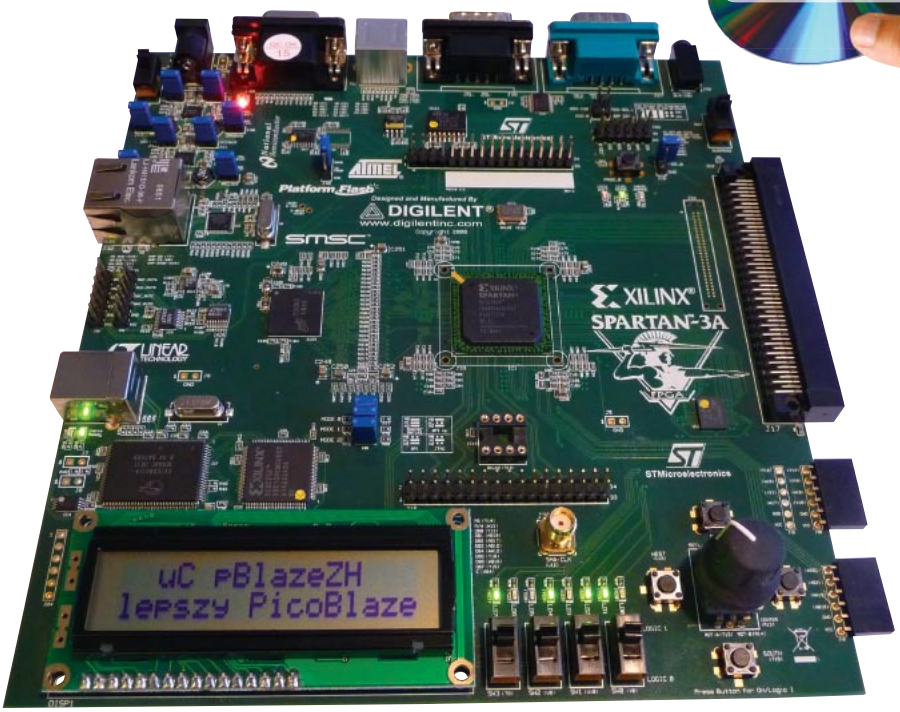
Mikrokontrolery w FPGA

Szybszy PicoBlaze i nie tylko dla układów PLD Xilinx



Czytelnikom Elektroniki Praktycznej z pewnością jest już znany 8-bitowy mikrokontroler PicoBlaze przeznaczony do implementacji w układach programowalnych Xilinx. Ogólna architektura tego mikrokontrolera, jak również przykładowy sposób jego wykorzystania, prezentowane były w EP 5-6/2005 oraz 4-6/2008. W tym artykule przedstawimy opis behawioralny w języku Verilog mikrokontrolera zgodnego z PicoBlaze, charakteryzującego się dodatkowo dwukrotnie szybszym rdzeniem realizującym rozkazy w ciągu jednego taktu zegara.

Oryginalny mikrokontroler PicoBlaze można pobrać ze strony producenta w postaci kodu w jednym z języków opisu sprzętu i następnie zaimplementować w wybranym układzie programowalnym CPLD lub FPGA. Dostarczany przez *Xilinx'a* kod w języku Verilog lub VHDL zawiera opis strukturalny mikrokontrolera zoptymalizowany dla konkretnej rodziny układów *Xilinx*. Opis strukturalny odwołuje się do bibliotek elementów charakterystycznych dla produktów firmy *Xilinx*. Tego mikrokontrolera nie można więc bezpośrednio zaimplementować w układach programowalnych pochodzących od innego producenta.

Nie jest to zgodne z udzielaną licencją. Dzięki zastosowaniu opisu behawioralnego prezentowany tutaj mikrokontroler może być wykorzystany do implementacji w układach programowalnych dowolnego producenta. Dodatkowo kod opisujący mikrokontroler można w łatwy sposób modyfikować dostosowując niektóre właściwości funkcjonalne mikrokontrolera do wymagań bieżącej aplikacji.

Mikrokontrolery osadzone w FPGA

Zarówno za pomocą mikrokontrolerów jak i układów programowalnych FPGA, można zaimplementować praktycznie dowolną funkcję logiczną. Jednak każda z tych technologii posiada swoje unikalne cechy i właściwości takie jak koszt implementacji, szybkość działania (wydajność), łatwość użycia itp. Ogólnie można powiedzieć, że mikrokontrolery nadają się doskonale do realizacji aplikacji sterujących zwłaszcza w sytuacji, gdy wymagania projektowe realizowanego algorytmu zmieniają się w szerokich granicach. Często zdarza się, że budowanie nawet stosunkowo prostych aplikacji sterujących w układach programowalnych wymaga żmudnej specyfikacji automatów sekwencyjnych. Nawet niewielka zmiana algorytmu sterującego pociąga za sobą w takim przypadku spore zmiany w specyfikacji automatu. Dla mikrokontrolerów programowanie sekwencji sterujących lub automatów sekwencyjnych w języku maszynowym jest najczęściej znacznie prostsze niż utworzenie odpowiednich struktur w układach programowalnych.

