Układy programowalne, część 1

Układy programowalne cieszą się dużą i ciągle rosnącą popularnością. Elektronicy często ocierają się o modne hasła (jak choćby PSoC czy FPGA), nieco gorzej jest z wiedzą o tym, co one w praktyce znaczą i jakie korzyści może z nich „wyciągnąć" elektronik.

Wszystkim zainteresowanym tematyką PLD proponujemy ekspresowy kurs, który przez kilka kolejnych miesięcy będziemy publikować w EP. W jego trakcie pokażemy zarówno łatwo dostępne (bezpłatne!) narzędzia do realizacji projektów, jak i układy, oczywiście wszystko w praktyce.

Zacznę od (nieco) rozczarowującego wstępu. Czytelnicy, którzy liczą na spektakularne pokazy możliwości współczesnych układów PLD, tzn. implementację w nich serwerów sieciowych, mikroprocesorów, kompletnych interfejsów komunikacyjnych czy choćby UART-ów, nie znajdą tu nic dla siebie. Zaczniemy znacznie banalniej, bowiem naszym „kursowym" układem programowalnym będzie GAL22V10 (w wersji z interfejsem JTAG, przystosowany do programowania w systemie), a językiem opisu sprzętu niemodny już dziś, ale bardzo skuteczny CUPL. Dopiero w dalszej kolejności sięgniemy po VHDL i „większe" układy CPLD.

Zniechęceni? Niepotrzebnie! Dowodem na popularność CUPL-a jest choćby fakt, że jest to standardowy język HDL (traktowany na równi z VHDL-em i Verilogiem) zaimplementowany w Protela DXP, a Atmel udostępnia kompletne środowisko projektowe z kompilatorem i symulatorem funkcjonalnym CUPL-a. Bardziej zaawansowanych Czytelników zachęcam do śledzenia opisów IP core'ów publikowanych w EP - tam będzie można znaleźć prawdziwie spektakularne opracowania.

Motto

Ludzie dzielą się na 10 kategorii: tych, którzy znają kod binarny i tych, którzy go nie znają...

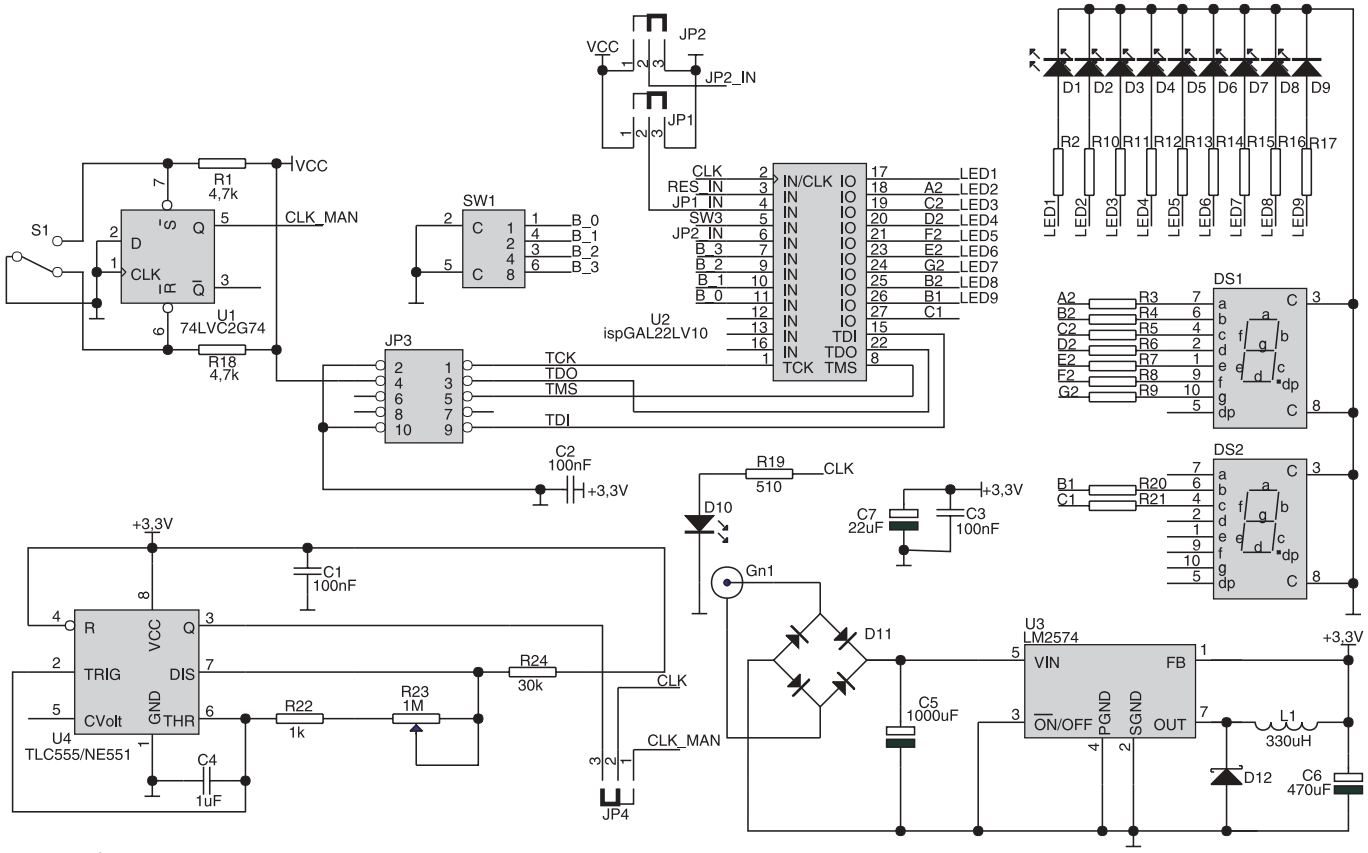
Paweł "Pelos" Dienwebel, www.pelos.pl

Plan kursu

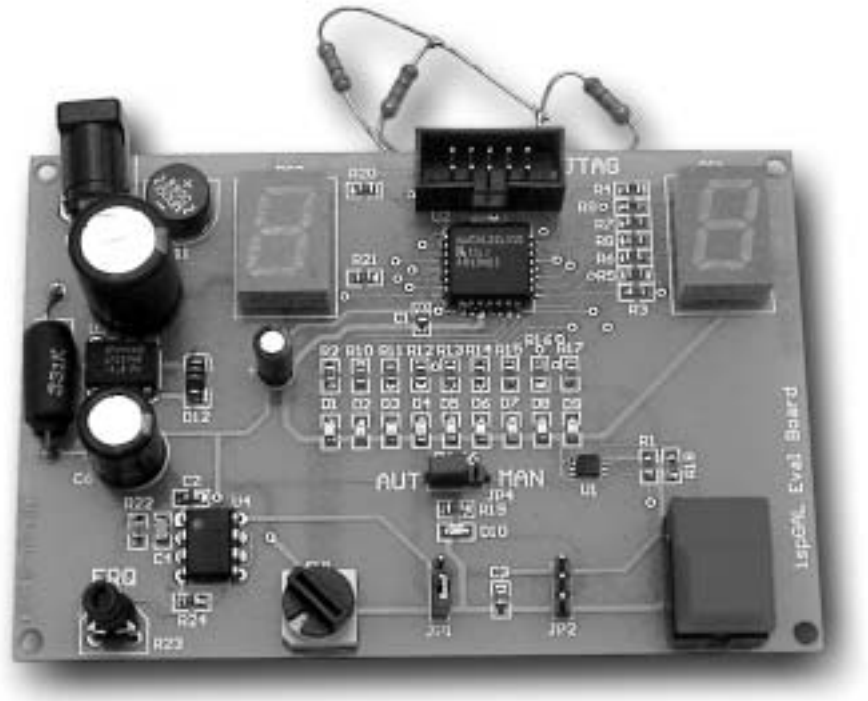
Specjalnie na potrzeby kursu powstał zestaw testowy AVT-559, który współpracuje z programatorem uniwersalnym UnISProg (AVT-560, EP1/2004). Wszystkie przykładowe projekty przedstawione w ramach kursu były uruchamiane i testowane na tym właśnie zestawie.

Kurs będzie się składał z następujących części:

1. Wstępu z opisem zestawu AVT-559 (EP3/2004).



Rys. 1. Schemat elektryczny zestawu testowego



Fot. 2. Wygląd zmontowanego zestawu testowego

2. Prezentacji architektur układów PLD, w tym przede wszystkim GAL22V10 (EP4/2004). Wiadomości zdobyte w tej części pozwolą „kursowiczom" poznać i zrozumieć najważniejsze różnice pomiędzy dostępnymi na rynku układami PLD i dzięki temu świadomie podejść do wyboru układu docelowego dla realizowanego projektu.

3. Opisu języka CUPL (EP5...7/2004), który w znacznym stopniu będzie oparty na przykładach, z którymi praktykujący elektronicy „cyfrowi" mieli okazję się - w nieco innym wykonaniu - zetknąć.

4. Prezentacji obsługi narzędzi wspomagających projektowanie: kompilatora-symulatora WinCUPL firmy Atmel i Protela 99SE (EP8...12/2004).

Podane terminy mogą nieco fluktuować, ale dołożę starań, aby ich dotrzymać.

Zestaw AVT-599

Schemat elektryczny tego zestawu pokazano na **rys. 1**. Zastosowano w nim układ PLD firmy Lattice - ispGAL22LV10, który jest ścisłym odpowiednikiem „klasycznego" GAL22V10 w obudowie PLCC28, ale przystosowanym do programowania w systemie. Podczas kupowania układów do zestawu należy zwrócić uwagę na oznaczenie „LV" w symbolu układu. Nadal są dostępne w sprzedaży układy ispGAL22V10, które są co prawda kompatybilne wewnętrznie i zewnętrznie z innymi układami GAL22V10 w obudowie PLCC28, ale mają wbudowany przestarzały i praktycznie zanikający interfejs służący do programowania wewnętrznej pamięci - *Lattice ISP-download*. W układach z literami „LV" w oznaczeniu, a także w ispGAL-ach nowej generacji (**tab. 1**) zastosowano interfejs zgodny z obowiązującymi obecnie standardami - JTAG. Alternatywą dla ispGAL22LV10 jest ispGAL22V10AV, który pobiera znacznie mniej prądu, ale - przynajmniej do ostatnich dni lutego 2004 - jest trudny do kupienia w naszym kraju.

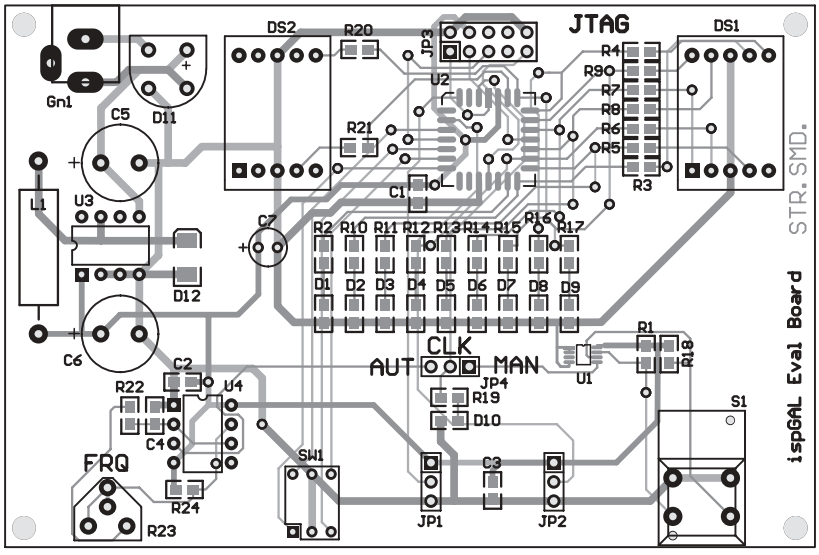
Układy PLD zastosowane w prezentowanym projekcie wymagają napięcia zasilającego o wartości 3,3 V. Zakup odpowiednich stabilizatorów jest rzeczą trudną, stąd decyzja o zastosowaniu stabilizatora impulsowego, wykonanego na układzie *SimpleSwitcher* firmy National Semiconductor (U3). Niebagatelną zaletą stabilizatora impulsowego jest zminimalizowanie strat mocy, w związku z czym można uniknąć konieczności stosowania radiatora. Zalecany zakres napięcia zasilającego wynosi 8...12 VDC.

Użytkownik zestawu ma do dyspozycji:

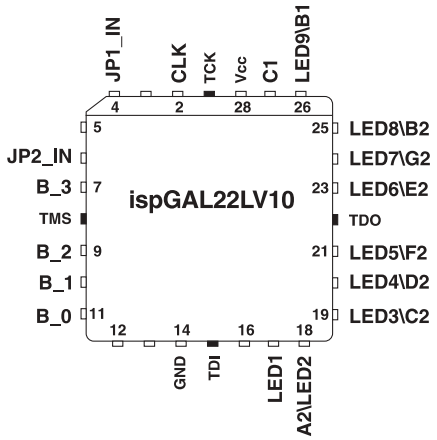
Tab. 1. Dostępne obecnie wersje układów ispGAL22V10

Oznaczenie	Interfejs ISP	Napięcie zasilania [V]	Najszybsze wersje		Pobór prądu
			t_{PD} [ns]	F_{max} [MHz]	
ispGAL22V10AC	JTAG	1,8	2,3	455	150 µA
ispGAL22V10AB	JTAG	2,5	2,3	455	7 mA
ispGAL22V10AV	JTAG	3,3	2,3	455	7 mA
ispGAL22LV10	JTAG	3,3	4,0	250	130 mA
ispGAL22V10	LatticeISP	5,0	7,5	111	140 mA

Uwaga: układy zaznaczone na szaro można stosować w zestawie AVT-599.



Rys. 3. Schemat montażowy płytki drukowanej zestawu



Rys. 4. Funkcje wyprowadzeń układu U2

- ręczny „generator“ sygnału zegarowego, wykonany na układzie U1 i przełączniku S1,
- generator sygnału zegarowego o regulowanej częstotliwości (za pomocą R23) wykonany na układzie TLC551 (U4), będący niskonapięciowym odpowiednikiem standardowego 555,
- nastawnik szesnastkowy SW1, służący do zadawania czterobitowej liczby binarnej,
- dwa jumpery (JP1 i JP2), służące do podawania stanów logicznych na wejścia układu U2 (ich zastosowanie określa użytkownik, budując aplikację),
- 9 diod LED,
- dwa wyświetlacze LED 7-segmentowe (w jednym wykorzystano tylko segmenty B i C).

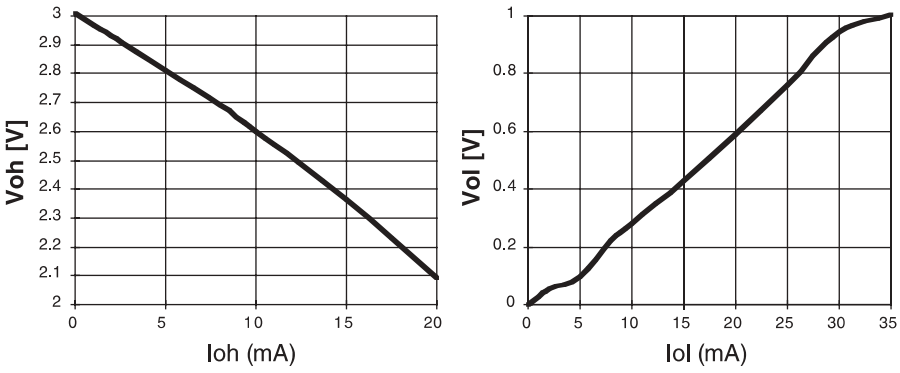
Segmenty wyświetlaczy połączono równolegle z diodami LED, świecą więc one jednocześnie. Źródło sygnału taktującego (ręczne/automatyczne) można wybrać za pomocą zwory JP4. Sygnał zegarowy jest monitorowany za pomocą diody LED (D10) - „1“ jest sygnalizowana jej świeceniem.

Programator UnISProg należy dołączyć do płytki ewaluacyjnej za pomocą kabla taśmowego zakończonego złączem ZWS10. Do tego celu służy gniazdo JP3.

Wygląd zmontowanego zestawu przedstawiono na **fot. 2**. Schemat montażowy płytki drukowanej pokazano na **rys. 3**. Większość zastosowanych elementów ma obudowy przystosowane do montażu powierzchniowego. Są one dość duże, nie powinno więc być problemu z ich przylutowaniem. Pewną trudność może sprawić jedynie przylutowanie układu U1. Najprostszą, a przy tym skuteczną, metodą jest przylutowanie wyprowadzeń w sposób standardowy, co wiąże się z powstaniem zwarć pomiędzy nimi. Nadmiar cyny usuwamy następnie za pomocą miedzianej plecionki, którą należy przyłożyć do wyprowadzeń ulokowanych z jednej strony obudowy i następnie ją przygrzać, co spowoduje wchłonięcie cyny pomiędzy druciki plecionki. Efekt końcowy jest - pomimo prostoty pomysłu - zaskakująco dobry.

Wątpliwość może wzbudzać fakt przylutowania układu U2 bezpośrednio do płytki drukowanej. Powodem tego jest duża liczba dopuszczalnych przez producenta cykli kasowania pamięci EEPROM wbudowanej w układy ispGAL22LV10 - wynosi ona co najmniej 10000 razy. Z punktu widzenia typowych prac ewaluacyjnych żywotność układów ispGAL22LV10 jest więc praktycznie nieograniczona.

Na **rys. 4** pokazano funkcje przypisane wyprowadzeniom układu U2. Większość wyjść jest obciążona dwoma diodami LED, co niesie za sobą ryzyko przeciążenia obwodów wyjściowych. Producent zaleca, żeby nie przekraczać maksymalnego natężenia (dla prądu wpływającego i wypływającego) 8 mA dla każdego z wyjść. Wartości rezystancji rezystorów ograniczających natężenie prądu płynącego przez diody i segmenty wyświetlaczy dobrano w taki sposób, aby nie przekroczyć bezpiecznego natężenia prądu. W wyjątkowych sytuacjach można obciążać wyjścia prądami o większym natężeniu, ale należy się wtedy liczyć ze zmianami napięcia na wyjściach buforów. Ich charakterystyki prądowo-napięciowe pokazano na **rys. 5**.



Rys. 5. Charakterystyki prądowo-napięciowe buforów wyjściowych w układzie ispGAL22LV10 (z lewej strony dla "1" na wyjściu, z prawej strony dla "0" na wyjściu)

WYKAZ ELEMENTÓW

Rezystory
R1, R18: 4,7kΩ 0805
R2, R21: 330Ω 0805
R3...R17, R20: 270Ω 0805
R19: 510Ω 0805
R22: 1kΩ 0805
R23: 1MΩ 0805
R24: 30kΩ 0805

Kondensatory
C1...C3: 100nF 0805
C4: 1µF 0805
C5: 1000µF/25V
C6: 470µF/16V
C7: 22µF/16V

Półprzewodniki
U1: 74LVC2G74
U2: ispGAL22LV10 lub ispGAL22V10AV w obudowie PLCC28
U3: LM2574 DIP8
U4: TLC555/NE551 DIP8
D1...D10: LED w obudowie 0805
D11: mostek 1A/100V
D12: dioda Schotky'ego 1A/30V
DS1, DS2: wyświetlacze LED WK 13 mm

Różne
L1: 330µH
Gn1: gniazdo zasilania DC
JP3: ZWS10
JP1, JP2, JP4: glod-pin 3x1 + jumpery
SW1: nastawnik HEX PT65
S1: przełącznik Digitast

Co dalej?

Za miesiąc przedstawimy architektury układów PLD, ze szczególnym uwzględnieniem budowy i możliwości konfiguracji układów GAL22V10. Będzie to nasz drugi, w tym cyklu, krok w stronę poznania PLD.

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

Układy programowalne, część 2

Tę część kursu przeznaczymy na przedstawienie tajników architektur układów PLD, w tym przede wszystkim układów GAL22V10/ispGAL22V10. Informacje tutaj zawarte są niezbędną podstawą do wprawnego posługiwania się układami GAL.

Historia układów PLD (*Programmable Logic Devices*) sięga końca lat 60., kiedy to powstały pierwsze teoretyczne opracowania, w których wykazywano, że jest możliwe zbudowanie programowanego układu realizującego dowolną logiczną funkcję kombinacyjną i synchroniczną (z wykorzystaniem rejestrów). Podstawą do tych rozważań była praca *An Investigation of the Laws of Thought* George'a Boole'a z 1854 roku, w której wykazał on, że dowolną, najbardziej nawet skomplikowaną funkcję logiczną można stworzyć za pomocą trzech funktorów logicznych: AND, OR i NOT. Wystarczyło więc stworzyć układ, w którym funktory te są ze sobą połączone za pomocą programowanych połączeń, co zapewnia jego maksymalną uniwersalność.

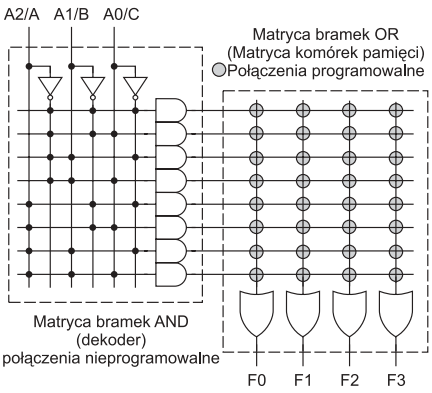
Na początku był chaos...

Jak łatwo zauważyć, możliwych sposobów połączenia funktorów ze sobą jest wiele. Historycznym przykładem układu PLD jest pamięć PROM, w której sygnały wejściowe są podawane na bramki AND o ustalonych połączeniach (tak że spełniają rolę dekodera adresowego pamięci), a ich wyjścia są połączone z programowalną matrycą bramek OR (**rys. 6**). Stosunkowo niewielka elastyczność takiego układu PLD i brak możliwości wygodnego projektowania automatów synchronicznych spowodowały, że prace badawcze trwały, a w ich wyniku powstały układy PLA (*Programmable Logic Array*). Różnią się one od pamięci wprowadzeniem programowanych połączeń w matrycy bramek AND (**rys. 7**), co spowodowało zwiększenie elastyczności tych układów, ułatwiło także optymalne wykorzystywanie ich zasobów.

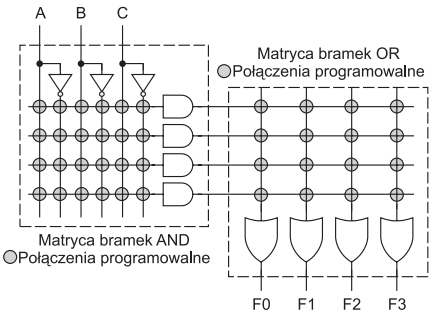
Kolejnym etapem rozwoju układów programowalnych były układy PAL (*Programmable Array Logic*), które charakteryzują się programowalną matrycą AND i skonfigurowaną na stałe (przez producenta) matrycą OR (**rys. 8**). Wprowadzono je do sprzedaży w roku 1978 jako remedium na problemy związane ze stosowaniem układów PLA: trudne przygotowywanie implementowanych projektów (nie było wtedy praktycznie żadnych narzędzi automatyzujących projektowanie!), duży pobór mocy, niewielka szybkość pracy.

