. Jeżeli z jakichś przyczyn projektant chce zapisać liczby w innym kodzie, musi je oznaczać specjalnymi przedrostkami, które zestawiono w **tab. 6**. Kompilator nie rozróżnia dużych i małych liter w przedrostkach, w związku z czym zapisy:

```
'b'100111 i 'B'100111
'h'fe19 i 'H'fe19
```

są traktowane jednakowo.

Interesującą możliwością oferowaną przez CUPL-a jest możliwość zastępowania cyfr nieistotnych w podawanej liczbie (na przykład podczas deklarowania zakresu adresów) znakiem 'X' (lub 'x'), co jest traktowane przez kompilator jako wartość dowolna - przykłady pokazano w **tab. 7**.

Operatory i funkcje arytmetyczne

Preprocesor CUPL-a pozwala korzystać z wielu operatorów arytmetycznych, które mogą być stosowane do obliczania argumentów

Zapis liczb szesnastkowych w CUPL-u

W odróżnieniu od wielu kompilatorów, CUPL dopuszcza możliwość zapisu liczb szesnastkowych bez konieczności poprzedzania liter A...F cyfrą, tzn. prawidłowe są obydwa zapisy: *a* i *0a*, *c* i *0c* itd.

Tab. 8. Obsługiwane przez CUPL-a operatory i funkcje arytmetyczne

Znak operatora	Przykład	Nazwa działania	Kolejność wykonywania
**	2**3	Potęgowanie	1
*	8*2	Mnożenie	2
/	3/2	Dzielenie	2
x%n	8%7	Modulo *n* dla liczby z zakresu 0...*x*	2
+	3+2	Dodawania	3
-	5-4	Odejmowanie	3
LOGa(x)	LOG2(x) LOG8(x) LOG16(x) LOG(x)	Logarytm z *x* o podstawie *a*	-

(zaznaczanych poleceniami $REPEAT lub $MACRO) dla makr wykorzystywanych w opisie projektu. Nie można z nich korzystać bezpośrednio w opisie projektowanego sprzętu, kompilator będzie bowiem zgłaszał błędy.

Zestawienie dostępnych w CUPL-u operatorów oraz funkcji arytmetycznych znajduje się w **tab. 8**. Wynikiem obliczenia wartości logarytmu jest zawsze liczba całkowita.

Operatory logiczne

Narzędziem niezbędnym podczas opisywania bloków cyfrowych są operatory logiczne, za pomocą których użytkownik może tworzyć dowolne zależności logiczne pomiędzy sygnałami występującymi w projektowanym układzie. Taki sposób opisywania projektów (za pomocą równań boole'owskich), jakkolwiek najbardziej uniwersalny, nie cieszy się wśród projektantów dużą popularnością, ponieważ CUPL oferuje szereg wygodniejszych sposobów opisu (o wyższym stopniu abstrakcji). Przedstawimy je w dalszej części artykułu.

W **tab. 9** zestawiono dostępne w języku CUPL operatory logiczne oraz ich położenie w hierarchii wykonywania działań.

Tab. 9. Operatory logiczne interpretowane przez CUPL-a

Znak operatora	Opis	Kolejność w hierarchii
!	NOT	1
&	AND	2
#	OR	3
$	XOR	4

Zmienne

Zmiennymi w języku CUPL nazywamy ciągi znaków (nazwy), które są przypisane wyprowadzeniom układu (wejściowym lub wyjściowym), wewnętrznym węzłom (tzw. węzłom „zagrzebanym" - *buried node*), można także tworzyć zmienne z sygnałów połączonych w grupy, często nazywane wektorami (o nich w dalszej części artykułu). W większości dostępnych na rynku kompilatorów języka CUPL w nazwach zmiennych są rozróżniane litery małe i duże, w związku z czym nazwy *ADRok* i *AdrOK* nie są równoważne. Deklarowane zmienne mogą zaczynać się cyfrą, literą lub znakiem podkreślenia i muszą w nazwie zawierać co najmniej jedną literę. Nazwa zmiennej nie może zawierać spacji, czyli nazwa *Adres ROM* nie jest prawidłowa (błąd zostanie automatycznie wychwycony przez program CUPLA), w przeciwieństwie do nazwy *Adres_ROM*. Nazwy zmiennych mogą składać się z maksymalnie 31 znaków. Nazwy dłuższe są przez kompilator automatycznie skracane do 31 znaków, co może powodować błędną identyfikację zmiennych.