Wadą mikrokontrolerów jest jednak to, że mają ograniczoną wydajność. Wszystkie instrukcje wykonywane są przez mikrokontrolery sekwencyjnie. Wraz ze wzrostem złożoności algorytmu rośnie ilość instrukcji niezbędna do jego realizacji, a tym samym wydajność całego systemu zmniejsza się. W kontekście wydajności systemu układy programowalne są znacznie bardziej elastyczne. Dany algorytm sterujący może być w układach PLD zrealizowany dwojako: sekwencyjnie lub też w pełni równolegle – zależnie od wymaganej szybkości działania. Równoległa realizacja algorytmu jest oczywiście szybsza, jednak wymaga większej liczby zasobów logicznych układu programowalnego.

Połączenie tych różnych światów – układów FPGA i mikrokontrolerów, czyli osadzenie mikrokontrolerów w układach FPGA – pozwala na wydobycie najlepszych cech obu technologii. Mikrokontrolery mogą realizować złożone algorytmy sterujące, których czas wykonania nie jest krytyczny, podczas gdy elementy systemu, w tym krytyczne czasowo funkcje układu operacyjnego, najlepiej jest zrealizować z użyciem zasobów logicznych układów FPGA. Takie podejście prowadzi do projektowania z podziałem zadań pomiędzy sprzęt i oprogramowanie (*Software Hardware Co-Design*). Można jednak dostrzec, że pewną ujemną stroną jest tutaj potrzeba stosowania innych narzędzi do projektowania sprzętu i innych do projek-

towania i testowania oprogramowania. Integrując w jednym układzie FPGA cały system z mikrokontrolerem i układami peryferyjnymi, realizujący zadania określonej aplikacji, w istocie uzyskujemy tzw. zintegrowany system cyfrowy – SoC (*System on-Chip*).

W przypadku mikrokontrolerów osadzanych w układach FPGA zapotrzebowanie na zasoby logiczne FPGA niezbędne do implementacji mikrokontrolera jest względnie stałe – niezależne od stopnia złożoności realizowanego przez mikrokontroler algorytmu. Te same bloki logiczne układu FPGA wykorzystywane są przez wiele różnych instrukcji mikrokontrolera co zmniejsza zapotrzebowanie na zasoby. Jednak wraz ze wzrostem złożoności algorytmu może wzrastać zapotrzebowanie mikrokontrolera na pamięć operacyjną jak również pamięć programu. W przypadku algorytmów realizowanych sprzętowo w układach FPGA, zapotrzebowanie na zasoby logiczne jest proporcjonalne do stopnia złożoności tego algorytmu. Czym algorytm jest bardziej złożony tym więcej potrzeba bloków logicznych FPGA do jego realizacji.

Zamierzając wykorzystać mikrokontroler osadzony w FPGA, dzięki któremu uzyskanie określonej funkcjonalności projektowanego systemu będzie charakteryzować się pewnymi zaletami w porównaniu z realizacją czysto sprzętową, możemy postąpić dwojako. Możemy albo zaprojektować własny mikrokontroler, specjalizowany do określonych zadań, charakteryzujący się specyficzną listą instrukcji, albo też dokonać miękkiej implementacji jednego z mikrokontrolerów istniejących w postaci układów ASIC. Zaletą pierwszego podejścia jest to, że projektowany mikrokontroler jest zoptymalizowany do realizacji określonych zadań. Zajmuje więc mniej zasobów logicznych układu FPGA, a także określone algorytmy realizuje w sposób bardziej efektywny, niż miało by to miejsce z użyciem typowych mikrokontrolerów ogólnego przeznaczenia. Z kolei wadą jest tu brak gotowych narzędzi ułatwiających tworzenie i uruchamianie programów.

Zaletą drugiego podejścia opierającego się na wykorzystaniu specyfikacji działania mikrokontrolera istniejącego np. jako ASIC jest to, że możemy skorzystać z dostępnych narzędzi programistycznych , w tym z kompilatorów wysokiego poziomu (np. języka C), a także wielu bibliotek programów przeznaczonych dla mikrokontrolera w wersji ASIC. Projektując miękką implementację takiego mikrokontrolera możemy się też pokusić o wprowadzenie pewnych ulepszeń pierwotnej architektury, co może owocować np. poprawą wydajności rdzenia. Nie bez znaczenia jest też kwestia pewnych subiektywnych upodobań projektanta dotyczących typu wykorzystywanego mikrokontrolera. Na przykład projektant mający pewne doświadczenie w tworzeniu aplikacji z mikrokontrolerami PIC *Microchip* z pewnością preferowałby właśnie taki mikrokontroler osadzony w FPGA, niż inny o mniej znanej architekturze.