Układy PLA i PAL były wykonywane w technologii bipolarnej z pamięcią konfigurującą typu PROM (jednokrotnie programowalną).

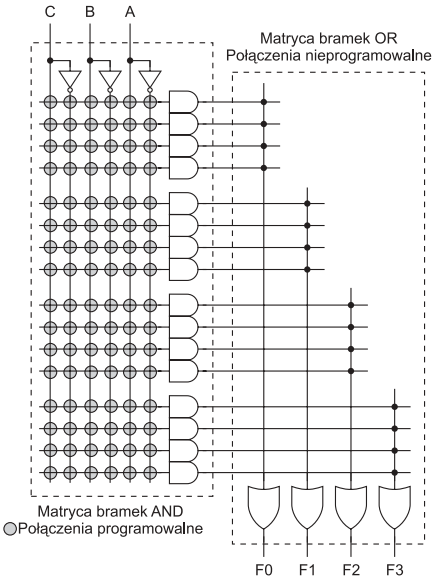
George Boole (1815-1864) w opublikowanym w 1854 roku "dziele życia" (*An Investigation of the Laws of Thought*) stworzył podstawy rozwoju techniki cyfrowej i później układów PLD.



Rys. 6. Budowa logiczna pamięci ROM/PROM



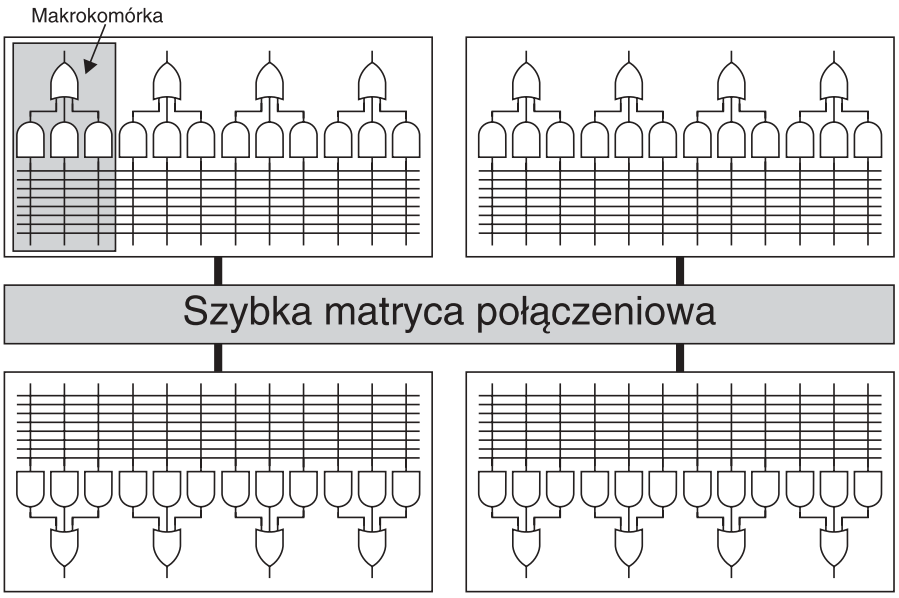
Rys. 7. Budowa logiczna układów PLA



Rys. 8. Budowa logiczna układów PAL

Bliżej współczesności...

Być może niektórzy Czytelnicy poczują się zawiedzeni, ale w tym momencie moglibyśmy przejść do omawiania architektury układów GAL22V10. Układy te powstały bowiem na bazie pomysłów z lat 70. Zaskakujące połączenie: nowoczesna (ciągle nieco awangardowa) technologia, bezpośrednio wykorzystująca pomysły z czasów - dla współczesnych - historycznych.



Rys. 9. Budowa logiczna układów CPLD

Zanim jednak przejdziemy do zgłębiania tajników układów GAL22V10, na chwilę powrócimy do historii rozwoju układów PLD, bo oferują one znacznie większe możliwości niż było to możliwe „za panowania" GAL22V10.

Ewolucja podążyła dwiema ścieżkami:

- Powiększania zasobów dostępnych w pojedynczych układach, poprzez powielanie komórek opartych na matrycach PAL. W ten sposób postały układy CPLD (*Complex Programmable Logic Devices* - **rys. 9**). Wszystkie współczesne układy CPLD wyposażono w nieulotne pamięci konfigurujące (EEPROM lub Flash), których zawartość może być wielokrotnie zmieniana.
- Zastosowaniu radykalnie odmiennej architektury, którą nazwano FPGA (*Field Programmable Gate Array* - **rys. 10**). Układy tego typu składają się z matryc jednakowych lub bardzo do siebie podobnych komórek, w których zintegrowano konfigurowalne zasoby logiczne. Połączenia pomiędzy nimi są możliwe dzięki traktom połączeniowym, których konfigurację może zmieniać użytkownik. W układach FPGA rolę pamięci konfigurującej spełnia zazwyczaj ulotna pamięć SRAM, której zawartość jest każdorazowo po włączeniu zasilania odtwarzana (dane są pobierane z zewnętrznej pamięci nieulotnej).

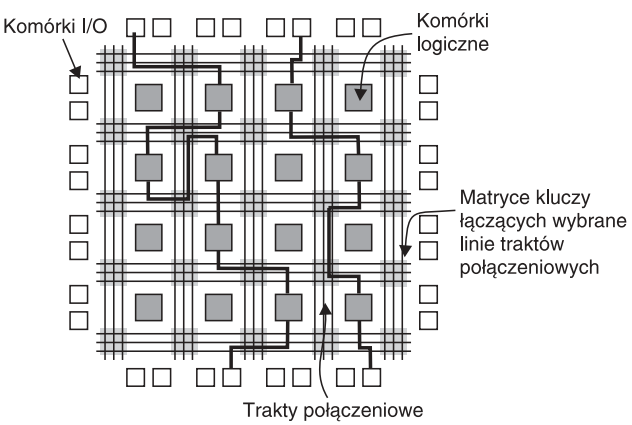
W ostatnich latach producenci wprowadzają do sprzedaży układy o nowatorskiej architekturze nazywanej SoC (*System on a Chip*) lub PSoC (*Programmable System on a Chip*), które składają się z mikroprocesora (często bardzo szybkiego) oraz dużego bloku logiki konfigurowalnej (**rys. 11**). Układy tego typu są coraz chętniej stosowane w aplikacjach, ponieważ pozwalają na budowanie w jednym układzie kompletnych urządzeń spełniających wymagania nawet bardzo zaawansowanych aplikacji.

Tab. 2. Funkcje sygnałów interfejsu JTAG

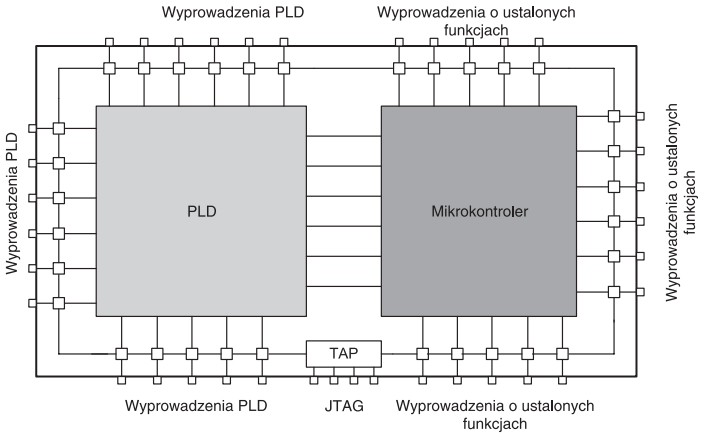
Nazwa	Opis funkcjonalny
TDI Test Data Input	Szeregowe wejście danych do konfiguracji i testowania. Dane z tego wejścia są synchronizowane narastającym zboczem sygnału zegarowego TCK.
TDO Test Data Output	Szeregowe wyjście danych wyprowadzanych z rejestru BST lub pamięci konfiguracji układu. Dane wyjściowe są synchronizowane opadającym zboczem sygnału zegarowego TCK.
TMS Test Mode Select	Wejście sterujące pracą automatu TAP zgodnie z grafem przedstawionym na rys. 13. Ustalenie wartości logicznej na tym wejściu powinno nastąpić przed narastającym zboczem sygnału zegarowego TCK.
TCK Test Clock	Wejście sygnału zegarowego, taktującego automat TAP oraz rejestr instrukcji.

ispGAL22V10 vs GAL22V10

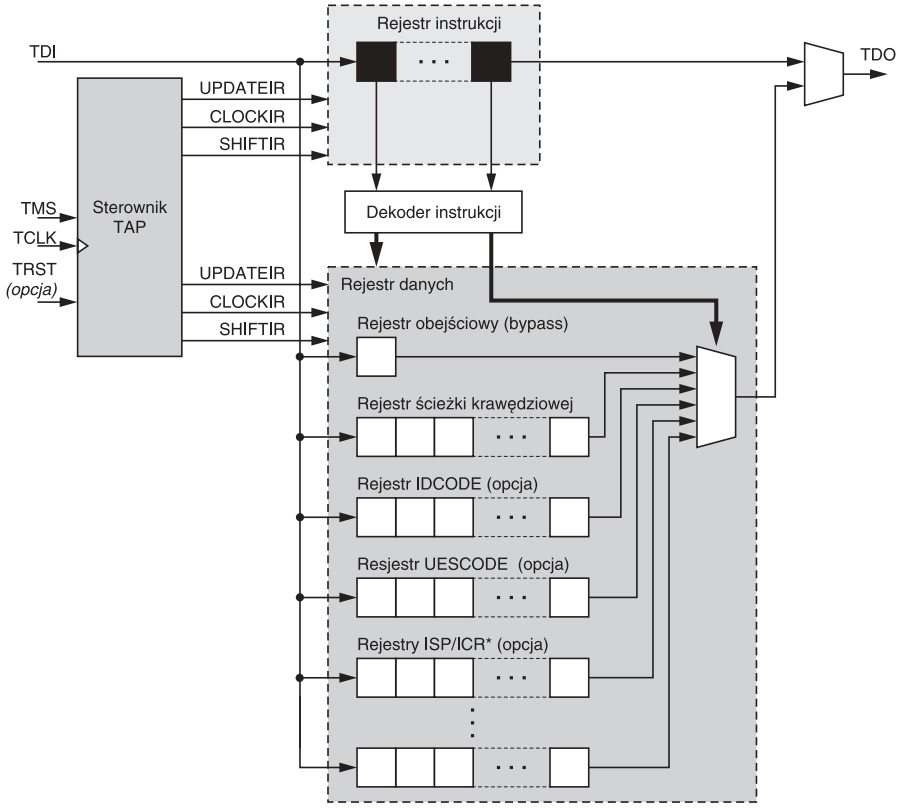
Układ ispGAL22V10 (w dowolnej wersji) jest funkcjonalnym odpowiednikiem standardowego układu GAL22V10. Kompatybilność dotyczy zarówno rozmieszczenia wyprowadzeń, jak i plików JEDEC wykorzystywanych do programowania układu.



Rys. 10. Budowa logiczna układów FPGA



Rys. 11. Budowa logiczna układów SoC



Rys. 12. Schemat blokowy jednej z wielu możliwych implementacji interfejsu JTAG

Tab. 3. Liczba programowalnych iloczynów dostępnych dla OLMC dołączonych do wyprowadzeń układu GAL22V10 (w obudowie PLCC28)

Numer wyprowadzenia I/O	Liczba iloczynów dostępnych dla OLMC
27	8
26	10
25	12
24	14
23	16
21	16
20	14
19	12
18	10
17	8

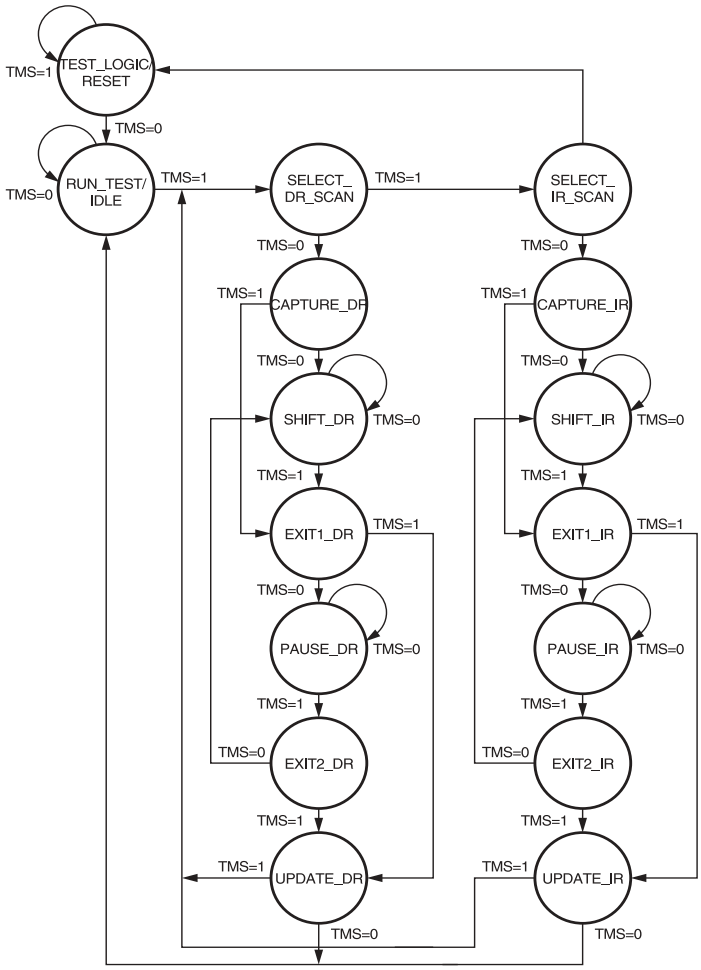
O programowaniu słów kilka

Programowanie ISP (*In System Programming*), obecnie tak modne wśród użytkowników mikrokontrolerów, jest wykorzystywane w układach CPLD i FPGA od początku lat '90. Obecnie obowiązuje jednolity standard - do programowania układów PLD jest powszechnie wykorzystywany interfejs JTAG. Niegdyś jego podstawowym zadaniem było umożliwienie testowania układów cyfrowych i połączeń między nimi, stopniowo zdobył on popularność głównie jako interfejs służący do programowania układów PLD w systemie. Obecnie JTAG jest dostępny nawet w układach o tak niewielkich zasobach logicznych jak w przypadku ispGAL22V10.

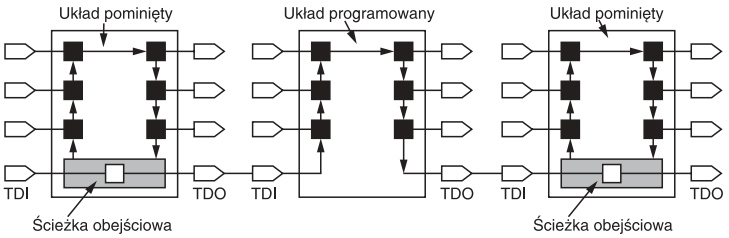
Układy wyposażone w JTAG mają wbudowane specjalne kontrolery zarządzające jego pracą (TAP - *Test Access Point* - **rys. 12**). TAP jest 16-stanowym automatem, którego cykl pracy pokazano na **rys. 13**. Przebiegiem cyklu pracy automatu TAP sterują cztery wyprowadzenia (TMS, TDI, TDO i TCK), których funkcje zestawiono w **tab. 2**. Obsługa programowania ISP została dodana do

Trwałość pamięci konfigurującej

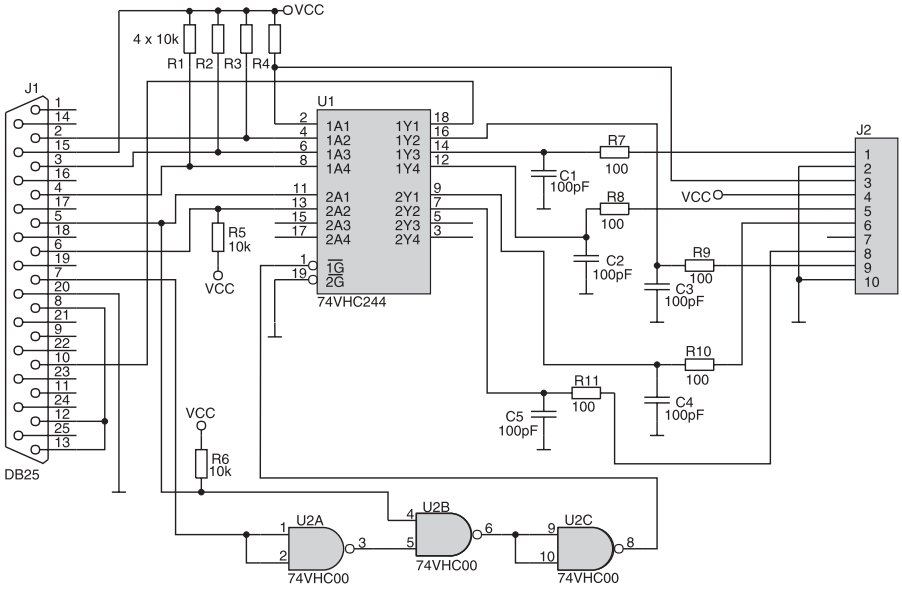
Pamięć EEPROM spełniająca w układzie ispGAL22V10 rolę pamięci konfigurującej jest duża, bowiem producent gwarantuje co najmniej 10000 cykli kasowanie-zapis.



Rys. 13. Cykl pracy automatu TAP



Rys. 14. Układy z interfejsem JTAG można łączyć w łańcuchy (na rysunku pominięto sygnały TMS i TCK, które są dostarczane do wszystkich układów jednocześnie)



Rys. 15. Schemat przykładowego programatora układów PLD firmy Lattice

standardowego JTAG-a „sztucznie" (bo, jak wspomniano, JTAG miał służyć do testowania układów po zamontowaniu w systemie), ale zostało to zrobione w taki sposób, że użytkownik tej „sztuczności" w żaden sposób nie odczuwa.

Za pomocą JTAG-a można programować zarówno pojedyncze układy (jak ma to miejsce m.in. w zestawie, który wykorzystamy podczas kursu), jak i wiele układów połączonych w łańcuch (**rys. 14**). Na rysunku, żeby nie zmniejszać jego czytelności nie narysowano sygnałów TMS i TCK, które są dostarczane równolegle do wszystkich układów wchodzących w skład łańcucha.

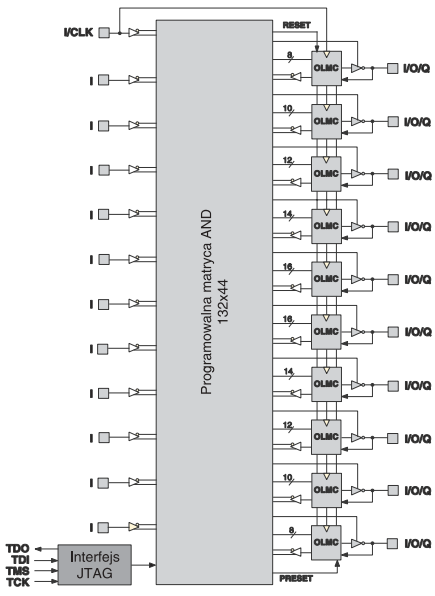
Korzystanie z JTAG-a, pomimo jego dość złożonej budowy, jest nadzwyczaj proste. Rolę programatora spełnia łatwy w wykonaniu interfejs (schemat elektryczny programatora ISP dla układów Lattice pokazano na **rys. 15**), za sterowanie jego pracą odpowiada specjalne oprogramowanie (ispVM), udostępniane przez firmę Lattice bezpłatnie. Najnowszą wersję tego programu oraz wersje dla Linuksa publikujemy także na CD-EP4/2004B. Sposób obsługi tego programu przedstawimy w jednym z kolejnych odcinków cyklu.

PAL vs GAL

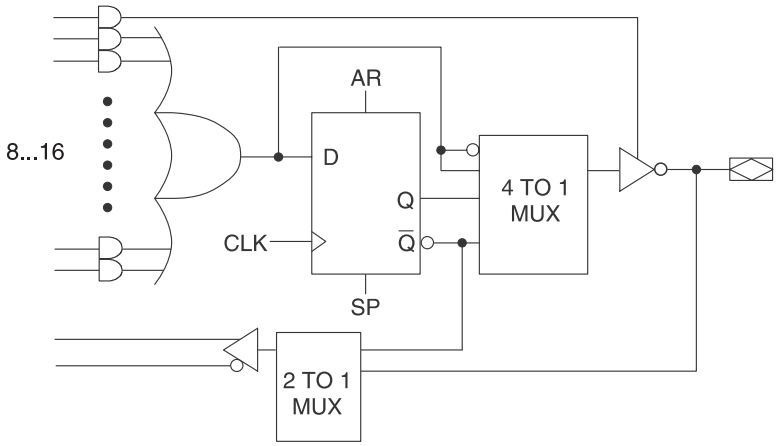
Układy GAL są reprogramowalnymi, uniwersalnymi wersjami układów PAL. Ich komórki wyjściowe są tak elastyczne, że można je skonfigurować w dowolny tryb obsługiwany przez układy PALxxR (z wyjściami rejestrowymi), PALxxH (bez inwerterów na wyjściu) i PALxxL (z inwerterami na wyjściu).

Nasz bohater: ispGAL22V10

Jak już wspomniano, architektura układu ispGAL22V10 jest bezpośrednim rozwinięciem klasycznych, bipolarnych PAL-i. Układ jest zbudowany (**rys. 16**) z 10 konfigurowalnych makrokomórek OLMC (*Output Logic Macro Cell*), na wyjściu których znajdują



Rys. 16. Architektura układów (isp)GAL22V10



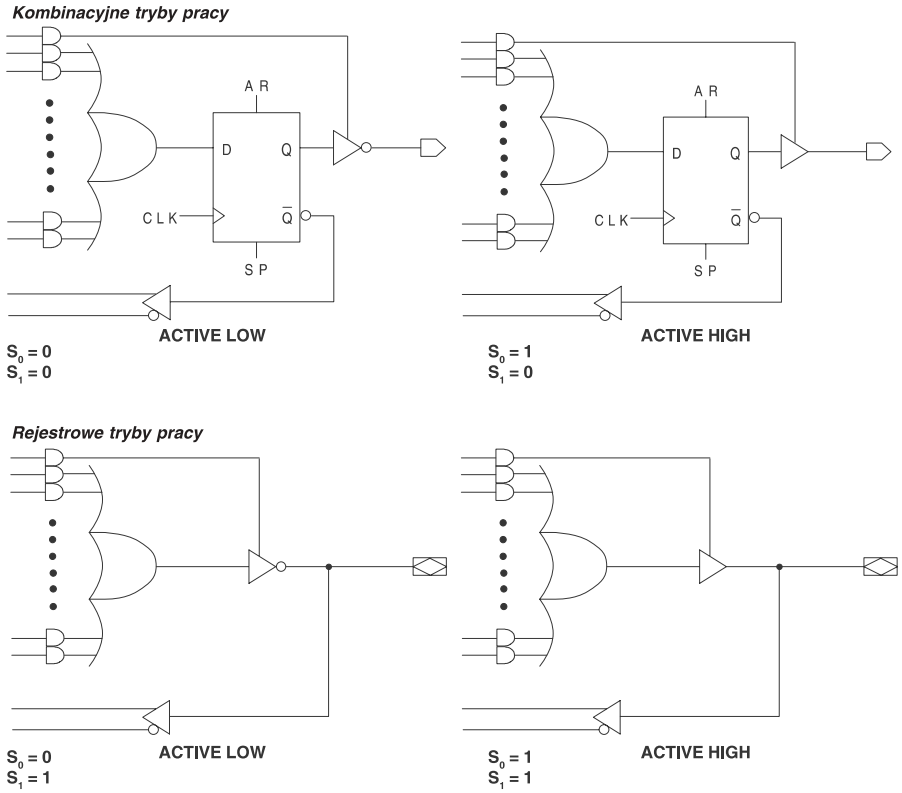
Rys. 17. Budowa komórki OLMC układu (isp)GAL22V10

się bufory trójstanowe. Sygnały na wejścia OLMC są podawane z programowanej matrycy bramek AND o organizacji 44 (liczba wejść bramek AND) x 132 (liczba bramek AND).