Zmienne indeksowane

Język CUPL jest wyposażony w wygodny mechanizm wspomagający tworzenie indeksowanych grup zmiennych (wektorów). Dzięki niemu można definiować grupy sygnałów o jednakowych nazwach (na przykład magistrale), różniące się między sobą wyłącznie cyframi indeksującymi. Dzięki temu, zamiast wymieniać wszystkie sygnały jak w przykładzie:

```
[A0, A1, A2, A3, A4, A5, A6, A7,
A8, A9, A10, A11]
```

można je zapisać w postaci:

```
[A0..A11]
```

Z nie do końca wyjaśnionych przez producenta przyczyn, cyfry indeksujące powinny mieścić się w przedziale 0...*n* (gdzie *n* oznacza dowolną liczbę całkowitą mniejszą od 32). W niektórych przypadkach zweryfikowanych przez autora jest możliwa poprawna kompilacja projektu, w którym zastosowano zmienne indek-

Tab. 10. Wykaz poleceń preprocesora (znak $ musi się znajdować w pierwszej kolumnie nowego wiersza)

$DEFINE	$IFDEF	$UNDEF
$ELSE	$IFNDEF	$REPEAT
$ENDIF	$INCLUDE	$REPEND
$MACRO	$MEND	

sowane w przedziale *m...n* (gdzie *m* oznacza dowolną naturalną liczbę dziesiętną większą od 0). Nie jest to jednak reguła, w związku z czym lepiej jest przestrzegać przedstawionego zalecenia. Zmienna zindeksowana cyfrą zero ma zawsze najmniejszą wagę (LSB).

Przykłady prawidłowo zindeksowanych zmiennych:

```
[low_byte_d0..low_byte_d7]
[cnt_data_in_0..cnt_data_in_31]
[data0..data7]
```

Wprowadzenie do numeru indeksu zera wiodącego powoduje, że zmienne (np. *adr_ok2* i *adr_ok02*) nie są sobie równoważne.

Komendy preprocesora

Kompilator jest wyposażony w preprocesor, który wyszukuje i wykonuje specyficzne polecenia, pozwalające wykonywać między innymi warunkową kompilację fragmentów opisu, samodzielnie definiować stałe wykorzystywane w opisie, a także korzystać w bieżącym projekcie z zawartości zewnętrznych plików (np. zawierających predefiniowane elementy lub bloki logiczne). Wszystkie te zadania preprocesor wykonuje przed rozpoczęciem pracy kompilatora. Wykaz poleceń interpretowanych przez preprocesor znajduje się w **tab. 10**. Wszystkie polecenia muszą się rozpoczynać w pierwszej linii wiersza znakiem $. Wielkość liter, jakimi zapisano polecenia dla preprocesora, nie ma żadnego znaczenia, są one zawsze rozpoznawane. W odróżnieniu od pozostałej części opisu HDL, koniec linii zawierającej polecenie dla preprocesora nie jest zaznaczany za pomocą średnika.

CUPL i operatory relacji

Dokuczliwą wadą CUPL-a jest brak możliwości korzystania z operatorów relacji (występują takie m.in. w ABEL-u, AHDL-u, VHDL-u i Verilogu), dzięki czemu opisywanie różnego rodzaju komparatorów i porównywanie wartości wektorów stałoby się bardzo łatwe.

Polecenie $DEFINE

Polecenie $DEFINE pozwala zdefiniować ciąg znaków, który zastąpi określony w poleceniu operator, liczbę lub symbol. Działa ono w każdym miejscu opisu, aż do odwołania go za pomocą polecenia $UNDEF.

Format polecenia $DEFINE jest następujący:

```
$DEFINE argument1 argument2
```

gdzie:

argument1 - ciąg znaków, któremu jest przypisywane nowe znaczenie,

argument2 - operator, liczba lub zmienna.

Po przypisaniu ciągowi znaków zastępstwa, można go używać w dowolnym miejscu programu w taki sam sposób jak wartości oryginalnej.

Przykłady:

```
$DEFINE ON 'b'1
$DEFINE OFF 'B'0
$DEFINE PORT_A 'h'3ff
```

Za pomocą tego polecenia można także zdefiniować własne symbole - operatory logiczne, przykłady:

$DEFINE { /* - alternatywny znak początku komentarza

$DEFINE } */ - alternatywny znak końca komentarza

$DEFINE / ! - alternatywny znak operatora negacji

$DEFINE * & - alternatywny znak operatora AND

$DEFINE + # - alternatywny znak operatora OR

$DEFINE:+: $ - alternatywny znak operatora XOR

$DEFINE end_proc 'h'ea - przypisanie stałej *end_proc* wartości EAh

$DEFINE ROM_ADDRESS 'b'10011101 - przypisanie stałej *ROM_ADDRESS* wartości 10011101b

Za pomocą polecenia $DEFINE można także definiować stałe o wartościach podanych jako zakres, przykłady:

$DEFINE der_osc 'b'110x - przypisuje stałej *der_osc* wartości zapisane dwójkowo: 1100 i 1101

$DEFINE adres 'd'[120..129] - przypisuje stałej *adres* wartości dziesiętne z przedziału 120...129.