Projektowanie mikrokontrolerów istniejących w postaci kodu w jednym z języków opisu sprzętu (tzw. miękkich implementacji) a także opis w językach HDL innych funkcji, bloków funkcjonalnych a także złożonych algorytmów sterujących staje się procesem mniej skomplikowanym dzięki wykorzystaniu metodologii projektowania typu *top-down* („od ogółu do szczegółu"). W języku opisu sprzętu specyfikuje się działanie projektowanego systemu na wysokim poziomie abstrakcji a proces przejścia od tej specyfikacji do najniższego poziomu bramek logicznych realizuje odpowiednie oprogramowanie. Znacząco upraszcza to proces projektowania. Jednak dzieje się to pewnym kosztem, którym zazwyczaj jest nieoptymalne wykorzystanie zasobów logicznych.

Oryginalny mikrokontroler PicoBlaze istnieje w postaci opisu struktury połączeń przerzutników, bramek logicznych, rejestrów przesuwających i innych elementów bibliotecznych charakterystycznych dla określonej rodziny układów *Xilinx*. Zakładając, że struktura ta jest optymalna jej implementacja wymaga minimalnej możliwej liczby zasobów logicznych docelowego układu programowalnego. Niemniej jednak bezpośrednio może być użyta tylko dla układów *Xilinx*.

Mikrokontroler pBlazeZH

Opracowany przez Autora artykułu mikrokontroler zgodny z PicoBlaze został roboczo nazwany pBlazeZH. Zasadnicze właściwości funkcjonalne tego mikrokontrolera są identyczne jak dla PicoBlaze. W szczególności można tu wymienić:

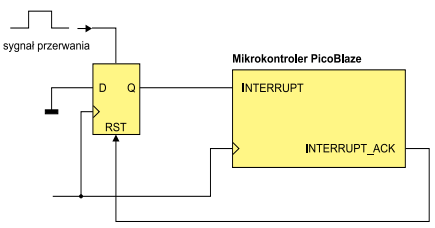
- szesnaście 8-bitowych rejestrów ogólnego przeznaczenia,
- możliwość zaadresowania 1 K słów pamięci programu,
- 8-bitowa jednostka arytmetyczno-logiczna dysponująca znacznikami przepełnienia (*CARRY*) oraz zera (*ZERO*),
- pamięci RAM o pojemności 64 bajtów,
- możliwość dołączenia 256 portów wejściowych i 256 wyjściowych,
- sprzętowy stos 32-poziomowy,
- wszystkie rozkazy wykonywane są w jednym takcie zegara z wyjątkiem rozkazów obsługi wejścia – wyjścia,
- krótki czas reakcji na przerwanie: typowo 2 takty zegara, maksymalnie 4 takty w najbardziej niekorzystnym przypadku.

Cechą, która wyróżnia pBlazeZH jest przede wszystkim to, że mikrokontroler ten dysponuje szybszym rdzeniem realizującym wszystkie rozkazy w jednym takcie zegara. Wyjątek od tej reguły stanowią rozkazy wejścia – wyjścia, które dla zachowania pełnej zgodności z PicoBlaze wykonywane są w dwóch taktach zegara. Dzięki temu sposób dołączenia urządzeń wejścia – wyjścia jest identyczny jak dla PicoBlaze i nie wymaga żadnej modyfikacji.

Przyśpieszenie działania mikrokontrolera pBlazeZH w stosunku do oryginalnego PicoBlaze uzyskano dzięki wykorzystaniu idei przetwarzania potokowego. Fazy pobrania i wykonania rozkazów dla pBlazeZH realizowane są za pośrednictwem prostego, dwustopniowego potoku. Podczas danego taktu zegara (mówiąc bardziej precyzyjnie – podczas narastającego zbocza sygnału zegarowego) pobierany jest kod następnego rozkazu do wykonania (pierwszy stopień potoku) i jednocześnie wykonywany jest rozkaz, którego kod został pobrany w poprzednim takcie zegara (drugi stopień potoku). Dzięki dość specyficznemu sterowaniu pierwszym stopniem potoku oraz szyną adresową mikrokontrolera również rozkazy skoków wykonywane są ciągu jednego taktu zegara. Dla rozkazów skoku nie zachodzi potrzeba wykonywania dodatkowego jałowego cyklu maszynowego przeznaczonego na opróżnienie potoku z pobranego wcześniej kodu rozkazu i pobranie kodu spod adresu pod który ma nastąpić skok. W takim przypadku od razu pobierany jest rozkaz znajdujący się pod właściwym adresem.