Budowę OLMC pokazano na **rys. 17**. W jej skład wchodzi przerzutnik D z wejściami: asynchronicznego zerowania (AR) i synchronicznego ustawiania (SP), na którego wejście D jest podawany sygnał wytwarzany zaprogramowaną przez użytkownika przez sumę iloczynów sygnałów wejściowych (oznaczonych literą I) i sygnałów podawanych zwrotnie na matrycę programowalną, wytwarzanych w innych OLMC. W zależności od lokalizacji OLMC, liczba dostępnych dla OLMC iloczynów waha się od 8 do 16 (według wzoru: „zewnętrzne" OLMC mają ich 8, następne 10, aż do 16 dla OLMC umieszczonych centralnie - **tab. 3**).

Multipleksery widoczne na rys. 17 są wykorzystywane wyłącznie w celu skonfigurowania trybu pracy OLMC (za ich konfigurację odpowiadają bezpieczniki S_0 i S_1, oddzielne dla każdej OLMC), nie można więc zmieniać ich stanu podczas pracy układu. Na **rys. 18** pokazano możliwe konfiguracje OLMC. W każdym z możliwych trybów pracy, sygnały wytwarzane w OLMC są zwrotnie przesyłane na programowalną matrycę AND, dzięki czemu funkcje logiczne realizowane w OLMC mogą być wykorzystywane przez inne OLMC (oczywiście, jeśli występuje taka konieczność). W przypadku pracy OLMC w trybie rejestrowym nie ma możliwości wykorzystania linii I/O jako wejściowej (jest na stałe skonfigurowana jako trójstanowe wyjście). Sygnał CLK dostarczany na wejścia zegarowe wszystkich przerzutników jest na stałe przypisany do wejścia I/CLK (wyprowadzenie numer 2 w obudowie PLCC28). Niestety, w układach GAL22V10 nie ma możliwości taktowania przerzutników niezależnymi sygnałami zegarowymi - w trybie rejestrowym OLMC mają do dyspozycji tylko jeden, wspólny sygnał. Ograniczenie to nie dotyczy sygnałów AR i SP, które są wytwarzane indywidualnie dla każdego przerzutnika.

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



Rys. 18. Możliwe sposoby skonfigurowania OLMC w układach (isp)GAL22V10

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

Układy programowalne, część 4

W tej części cyklu dokończymy omówienie poleceń preprocesora kompilatora CUPL, omówimy także rozszerzenia nazw sygnałów przydatne podczas realizacji projektów na układach GAL16V8, 18V8, 20V8 i 22V10.

Polecenie $ENDIF

Jak wspomniano w poprzednim odcinku cyklu, polecenie to służy do zaznaczania końca opisu HDL kompilowanego warunkowo, którego początek wskazuje jedno z poleceń: $IFDEF lub $IFNDEF.

Format polecenia jest następujący:

```
$ENDIF
```

Polecenia warunkowej kompilacji mogą być zagnieżdżane, przy czym należy pamiętać o tym, żeby każdy poziom zagnieżdżenia zaznaczony poleceniem $IFDEF lub $IFNDEF został „zamknięty” poleceniem $ENDIF.

Przykład:

```
$IFDEF argument_1
pin 1 = we1;
pin 2 = we2;
$IFDEF argument_2
pin 3 = ramka_a;
pin 4 = rom_sel;
$ENDIF
pin 5 = in_rcs;
pin 6 = rs_dek;
$ENDIF
```

Polecenie $ELSE

Jest to polecenie, za pomocą którego można tworzyć lokalne rozgałęzienia kompilacji warunkowej, której obszar zaznaczono jednym z poleceń: $IFDEF lub $IFNDEF.

Działanie polecenia jest następujące: jeżeli warunek testowany przez polecenia $IFDEF i $IFNDEF jest spełniony (czyli opis znajdujący się za nimi jest kompilowany), to opis po poleceniu $ELSE jest ignorowany. Jeżeli natomiast testowany przez polecenia $IFDEF i $IFNDEF warunek nie jest spełniony, opis HDL znajdujący się po nich jest ignorowany, a kompilacji jest poddawany opis po poleceniu $ELSE.

Format polecenia jest następujący:

```
$ELSE
```

Przykład:

```
$DEFINE Wersja 1
$IFDEF Wersja
pin 1 = mem_req;
pin 2 = io_req;
$ELSE
pin 1 = io_req;
pin 2 = mem_req;
$ENDIF
```

W przedstawionym przykładzie można zdecydować o sposobie przypisania sygnałów do wyprowadzeń układu zmieniając nazwę zdefiniowanej stałej lub przez usunięcie linii zawierającej jej definicję ($DEFINE Wersja 1).

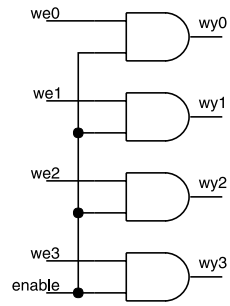
Polecenie $REPEAT

Za pomocą tego polecenia można automatyzować tworzenie opisów HDL, które składają się z wielu takich samych bloków funkcjonalnych, różniących się jedynie indeksem. Zakres wartości indeksu musi się mieścić w zakresie 0...1023.

Format tego polecenia jest następujący:

```
$REPEAT index=[liczba_0, liczba_1,...liczba_n]
    powielany opis
    z elementami
    indeksowanymi
$REPEND
```

Preprocesor przed kompilacją projektu „rozwija” opis, tworząc odpowiednią (wynikającą z zakresu indeksu) liczbę bloków funkcjonalnych. Indeksowanie nie musi przebiegać kolejno od liczby 0 do liczby *n*, ale w takim przypadku konieczne jest jawne podanie kolejnych wartości indeksów



Rys. 19

([liczba_0, liczba_1,...liczba_n]). Jeżeli indeksowanie ma przebiegać kolejno w podanym przedziale, użytkownik może podać tylko najmniejszą i największą wartość z przedziału indeksowania ([liczba_1..liczba_n]).

Przykład:

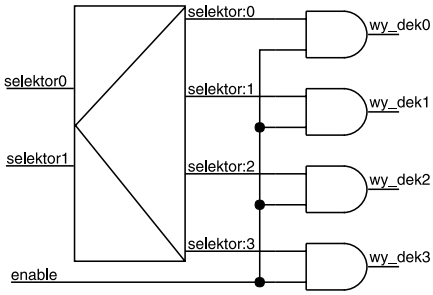
```
$REPEAT i = [0..3]
  wy{i} = we{i} & enable;
$REPEND
```

Wynikiem działania preprocesora jest następujący opis HDL, odpowiadający układowi pokazanemu na **rys. 19**:

```
wy0 = we0 & enable;
wy1 = we1 & enable;
wy2 = we2 & enable;
wy3 = we3 & enable;
```

Oczywiście, za pomocą polecenia $REPEAT można „rozwijać” znacznie bardziej skomplikowane bloki funkcjonalne, niż pokazany w przykładzie. Podczas „rozwijania” opisu można wykorzystywać także operatory arytmetyczne, czego przykład pokazano poniżej:

```
FIELD licznik = [wy2..wy0];
```

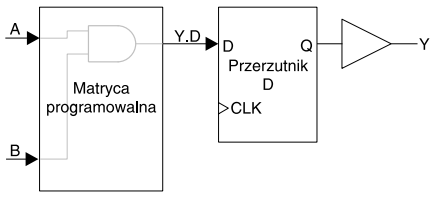


Rys. 20

```
SEQUENCE licznik {
$REPEAT i = [0..7]
  PRESENT {i}
  IF zliczaj & reset NEXT {(i+1)%(8)};
  IF !reset NEXT {0};
  DEFAULT NEXT {i};
$REPEND
}
```

Przedstawiony przykład po działaniu preprocesora wygląda następująco:

```
FIELD licznik = [wy2..wy0];
SEQUENCE licznik {
PRESENT 0
  IF zliczaj & reset NEXT 1;
  IF !reset NEXT 0;
  DEFAULT NEXT 0;
PRESENT 1
  IF zliczaj & reset NEXT 2;
  IF !reset NEXT 0;
  DEFAULT NEXT 1;
PRESENT 2
  IF zliczaj & reset NEXT 3;
  IF !reset NEXT 0;
  DEFAULT NEXT 2;
PRESENT 3
  IF zliczaj & reset NEXT 4;
  IF !reset NEXT 0;
  DEFAULT NEXT 3;
PRESENT 4
  IF zliczaj & reset NEXT 5;
  IF !reset NEXT 0;
  DEFAULT NEXT 4;
PRESENT 5
  IF zliczaj & reset NEXT 6;
  IF !reset NEXT 0;
  DEFAULT NEXT 5;
PRESENT 6
  IF zliczaj & reset NEXT 7;
  IF !reset NEXT 0;
  DEFAULT NEXT 6;
PRESENT 7
  IF zliczaj & reset NEXT 0;
  IF !reset NEXT 0;
  DEFAULT NEXT 7;
}
```



Rys. 21

Polecenie $REPEND

Uważni Czytelnicy zauważyli z pewnością, że polecenie $REPEND jest używane do zaznaczania końca fragmentu opisu HDL, który jest „rozwijany" zgodnie z ustalonym przez użytkownika indeksowaniem. Każde polecenie $REPEAT musi zostać odwołane przez $REPEND, nie jest możliwe zagnieżdżanie poleceń $REPEAT.

Format tego polecenia jest następujący:

```
$REPEND
```

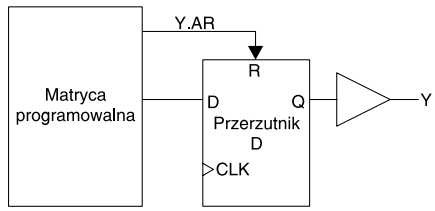
Polecenie $MACRO

Za pomocą tego polecenia użytkownik może tworzyć własne makrofunkcje wywoływane w opisie HDL nadaną im nazwą i odpowiednimi parametrami. Jeżeli w projekcie makrofunkcja nie jest wykorzystywana, to jej opis HDL nie jest kompilowany.

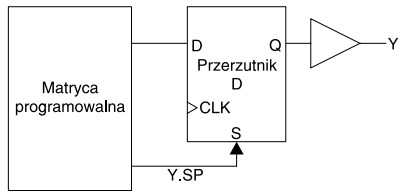
W opisie makrofunkcji można używać operatorów arytmetycznych, przy czym – podobnie jak ma to miejsce w przypadku polecenia $REPEAT – wszystkie działania muszą być ujęte w nawiasy półokrągłe.

Format polecenia:

```
$MACRO nazwa argument_1 argument_
2...argument_n
      opis HDL makofunkcji
$MEND
```



Rys. 22

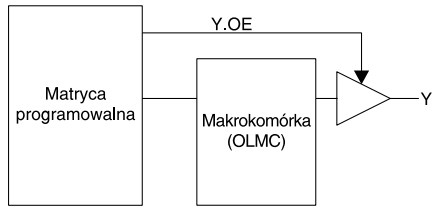


Rys. 23

Za pomocą makrofunkcji można na przykład tworzyć biblioteki gotowych bloków logicznych, na przykład będących odpowiednikami standardowych układów TTL.

Przykład (odpowiednik układu TTL 7474 – podwójny przerzutnik D):

```
$MACRO TTL7474 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10
P11 P12 P13 P14
P5.d = P2;
P5.ck = !P3;
P5.ar = !P1;
P5.ap = !P4;
P6 = !P2;
P9.d = P12;
```



Rys. 24

W przypadku, gdy nie wykorzystujemy wejść lub wyjść makrofunkcji, w chwili jej wywołania w miejsca parametrów odpowiadającym liniom niewykorzystanym należy wstawić słowo kluczowe NC.

```
P9.ck = !P11;
P9.ar = !P13;
P9.ap = !P10;
P8 = !P12;
$MEND
```

Przykład (odpowiednik układu TTL 7402 – czterech dwuwejściowych bramek NOR):

```
$MACRO TTL7402 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10
P11 P12 P13 P14
P3 = !(P1 # P2);
P6 = !(P4 # P5);
P8 = !(P9 # P10);
P11 = !(P12 # P13);
$MEND
```

Tworzone za pomocą makrofunkcji bloki funkcjonalne mogą być parametryzowane, co oznacza, że odpowiednio przygotowany opis licznika lub dekodera może być wykorzystywany w opisie projektu dla dowolnej liczby bitów (czyli np. jako licznik 4- i 12-bitowy lub jako dekoder 2->4 i 3->8 linii).

Przykład:

```
$MACRO dekoder l_bitow sel wy inh;
  FIELD dek = [sel{l_bitow-1}..0];
  $REPEAT i = [{(2**l_bitow)-1}..0]
     wy{i} = dek:'h'{i} & inh;
  $REPEND
$MEND
```

Wywołanie tej makrofunkcji (w wyniku rozwinięcia której powstaje dekoder z wejściem zezwalającym) wygląda na przykład następująco:

```
dekoder (2, selektor, wy_dek, inh_ext);
```

Po takim wywołaniu preprocesor generuje następujący opis HDL (odpowiadający mu, uproszczony schemat logiczny pokazano na **rys. 20**):

```
FIELD dek = [selektor1..0];
  wy_dek3 = dek:'h'3 & inh_ext;
  wy_dek2 = dek:'h'2 & inh_ext;
  wy_dek1 = dek:'h'1 & inh_ext;
  wy_dek0 = dek:'h'0 & inh_ext;
```

Dobrym zwyczajem jest przechowywanie makrodefinicji w zewnętrznym pliku. W CUPL-u przyjęto, że rozszerzeniem nazw plików zawierających makrofunkcje jest litera „m" (*.m). Korzystanie z tych plików umożliwia polecenie $INCLUDE, które opisano w poprzednim numerze EP.

Polecenie $MEND

Jest to polecenie, za pomocą którego jest zaznaczany koniec każdej makrofunkcji (czyli polecenia $MACRO i $MEND muszą tworzyć pary i nie mogą być zagnieżdżane).

Format:

```
$MEND
```

Rozszerzenia nazw zmiennych

Rozszerzenia nazw zmiennych są wykorzystywane w celu określenia specjalnej funkcji sygnału, źródła jego pochodzenia lub celu. Dzięki rozszerzeniom użytkownik może bezpośrednio operować na sygnałach niedostępnych na zewnątrz układu scalonego.

Tab. 11. Rozszerzenia nazw zmiennych przydatne podczas korzystania z układów GAL22V10

Nazwa	Położenie względem znaku równości w równaniach logicznych	Opis
.AR	L	Asynchronous Reset – asynchroniczne zerowanie przerzutnika
.SP	L	Synchronous Preset – synchroniczne ustawianie przerzutnika
.OE	L	Output Enable – sygnał sterujący pracą bufora trójstanowego
.D	L	Wejście danych przerzutnika D

Przykład:

```
pin 1 = A;
pin 2 = B;
pin 19 = Y;
Y.D = A & B;
```

Wynikiem przedstawionego zapisu jest przypisanie do wejścia D przerzutnika iloczynu logicznego sygnałów A i B, jak pokazano to na **rys. 21**.

CUPL obsługuje 42 rodzaje rozszerzeń, z których dla początkujących (przy założeniu, że dalsza część kursu będzie poświęcona głównie projektom realizowanym na układach GAL22V10) najistotniejsze są te, które przedstawiono w **tab. 11**. Na **rys. 22...24** pokazano graficzną interpretację znaczenia przedstawionych rozszerzeń.

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

Układy programowalne, część 5

Czytelnicy, którzy cierpliwie przebrnęli przez wprowadzenie do języka CUPL, znajdą teraz nieco „praktycznej" satysfakcji. Przechodzimy bowiem (powoli) do prezentacji przykładowych projektów implementowanych w układzie ispGAL22V10, który jest „sercem" zestawu ewaluacyjnego AVT-559, opisanego w EP3/2004.

Każda próba systematyzacji nabywanej wiedzy budziła we mnie bunt: po co zaczynać prace od podstaw, skoro chciałbym od razu zająć się zagadnieniami poważnymi? Pewnie wśród Czytelników jest wiele osób podobnie podchodzących do tematu (tak przynajmniej wynika z listów, a przychodzi ich zadziwiająco – jak na PLD – dużo), ale teraz już wiem, że systematyczne pokonywanie etapów poznania jest (przeciętnie rzecz ujmując) lepszym wyjściem, niż porywanie się od razu na zbudowanie kontrolera sieci Ethernet (to oczywiście tylko przykład).

List. 1. Przykładowy opis ilustrujący strukturę pliku źródłowego **.pld*

```
Name   proba;
Partno 999;
Revision  01;
Date   21/05/2004;
Designer  PZb;
Company   EP;
Assembly  PCB01;
Location  U2;
Device G16V8;
Format j;

/***** Wejscia *****/
pin 1 = CLK;  /* Zegar */
pin 2 = CL;   /* Zerowanie */
pin 4 = CA;   /* Wejscie czujnika A */
pin 5 = CB;   /* Wejscie czujnika B */

/***** Wyjscia *****/
pin 12 = ERROR;   /* Wyjscie wskazujace blad */
pin 14 = C_PLUS;
/* Wyjscie zwiekszajace licznik */
pin 15 = C_MINUS;
/* Wyjscie zmniejszajace licznik */
pin [16..19] = [Q0..Q3];

/***** Deklaracje pomocnicze *****/
field NUMER_STANU = [Q2..0];
field WEJSCIA = [CB,CA,CL];

PAUZA = WEJSCIA:'b'000;
A = WEJSCIA:'b'010;
B = WEJSCIA:'b'100;
ERR = WEJSCIA:'b'110;
CLR = WEJSCIA:'b'XX1;

$define S0 'b'000
$define S1 'b'001
$define S2 'b'010
$define S3 'b'011
$define S4 'b'100
$define S5 'b'101
$define S6 'b'110

/***** Opis HDL *****/
sequence NUMER_STANU {
present S0  if A         next S1;
            if B         next S4;
            if PAUZA     next S0;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
present S1  if PAUZA     next S2;
            if A         next S1;
            if CLR       next S0;
            if ERR       next S0     out ERROR;

present S2 if B          next S3;
            if PAUZA     next S2;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
present S3 if PAUZA      next S0     out C_PLUS;
            if B         next S3;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
present S4 if PAUZA      next S5;
            if B         next S4;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
present S5 if A          next S6;
            if PAUZA     next S5;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
present S6 if PAUZA      next S0     out C_MINUS;
            if A         next S6;
            if CLR       next S0;
            if ERR       next S0     out ERROR;
}
```

Zaczniemy więc od przydatnych „banałów", stopniowo przechodząc do przykładów, które pokażą prawdziwe możliwości „małych" układów PLD.

Format pliku wejściowego

Standardowym rozszerzeniem nazwy pliku źródłowego dla kompilatorów CUPL-a jest **.pld*. Format pliku źródłowego i jego organizacja są takie same, niezależnie od tego, w jakim systemie będzie on kompilowany.

Plik źródłowy (przykład pokazano na **list. 1**) składa się z trzech części.

1. Nagłówek, w skład którego wchodzą następujące pola rozpoczynające się od słów kluczowych:

```
Name        proba;
Partno      999;
Revision    01;
Date        21/05/2004;
Designer    PZb;
Company     EP;
Assembly    PCB01;
Location    U2;
Device      G22V10;
Format      ij;
```

Każda linia musi być zakończona średnikiem. Znaczenie poszczególnych wpisów jest następujące:

`Name` – zawiera nazwę projektu, której maksymalna długość wynosi 32 znaki. Nazwa ta nie musi być taka sama jak nazwa pliku źródłowego (**.pld*), ale należy pamiętać, że pliki będące wynikiem kompilacji projektu będą nosiły nazwy takie same jak nazwa wpisana w to pole (będą się różniły tylko rozszerzeniami).

`Partno` – pole służące do wpisania firmowego oznaczenia projektowanego układu, co ma ułatwić identyfikację układu.

`Revision` – numer wersji projektu. Aktualizacja tego numeru w niektórych systemach CUPL odbywa się automatycznie po zmianie zawartości pliku źródłowego, w niektórych wersjach CUPL-a (m.in. w wersji atmelowskiej dla Windows) numer wersji nie jest aktualizowany automatycznie. Słowo kluczowe `revision` można zastąpić skrótem `rev`.

`Date` – w niektórych wersjach CUPL-a jest tu wstawiana data utworzenia pliku źródłowego, w niektórych wersjach jest automatycznie wprowadzana data ostatniej aktualizacji.

`Designer` – pole przeznaczone na wpisanie nazwiska projektanta.

`Company` – pole przeznaczone na wpisanie nazwy firmy, w której projekt jest realizowany.

`Assembly` – identyfikator płytki drukowanej, na której ma być montowany projektowany układ. Alternatywnie słowo kluczowe `assembly` można zastąpić skrótem `assy`.

`Location` – pole przeznaczone na współrzędne określające miejsce montażu projektowanego układu na płytce drukowanej. Alternatywnie słowo kluczowe `location` można zastąpić skrótem `loc`.

`Device` – w tym polu jest wpisywana mnemoniczna nazwa określająca docelowy układ PLD. W zależności od systemu, w którym zintegrowano kompilator CUPL-a, liczba dostępnych układów i odnoszące się do nich mnemoniki mogą być różne. W naszym przypadku (układ GAL22V10 w obudowie PLCC28, który zastosowano w zestawie AVT-559) będziemy stosować nazwę `g22v10lcc`. Jeżeli realizowany projekt nie będzie implementowany w konkretnym typie układu PLD, po słowie kluczowym `device` można wpisać `virtual`, co oznacza wirtualny układ PLD. W CUPL-u układ wirtualny ma architekturę PAL z opcjonalnymi rejestrami wejściowymi i nieograniczoną liczbą wejść bramek AND i OR ulokowanych w „matrycy programowanej".

`Format` – za pomocą dyrektyw wpisanych w tej linii projektant może określić, jakie pliki wynikowe są tworzone podczas kompilacji pliku **.pld*. Dostępne są następujące opcje:
h – powoduje utworzenie pliku

w formacie ASCII-hex

i – powoduje utworzenie pliku w formacie Signetics HL,

j – powoduje utworzenie pliku w formacie JEDEC (najczęściej stosowane w przypadku układów GAL).

Wpisanie dwóch lub trzech liter odpowiadających opisanym opcjom po słowie kluczowym `format` powoduje wygenerowanie przez kompilator odpowiedniej liczby plików wynikowych.

Jak widać, w nagłówku znajduje się wiele zbędnych informacji, których wprowadzanie można pominąć. W takim przypadku kompilator każdorazowo informuje użytkownika o braku oczekiwanej linii w pliku, ale poddaje go normalnej kompilacji. Aby uniknąć niepotrzebnego alarmowania, można w zbędne pola wpisać dowolne słowo (np. „brak") lub literę.