Polecenie $UNDEF

Polecenie $UNDEF odwraca działanie polecenia $DEFINE dla wskazanego argumentu. Format polecenia jest następujący:

```
$UNDEF argument
```

gdzie:

argument - ciąg znakowy użyty w komendzie $DEFINE.

Polecenie $UNDEF można stosować do przedefiniowania ciągu znakowego, przykład:

```
$DEFINE S0 'B'0010
....
....
$UNDEF S0
$DEFINE S0 'B'1000
```

Polecenie $INCLUDE

Za pomocą polecenia $INCLUDE użytkownik może wykorzystać zasoby (np. przetestowane modele bloków cyfrowych) przechowywane w innych plikach. Przykład:

```
$INCLUDE nazwa_pliku
```

gdzie:

nazwa_pliku - to nazwa zewnętrznego pliku, do zawartości którego odwołuje się użytkownik w opisie projektu.

Podanie samej nazwy pliku (ewentualnie z rozszerzeniem) powoduje poszukiwanie przez kompilator pliku w bieżącym (domyślnym) katalogu. Aby uniknąć niejednoznaczności, zamiast samej nazwy można podawać kompletną ścieżkę dostępu do pliku.

Dopuszczalne jest zagnieżdżanie odwołań za pomocą polecenia $INCLUDE, czyli plik dołączany (zewnętrzny) może się tak-

Reguły indeksowania zmiennych

Zakres indeksowania powinien się mieścić w przedziale:

0...*n*, przy czym *n* < 32 (*n* jest zawsze liczbą dziesiętną)

że odwoływać do pliku dołączanego. Dopuszczalna liczba zagnieżdżeń nie została jawnie określona, z doświadczeń wynika, że CUPL bez trudu radzi sobie nawet z 20-krotnymi.

Polecenie $IFDEF

Za pomocą polecenia $IFDEF można poddać kompilacji warunkowej wybrane fragmenty opisu umieszczone w pliku.

Przykład:

```
$IFDEF argument
```

gdzie:

argument - to nazwa stałej, której obecność deklaracji (za pomocą polecenia $DEFINE) jest sprawdzana przez kompilator.

Fragment pliku poddawany kompilacji zaczyna się od miejsca zdefiniowania (za pomocą polecenia $DEFINE) stałej będącej argumentem polecenia $IFDEF, aż do miejsca wystąpienia jednego z poleceń: $ELSE lub $ENDIF. W przypadku, gdy stała będąca argumentem polecenia $IFDEF nie została zdefiniowana za pomocą polecenia $DEFINE, opis zawarty w pliku jest ignorowany aż do momentu wystąpienia jednego z poleceń: $ELSE lub $ENDIF.

Przykład:

```
$IFDEF argument_1
outA=inA & inB;
outB=inC # inA;
$ENDIF
```

Przedstawiona powyżej część opisu nie będzie brana przez kompilator pod uwagę, jeżeli wcześniej nie wystąpi polecenie:

```
$DEFINE argument_1
```

Polecenie $IFNDEF

Polecenie $IFNDEF działa przeciwnie do opisanego wcześniej polecenia $IFDEF, tzn. kompilowana jest ta część opisu, która znajduje się pomiędzy poleceniem $IFNDEF a jednym z poleceń: $ELSE lub $ENDIF, lecz tylko wtedy, gdy stała będąca argumentem polecenia nie została wcześniej zdefiniowana za pomocą polecenia $DEFINE.

Przykład:

```
$IFNDEF argument
```

gdzie:

argument - to nazwa stałej, której obecność deklaracji (za pomocą polecenia $DEFINE) jest sprawdzana przez kompilator.

Poniższy fragment opisu:

```
$IFNDEF argument_1
outA=inA & inB;
outB=inC # inA;
$ENDIF
```

będzie kompilowany tylko wtedy, jeżeli wcześniej nie zdefiniowano stałej *argument_1*. W przeciwnym przypadku, ta część opisu zostanie pominięta przez kompilator.

Piotr Zbysiński, EP
piotr.zbysinski@ep.com.pl