Specyficzny sposób pobierania i wykonywania rozkazów przez mikrokontroler pBlazeZH wymaga stosowania pamięci programu o odczycie synchronicznym. Dane (kod rozkazu) mogą się zmieniać na wejściu INSTRUCTION mikrokontrolera (wyjściu danych pamięci programu) tylko w odpowiedzi na narastające zbocze sygnału zegarowego.

Mikrokontroler pBlazeZH nieco inaczej przetwarza sygnał przerwania. Możliwe jest bezpośrednie dołączenie tego sygnału do wejścia INTERRUPT. W przypadku PicoBlaze zalecany sposób dołączenia sygnału przerwania pokazano na **rys. 1**. Mikrokontroler pBlazeZH w swojej strukturze integruje już przerzutnik synchronizujący sygnał przerwania z sygnałem zegara, nie ma więc potrzeby stosowania zewnętrznego przerzutnika. Sygnał potwierdzenia przyjęcia przerwania *INTERRUPT_ACK* generowany jest identycznie jak dla PicoBlze. Dlatego też zastosowanie dla pBlazeZH układu z rys. 1



Rys. 1. Zalecany przez *Xilinx* sposób dołączenia sygnału przerwania do mikrokontrolera PicoBlaze

List. 1. Kod opisujący mikrokontroler pBlazeZH

```
module pBlazeZH(input [7:0] IN_PORT,
                input INTERRUPT,RESET,CLK,
                input [17:0] INSTRUCTION,
                output [7:0] OUT_PORT,PORT_ID,
                output reg READ_STROBE,WRITE_STROBE,INTERRUPT_ACK,
                output [9:0] ADDRESS);
reg [7:0] REGISTERS [0:15];
reg [7:0] SCRATCHPAD [0:63];
reg INT_ENABLE,CARRY,ZERO,PR_CARRY,PR_ZERO;
reg [9:0] PC,pc_next;
wire [9:0] pcp1,TOS;
reg [9:0] STACK [31:0];
reg [4:0] sp,sp_next;
reg jmp,ar,ac,az,cy,z,int_sync;
wire [9:0] aaa=INSTRUCTION[9:0];
wire [7:0] kk=INSTRUCTION[7:0];
wire [3:0] sX=INSTRUCTION[11:8];
wire [3:0] sY=INSTRUCTION[7:4];
wire [7:0] DO_SP,ALU_AND;
wire fetch=(INSTRUCTION[17:13]==5'b00011);
wire store=(INSTRUCTION[17:13]==5'b10111);
wire reti=(INSTRUCTION[17:13]==5'b11100);

wire [7:0] DO2,AI1,ALU_SR,ALU_SL;
wire [7:0] AI2=(INSTRUCTION[12])?DO2:kk;
reg [7:0] AO,regs_in;
wire parity=^AO;
wire z0=~(|AO);
wire cin;
reg sr_in;
reg [8:0] ALU_ADD_SUB;
wire [5:0] SP_ADDR=AI2[5:0];
wire go=PC!=10'h3ff;
wire rd_strobe=INSTRUCTION[17:13]==5'b00010;
wire wr_strobe=INSTRUCTION[17:13]==5'b10110;
wire skip=(rd_strobe&~READ_STROBE)|(wr_strobe&~WRITE_STROBE);
wire int_req=INT_ENABLE&int_sync&~skip&~READ_STROBE&~WRITE_STROBE;

assign ADDRESS=pc_next;
assign PORT_ID=AI2;
assign OUT_PORT=AI1;
assign pcp1=(int_req|(skip&go))?PC:PC+1;
assign TOS=STACK[sp];

always @(posedge CLK) //(1)
if(RESET) int_sync<=0; else
begin
 if(INT_ENABLE&~int_sync) int_sync<=INTERRUPT;
 if(INTERRUPT_ACK) int_sync<=0;
end

always @(posedge CLK) //(2)
 if(jmp) STACK[sp_next]<=pcp1;

always @(*) //(3)
 if(READ_STROBE) regs_in=IN_PORT;
 else
 if(fetch) regs_in=DO_SP;
 else regs_in=AO;

always @(posedge CLK) //(4)
 if(~int_req&~skip&(ar|READ_STROBE|fetch))REGISTERS[sX]<=regs_in;

always @(posedge CLK) //(5)
 if(store) SCRATCHPAD[SP_ADDR]<=AI1;

assign DO2=REGISTERS[sY]; //(6)
assign AI1=REGISTERS[sX];
assign DO_SP=SCRATCHPAD[SP_ADDR];

always @(*) //(7)
 casex ({INTERRUPT_ACK,CARRY,ZERO,INSTRUCTION[17:10]})
  11'b0xx_1100_00xx: begin pc_next=aaa; jmp=1; sp_next=sp+1; end //call
  11'b01x_1100_0110: begin pc_next=aaa; jmp=1; sp_next=sp+1; end //if CARRY
  11'b00x_1100_0111: begin pc_next=aaa; jmp=1; sp_next=sp+1; end //if NOT CARRY
  11'b0x1_1100_0100: begin pc_next=aaa; jmp=1; sp_next=sp+1; end //if ZERO
  11'b0x0_1100_0101: begin pc_next=aaa; jmp=1; sp_next=sp+1; end //if NOT ZERO
  11'b0xx_1101_00xx: begin pc_next=aaa; jmp=0; sp_next=sp; end //jump
  11'b01x_1101_0110: begin pc_next=aaa; jmp=0; sp_next=sp; end //if CARRY
  11'b00x_1101_0111: begin pc_next=aaa; jmp=0; sp_next=sp; end //if NOT CARRY
  11'b0x1_1101_0100: begin pc_next=aaa; jmp=0; sp_next=sp; end //if ZERO
  11'b0x0_1101_0101: begin pc_next=aaa; jmp=0; sp_next=sp; end //if NOT ZERO
  11'b0xx_1010_10xx: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURN
  11'b01x_1010_1110: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURN if CARRY
  11'b00x_1010_1111: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURN if NOT CARRY
  11'b0x1_1010_1100: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURN if ZERO
  11'b0x0_1010_1101: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURN if NOT ZERO
  11'b1xx_xxxx_xxxx: begin pc_next=18'h3ffff; jmp=1; sp_next=sp+1; end // interrupt event
  11'b0xx_1110_0000: begin pc_next=TOS; jmp=0; sp_next=sp-1; end //RETURNI
  default: begin pc_next=pcp1; jmp=0; sp_next=sp; end
 endcase
/************************  ALU ****************************/
assign ALU_AND=AI1&AI2; //(8)
assign cin=INSTRUCTION[13]?CARRY:1'b0;
assign ALU_SR={sr_in,AI1[7:1]};
assign ALU_SL={AI1[6:0],sr_in};

always @(*)
 if(INSTRUCTION[14]) ALU_ADD_SUB=AI1-AI2-cin;
 else ALU_ADD_SUB=AI1+AI2+cin;

always @(*)
 case(INSTRUCTION[2:0])
```