2. Deklaracje wejść, wyjść i węzłów, które służą do określania zewnętrznego interfejsu projektowanego układu oraz „ręcznego" przypisania sygnałów wewnętrznych do określonych (charakterystycznych) miejsc wewnątrz układu scalonego (węzłów wewnętrznych, często zwanych węzłami „zagrzebanymi").

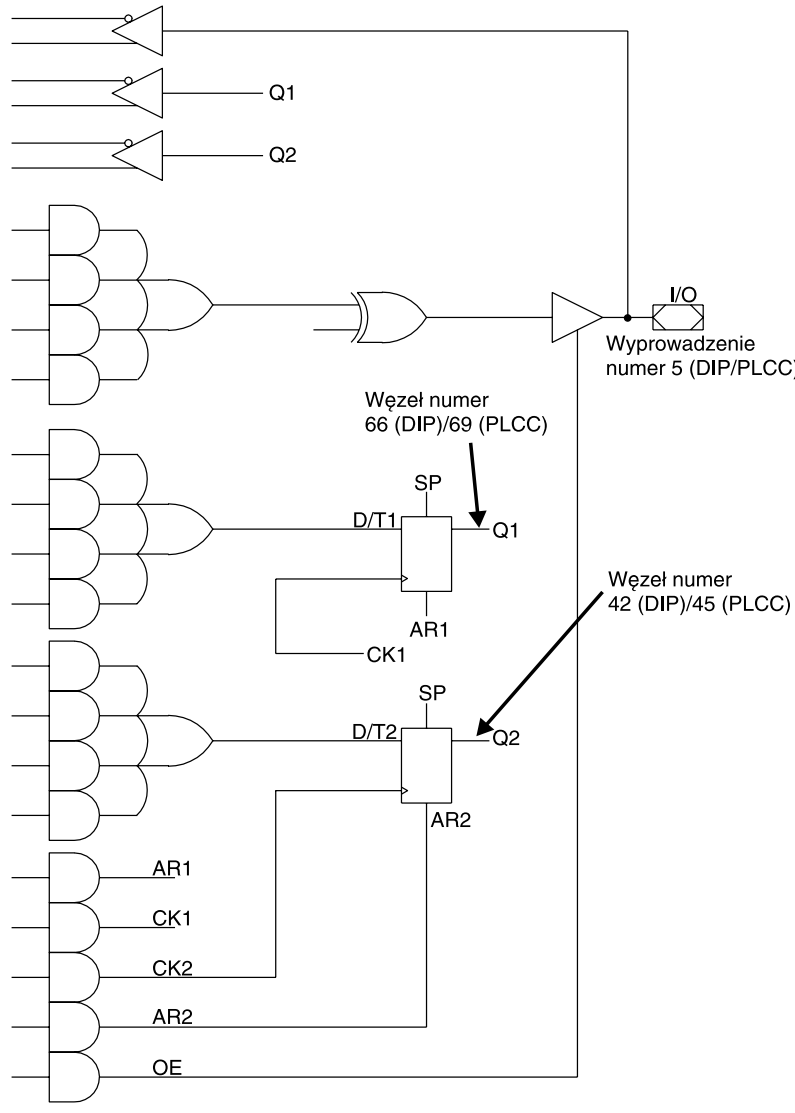
Przykładowe deklaracje przedstawiono poniżej:

```
/***** Wejscia *****/
PIN 1 = CLK;/* wejscie zegarowe */
PIN 2 = RES;/* wejscie zerowania */
PIN 3 = RxT;/* wejscie danych */

/***** Wyjscia *****/
PIN 19 = wy_clk_8; /* wyjscie preskalera 1:8 */
PIN 18 = wy_clk_4; /* wyjscie preskalera 1:4 */

/*****
Przypisanie sygnałów do węzłów wewnętrznych
*****/
PINNODE 213 = Q_X;
/* przypisanie sygnału Q_X do węzła 213 */
PINNODE 199 = RES_XTAL;
/* przypisanie sygnału RES_XTAL do węzła 199 */
```

Jak widać, w deklaracjach projektant nie określa w jawny sposób kierunku wyprowadzeń (wejście/wyjście), ustala to kompilator na podstawie opisu HDL. Deklarowanie węzłów zagrzebanych nie jest niezbędne i w praktyce (zwłaszcza w przypadku realizacji projektów na układy PLD o niewielkich zasobach) jest rzadko stosowane. W przypadku korzystania z bezpośrednich odwołań do węzłów zagrzebanych należy pamiętać, że liczby określające ich numery zależą od typu obudowy. Przykładowo, w przypadku układu ATV2500 firmy Atmel wyjście przerzutnika Q1 z komórki przypisanej

Węzły zagrzebane...

...są to miejsca w strukturze logicznej układów PLD określane w języku CUPL liczbą z zakresu 0...512. Informacje o lokalizacji takich węzłów były publikowane w notach katalogowych układów

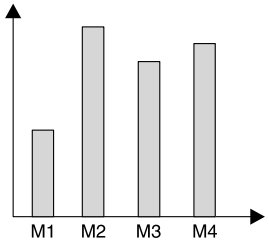


Rys. 25

Tab. 12. Dostępne sposoby minimalizacji funkcji logicznych w CUPL-u

n	Algorytm minimalizacji	Opis
0	Bez minimalizacji	Optymalizacja wyłączona, opcja zalecana podczas implementacji projektu w pamięci PROM/EPROM/EEPROM.
1	Quick	Metoda zrównoważona, zapewniająca krótki czas optymalizacji, wymagająca niewielkich zasobów pamięci, charakteryzująca się relatywnie słabą skutecznością optymalizacji.
2	Quine McCluskey	Najbardziej skuteczna minimalizacja, wymagająca dużych zasobów pamięci i - w przypadku dużej liczby zmiennych - wymagająca długotrwałych obliczeń.
3	Presto	Średnia skuteczność minimalizacji, nie wymaga pamięci o dużej pojemności. Szczególnie dobre wyniki daje podczas minimalizacji projektów implementowanych w układach IFL.
4	Expresso	Metoda o większej skuteczności minimalizacji niż Presto. Szczególnie dobre wyniki daje podczas minimalizacji projektów implementowanych w układach IFL.

Uwaga! Układy IFL (*Integrated Fuse Logic*) firmy Signetics nie są obecnie produkowane.



Rys. 26

wyprowadzeniu 5 (obudowa DIP) ma numer 66, a ten sam węzeł w przypadku układu w obudowie PLCC ma numer 69 (**rys. 25**).

W tej części opisu mogą się znaleźć (ale nie muszą, ma to znaczenie wyłącznie porządkowe) także deklaracje pomocnicze, jak na przykład:

```
field COUNT = [WY1..0];
field WYJSCIA = [wy1,acc_xa,re_out];
```

W przypadku odwoływania się w takich deklaracjach do zmiennych indeksowanych należy przestrzegać następujących zasad:
– nie należy w jednym polu odwoływać się do zmiennych indeksowanych i nieindeksowanych (jak np. `field [wy1..0,a,b,ext]`),
– nie należy w jednym polu używać dwóch lub więcej zmiennych o takim samym indeksie (jak np. `field [x1,y1,c2]`).

W obszarze deklaracji można (choć może to także nastąpić w dowolnym innym miejscu opisu HDL) określić sposób minimalizacji funkcji logicznej generującej określoną zmienną. Do tego celu służy słowo kluczowe MIN. Format deklaracji jest następujący:

```
MIN zmienna.rozszerzenie = n;
```

gdzie:

`zmienna` – jest to nazwa funkcji poddawanej minimalizacji,

`rozszerzenie` – opcjonalne rozszerzenie nazwy, stosowane na przykład w przypadku, gdy minimalizowana zmienna jest przypisana do wejścia D przerzutnika,

`n` – liczba z zakresu 0...4, która określa sposób (algorytm) minimalizacji (zgodnie z **tab. 12**).

Szacowane przez producenta wartości (uśrednione dla różnych projektów) współczynnika minimalizacji pokazano na **rys. 26** (podano za dokumentacją firmy Logical Devices).

3. Opis HDL, który może zostać przygotowany za pomocą:
– równań logicznych (Boole'a),
– tablic prawdy,
– opisu automatu (tekstowy odpowiednik grafu przejść).

Przykładowy fragment opisu projektowanego układu pokazano poniżej:

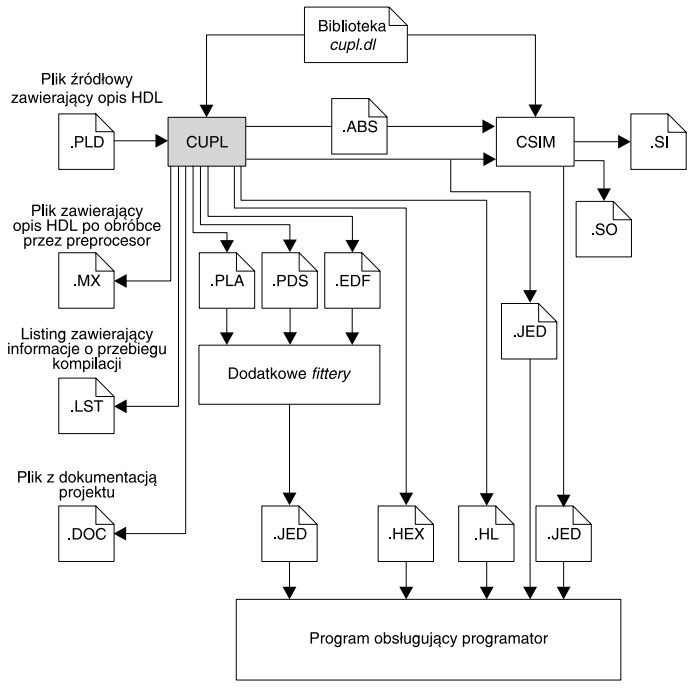
```
/***** Opis HDL *****/
OUTP = !ADR0 & !ADR1 & INP0
   # ADR0 & !ADR1 & INP1
   # !ADR0 & ADR1 & INP2
   # ADR0 & ADR1 & INP3;

CLK_OUT = !SELECT & SYNC
   # SELECT & ASYNC;

field COUNT = [WY1..0];
$define S0 'b'00
$define S1 'b'01
$define S2 'b'10
$define S3 'b'11

sequence COUNT {
present S0 next S1;
present S1 next S2;
present S2 next S3;
present S3 next S0;
}
```

Warto wiedzieć

Z wykresu pokazanego na rys. 26 wynika, że najskuteczniejszy jest algorytm minimalizacji Quine McCluskey. Warto jednak wziąć pod uwagę, że czas obliczeń rośnie wykładniczo (zgodnie ze wzorem $3^n/n$) wraz ze wzrostem liczby zmiennych wejściowych (n).

Pliki tworzone podczas kompilacji

Kompilator CUPL składa się z kilku programów (CUPLA – parser, CUPLB – fitter, CUPLC – generator plików wyjściowych, CUPLX – preprocesor, CUPLM – minimalizator), które wywoływane kolejno realizują



Rys. 27

etapy kompilacji. Ponieważ środowiska IDE, w które wbudowano CUPL-a (zajmiemy się ich przybliżeniem w kolejnych odcinkach cyklu), samodzielnie uruchamiają te programy i zarządzają obiegiem plików pomiędzy nimi, my skupimy się na przedstawieniu sposobu wymiany danych wyłącznie pomiędzy kompilatorem, symulatorem i dodatkowymi programami, jak na przykład edytory schematów, programy obsługujące programatory itp.

Na **rys. 27** pokazano obieg plików pomiędzy kompilatorem, symulatorem i opcjonalnymi programami dodatkowymi. Opis funkcji poszczególnych plików znajduje się w **tab. 13**.

Układy kombinacyjne

Po sporej dawce rozważań teoretycznych przechodzimy do pierwszych przykładów. W tej części kursu przedstawimy sposoby projektowania układów kombinacyjnych.

Układy kombinacyjne są to takie układy cyfrowe, których stany wyjściowe w danej chwili zależą jedynie od aktualnego stanów wejść (są one pozbawione pamięci historii).

Podstawowymi, powszechnie stosowanymi, elementami kombinacyjnymi są bramki logiczne i to od przedstawienia ich opisu zaczniemy opisywanie sprzętu w CUPL-u.

Na **list. 2** znajduje się przykładowy opis bramek logicznych, który wykonano za pomocą równań boole'owskich (z wykorzystaniem operatorów logicznych, które przedstawiono w EP5/2004). Taki sam efekt (czyli implementację w układzie PLD bramek logicznych) można uzyskać w nieco inny sposób, a mianowicie z wykorzy-

Tab. 13. Rozszerzenia nazw plików tworzonych przez kompilator CUPL (nazwy plików są takie, jak zadeklarowano w polu name pliku **.pld*)

Rozszerzenie	Tworzony przez	Funkcja pliku
PLD	Projektanta	Zawiera opis HDL projektowanego układu PLD.
Pliki dokumentacyjne		
DOC	Kompilator	Plik dokumentujący sposób zaimplementowania projektu w układzie docelowym, łącznie z mapą przepaleń i rozmieszczeniem sygnałów dołączonych do wyprowadzeń układu PLD.
ABS	Kompilator	Plik binarny zawierający informacje niezbędne dla poprawnej pracy symulatora.
MX	Kompilator	Plik zawierający „rozwinięty" opis projektu, czyli zawierający jawne opisy makrofunkcji, opisy generowane przez preprocesor, a także opisy dołączane do pliku źródłowego (pobierane z zewnętrznych plików bibliotecznych).
LST	Kompilator	Plik zawierający listing programu z ponumerowanymi liniami. Błędy wykryte podczas kompilacji są umieszczana na końcu pliku. Zawierają one odwołania do linii, w której wykryto błąd.
Pliki przejściowe		
PLA	Kompilator	Plik zawiera informacje umożliwiające implementację projektów w układach PLA.
PDS	Kompilator	Plik zawierający opis projektu w języku PALASM.
EDF	Kompilator	Plik w formacie EDIF (presyntezowany), który można wykorzystać do implementacji projektu w dowolnym układzie PLD.
Pliki zawierające informacje niezbędne do programowania układów		
JED	Kompilator	Plik zawierający informacje umożliwiające zaprogramowanie układu. Stosowany dla większości układów PLD. Format pliku został ustandaryzowany przez komitet JEDEC (dokument JESD-3) i zaaprobowany przez stowarzyszenie EIA.
HEX	Kompilator	Plik zawierający informacje umożliwiające zaprogramowanie układu. Stosowany do programowania pamięci.
HL	Kompilator	Plik zawierający informacje umożliwiające zaprogramowanie układów IFL firmy Signetics.
Pliki symulacyjne		
SI	Projektanta	Plik wejściowy dla symulatora funkcjonalnego. Zawiera wektory wejściowe (pobudzenia) i - opcjonalnie - wyjściowe (odpowiedzi).
SO	Kompilator	Plik zawierający wyniki symulacji prowadzonej przez CUPL-a. Zawiera także informacje o błędach wykrytych podczas symulacji.

List. 2. Opis bramek logicznych za pomocą równań logicznych

```
Name            bramki;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/*stany na wejsciach a (b_0 na plytce) i b (b_1)*/
/*ustala sie za pomoca nastawnika SW1, pozycje: 0...3*/

/***** Wejscia *****/
Pin 11 =  a;
Pin 10 =  b;

/***** Wyjscia *****/
Pin 17 = inva; /* D1 */
Pin 18 = and; /* D2 */
Pin 19 = nand; /* D3 */
Pin 20 = or; /* D4 */
Pin 21 = nor; /* D5 */
Pin 23 = xor; /* D6 */
Pin 24 = xnor; /* D7 */

/***** Opis HDL *****/
inva = !a;       /* inwerter sygnalu z wejscia A*/
and  = a & b;    /* bramka AND */
nand = !(a & b); /* bramka NAND */
or   = a # b;    /* bramka OR */
nor  = !(a # b); /* bramka NOR */
xor  = a $ b;    /* bramka ExOR */
xnor = !(a $ b); /* bramka ExNOR */
```

staniem tablic prawdy (**list. 3**).

Ten drugi sposób jest nieco bardziej rozwlekły, ale miał za zadanie zilustrować możliwość uzyskania takiego samego efektu za pomocą różnych sposobów opisu. Przedstawiony sposób tablicowego opisu bramek nie jest jedynym możliwym. Przykładowo, zamiast korzystać z zadeklarowanego w pliku źródłowym (list. 3) pola `wejscia` tablicę można zbudować korzystając z jawnie podanych wejść a i b, przykładowo:

```
/* bramka AND */
table [a,b] => and {
   'b'00  => 0;
   'b'01  => 0;
   'b'10  => 0;
   'b'11  => 1;
}
```

Także wartości bitów wejściowych, można zapisać inaczej niż to pokazano w przedstawionych przykładach. Przykładowo, stany wejściowe można podawać jawnie (w tym przypadku zapisy <!'b'10 i [1,0] są równoważne). Taki zapis pokazano poniżej:

```
/* bramka AND */
table [a,b] => and {
   [0,0]  => 0;
   [0,1]  => 0;
   [1,0]  => 0;
   [1,1]  => 1;
}
```

Jeżeli z jakichś przyczyn wygodniejsze jest posługiwanie się sposobem zapisu liczb innym niż binarny, tę samą tablicę można zapisać na przykład w taki sposób:

```
/* bramka AND */
table [a,b] => and {
   'd'0 => 'b'0;     /* zapis dziesietny/
                        binarny */
   'o'1   => 'b'0;   /* zapis osemkowy/
                        binarny */
   'h'2 => 'o'0;     /* zapis szesnastkowy/
                        osemkowy */
   'b'11  => 'h'1;   /* zapis binarny/
                        szesnastkowy */
}
```

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

List. 3. Opis bramek logicznych za pomocą tablic prawdy

```
Name            bram_tab;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/* stany na wejsciach a (b_0 na plytce)
      i b (b_1) */
/* ustala sie za pomoca nastawnika SW1, pozy-
      cje: 0...3 */

/***** Wejscia *****/
Pin 11 =  a;
Pin 10 =  b;

/***** Wyjscia *****/
Pin 17 = inva; /* D1 */
Pin 18 = and; /* D2 */
Pin 19 = nand; /* D3 */
Pin 20 = or; /* D4 */
Pin 21 = nor; /* D5 */
Pin 23 = xor; /* D6 */
Pin 24 = xnor; /* D7 */

/***** Deklaracje pomocnicze *****/
field wejscia = [a,b];

/***** Opis HDL *****/
table a => inva {    /* inwerter sygnalu
                        z wejscia A*/
   0   => 1;
   1   => 0;
}

table wejscia => and {  /* bramka AND */
    'b'00  => 0;
    'b'01  => 0;
    'b'10  => 0;
    'b'11  => 1;
}

table wejscia => nand   { /* bramka NAND */
    'b'00  => 1;
    'b'01  => 1;
    'b'10  => 1;
    'b'11  => 0;
}

table wejscia => or {   /* bramka OR */
    'b'00  => 0;
    'b'01  => 1;
    'b'10  => 1;
    'b'11  => 1;
}

table wejscia => nor {  /* bramka NOR */
    'b'00  => 1;
    'b'01  => 0;
    'b'10  => 0;
    'b'11  => 0;
}

table wejscia => xor {  /* bramka ExOR */
    'b'00  => 0;
    'b'01  => 1;
    'b'10  => 1;
    'b'11  => 0;
}

table wejscia => xnor { /* bramka ExNOR */
    'b'00  => 1;
    'b'01  => 0;
    'b'10  => 0;
    'b'11  => 1;
}
```

Układy programowalne, część 6

Kontynuujemy prezentację przykładowych opisów w języku CUPL, w tej części skupiając się na układach kombinacyjnych. Kody źródłowe prezentowanych projektów wraz z plikami symulacyjnymi publikujemy na CD-EP8/2004B. Gorąco zachęcamy do samodzielnych prób, do których można wykorzystać zestaw ewaluacyjny AVT-559.

Jak już wcześniej wspomniano, za pomocą języka CUPL można opisywać projektowane sprzętowe bloki funkcjonalne na wiele sposobów. Najbardziej oczywistym i przy tym najmniej wygodnym są równania boole'owskie, odpowiadające w nomenklaturze mikroprocesorowej pisaniu programów w asemblerze. Pokażemy teraz kilka przykładów rozwiązania prostych, aczkolwiek często napotykanych w praktyce, problemów za pomocą różnych sposobów opisu.

Dekoder adresowy

Zaprojektujemy dekoder adresowy z trzema wyjściami (RAM_SEL, IO_SEL, ROM_SEL), które uaktywniają (poziom aktywny tych sygnałów to „1") bloki peryferyjne systemu cyfrowego w zależności do stanu wejść *adresowych Adr4...Adr0* (32 różne adresy). Mapę przykładowego obszaru adresowego pokazano na **rys. 28** (zaciemnione pola wskazują peryferia aktywne pod danym adresem).

RAM_SEL	ROM_SEL	IO_SEL
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10
11	11	11
12	12	12
13	13	13
14	14	14
15	15	15
16	16	16
17	17	17
18	18	18
19	19	19
20	20	20
21	21	21
22	22	22
23	23	23
24	24	24
25	25	25
26	26	26
27	27	27
28	28	28
29	29	29
30	30	30
31	31	31

Rys. 28

Zminimalizowane równania logiczne zapewniające realizację przez układ PLD funkcji zgodnie z podaną specyfikacją dla wyjść IO_SEL i RAM_SEL przedstawiono na **list. 4**. Skonstruowanie tych równań, jakkolwiek możliwe, jest jednak dość kłopotliwe i znacznie utrudnia wprowadzenie do projektu ewentualnych zmian jak np. przesunięcia lokalizacji peryferiów w przestrzeni adresowej. Znacznie lepszym i wygodniejszym wyjściem jest zapisanie projektu w sposób pokazany na **list. 5**. W opisie tym zastosowano operator przypisania (:), za pomocą którego wcześniej zadeklarowanym wektorom są przypi-

Kierunek linii I/O

Projektant przygotowując opis HDL za pomocą CUPL-a nie musi (nie ma jak) zadeklarować kierunków sygnałów przypisanych do wyprowadzeń (wejścia/wyjścia/wejścia-wyjścia). Kompilator ustala kierunki samoczynnie na podstawie opisu i w odniesieniu do fizycznych możliwości docelowego układu PLD.

Inne możliwości stosowania operatora przypisania

Operator przypisania można wykorzystać do skrócenia zapisu równań logicznych dla operatorów działań: &, # i $.

Przykładowo zapisy:

```
[A3,A2,A1,A0]:&
[B3,B2,B1,B0]:#
[A,B,C,D]:$
```

odpowiadają równaniom:

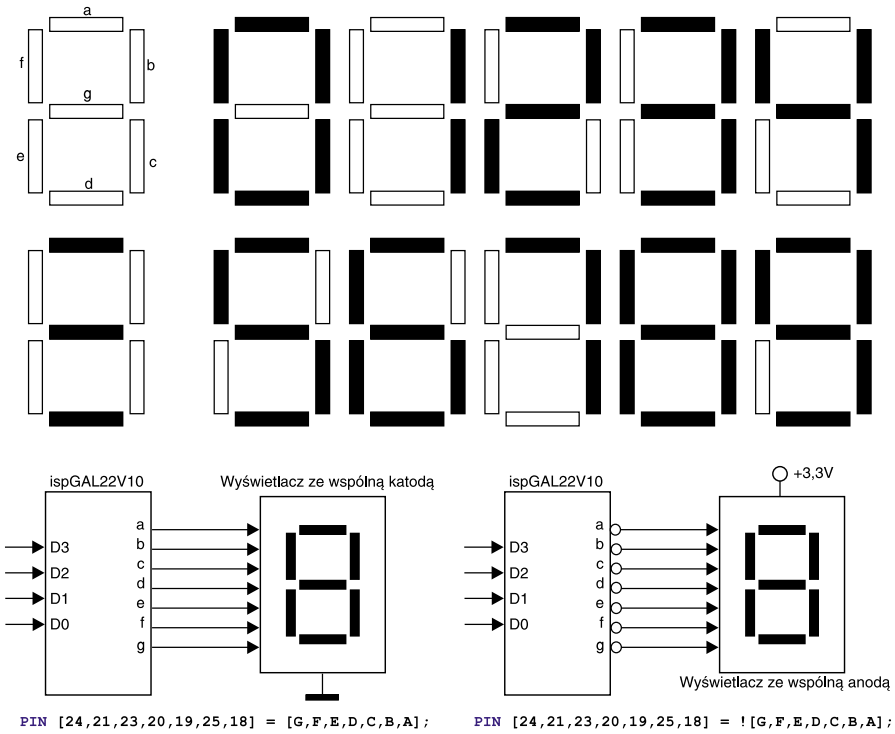
```
A3 & A2 & A1 & A0
B3 # B2 # B1 # B0
A $ B $ C $ D
```

List. 4. Równania boole'owskie funkcji logicznych dla wyjść IO_SEL i RAM_SEL (funkcje zgodnie z rys. 28)

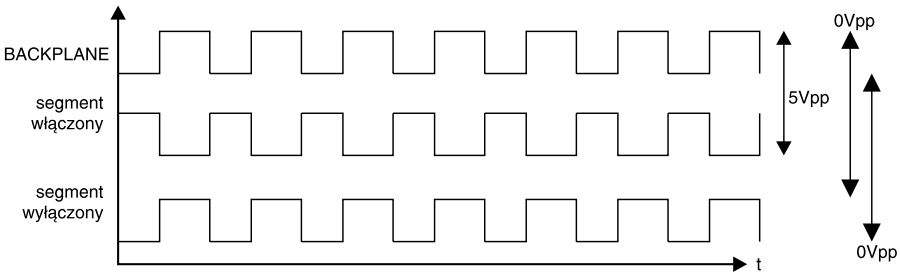
```
IO_SEL = Adr1 & Adr2 & !Adr3 & !Adr4
 # !Adr1 & !Adr2 & Adr3 & !Adr4
 # !Adr0 & Adr1 & !Adr2 & Adr3 & !Adr4
 # !Adr0 & !Adr1 & Adr2 & !Adr3 & Adr4
 # Adr0 & Adr1 & Adr2 & Adr3 & Adr4

RAM_SEL = Adr0 & Adr1 & Adr3 & !Adr4
 # !Adr1 & Adr2 & Adr3 & !Adr4
 # !Adr0 & Adr1 & Adr2 & Adr3 & !Adr4
 # !Adr2 & Adr4
 # Adr0 & Adr2 & !Adr3 & Adr4
 # !Adr0 & Adr1 & Adr2 & !Adr3 & Adr4
```

List. 5. Listing projektu dekodera adresów z pięcioma wejściami i trzema wyjściami (funkcje zgodnie z rys. 28)

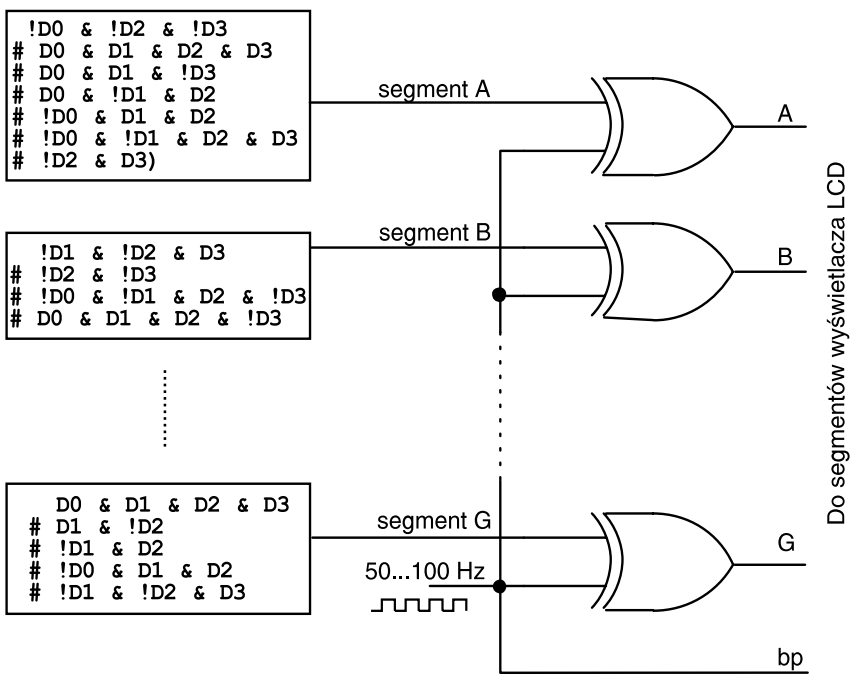
```
Name            dekoder;
Partno          brak;
Revision        brak;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/* Adres ustala sie za pomoca nastawnika
/* SW1(Adr0...Adr3)
/*     az jumpera JP1 (Adr4)            */

/***** Wejscia *****/
PIN [7,9..11] = [Adr3..0];
PIN 4 = Adr4; /* Jumper JP1 */

/***** Wyjscia *****/
PIN [26,23,17] = [RAM_SEL,IO_SEL,ROM_SEL];

/***** Deklaracje pomocnicze *****/
field ADRES = [Adr4..0];
serport_tx = ADRES:['d'6..'d'10];
serport_rx = ADRES:'d'20 # ADRES:'d'31;
ram_rd = ADRES:['d'11..'d'19];
ram_wr = ADRES:['d'21..'d'27];

/***** Opis HDL *****/
RAM_SEL = ram_rd # ram_wr;
IO_SEL = serport_tx # serport_rx;
ROM_SEL = ADRES:['d'0..'d'5] # ADRES:
['d'28..'d'30];
```



Rys. 29



Rys. 30



Rys. 31

List. 6. Projekt dekodera wyświetlacza 7-segmentowego opisanego równaniami logicznymi

```
Name            dek_wys;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/*Stany na wejsciach D3...D0 ustala sie za */
/*    pomoca nastawnika SW1                 */
/*****************************/
/*          a                */
/*        -----              */
/*       |     |             */
/*     f|     |b             */
/*      |  g  |              */
/*       -----               */
/*      |     |              */
/*     e|     |c             */
/*      |     |              */
/*       -----               */
/*          d                */
/*                           */
/*****************************/

/***** Wejscia *****/
PIN [7,9..11] = [D3..0];

/***** Wyjscia *****/
PIN [24,21,23,20,19,25,18] = [G,F,E,D,C,B,A];

/***** Deklaracje pomocnicze *****/
Field dana = [D3..0];
Field segment = [A,B,C,D,E,F,G];

/***** Opis HDL *****/
A = !D0 & !D2 & !D3
  # D0 & D1 & D2 & D3
  # D0 & D1 & !D3
  # D0 & !D1 & D2
  # !D0 & D1 & D2
  # !D0 & !D1 & D2 & D3
  # !D2 & D3;

B =  !D1 & !D2 & D3
  # !D2 & !D3
  # !D0 & !D1 & D2 & !D3
  # D0 & D1 & D2 & !D3;

C = !D1 & !D2 & D3
  # !D0 & D1 & D2 & !D3
  # D0 & D1 & !D3
  # !D1 & !D3;

D = !D0 & !D2 & !D3
  # D0 & D1 & D2 & D3
  # D0 & D1 & !D2
  # D0 & !D1 & D2
  # !D0 & D1 & D2
  # !D1 & !D2 & D3
  # !D0 & D1 & !D2 & D3
  # !D0 & !D1 & D2 & D3;

E = !D0 & !D2 & !D3
  # !D0 & !D1 & !D2 & D3
  # !D0 & D1 & D2 & !D3;

F = !D0 & !D1 & !D3
  # !D1 & !D2 & D3
  # D0 & !D1 & D2 & !D3
  # !D0 & D1 & D2 & !D3;

G = D0 & D1 & D2 & D3
  # D1 & !D2
  # !D1 & D2
  # !D0 & D1 & D2
  # !D1 & !D2 & D3;
```

sywane oczekiwane wartości lub ich przedziały, jak np.: `serport_tx = ADRES:['d'6..'d'10]`. Przypisanie może mieć także postać równania logicznego, jak np.: `serport_rx = ADRES:'d'20 # ADRES:'d'31`. Tak zapisane równania kompilator sam rozwinie do postaci „czystych" równań logicznych, znacznie ułatwiając projektantowi diagnostykę projektu i jego ewentualną modyfikację.

Dekoder-sterownik wyświetlacza 7-segmentowego

W kolejnym przykładzie przedstawimy trzy możliwe sposoby opisu dekodera 7-segmentowego współ-

List. 7. Projekt dekodera wyświetlacza 7-segmentowego opisanego równaniami logicznymi o postaci łatwej do weryfikacji przez projektanta

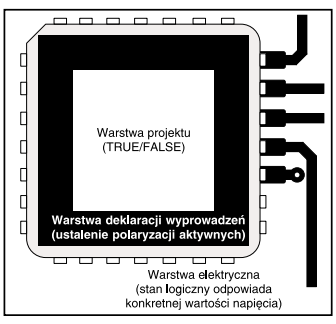
```
Name            dek_wys;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/* Stany na wejsciach D3...D0 ustala sie za  */
/*   pomoca nastawnika SW1                   */
/*************************/
/*          a            */
/*       -----           */
/*      |     |          */
/*     f|     |b         */
/*      |  g  |          */
/*       -----           */
/*      |     |          */
/*     e|     |c         */
/*      |     |          */
/*       -----           */
/*         d             */
/*                       */
/*************************/

/***** Wejscia *****/
PIN [7,9..11] = [D3..0];

/***** Wyjscia *****/
PIN [24,21,23,20,19,25,18] = [G,F,E,D,C,B,A];

/***** Deklaracje pomocnicze *****/
Field dana = [D3..0];
Field segment = [A,B,C,D,E,F,G];

$define ON  'b'1        /* Segment swieci     */
$define OFF 'b'0        /* Segment nie swieci */

/***** Opis HDL *****/
/*              a    b    c    d    e    f    g  */
segment   =
/* 0 */       [ ON,  ON,  ON,  ON,  ON,  ON, OFF] & dana:0
/* 1 */    #  [OFF,  ON,  ON, OFF, OFF, OFF, OFF] & dana:1
/* 2 */    #  [ ON,  ON, OFF,  ON,  ON, OFF,  ON] & dana:2
/* 3 */    #  [ ON,  ON,  ON,  ON, OFF, OFF,  ON] & dana:3
/* 4 */    #  [OFF,  ON,  ON, OFF, OFF,  ON,  ON] & dana:4
/* 5 */    #  [ ON, OFF,  ON,  ON, OFF,  ON,  ON] & dana:5
/* 6 */    #  [ ON, OFF,  ON,  ON,  ON,  ON,  ON] & dana:6
/* 7 */    #  [ ON,  ON,  ON, OFF, OFF, OFF, OFF] & dana:7
/* 8 */    #  [ ON,  ON,  ON,  ON,  ON,  ON,  ON] & dana:8
/* 9 */    #  [ ON,  ON,  ON,  ON, OFF,  ON,  ON] & dana:9
/* A */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:A
/* B */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:B
/* C */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:C
/* D */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:D
/* E */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:E
/* F */    #  [ ON, OFF, OFF,  ON, OFF, OFF,  ON] & dana:F;
}
```

List. 8. Projekt dekodera wyświetlacza 7-segmentowego opisanego równaniami za pomocą tablicy prawdy

```
Name            dek_wys;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/* Stany na wejsciach D3...D0 ustala sie za  */
/*   pomoca nastawnika SW1                   */
/*************************/
/*          a            */
/*       -----           */
/*      |     |          */
/*     f|     |b         */
/*      |  g  |          */
/*       -----           */
/*      |     |          */
/*     e|     |c         */
/*      |     |          */
/*       -----           */
/*         d             */
/*                       */
/*************************/

/***** Wejscia *****/
PIN [7,9..11] = [D3..0];

/***** Wyjscia *****/
PIN [24,21,23,20,19,25,18] = [G,F,E,D,C,B,A];

/***** Deklaracje pomocnicze *****/
Field Dana = [D3..0];
Field Segment = [A,B,C,D,E,F,G];

/***** Opis HDL *****/
Table Dana => Segment {

/*  Wejscia      Wyjscia segmentowe             */
/*  ------------------------------------       */
/*  AAAA                                        */
/*  3210          ABCDEFG                       */
'b'0000   =>  'b'1111110; /* 0                 */
'b'0001   =>  'b'0110000; /* 1                 */
'b'0010   =>  'b'1101101; /* 2                 */
'b'0011   =>  'b'1111001; /* 3                 */
'b'0100   =>  'b'0110011; /* 4                 */
'b'0101   =>  'b'1011011; /* 5                 */
'b'0110   =>  'b'1011111; /* 6                 */
'b'0111   =>  'b'1110000; /* 7                 */
'b'1000   =>  'b'1111111; /* 8                 */
'b'1001   =>  'b'1111011; /* 9                 */
'b'1010   =>  'b'1001001; /* blad              */
'b'1011   =>  'b'1001001; /* blad              */
'b'1100   =>  'b'1001001; /* blad              */
'b'1101   =>  'b'1001001; /* blad              */
'b'1110   =>  'b'1001001; /* blad              */
'b'1111   =>  'b'1001001; /* blad              */
}
```

Niuanse negacji

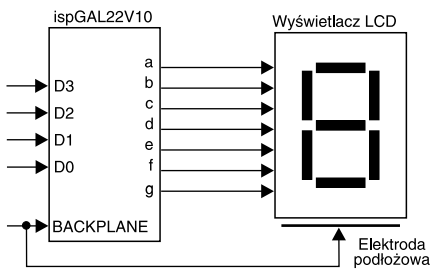
Twórcy CUPL-a przyjęli, że podczas przygotowywania opisu sprzętu projektant rozważa wyłącznie wartości TRUE/FALSE, natomiast o polaryzacji (aktywne „0”/aktywne „1”) sygnału decyduje podczas deklarowania wyprowadzeń.



pracującego z wyświetlaczem LED o wspólnej katodzie. Na **rys. 29** pokazano sposób wyświetlania znaków z zakresu 0...9 oraz znaków o kodach powyżej 9 (palą się wyłącznie poziome segmenty wyświetlacza).

Czytelnicy o największym zacięciu do posługiwania się mapami Karnaugh mogą spróbować zweryfikować poprawność równań logicznych, za pomocą których opisano dekoder w przypadku pokazanym na **list. 6**. Te same równania można zapisać w wygodniejszej postaci (**list. 7**), a ich przekształceniem do postaci pokazanej na list. 6 zajmie się kompilator. Jak można zauważyć, za pomocą deklaracji `define` stałych ON i OFF przypisano wartości bitów (odpowiednio) „1” i „0”, co pozwala posługiwać się w dalszej części opisu czytelnymi nazwami. Taki sposób opisania działania dekodera ułatwia diagnostykę projektu oraz wprowadzania do niego zmian.

Kolejnym możliwym sposobem opisania dekodera jest zawarcie zależności pomiędzy stanami na jego wejściach i wyjściach w tablicy prawdy (**list. 8**). Poszczególnym wartościom wektora wejściowego `Dana` przypisywane są odpowiednie wartości wyjściowe (wektor `Segment`), a całość jest ulokowana w tablicy zaczynającej się od słowa kluczowego `Table`. Obydwa wektory zdefiniowano w polu deklaracji pomocniczych za pomocą słów kluczowych `Field`.



Rys. 32

List. 9. Jeden z możliwych sposobów dodania do funkcji sterującej segmentem A wyświetlacza inwertera sterowanego sygnałem BACKPLANE

```
A = (!D0 & !D2 & !D3
  # D0 & D1 & D2 & D3
  # D0 & D1 & !D3
  # D0 & !D1 & D2
  # !D0 & D1 & D2
  # !D0 & !D1 & D2 & D3
  # !D2 & D3)
  $ BACKPLANE;
```

W przypadku, gdy zaprojektowany sterownik będzie współpracował z wyświetlaczem LED o wspólnej anodzie wystarczy zmienić aktywny stan (z wysokiego na niski) na wyjściach dekodera. Najprostszym sposobem jest zastąpienie linii `PIN [24,21,23,20,19,25,18] = [G,F,E,D,C,B,A];` linią `PIN [24,21,23,20,19,25,18] = ![G,F,E,D,C,B,A];` (w której linie wyjściowe portów zostały zanegowane).

Prezentowany dekoder można łatwo dostosować do sterowania 7-segmentowego wyświetlacza LCD. W tym celu wszystkie wyjścia zasilające segmenty wyświetlacza powinny zostać wyposażone w sterowane inwertery (wykonane np. na bramkach ExOR), które dostarczą do segmentów „świecących" sygnał w przeciwfazie w stosunku do sygnału zasilającego podłoże (*backplane*) wyświetlacza, jak to pokazano na **rys**. **30**. Sterowane inwertery najprościej można uzyskać w CUPL-u XOR-ując funkcje tworzące sygnały sterujące segmentami z sygnałem BACKPLANE, np. w taki sposób jak to pokazano na **list**. **9** (pokazano przykład tylko dla jednego wyjścia).

List. 10. Projekt multipleksera 4-wejściowego

```
Name            mux;
Partno          brak;
Revision        brak;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/* Nastawnik SW1 sluzy do zmiany stanow  */
/*      na wejsciach X3...X0             */
/* Jumpery Sw1 i Sw2 spelniaja role      */
/*      elementow adresujacych           */
/*   aktywne wejscie multipleksera 4x1   */

/***** Wejscia *****/
PIN [7,9..11] = [X0,X1,X2,X3];
PIN [4,6] = [SEL1..0];

/***** Wyjscia *****/
PIN [25] = Y;

/***** Deklaracje pomocnicze *****/
Field SELEKTOR = [SEL1..0];

/***** Opis HDL *****/
Y = (X0 & SELEKTOR:0)
 # (X1 & SELEKTOR:1)
 # (X2 & SELEKTOR:2)
 # (X3 & SELEKTOR:3);
```

Schemat blokowy ilustrujący sposób tworzenia sygnałów sterujących segmentami wyświetlacza LCD pokazano na **rys**. **31**. Na **rys**. **32** przedstawiono sposób dołączenia wyświetlacza do dekodera zaimplementowanego w układzie PLD.

Przedstawiony mechanizm tworzenia tablic prawdy w CUPL-u pozwala na łatwą i wygodną implementacje w układach PLD najróżniejszych tablic przekodowań (transkoderów), często określanych mianem *look-up table*.

Multiplekser

Podobnym do dekodera-sterownika wyświetlacza 7-segmentowego przykładem projektu układu kombinacyjnego jest multiplekser. W artykule pokażemy implementację pojedynczego multipleksera 4-wejściowego.

Najbardziej oczywistym sposobem opisu jest równanie przypisujące wyjściu multipleksera stan występujący na zaadresowanym wejściu. Takie równanie może mieć postać jak poniżej:

```
Y = !SEL0 & !SEL1 & X0
 # SEL0 & !SEL1 & X1
 # !SEL0 & SEL1 & X2
 # SEL0 & SEL1 & X3
```

Podobnie, jak miało to miejsce we wcześniejszych przykładach, taki sposób opisu, jakkolwiek skuteczny, nie jest wygodny. Zdecydowanie lepiej sprawdza się w praktyce (ze względu na wygodę, formalnie obydwa zapisy są praktycznie równoważne) opis pokazany na **list**. **10**.

Jak widać, liczba oferowanych przez CUPL-a sposobów opisu układów kombinacyjnych nie jest duża, ale w zupełności wystarczy do realizacji każdego zadania inżynierskiego, dając przy tym możliwości wybrania przez projektanta sposobu najbardziej mu odpowiadającego.

Za miesiąc przedstawimy kilka przykładów układów synchronicznych, w kolejnych zajmiemy się prezentacją narzędzi.

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

Układy programowalne, część 7

Siódmą część cyklu poświęcamy przedstawieniu przykładów opisu klasycznych układów synchronicznych, w tym przede wszystkim liczników. Zaczniemy od opisu HDL najważniejszych elementów takich układów: przerzutników.

Automaty – krok 1

Klasyczne układy synchroniczne, często zwane automatami, budowane są na bazie przerzutników, które spełniają w nich rolę pamięci stanu. W zasadzie, poza przerzutnikami RS, wszystkie inne rodzaje przerzutników (T, D, JK, JK-MS) nadają się do spełniania roli pamięci stanu w automatach. W komórkach OLMC układów ispGAL22V10 (taki zastosowano w zestawie AVT-599) znajdują się przerzutniki typu D z wejściami synchronicznego ustawiania i asynchronicznego zerowania.

Na **list. 11...14** znajdują się opisy HDL czterech przerzutników, których praca jest synchronizowana (oprócz przerzutnika RS, który nie wymaga sygnału zegarowego) przez sygnał zegarowy. Użytkownik korzystający z zestawu AVT-599 może wybrać jego źródło (za pomocą jumpera JP4).

Ponieważ – jak już wcześniej wspomniano – w komórkach OLMC układów GAL22V10 znajdują się przerzutniki typu D, opis przedstawiony na list. 11 przygotowano nieco „na wyrost". Zastosowanie w równaniach synchronicznych wejść zerowania (RESET) i ustawiania (PRESET) powoduje jednak, że możliwości funkcjonalne implementowanego przerzutnika są takie same, niezależnie od jego budowy w docelowym układzie PLD. Zwiększamy w ten sposób uniwersalność opisu, który „pasuje" do każdego układu PLD z wbudowanymi jakimikolwiek przerzutnikami D. Z tego samego powodu w opisie nie są wykorzystane dostępne w GAL22V10 sygnały asynchronicznego zerowania Q.AR, NIE_Q.AR oraz synchronicznego ustawiania Q.SP i NIE_Q.SP.