List. 1. c.d.

```
  3'b110: sr_in=1'b0;
  3'b111: sr_in=1'b1;
  3'b100: sr_in=AI1[0];
  3'b010: sr_in=AI1[7];
  3'b000: sr_in=CARRY;
  default: sr_in=CARRY;
 endcase

always @(*)
 casex({INSTRUCTION[17:13],INSTRUCTION[3]})
  9'b00000x: {ar,ac,az,cy,z,AO}={5'b10000,AI2}; // LOAD
  9'b00101x: begin {ar,ac,az}={3'b111}; cy=1'b0; z=z0; AO=ALU_AND; end // AND
  9'b00110x: begin {ar,ac,az}={3'b111}; cy=1'b0; z=z0; AO=AI1|AI2; end // OR
  9'b00111x: begin {ar,ac,az}={3'b111}; cy=1'b0; z=z0; AO=AI1^AI2; end // XOR
  9'b01001x: begin {ar,ac,az}={3'b011}; cy=parity; z=z0; AO=ALU_AND; end // TEST
  9'b011xxx: begin {ar,ac,az}={3'b111}; {cy,AO}=ALU_ADD_SUB; z=z0; end // SUB, SUBCY
  9'b01010x: begin {ar,ac,az}={3'b011}; {cy,AO}=ALU_ADD_SUB; z=z0; end // COMPARE
  9'b100001: begin {ar,ac,az}={3'b111}; cy=AI1[0]; AO=ALU_SR;
                   z=((INSTRUCTION[0])?1'b0:z0); end // SR0,SR1
  9'b100000: begin {ar,ac,az}={3'b111}; cy=AI1[7]; AO=ALU_SL;
                   z=((INSTRUCTION[0])?1'b0:z0); end // SL0,SR1
  default: begin {ar,ac,az}={3'b000}; cy=1'b0; z=1'b0; AO=8'd0; end
 endcase
/*****************************************************************/
always @(posedge CLK) //(9)
 if(rd_strobe&~int_req&~READ_STROBE) READ_STROBE<=1'b1;
  else READ_STROBE<=1'b0;

always @(posedge CLK) //(10)
 if(wr_strobe&~int_req&~WRITE_STROBE) WRITE_STROBE<=1'b1;
  else WRITE_STROBE<=1'b0;

always @(posedge CLK) //(11)
begin
 if(RESET) begin
  PC<=10'h3ff; sp<=5'd0;  INT_ENABLE<=1'b0; INTERRUPT_ACK<=1'b0;
 end else
 begin
   if(INTERRUPT_ACK) INTERRUPT_ACK<=1'b0;
   if(~skip)
   begin
    PC<=pc_next; sp<=sp_next;
    if(ac&~int_req) CARRY<=cy;
    if(az&~int_req) ZERO<=z;
   if(int_req)
   begin
    PR_CARRY<=CARRY; PR_ZERO<=ZERO;
    INT_ENABLE<=1'b0; INTERRUPT_ACK<=1'b1;
   end else
   begin
    if(reti)
    begin
     CARRY<=PR_CARRY; ZERO<=PR_ZERO;
     if(INSTRUCTION[0]) INT_ENABLE<=1'b1;
     else INT_ENABLE<=1'b0;
    end else
    if(INSTRUCTION[17:13]==5'b11110) //ENABLE, DISABLE INT
    begin
     if(INSTRUCTION[0]) INT_ENABLE<=1'b1;
     else INT_ENABLE<=1'b0;
    end
   end
  end
  end
end
endmodule
--
```