Automaty – krok 2

W zależności od sposobu tworzenia sygnałów wyjściowych, automaty dzielą się na automaty Moore'a (schemat blokowy pokazano na **rys. 33**) i Mealy'ego (schemat blokowy pokazano na **rys. 34**). Różnica pomiędzy nimi polega na tym, że sygnały wyjściowe w automacie Mealy'ego są tworzone w postaci funkcji kombinacyjnej dwóch zmiennych: bitów wyjściowych przerzutników pamiętających aktualny stan automatu (informacja pobierana z wyjść prze-

List. 11. Opis w języku CUPL przerzutnika typu D z synchronicznymi wejściami zerowania i ustawiania

```
Name            d_ff;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;       /* Wejscie zegarowe */
PIN 6 = DATA;      /* Stan na wejsciu ustala JP2 */
PIN 4 = PRESET;    /* Wejscie synchronicznego ustawiania - JP1*/
PIN 11 = RESET;    /* Wejscie synchronicznego zerowania - B_0 SW1 */

/***** Wyjscia *****/
PIN [17, 18] = [Q, NIE_Q]; /* Wyjscia przerzutnika */

/***** Opis HDL *****/
Q.D = PRESET # (DATA & !RESET);
NIE_Q.D = RESET # (!DATA & !PRESET);
```

List. 12. Opis w języku CUPL przerzutnika typu T z synchronicznymi wejściami zerowania i ustawiania

```
Name            t_ff;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;        /* Wejscie zegarowe */
PIN 6 = T;          /* Stan na wejsciu T ustala JP2 */
PIN 4 = PRESET;     /* Wejscie synchronicznego ustawiania - JP1*/
PIN 11 = RESET;     /* Wejscie synchronicznego zerowania - B_0 SW1 */

/***** Wyjscia *****/
PIN [17, 18] = [Q, NIE_Q]; /* Wyjscia przerzutnika */

/***** Opis HDL *****/
Q.D = PRESET # (!RESET & !T & Q) # (!RESET & T & NIE_Q);
NIE_Q.D = RESET # (!PRESET & !T & NIE_Q) # (!PRESET & T & Q);
```

List. 13. Opis w języku CUPL przerzutnika typu JK z synchronicznymi wejściami zerowania i ustawiania

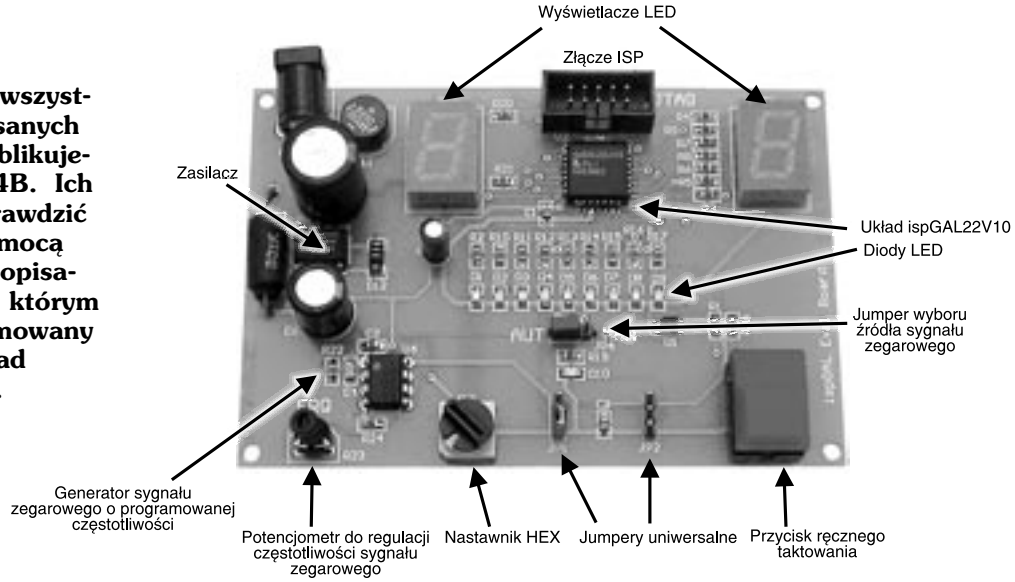
```
Name            jk_ff;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;        /* Wejscie zegarowe */
PIN 4 = PRESET;     /* Wejscie synchronicznego ustawiania - JP1*/
PIN 6 = RESET;      /* Wejscie synchronicznego zerowania - JP2 */
PIN 7 = J;          /* Wejscie synchronicznego zerowania - B_0 SW1 */
PIN 9 = K;          /* Wejscie synchronicznego zerowania - B_1 SW1 */

/***** Wyjscia *****/
PIN [17, 18] = [Q, NIE_Q]; /* Wyjscia przerzutnika */

/***** Opis HDL *****/
Q.D = PRESET # (J & NIE_Q & !RESET) # (!K & Q & !RESET);
NIE_Q.D = RESET # (!J & NIE_Q & !PRESET) # (K & Q & !PRESET);
```

List. 14. Opis w języku CUPL przerzutnika typu RS (na bramkach NAND)

```
Name            rs_ff;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 4 = S;     /* Wejscie ustawiania - JP1*/
PIN 6 = R;     /* Wejscie zerowania - JP2 */

/***** Wyjscia *****/
PIN [17, 18] = [Q, NIE_Q]; /* Wyjscia przerzutnika */

/***** Opis HDL *****/
Q = !S # (R & Q);
NIE_Q = !R # (S & NIE_Q);
```

Można także w praktyce

Programy źródłowe wszystkich projektów opisanych w ramach kursu publikujemy na CD-EP9/2004B. Ich działanie można sprawdzić w praktyce za pomocą zestawu AVT-599 (opisany w EP3/2004), w którym zastosowano programowany w systemie układ ispGAL22V10.



rzutników spełniających rolę pamięci stanu) i aktualnego stanu wejść automatu. Użytkownik odpowiednio przygotowując opis automatu może wymusić implementację projektu w jednym lub drugim rodzaju automatu. W większości przypadków walory automatów Moore'a są na tyle duże, że implementowane w nim projekty spełniają wymagania typowych aplikacji.

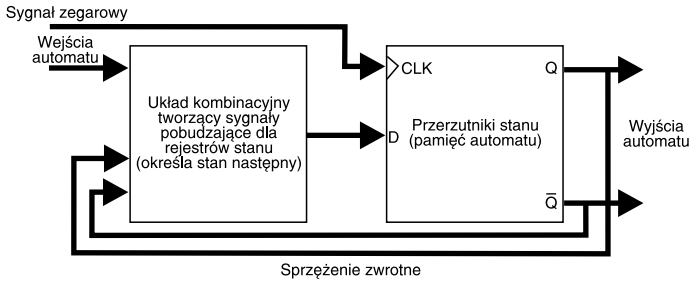
Prezentację opisu automatów zaczniemy od 4-bitowego licznika liczącego w cyklu modulo 16 w dwóch kierunkach. Licznik wyposażono w wejście sychronicznego zerowania. Jego opis HDL przedstawiono na **list. 15**. Ze względu na wygodę zastosowano w nim opis przejść warunkowych pomiędzy kolejnymi stanami automatu, co pozwala na wygodne odwzorowanie standardowego grafu przejść. Ten sam efekt funkcjonalny można uzyskać za pomocą przypisania do wejścia D każdego przerzutnika (Q0.d, Q1.d, Q2.d i Q3.d) wyjściowego funkcji logicznej, ale nie jest to zadanie łatwe do wykonania, na co dowodem może być przedstawione poniżej równanie dla przerzutnika Q3.d:

```
Q3.d =>
  Q0 & Q1 & Q2 & Q3 & !RESET & U_D
  # !Q0 & !Q1 & !Q2 & !Q3 & !RESET & U_D
  # Q0 & Q1 & Q2 & !Q3 & !RESET & !U_D
  # !Q1 & !Q2 & Q3 & !RESET & !U_D
  # Q0 & !Q2 & Q3 & !RESET & U_D
  # !Q0 & Q1 & Q3 & !RESET
  # Q0 & Q1 & !Q2 & Q3 & !RESET & !U_D
  # !Q1 & Q2 & Q3 & !RESET
```

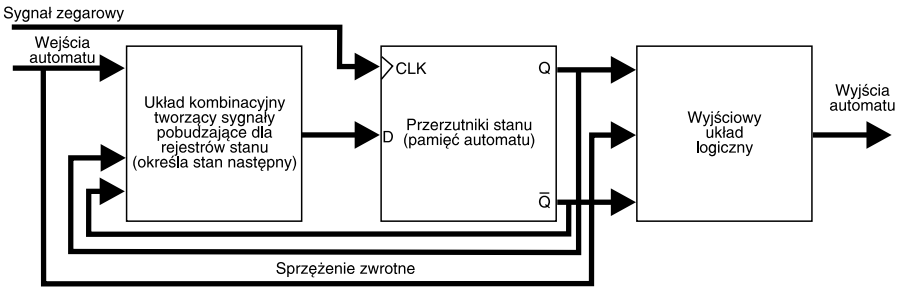
Wykonywanie takich „sztuczek" zaprzecza wygodzie korzystania z języka wysokiego poziomu, więc potraktujmy przedstawioną możliwość jedynie jako przykład możliwości CUPL-a, a nie zalecany sposób projektowania.

O tym, że opis licznika za pomocą równań logicznych nie jest najwygodniejszym sposobem realizacji projektu, nie trzeba nikogo przekonywać. Trzeba jednak przyznać, że także opis kolejnych przejść, jakkolwiek znacznie bardziej czytelny nie jest pozbawiony wady: jest po prostu bardzo długi i w przypadku konieczności wprowadzenia jakiejkolwiek zmiany (np. modyfikacji długości cyklu zliczania lub wprowadzenia dodatkowych sygnałów sterujących), poprawianie tak przygotowanego opisu nie jest wygodne. Sytuację uprości zastosowanie komendy preprocesora `$REPEAT`, jak to pokazano na **list. 16**. Wygodę stosowania zapisów „kompaktowych" zilustrowano na **list. 17**, na którym pokazano opis HDL licznika *modulo 10*, który jest odpowiednikiem funkcjonalnym liczników z list. 15 i 16.

Przedstawione opisy dotyczą pojedynczego licznika, który nie może



Rys. 33



Rys. 34

List. 15. Opis 4-bitowego licznika góra/dół z synchronicznym wejściem zerującym i jawnymi deklaracjami wartości przypisanych kolejnym stanom

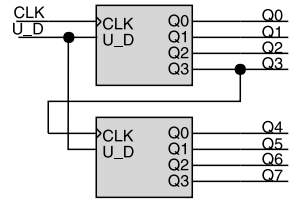
```
Name            cnt_ud_mod16;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;    /* Wejscie zegarowe */
PIN 4 = RESET;  /* Wejscie asynchronicznego */
                /* zerowania - JP1*/
PIN 6 = U_D;    /* Wybor kierunku zliczania */
                /* - JP2 */

/***** Wyjscia *****/
PIN [17..20] = [Q3..0]; /* Wyjscia licznika */

/***** Deklaracje pomocnicze *****/
field licznik = [Q3..0];
$define S0 ,b'0000
$define S1 ,b'0001
$define S2 ,b'0010
$define S3 ,b'0011
$define S4 ,b'0100
$define S5 ,b'0101
$define S6 ,b'0110
$define S7 ,b'0111
$define S8 ,b'1000
$define S9 ,b'1001
$define S10 ,b'1010
$define S11 ,b'1011
$define S12 ,b'1100
$define S13 ,b'1101
$define S14 ,b'1110
$define S15 ,b'1111

field mode = [RESET,U_D];
up = mode:0;  /* w gore */
down = mode:1; /* w dol */
clear = mode:[2..3]; /* zerowanie */

/***** Opis HDL *****/
sequence licznik {
present S0      if up           next S1;
                if down         next S15;
                if clear        next S0;
present S1      if up           next S2;
                if down         next S0;
                if clear        next S0;
present S2      if up           next S3;
                if down         next S1;
                if clear        next S0;
present S3      if up           next S4;
                if down         next S2;
                if clear        next S0;
present S4      if up           next S5;
                if down         next S3;
                if clear        next S0;
present S5      if up           next S6;
                if down         next S4;
                if clear        next S0;
present S6      if up           next S7;
                if down         next S5;
                if clear        next S0;
present S7      if up           next S8;
                if down         next S6;
                if clear        next S0;
present S8      if up           next S9;
                if down         next S7;
                if clear        next S0;
present S9      if up           next S10;
                if down         next S8;
                if clear        next S0;
present S10     if up           next S11;
                if down         next S9;
                if clear        next S0;
present S11     if up           next S12;
                if down         next S10;
                if clear        next S0;
present S12     if up           next S13;
                if down         next S11;
                if clear        next S0;
present S13     if up           next S14;
                if down         next S12;
                if clear        next S0;
present S14     if up           next S15;
                if down         next S13;
                if clear        next S0;
present S15     if up           next S0;
                if down         next S14;
                if clear        next S0;
}
```

List. 16. Opis 4-bitowego licznika góra/dół z synchronicznym wejściem zerującym i skróconym zapisem deklaracji wartości przypisanych kolejnym stanom

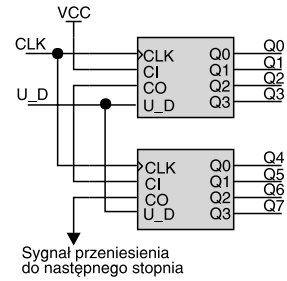
```
Name            cnt_ud_mod16_r;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;     /* Wejscie zegarowe */
PIN 4 = RESET;   /* Wejscie asynchronicznego */
                 /*zerowania - JP1*/
PIN 6 = U_D;     /* Wybor kierunku zliczania */
                 /*- JP2 */

/***** Wyjscia *****/
PIN [17..20] = [Q3..0]; /* Wyjscia licznika*/

/***** Deklaracje pomocnicze *****/
field licznik = [Q3..0];
$REPEAT i = [0..15]
$DEFINE S{i} {i}
$REPEND

field mode = [RESET,U_D];
up = mode:0;          /* w gore */
down = mode:1;        /* w dol */
clear = mode:[2..3];  /* zerowanie */

/***** Opis HDL *****/
sequence licznik {
PRESENT S0
   IF up NEXT S1;
   IF down NEXT S15;
   IF clear NEXT S0;
$REPEAT i = [1..15]
PRESENT S{i}
   IF up NEXT S{(i+1)%16};
   IF down NEXT S{(i-1)%16};
   IF clear NEXT S0;
$REPEND
}
```

List. 17. Opis 4-bitowego licznika góra/dół liczącego w cyklu modulo 10 z synchronicznym wejściem zerującym i skróconym zapisem deklaracji wartości przypisanych kolejnym stanom

```
Name            cnt_ud_mod10_r;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;     /*Wejscie zegarowe */
PIN 4 = RESET;   /*Wejscie asynchronicznego*/
                 /*zerowania - JP1*/
PIN 6 = U_D;     /*Wybor kierunku zliczania*/
                 /*- JP2 */

/***** Wyjscia *****/
PIN [17..20] = [Q3..0]; /*Wyjscia licznika*/

/***** Deklaracje pomocnicze *****/
field licznik = [Q3..0];
$REPEAT i = [0..9]
$DEFINE S{i} {i}
$REPEND

field mode = [RESET,U_D];
up = mode:0;            /* w gore */
down = mode:1;          /* w dol */
clear = mode:[2..3];    /* zerowanie */

/***** Opis HDL *****/
sequence licznik {
PRESENT S0
   IF up NEXT S1;
   IF down NEXT S9;
   IF clear NEXT S0;
$REPEAT i = [1..9]
PRESENT S{i}
   IF up NEXT S{(i+1)%10};
   IF down NEXT S{(i-1)%10};
   IF clear NEXT S0;
$REPEND
}
```



Rys. 35



Rys. 36

być łączony w synchroniczne kaskady z innymi, co pozwoliłoby na zwiększenie długości zliczanego słowa. Jest to dość poważna wada przedstawionego rozwiązania, warto więc by było wyposażyć liczniki w wejście i wyjście przeniesienia, które umożliwią ich łączenie w wielobitowe zespoły liczące. Jedyną możliwością zwiększenia długości zliczania jest połączenie ich w sposób pokazany na **rys**. **35**, ale rozwiązanie to ma wadę: naruszana jest synchroniczność licznika, co w przypadku większych częstotliwości taktowania może spowodować niepoprawną jego pracę.

Na **list**. **18** pokazano przykładowy opis 4-bitowego licznika liczącego w cyklu modulo 10, wyposażonego w wejście (CI) i wyjście (CO) przeniesienia (sposób szeregowego łączenia takich liczników pokazano na **rys**. **36**). Opis tego licznika nie różni się zbytnio od wcześniej przedstawionych, należy zwrócić jedynie uwagę na to, że w każdym stanie rozpatrywanych jest więcej warunków, z których jeden (`if others`) zapewnia zatrzymanie się licznika w bieżącym stanie z jego podtrzymaniem. Drugą rzeczą, na którą warto zwrócić uwagę, jest występujące w dwóch miejscach polecenie `out CO`, za pomocą którego „wyprowadzany" jest sygnał przeniesienia.

Automaty – krok 3

Automaty opisywane w języku CUPL można wykorzystać do generowania sygnałów synchronicznych i asynchronicznych, które będą wy-

List. 18. Opis 4-bitowego licznika góra/dół liczącego w cyklu modulo 10 z synchronicznym wejściem zerującym i możliwością łączenia ze sobą liczników w kaskady synchroniczne

```
Name            cnt_ud_mod10_rc;
Partno          U1;
Revision        01;
Date            20/05/04;
Designer        PZb;
Company         EP;
Location        brak;
Assembly        brak;
Device          g22v10lcc;

/***** Wejscia *****/
PIN 2 = CLK;      /*Wejscie zegarowe */
PIN 4 = RESET;    /*Wejscie asynchronicznego*/
                  /*zerowania - JP1*/
PIN 6 = U_D;      /* Wybor kierunku zliczania*/
                  /*- JP2 */
PIN 11 = CI;

/***** Wyjscia *****/
PIN [17..20] = [Q3..0]; /*Wyjscia licznika*/
PIN 26 = CO;

/***** Deklaracje pomocnicze *****/
field licznik = [Q3..0];
$define S0 ,b'0000
$define S1 ,b'0001
$define S2 ,b'0010
$define S3 ,b'0011
$define S4 ,b'0100
$define S5 ,b'0101
$define S6 ,b'0110
$define S7 ,b'0111
$define S8 ,b'1000
$define S9 ,b'1001

field mode = [U_D,CI,RESET];
up = mode:'b'010;        /* w gore */
down = mode:'b'110;      /* w dol */
others = mode:'b'x00;
clear = mode:'b'xx1;
```

List. 18 - cd

```
/***** Opis HDL *****/
sequence licznik {
present S0      if up           next S1;
                if down         next S9;
                if others       next S0;
                if clear        next S0;
present S1      if up           next S2;
                if down         next S0 out CO;
                if others       next S1;
                if clear        next S0;
present S2      if up           next S3;
                if down         next S1;
                if others       next S2;
                if clear        next S0;
present S3      if up           next S4;
                if down         next S2;
                if others       next S3;
                if clear        next S0;
present S4      if up           next S5;
                if down         next S3;
                if others       next S4;
                if clear        next S0;
present S5      if up           next S6;
                if down         next S4;
                if others       next S5;
                if clear        next S0;
present S6      if up           next S7;
                if down         next S5;
                if others       next S6;
                if clear        next S0;
present S7      if up           next S8;
                if down         next S6;
                if others       next S7;
                if clear        next S0;
present S8      if up           next S9 out CO;
                if down         next S7;
                if others       next S8;
                if clear        next S0;
present S9      if up           next S0;
                if down         next S8;
                if others       next S9;
                if clear        next S0;
}
```

stępować wraz z określonymi stanami automatów. Do tego celu służy polecenie `out`, które w przykładzie pokazanym na list. 18 wykorzystano do „wyprowadzenia" z licznika sygnału przeniesienia CO.

Z polecenia `out` można skorzystać na dwa sposoby, uzyskując różne wyniki:

- jeżeli chcemy uzyskać sygnał synchronizowany przebiegiem zegarowym (czyli uzyskiwany na wyjściu przerzutnika taktowanego tym samym sygnałem zegarowym, którym jest taktowany automat), to należy korzystać z następującego zapisu:

```
sequence licznik {
present S0      next S1;
present S1      next S2;
present S2      if A next S3 out CO;
                if !A next S1;
present S3      next S0;
}
```

lub, gdy generowanie sygnału jest bezwarunkowe:

```
equence licznik {
present S0      next S1;
present S1      next S2;
present S2      next S3   out CO;
present S3      next S0;
}
```

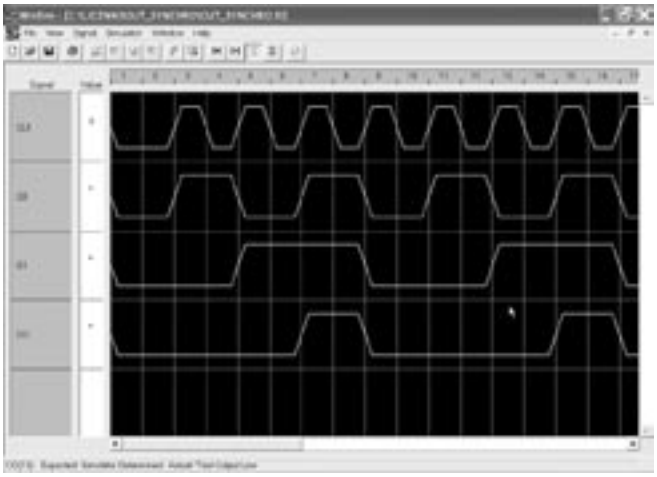
Jakkolwiek takie rozwiązanie jest technicznie eleganckie, należy pamiętać, że generowany sygnał pojawi się na wyjściu opóźniony o jeden takt zegarowy (jeżeli jest generowany w stanie S2, to na wyjściu pojawi się na czas trwania stanu S3). Widać to na **rys. 37** (źródło symulowanego automatu 4-stanowego jest dostępne na CD-EP9/2004B w katalogu *\OUT_synchro* i na stronie internetowej EP w dziale *Download*).

- w przypadku, gdy sygnał wyjściowy ma być wytwarzany w układzie kombinacyjnym (może wtedy zawierać zakłócenia szpilkowe wywołane przez opóźnienia w funktorach logicznych tworzących ten sygnał), zapis w języku CUPL jest następujący:

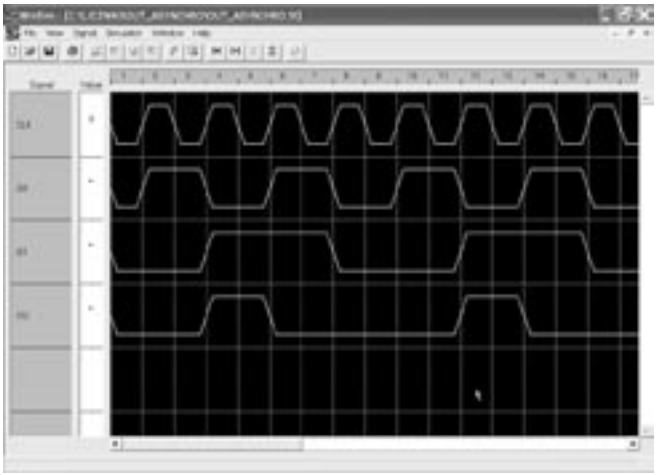
```
sequence licznik {
present S0      next S1;
present S1      next S2;
present S2      next S3;
                out CO;
present S3      next S0;
```

W odróżnieniu od synchronicznego „wyprowadzania" sygnału CO, tym razem pojawia się on dokładnie podczas stanu, do którego go przypisano (**rys. 38**). Nieco więcej zabiegów w tym przypadku wymaga opisanie układu generującego warunkowo sygnał wyjściowy, ponieważ jest on przypisany do danego stanu – jeżeli automat się w nim znajdzie, sygnał wyjściowy na pewno się pojawi. Z tego wynika konieczność wcześniejszego, niż ma to miejsce w układach synchronicznych, rozpatrywania warunków zmiany stanu.