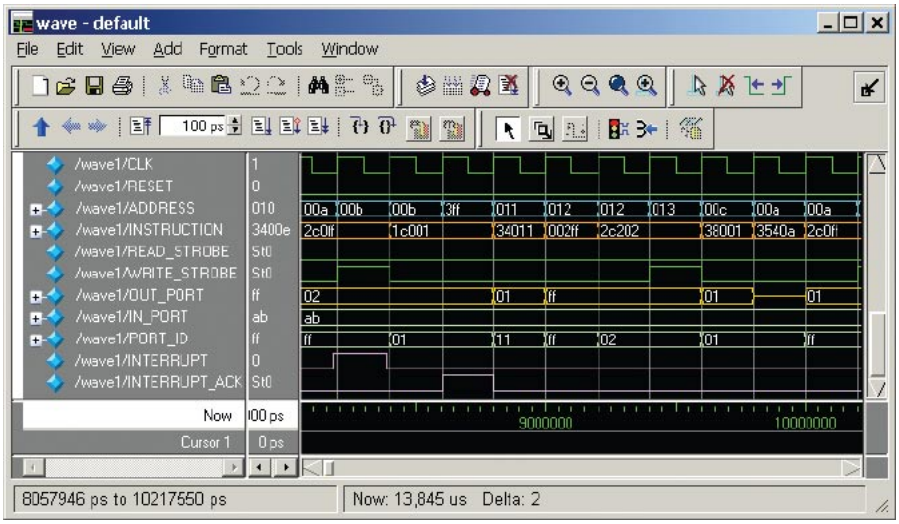
będzie również poprawne. Dodatkową cechą pBlazeZH jest to, że sygnał żądania obsługi przerwania na wejściu INTERRUPT może trwać zaledwie jeden takt zegara. Również czas reakcji na przerwanie jest krótszy niż dla PicoBlaze. W najbardziej niekorzystnym przypadku, gdy żądanie obsługi przerwania wystąpi tuż przed realizacją instrukcji wejścia – wyjścia, czas ten może wynieść 4 takty zegara. Dla wszystkich innych instrukcji przerwanie przyjmowane jest w ciągu 2 taktów zegara.

Ogólny schemat architektury mikrokontrolera pBlazeZH jest zbliżony do PicoBlaze, zaś lista realizowanych rozkazów dokładnie identyczna z pierwowzorem. Bliższe informacje dotyczące m. in. listy rozkazów można znaleźć w oryginalnej dokumentacji do mikrokontrolera PicoBlaze jak również we wspomnianych na początku publikacjach w miesięczniku Elektronika Praktyczna.

Opis behawioralny mikrokontrolera pBlazeZH

Na **list. 1** przedstawiono kod z opisem w języku Verilog mikrokontrolera pBlazeZH. Ogólnie można zauważyć, że pomimo dość długiej listy realizowanych rozkazów kod opisujący mikrokontroler jest względnie krótki i dość zwarty. Jest też znacznie krótszy od strukturalnego opisu oryginalnego mikrokontrolera PicoBlaze.