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



Rys. 37



Rys. 38

Układy programowalne, część 8

Zbliżamy się do końca cyklu, więc po sporej dawce informacji o języku CUPL, przechodzimy do przedstawienia narzędzi, w których ten właśnie język HDL został zaimplementowany. W tej części przedstawiamy możliwości bezpłatnych programów: edytora schematów RimuSCH i kompilatora WinCUPL.

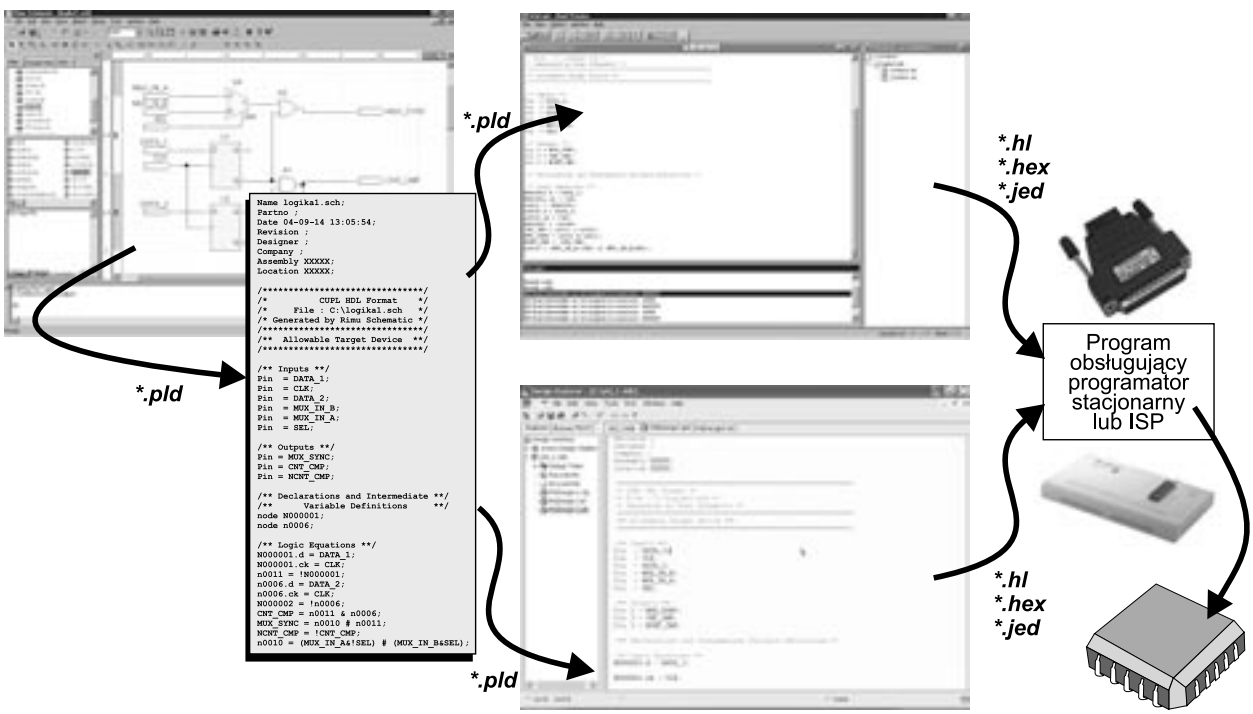
„Plotki o mojej śmierci są nieco przesadzone" powiedział niegdyś Mark Twain. Powiedzenie to doskonale pasuje do obecnej sytuacji CUPL-a: z mody nieco wyszedł, ale jest ciągle stosowany m.in. w pakietach projektowych Protel (także w najnowszej wersjach DXP i 2004). Warto zwrócić uwagę na fakt, że kompilatory wbudowane w pakiety firmy Altium (Protel 99SE/DXP/2004) obsługują blisko 6000 typów układów SPLD/CPLD oraz ponad 8500 typów układów FPGA pochodzących od 20 producentów. CUPL jest także sztandarowym językiem HDL firmy Atmel, która udostępnia na swojej stronie internetowej bezpłatną wersję WinCUPL-a. Kompilator CUPL-a zastosowano także w pakiecie ProChip Designer firmy Atmel, który służy m.in. do realizacji projektów na układy FPSLIC (rdzeń AVR i FPGA w jednej obudowie).

Do grona producentów narzędzi ułatwiających korzystanie z CUPL-a dołączyła także nowozelandzka firma Hutson (*http://www.hutson.co.nz*), w ofercie której znajduje się program Rimu Schematic. Za jego pomocą można konwertować schematy elektryczne do postaci opisu tekstowego w języku CUPL. Program ten nie ma wbudowanego kompilatora, nie może on więc pracować całkowicie samodzielnie – służy wyłącznie jako wygodny konwerter schematów do opisu HDL. Prezentację narzędzi rozpoczniemy od tego właśnie programu, którego możliwości w bezpłatnej wersji są w zupełności wystarczające do zrealizowania projektów na układach GAL i większych.

RimuSCH:
dla tych co nie lubią pisać

Pomimo tego, że CUPL należy do zdecydowanie najprostszych języków HDL, wielu jego potencjalnych użytkowników obawia się podejmowania samodzielnych prób z przygotowywaniem za jego pomocą własnych opisów projektowanych układów. Uważają (i słusznie), że łatwiej byłoby narysować schemat logiczny, który jakieś narzędzie przekonwertuje do postaci „zrozumiałej" dla programowanego układu PLD. Z pomocą przychodzi nam RimuSCH – dostępny bezpłatnie (publikujemy go na CD-EP10/2004B) edytor schematów wyposażony m.in. w interfejs eksportu listy połączeń w formacie CUPL-a. Program wymaga instalacji, która przebiega w sposób typowy dla systemu Windows (działa także z Windows XP).

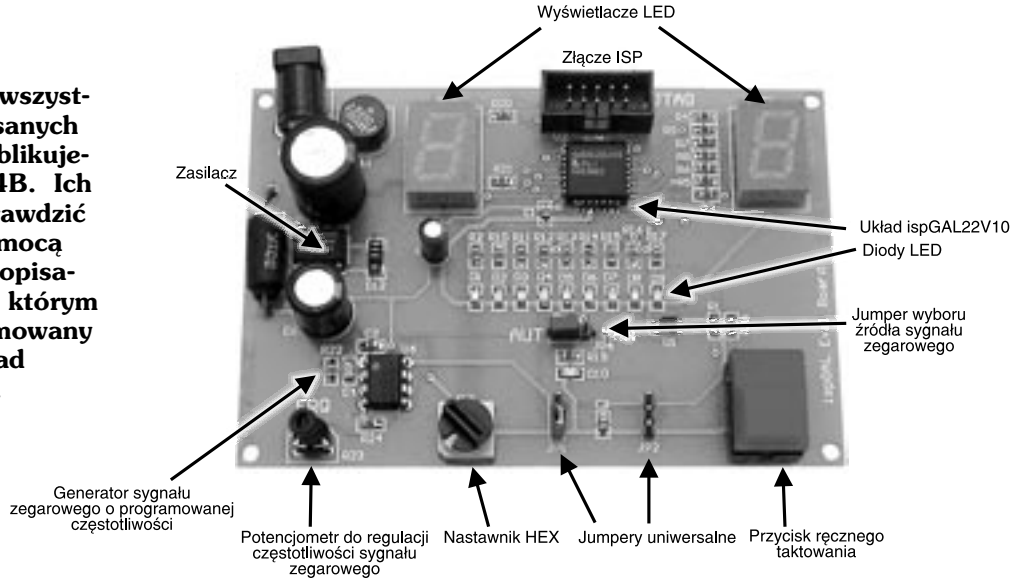
Na **rys. 39** pokazano ulokowanie RimuSCH w typowym cyklu projektowym. Rolę syntezera logicznego może spełniać dowolny kompilator CUPL-a, także jego wersja DOS-owa. Na rysunku przedstawiono dwa zalecane programy:



Rys. 39

Można także w praktyce

Programy źródłowe wszystkich projektów opisanych w ramach kursu publikujemy na CD-EP9/2004B. Ich działanie można sprawdzić w praktyce za pomocą zestawu AVT-599 (opisany w EP3/2004), w którym zastosowano programowany w systemie układ ispGAL22V10.



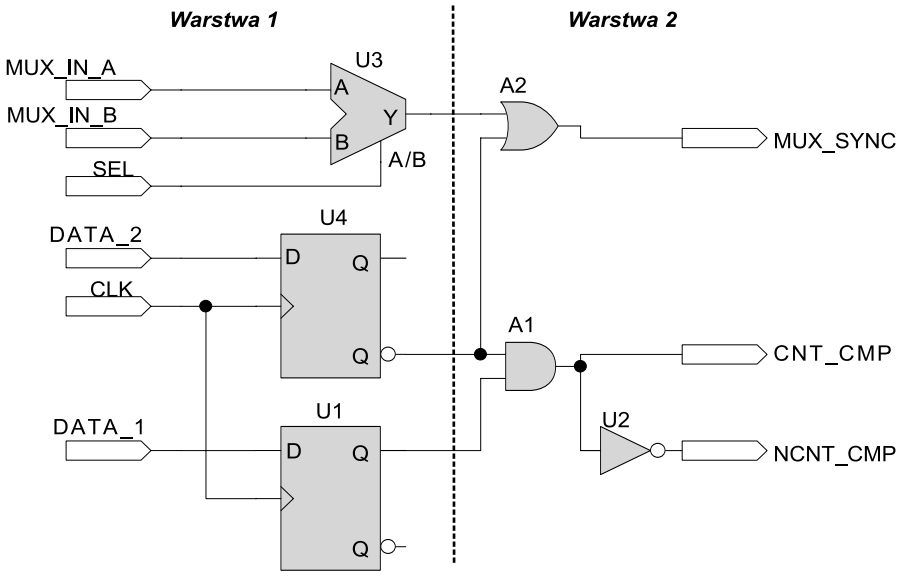
– WinCUPL, którego najpoważniejszą (w zasadzie jedyną) wadą są mocno ograniczone biblioteki układów docelowych (do układów zgodnych z oferowanymi przez Atmela), co jest o tyle oczywiste, że ta wersja programu jest własnością Atmela. Niebagatelną zaletą tego programu jest możliwość nieodpłatnego (przy tym legalnego) korzystania z niego.
– Protel 99SE lub nowszy, którego wadą jest ograniczony czas legalnego korzystania (ze względu na ograniczenia wersji ewaluacyjnej), za to biblioteki obsługiwanych układów są niezwykle bogate.

Tak więc, kompilator można dobrać do indywidualnych potrzeb i wymagań. W przypadku projektów prezentowanych w tym cyklu artykułów, do ich przetestowania w zestawie AVT-599 w zupełności wystarczy WinCUPL.

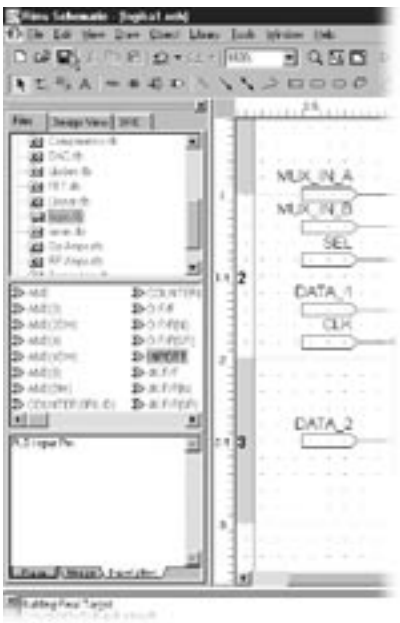
Projekt w dwóch plikach

Kompletny projekt CUPL składa się z dwóch plików o rozszerzeniach: **.pld* (źródłowy) oraz **.si* (symulacyjny - jeżeli symulator jest wykorzystywany podczas projektowania). Przeniesienie projektu pomiędzy programami wykorzystywanymi podczas projektowania oznacza w praktyce przeniesienie tych dwóch plików.

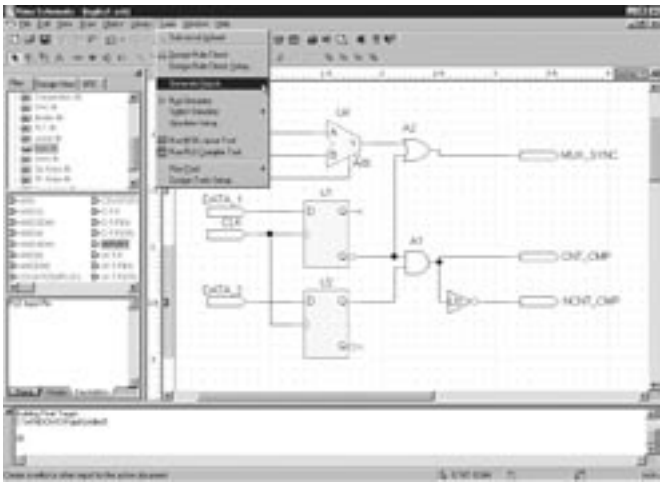
Na **rys. 40** pokazano schemat logiczny przykładowego projektu, który przygotowano za pomocą edytora RimuSCH. Podczas rysowania schematów logicznych przeznaczonych do konwersji do postaci HDL, należy korzystać wyłącznie z biblioteki *logic.rlb*, która jest dostarczana w standardowej instalacji edytora (**rys. 41**). W jej ramach są dostępne wszystkie standardowe funktory logiczne (bram-



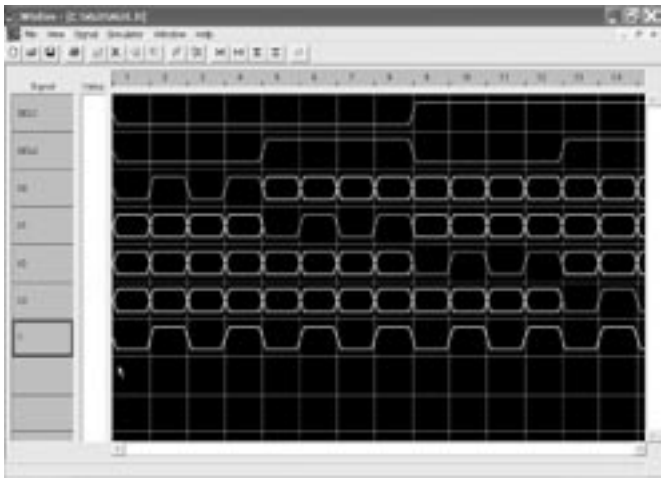
Rys. 40



Rys. 41



Rys. 42



Rys. 44

ki AND, NAND, OR, NOR, ExOR i ExNOR, także w wersjach o dużej liczbie wejść, ich symbole są dostępne w dwóch notacjach), przerzutniki JK, T, D i SR, a także multipleksery. Twórca RimuSCH zawarł ponadto w bibliotece *logic.rlb* dwa złożone elementy biblioteczne – 8-bitowe liczniki góra-dół z wyjściami trójstanowymi, które – ze względu na przyjęty sposób opisu – nie we wszystkich układach PLD dają się wygodnie zaimplementować.

Po narysowaniu schematu można go poddać weryfikacji elektrycznej za pomocą standardowego narzędzia ERC wbudowanego w RimuSCH (pozwala wychwycić „grube" pomyłki w narysowanym schemacie), a następnie wyeksportować do formatu pliku źródłowego **.pld* z opisem w języku CUPL. Wymaga to wybrania w menu opcji *Tools> Generale Report...* (**rys. 42**), następnie w wyświetlonym oknie (**rys. 43**) wybieramy format pliku wyjściowego (na liście *Report Type*). W ten sposób uzyskujemy plik źródłowy z opisem projektu przedstawionego na schemacie (rys. 40) w języku CUPL. Opis HDL przedstawiono na **list. 19**. Niestety, pokazany przykład ilustruje drobną słabość wbudowanego w RimuSCH generatora plików wyjściowych: w przypadku projektów wielopoziomowych (tzn. takich, w których występują zagrzebane węzły logiczne wykorzystywane do implementacji funkcji logicznych – rys. 40) edytor samodzielnie tworzy nazwy węzłów (jak *N000001* lub *n0006* na list. 19), ale nie deklaruje ich w pliku źródłowym. Powoduje to, że wygenerowany plik nie daje się skompilować. Konieczne jest uzupełnienie go o deklaracje węzłów zagrzebanych:

```
node N000001;
node n0006;
```

lub węzłów przypisanych do wyprowadzeń docelowego układu:

```
pinnode = N000001;
pinnode = n0006;
```

Rys. 43

Kompilacja opisu – możliwość pierwsza

Uzyskany opis poddamy kompilacji za pomocą programu WinCUPL. Ponieważ RimuSCH generuje plik w formacie **.pld*, sformatowany zgodnie ze specyfikacją CUPL-a, można wykorzystać go jak standardowy plik projektowy. W przypadku takiej konieczności, opis uzyskany z RimuSCH należy zmodyfikować o wcześniej wspomniane deklaracje, co umożliwi jego kompilację.

W zależności od potrzeb, WinCUPL umożliwia kompilację docelową, tzn. z uwzględnieniem ograniczeń architektury fizycznych układów lub – co jest przydatne podczas weryfikacji poprawności opisu HDL – kompilację na układ wirtualny (*virtual device*), który nie ma żadnych ograniczeń wynikających z budowy jego mikrokomórek, buforów I/O itp. Podczas pierwszych samodzielnych prób zdecydowanie łatwiejsze będzie

Nie do końca doskonały

RimuSCH generuje niepoprawne opisy dla projektów składających się z więcej niż jednej warstwy. Konieczne jest uzupełnienie wygenerowanego opisu o deklaracje węzłów zagrzebanych (*node*) oraz wyprowadzonych na zewnątrz układu (*pinnode*).

List. 19. Plik źródłowy z opisem projektu pokazanego na rysunku 40, uzyskany za pomocą edytora RimuSCH

```
Name logika1.sch;
Partno ;
Date 04-09-14 13:05:54;
Revision ;
Designer ;
Company ;
Assembly XXXXX;
Location XXXXX;

/*******************************/
/*       CUPL HDL Format       */
/*    File : C:logika1.sch     */
/* Generated by Rimu Schematic */
/*******************************/
/**  Allowable Target Device  **/
/*******************************/

/** Inputs **/
Pin  = DATA_1;
Pin  = CLK;
Pin  = DATA_2;
Pin  = MUX_IN_B;
Pin  = MUX_IN_A;
Pin  = SEL;

/** Outputs **/
Pin = MUX_SYNC;
Pin = CNT_CMP;
Pin = NCNT_CMP;

/** Declarations and Intermediate Variable
Definitions */

/** Logic Equations **/
N000001.d = DATA_1;

N000001.ck = CLK;

n0011 = !N000001;
n0006.d = DATA_2;

n0006.ck = CLK;

N000002 = !n0006;
CNT_CMP = n0011 & n0006;
MUX_SYNC = n0010 # n0011;
NCNT_CMP = !CNT_CMP;
n0010 = (MUX_IN_A&!SEL) # (MUX_IN_B&SEL);
```

List. 20. Plik zawierający opis symulacji multipleksera z list. 10 (EP8/2004)

```
Name     mux;
PartNo   brak;
Date     20/05/04;
Revision brak;
Designer PZb;
Company  EP;
Assembly brak;
Location brak;
Device   g22v10lcc;

ORDER: SEL1, SEL0, X0, X1, X2, X3, Y;
VECTORS:
000XXX*
001XXX*
000XXX*
001XXX*
01X0XX*
01X1XX*
01X0XX*
01X1XX*
10XX0X*
10XX1X*
10XX0X*
10XX1X*
11XXX0*
11XXX1*
11XXX0*
11XXX1*
```

List. 21. Pliki z wynikami symulacji z formatowaniem i bez niego

```
Bez formatowania
================
     SS
     EE
     LLXXXX
     100123Y
================
0001: 000XXXL
0002: 001XXXH
0003: 000XXXL
0004: 001XXXH
0005: 01X0XXL
0006: 01X1XXH
0007: 01X0XXL
0008: 01X1XXH
0009: 10XX0XL
0010: 10XX1XH
0011: 10XX0XL
0012: 10XX1XH
0013: 11XXX0L
0014: 11XXX1H
0015: 11XXX0L
0016: 11XXX1H

Z formatowaniem
=====================
     SS
     EE
     LL    XXXX
     10    0123  Y
=====================
0001: 00    0XXX  L
0002: 00    1XXX  H
0003: 00    0XXX  L
0004: 00    1XXX  H
0005: 01    X0XX  L
0006: 01    X1XX  H
0007: 01    X0XX  L
0008: 01    X1XX  H
0009: 10    XX0X  L
0010: 10    XX1X  H
0011: 10    XX0X  L
0012: 10    XX1X  H
0013: 11    XXX0  L
0014: 11    XXX1  H
0015: 11    XXX0  L
0016: 11    XXX1  H
```

Rys. 45

korzystanie z możliwości układu wirtualnego tym bardziej, że diagnostyka błędów wbudowana w WinCUPL-u nie jest najwyższych lotów. Zdarza się, że komunikat o błędzie wyprowadza projektanta na manowce, sugerując błędy w innych miejscach niż występują w rzeczywistości.

Symulacja w WinCUPL-u

Debugowanie projektu ułatwia wbudowany w WinCUPL-a symulator funkcjonalny, który prezentuje wyniki symulacji w postaci graficznej (na **rys. 44** pokazano widok okna symulatora z wynikami weryfikacji projektu dwuwejściowego multipleksera – jego opis pokazano w EP8/2004 na list. 10).

Do przeprowadzenia symulacji konieczny jest plik wejściowy **.si*, zawierający listę weryfikowanych sygnałów oraz informacje o wektorach wejściowych (pobudzeniach). Na **list. 20** pokazano plik zawierający opis przebiegu symulacji. Jest on dość prosty w analizie:

– Sekcja zaczynająca się od słowa `ORDER` zawiera listę sygnałów wejściowych, wyjściowych i węzłów wewnętrznych, których stany będą monitorowane lub wymuszane.
– Sekcja zaczynająca się od słowa `VECTORS` zawiera informacje o kolejnych (w wierszach) stanach wejść i wyjść. Stany wyjściowe projektant może określić samodzielnie, opisując to, czego się spodziewa, oddać też inicjatywę w ręce symulatora, który określi stany na wyjściach układu.

Tab. 14. Sygnały wykorzystywanych w plikach **.si*

Oznaczenie	Opis	Dotyczy...	Wymuszenie stosowane w graficznym edytorze przebiegów WinCUPL	Symbol
0	Zero logiczne	...wejść	Kliknięcie w dolnej lewej części komórki	–
1	Jedynka logiczna	...wejść	Kliknięcie w górnej lewej części komórki	–
C	Sygnał zegarowy „0-1-0”	...wejść	-[1]	
K	Sygnał zegarowy „1-0-1”	...wejść	-[1]	
X	Stan nieistotny	...wejść	Kliknięcie w środkowej części komórki	
P	Wstępne ładowanie rejestrów	...wejść	Lewy przycisk ALT + kliknięcie myszką w środkowej części komórki	–
H	Stan wysoki na wyjściu testowanego układu	...wyjść	Kliknięcie w dolnej prawej części komórki	–
L	Stan niski na wyjściu testowanego układu	...wyjść	Przycisk SHIFT + kliknięcie w górnej prawej części komórki	–
Z	Stan wysokiej impedancji	...wyjść	Kliknięcie w środkowej części komórki	
*	Stan wyliczany przez symulator	...wyjść	Lewy przycisk CTRL + kliknięcie myszką w środkowej części komórki	–

Interpreter wbudowany w symulator CUPL-a umożliwi m.in. proste formatowanie wyników symulacji generowanych do pliku tekstowego (**.so*, rodzaj raportu z symulacji). Za pomocą znaku `%` można pomiędzy nazwami sygnałów wstawić zadaną liczbę spacji, co ułatwi pogrupowanie sygnałów w taki sposób, aby zwiększyć czytelność raportu. Przykłady plików **.so* bez formatowania i z formatowaniem, uzyskanym poprzez zapisanie linii ze słowem kluczowym `ORDER` w następujący sposób:

```
ORDER: SEL1, SEL0, %4, X0, X1, X2, X3, %2, Y;
```

pokazano na **list. 21**.