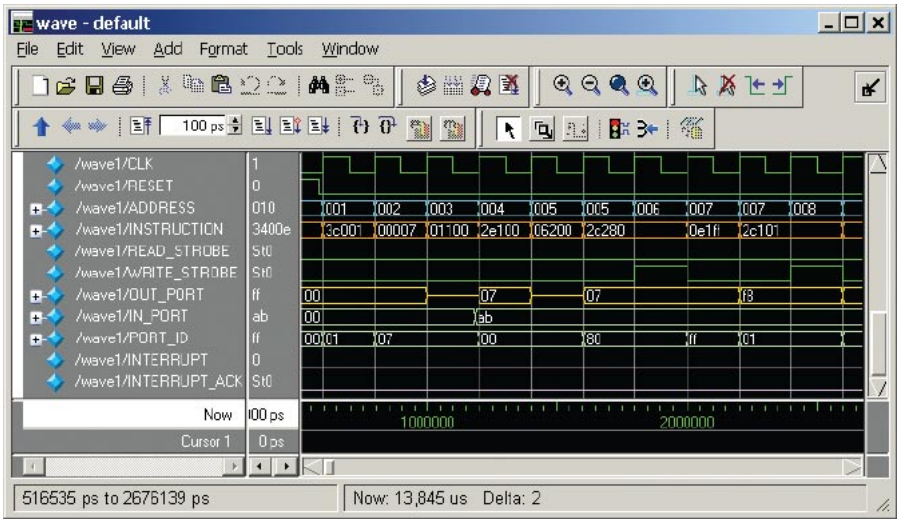
Początkowe linie opisu mikrokontrolera zawierają szereg deklaracji sygnałów i zmiennych używanych w dalszej części opisu. W linii (1) znajduje się opis procesu sekwencyjnego, który odpowiada za warunkową synchronizację zewnętrznego sygnału zgłoszenia przerwania z sygnałem zegarowym. W linii (2) kolejny proces sekwencyjny realizuje obsługę stosu a dokładnie zapis na stos bieżącej wartości licznika rozkazów zwiększonej o 1 (sygnał *pcp1*). Linia (3) modeluje multiplekser, który ustala źródło dla przypisań do rejestrów ogólnego przeznaczenia. Może to być albo port wejściowy (*PORT_IN*) albo wartość odczytana z pamięci podręcznej RAM (*DO_SP*) lub też wyjście jednostki arytmetyczno – logicznej ALU. Linia (4) modeluje zapis wartości wybranej przez multiplekser opisany w linii (3) do banku rejestrów ogólnego przeznaczenia (dokładnie do rejestru identyfikowanego przez wartość sygnału *sX*). Zapis odbywa się wówczas, gdy spełnione są pewne warunki określone wyrażeniem instrukcji warunkowej. Kolejna linia (5) opisuje zapis do pamięci RAM, wtedy gdy bieżącym rozkazem jest rozkaz *STORE*. W linii (6) zawarte są trzy instrukcje, które odpowiadają za asynchroniczny odczyt danych z banku rejestrów ogólnego przeznaczenia oraz pamięci podręcznej RAM. W linii (7) opisano dość długi proces kombinacyjny, który odpowiada za



Rys. 2. Fragment symulacji behawioralnej mikrokontrolera pBlazeZH

wyznaczenie następnych (w kolejnym takcie zegara) wartości licznika rozkazów (*pc_next*) oraz wskaźnika stosu (*sp_next*) a także identyfikuje czy aktualnie wykonywany rozkaz jest rozkazem wymagającym zapisu bieżącej wartości licznika rozkazów na szczyt stosu (sygnał *jmp*). Opis rozpoczynający się w linii (8) modeluje jednostkę arytmetyczno – logiczną. Jest to zasadniczy blok funkcjonalny mikrokontrolera. Wewnątrz ostatniego z procesów kombinacyjnych jednostki ALU, oprócz zmiennych identyfikujących wyjście ALU (*AO*) oraz stan znaczników CARRY (*cy*) i ZERO (*z*) zdefiniowane są też trzy zmienne: *ar*, *ac* oraz *az*. Zależnie od wykonywanego aktualnie rozkazu przechowują one informację o tym czy kolejno: należy wykonać zapis do wybranego rejestru ogólnego przeznaczenia oraz zmodyfikować znaczniki (flagi) CARRY oraz ZERO. W liniach (9) i (10) opisane są dwa procesy sekwencyjne, które odpowiadają za wygenerowanie sygnałów strobujących podczas obsługi rozkazów wejścia – wyjścia: READ_STROBE oraz WRITE_STROBE. W ostatnim procesie sekwencyjnym opisanym w linii (11) odbywa się aktualizacja wartości licznika rozkazów (*PC*), wskaźnika stosu (*sp*) a także znaczników CARRY i ZERO – wszystko na podstawie wartości sygnałów wypracowanych przez procesy kombinacyjne jednostki ALU oraz proces opisany w linii (7). Proces sekwencyjny opisany w linii (11) odpowiada również za przyjęcie przerwania (przechowanie stanu znaczników CARRY i ZERO oraz zablokowanie przerwań) i wygenerowanie sygnału potwierdzenia INTERRUPT_ACK a także za obsługę wewnętrznego sygnału sterującego blokadą przerwań (*INT_ENABLE*).