Pobudzenia wejść i stany wyjść oznaczane są symbolami, które zebrano w **tab. 14**. Pliki pobudzeń można tworzyć ręcznie za pomocą dowolnego edytora tekstów, można także skorzystać z wbudowanego w WinCUPL-a edytora przebiegów, który znacznie ułatwia (można ręcznie określić stan przypisany określonej liczbie umownych jednostek czasu – **rys. 45**) i przyspiesza zarówno tworzenie samych pobudzeń jak i weryfikację reakcji układu na nie. Twórcy edytora przebiegów zastosowali pomysłowy sposób określania stanów za pomocą myszki: zależy on od miejsca w obrębie komórki wyznaczającej pojedynczy krok na osi czasu, w którym użytkownik kliknie myszką. Przykładowo, kliknięcie w środkowej części komórki powoduje przypisanie stanu wysokiej impedancji, w prawej dolnej części komórki – stanu L, w lewej górnej – stanu „1”, a kliknięcie w środkowej części z jednoczesnym przytrzymaniem klawisza ALT wymusza załadowanie rejestrów predefiniowaną wartością początkową. Niestety, z nieustalonych przyczyn, nie jest możliwe wprowadzenie w ten sposób sygnałów zegarowych i to wbrew zapisom w dokumentacji WinCUPL-a.

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

Układy programowalne, część 9

W ostatnim odcinku cyklu przedstawimy jedno z najwygodniejszych, pośród obecnie dostępnych, uniwersalnych narzędzi do realizacji projektów w układach programowalnych: pakiet Protel 99SE. Pakiet ten, podobnie do następcy - Protela DXP - jest wyposażony w kompilator CUPL-a, który współpracuje z systemowym edytorem schematów. Niewątpliwą zaletą pakietów Protel jest możliwość realizacji całego projektu urządzenia w jednym środowisku, bez konieczności oswajania się z różnymi interfejsami użytkowników i odmiennymi filozofiami obsługi programów.

Kompilator CUPL-a, w jaki wyposażono Protele, jest minimalnie zmodyfikowanym wcieleniem pierwszej wersji CUPL-a dla Windows, jaka powstała wiele lat temu w firmie Data I/O. Firma Protel (później Altium) nie przykładała się specjalnie do powiększania jego walorów użytkowych, stąd większość wad oryginalnego kompilatora (w tym przede wszystkim niezbyt przejrzysta diagnostyka błędów) jest odczuwalna do dziś. Dodano jedynie kilka prostych rozszerzeń funkcjonalnych (jak na przykład kreator projektów), poprawiono także współpracę kompilatora z edytorem przebiegów, który ułatwia analizę wyników symulacji. Ze względu na specyfikę środowiska protelowskiego nieco inaczej niż miało to miejsce w WinCUPL-u wygląda obsługa kompilatora. Jemu właśnie poświęcamy ten odcinek kursu.

Czarownik Ci pomoże

Ogromnym ułatwieniem podczas realizacji projektów w układach PLD, zwłaszcza dla początkujących użytkowników, jest wbudowany w Protela 99SE kreator projektów (*wizard*). Dzięki niemu większość typowych problemów program rozstrzyga interaktywnie z projektantem, zadając mu kolejno proste pytania, których odpowiedzi pozwalają uzyskać „szkielet" opisu HDL.

Jak wiadomo Protel 99SE przechowuje wszystkie pliki wchodzące w skład projektu w jednym pliku o rozszerzeniu **.ddb*. Plik taki należy utworzyć dla każdego nowego projektu, co

Rys. 46

Rys. 47

wymaga wybrania w menu opcji *File>New*. W wyświetlonym oknie *New Design Database* należy podać docelową lokalizację pliku oraz nazwę pliku (rozszerzenie **.ddb* nie jest nadawane automatycznie!). Plik wykorzystany w przykładzie nosi nazwę *PLD_proj.ddb* i jest ulokowany w głównym katalogu dysku C.

Kolejnym krokiem jest utworzenie pliku zawierającego opis HDL w postaci tekstowej (w języku CUPL) lub schematu. Ponieważ w tej części artykułu zajmiemy się przedstawieniem sposobu realizacji projektu opisanego tekstowo, mamy do wyboru dwie drogi:

- skorzystanie z kreatora projektu,
- utworzenie „pustego" pliku tekstowego, w którym trzeba będzie przygotować cały opis HDL.

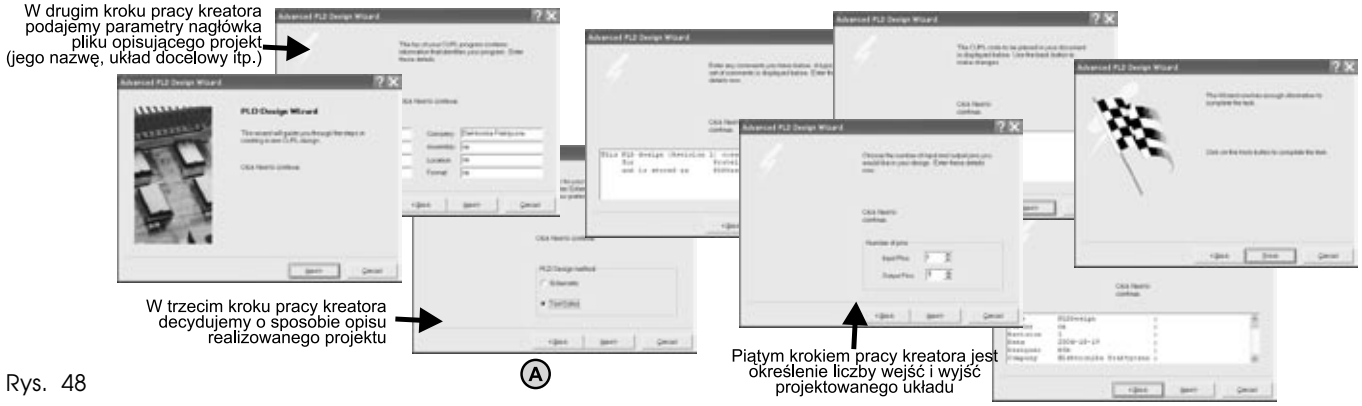
Ze względu na wygodę warto skorzystać z interaktywnej pomocy kreatora. W tym celu w menu wybieramy opcję *File>New...*, co powoduje wyświetlenie okna *New Document*, którego widok pokazano na **rys. 46**. Wybieramy w tym oknie zakładkę *Wizards*, a w niej *PLD-CUPL Wizard* (**rys. 47**). Powoduje to zainicjowanie pracy kreatora, którego jeden z kilku możliwych wariantów przebiegu pokazano na **rys. 48**. Na rysunku tym zaznaczono najistotniejsze z punktu widzenia projektanta miejsca, w których kreator „dopytuje" się o najważniejsze parametry realizowanego projektu.

List. 22. Szkielet opisu HDL wygenerowany przez kreatora projektów PLD z pakietu CUPL

```
Name        PLDDesign  ;
Partno      na         ;
Revision    1          ;
Date        2004-10-17 ;
Designer    PZb        ;
Company     EP         ;
Assembly    na         ;
Location    na         ;
Device      G22V10LCC  ;
Format      JEDEC      ;

/*************************************************
************************/
/* This PLD design (Revision 1) created on
2004-10-17                */
/*      for                  Protel Interna-
tional                    */
/*      and is stored as     PLDDesign
*/
/*************************************************
************************/

/** Inputs  **/
Pin 1   =  Input_1  ;
Pin 2   =  Input_2  ;
Pin 3   =  Input_3  ;
Pin 4   =  Input_4  ;

/** Outputs **/
Pin 5   =  Output_5 ;
Pin 6   =  Output_6 ;
Pin 7   =  Output_7 ;
Pin 8   =  Output_8 ;
Pin 9   =  Output_9 ;
Pin 10  =  Output_10;
Pin 11  =  Output_11;

/** Declarations and Intermediate Variables
**/

/** Logic Equations **/
```

Uniwersalne narzędzie

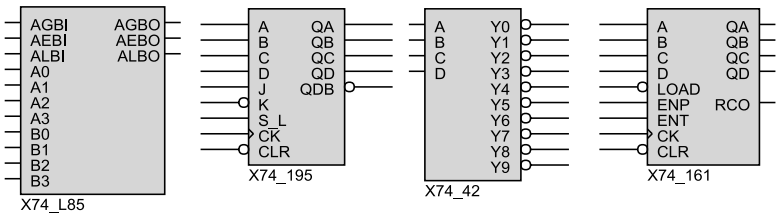
Protel 99SE umożliwia realizację projektów w układach PLD pochodzących od różnych producentów. W sumie do dyspozycji projektantów jest ponad 200 różnych typów układów PLD, począwszy od pamięci PROM, aż po układy CPLD i FPGA różnych producentów.



Rys. 48

Na skróty, ku wygodzie!

Przykładowe symbole biblioteczne, funkcjonalne odpowiedniki układów TTL



Standardowo Protel 99SE jest wyposażony w bibliotekę symboli służących do projektowania układów PLD, wśród których są dostępne m.in. wygodne w stosowaniu wybrane odpowiedniki funkcjonalne układów TTL.

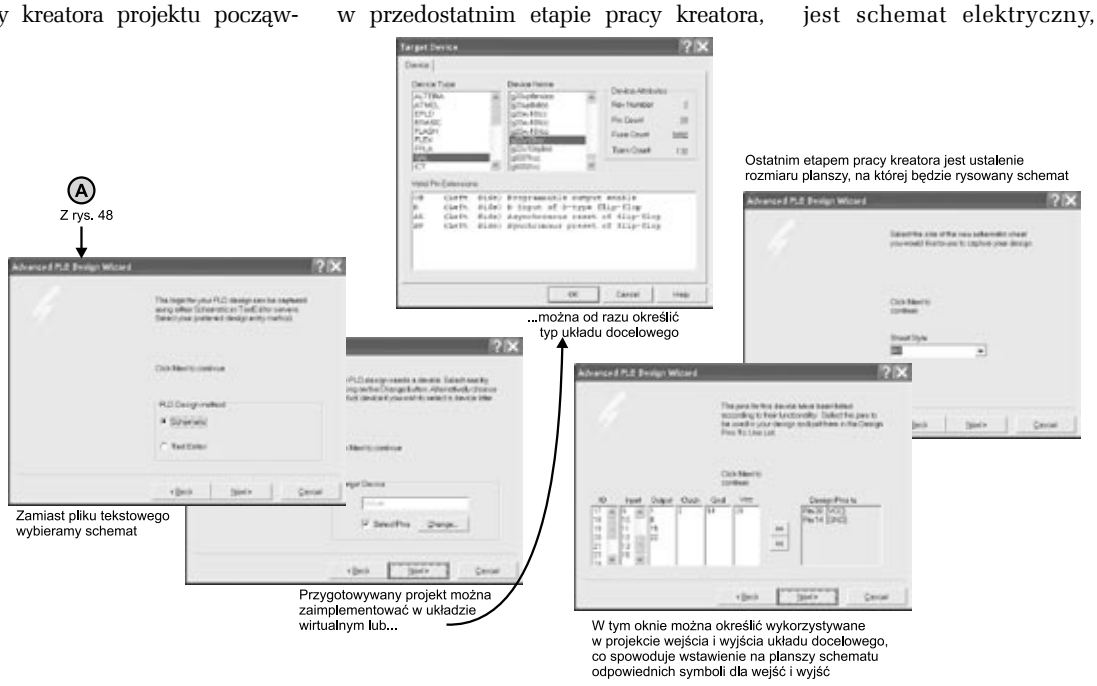
Układ wirtualny

Projektanci korzystający z CUPL-a mogą realizować projekty na układy wirtualne (virtual device), które są pozbawione ograniczeń charakterystycznych dla układów rzeczywistych (jak choćby liczba termów matrycy OR, maksymalna liczba wejść bramek AND w matrycy programowalnej, przypisanie sygnałów globalnych zegarowych itp.).

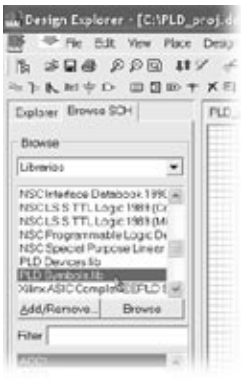
W wyniku działania kreatora powstaje plik tekstowy zawierający podstawowe elementy opisu HDL - uzyskany przykładowy szkielet opisu HDL pokazano na **list. 22**.

Nieco inaczej wygląda inicjacja projektu, w którym zamiast tekstowego opisu HDL użytkownik chciałby wykorzystać bardziej przyjazny opis schematowy. Na **rys. 49** pokazano przebieg ścieżki pracy kreatora projektu począwszy od chwili wybrania jako pliku projektowego schematu (punkt oznaczony literą „A" na rys. 48). W zależności od zamiarów, projektant może realizować projekt na układzie wirtualnym, może także wskazać rzeczywisty układ docelowy. Kolejnym krokiem jest określenie wykorzystywanych w projekcie linii wejściowych i wyjściowych (muszą się one znaleźć w okienku *Design Pins to* w przedostatnim etapie pracy kreatora, pokazanym na rys. 49), co zaowocuje automatycznym wprowadzeniem na planszę schematu odpowiednich symboli. Można je oczywiście później modyfikować, ale warto (zwłaszcza przy pierwszych projektach) poświęcić chwilę na podanie tych parametrów - uprości to dalsze prace nad projektem.

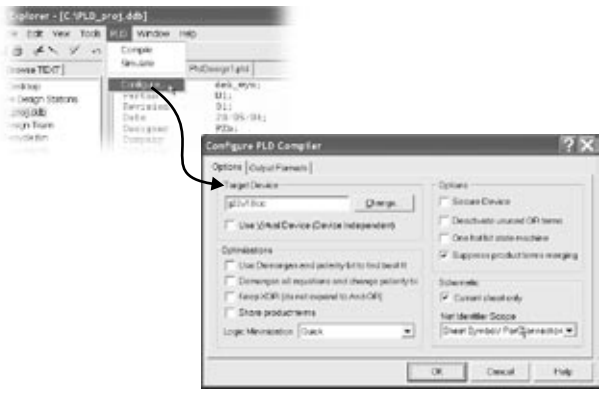
Na tym, w zasadzie, kończy się praca kreatora projektu. Jej efektem jest schemat elektryczny, na którym



Rys. 49



Rys. 50



Rys. 51

są umieszczone symbole wejść i wyjść (zgodnie z opisem podanym przez użytkownika) oraz symbole zasilania (które jednak do niczego nie służą i można jej usunąć z planszy).

Biblioteki PLD dla edytora schematów

Protel 99SE wyposażono w bibliotekę symboli (dla edytora schematów) elementów logicznych przygotowanych z myślą o realizacji projektów na układach PLD. Biblioteka o nazwie *PLD Symbols.lib* (**rys. 50**) jest przechowywana w pliku *PLD.ddb* i zawiera, oprócz podstawowych funktorów logicznych (jak m.in. porty wejściowe, wyjściowe i dwukierunkowe, bramki AND, OR, NOR, NOT itp.) wiele „makrofunkcji", będących odpowiednikami układów) TTL. Symbole te są oznaczane w następujący sposób:

```
X74_aaa,
```

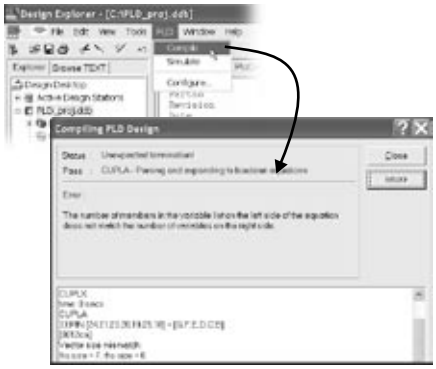
gdzie *aaa* oznacza dwu- lub trzycyfrowy symbol zaczerpnięty z oznaczenia układu TTL. Przykładowo symbol X74_42 odpowiada funkcjonalnie dekoderowi 7442, a symbol X74_151 odpowiada funkcjonalnie multiplekserowi 74151.

Kompilacja projektu

Kompilator wbudowany w Protela 99SE w większości przypadków nie wymaga konfiguracji - domyślne nastawy zazwyczaj umożliwiają poprawną kompilację opisu HDL. Warto jednak zdawać sobie sprawę z możliwości zmodyfikowania parametrów syntezy logicznej, co w pewnych sytuacjach (na przykład niewystarczającej liczby termów wejściowych makrokomórki) może mieć decydujący wpływ na jakość implementacji projektu.

Okno konfiguracji kompilatora jest wyświetlane po wybraniu w menu opcji *PLD>Configure* (**rys. 51**). Domyślnie jest wyświetlana zakładka *Options*, na której można ustalić typ docelowego układu PLD (*Target Device*, ewentualnie *Use Virtual Device*), określić sposób optymalizacji opisu HDL i kodowania stanów automatów i wybrać algorytm minimalizacji opisu (*Logic Minimization*). W większości przypadków nie ma jednak konieczności modyfikowania ustawień dostępnych na tej zakładce. Druga z zakładek okna *Configure PLD Compiler* (**rys. 52**) służy do ustalenia

a)



b)

Rys. 53

Rys. 54

Symulacja i oś czasu

Symulator układów PLD wbudowany w Protela 99SE jest symulatorem funkcjonalnym, nie ma więc możliwości zweryfikowania wpływu parametrów czasowych układów docelowych na ich działanie w fizycznym urządzeniu.

jakie pliki będą tworzone podczas pracy kompilatora, za pomocą dostępnych ustawień można także skonfigurować zawartość pliku raportu z kompilacji. W przypadku kompilowania schematu zawierającego opis projektu można uaktywnić automatyczny podgląd pliku HDL (opcja *Schematic*, *View PLD File*), który zawiera tekstowy opis schematu.

Po skonfigurowaniu kompilatora możemy podjąć próbę skompilowania projektu. W tym celu w menu wybieramy opcje: *PLD>Compile*, co powoduje natychmiastowe uruchomienie kompilatora. W zależności od przebiegu kompilacji, wyświetlone okno zawiera informacje o wykrytych błędach (**rys. 53a**) lub o pomyślnym zakończeniu kompilacji (**rys. 53b**).

Kolejnym krokiem realizacji projektu jest jego symulacja. Protel 99SE wyposażono w niezły symulator funkcjonalny, za pomocą którego można wiarygodnie zweryfikować uzyskane wyniki.

Symulacja projektu

Do symulacji projektu opisanego w CUPL-u niezbędny jest plik **.si*, zawierający wektory pobudzeń oraz (opcjonalnie) odpowiedzi. Jest to taki sam plik jak w przypadku wszystkich innych wersji CUPL-a. Nazwa pliku **.si* musi być identyczna jak nazwa projektu!

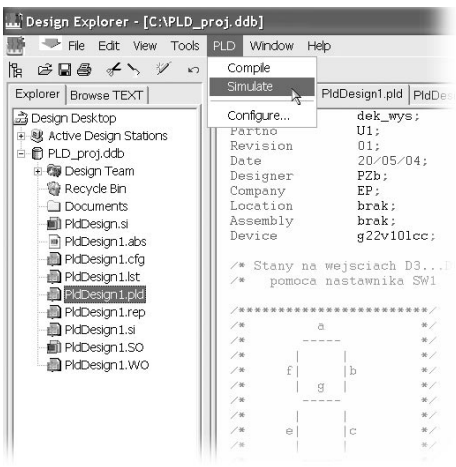
Do symulacji niezbędny jest jeszcze drugi plik - o rozszerzeniu **.abs* - uzyskiwany podczas kompilacji (w zakładce *Output Formats* okna *Configure PLD Compiler* należy zaznaczyć opcję *Simulation*, *Absolute ABS*).

Te dwa pliki umożliwiają przeprowadzenie symulacji. Jej uruchomienie wymaga wskazania w oknie menadżera projektów pliku **.pld* lub **.sch* (w zależności od przyjętego sposobu opisania projektu) i wybrania w menu opcji *PLD>Simulate* (**rys. 54**). Należy pamiętać, że symulacja nie zostanie przeprowadzona, jeżeli w oknie menadżera projektu będzie zaznaczony inny plik, niż ten, który zawiera opis HDL!

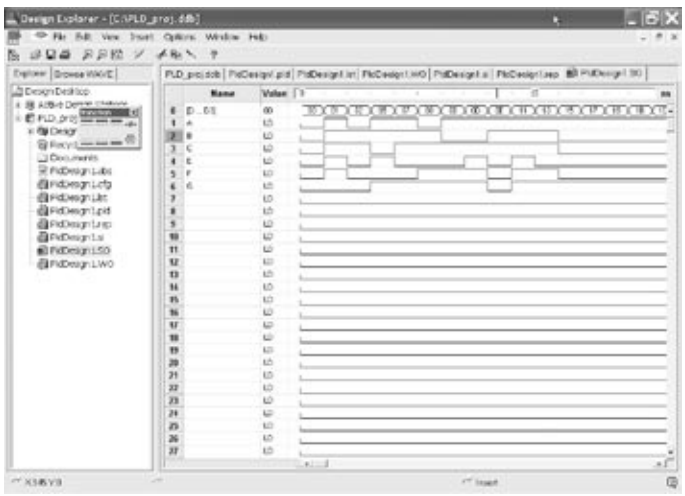
Wynikiem pracy symulatora jest między innymi plik **.so*, który ma postać tekstową (podobnie jak w WinCUPL-u i innych mutacjach CUPL-a), ale w przypadku zainstalowania Protela 99SE z serwerem *Waveform Display*, wyniki symulacji są prezentowane w postaci przebiegów (**rys. 55**). Należy pamiętać, że prezentowana przez symulator skala czasu nie ma odzwierciedlenia w rzeczywistości!

Warto wziąć także pod uwagę, że pomimo deklarowanej przez producenta możliwości edycji przebiegów ilustrujących działanie układu (są one wyświetlane przecież w systemowym edytorze przebiegów), ich modyfikacja i poprawne przechowywanie w pliku projektu **.ddb* jest (a raczej bywa) możliwe w wersjach z zainstalowanym *Service Packiem 6*. Piszę „bywa”, ponieważ Protel 99SE z SP6 po zainstalowaniu na dwóch komputerach zachowuje się - bez wyraźnej przyczyny - odmiennie. Stąd zachęta do przyjęcia zasady, że modyfikacje wektorów pobudzeń i odpowiedzi należy przeprowadzać w pliku **.si*, a edytorowi przebiegów pozostawić wyłącznie funkcję przeglądarki.

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



Rys. 55



Rys. 56