Zwróćmy uwagę, że wewnątrz procesu sekwencyjnego w linii (11) zasadnicze czynności, również z punktu widzenia funkcjonowania całego mikrokontrolera (np. aktualizacja licznika rozkazów), podejmowane są tylko wtedy, gdy sygnał *skip* przyjmuje wartość 0. Z kolei sygnał *skip* jest ustawiany (w części deklaracyjnej na początku modułu) wówczas gdy realizowanym rozkazem jest rozkaz obsługi wejścia – wyjścia. Aktywny sygnał *skip* wstrzymuje więc normalną pracę mikrokontrolera na jeden takt zegara. Dzięki temu rozkazy skoków realizowane są tak jak w PicoBlze w ciągu dwóch taktów zegara. Ma to pewne zalety z punktu widzenia wydajności całego systemu (mikrokontrolera PicoBlaze/pBlazeZH wraz z dołączonymi układami wejścia – wyjścia) a konkretnie maksymalnej częstotliwości zegara taktującego, zwłaszcza gdy system zawiera większą liczbę układów wejścia – wyjścia.



Rys. 3. Fragment symulacji behawioralnej mikrokontrolera pBlazeZH – obsługa przerwań

Symulacje

Na **rys. 2** i **rys. 3** przedstawiono przykładowe fragmenty symulacji behawioralnej działania mikrokontrolera pBlazeZH. W czasie symulacji mikrokontroler realizował bardzo prosty program, którego kod pokazany jest na **list. 2**. Lewa skrajna kolumna kodu, zawierająca liczby heksadecymalne, reprezentuje adres. Kolejna kolumna to 5-cio cyfrowy (18 bitów) kod rozkazu. W pozostałych kolumnach zawarte są etykiety i mnemoniki asemblera.

Rys. 2 przedstawia fragment symulacji działania mikrokontrolera tuż po wycofaniu sygnału RESET. Mikrokontroler realizuje wówczas sekwencję instrukcji wymiany pewnej wartości stałej (07h) pomiędzy rejestrami roboczymi oraz pamięcią RAM. Wartość ta zostaje ostatecznie wysłana do portu wyjściowego o adresie 80h, potwierdzając tym samym prawidłowe funkcjonowanie mikrokontrolera w zakresie obsługi rejestrów roboczych, pamięci RAM oraz operacji na portach wyjściowych.

Przebiegi czasowe na rys. 2 pokazują również, że poszczególne instrukcje z wyjątkiem rozkazów obsługi portów wejścia – wyjścia realizowane są w ciągu jednego taktu zegara. Podczas pracy mikrokontrolera na portach OUT_PORT oraz PORT_ID zmieniają się wartości danych , zależnie od realizowanego rozkazu. Jednak porty te przyjmują wartość stabilną już w pierwszym takcie zegara gdy wykonywanym rozkazem jest rozkaz obsługi wejścia – wyjścia.

Rys. 3 ilustruje sytuację w której zgłaszane jest żądanie obsługi przerwania. Sygnał przerwania na wejściu INTERRUPT pojawia się w drugim takcie zegara podczas realizacji operacji zapisu danych do portu wyjściowego (w czasie aktywnego sygnału WRITE_STROBE). Przerwanie zostaje przyjęte dwa takty zegara później, o czym informuje aktywny stan wyjścia INTERRUPT_ACK oraz obecność na szynie adresowej ADDRESS wektora przerwania 3FFh. Podprogram obsługi przerwania realizuje tylko czynność wysłania do portu wyjściowego o adresie 02h wartości FFh i powrót z przerwania wraz z odblokowaniem możliwości przyjmowania kolejnych przerwań (instrukcja RETURNI ENABLE). Przerwanie zostało przyjęte po tym jak mikrokontroler wykonał rozkaz pobrany spod adresu 00Bh. Jak widać na rys. 3, powrót z przerwania następuje do prawidłowego adresu 00Ch. Pod tym adresem znajduje się

List. 2. Kod programu zapisanego w asemblerze KCPSM3, użytego do symulacji

```
 000                 ADDRESS 000
 000  3C001          ENABLE INTERRUPT
 001  00007          LOAD s0, 07
 002  01100          LOAD s1, s0
 003  2E100          STORE s1, 00
 004  06200          FETCH s2, 00
 005  2C280          OUTPUT s2, 80
 006  0E1FF          XOR s1, FF
 007  2C101          OUTPUT s1, 01
 008  3000A          CALL LOOP[00A]
 009  3400E          JUMP END[00E]
 00A           LOOP:
 00A  2C0FF          OUTPUT s0, FF
 00B  1C001          SUB s0, 01
 00C  3540A          JUMP NZ,
LOOP[00A]
 00D  2A000          RETURN
 00E            END:
 00E  000FF          LOAD s0, FF
 00F  2C0FF          OUTPUT s0, FF
 010  3400E          JUMP END[00E]
 011            INT:
 011  002FF          LOAD s2, FF
 012  2C202          OUTPUT s2, 02
 013  38001          RETURNI ENABLE
 3FF                 ADDRESS 3FF
 3FF  34011          JUMP INT[011]
--
```

instrukcja skoku pod adres 00Ah. Również realizacja tej instrukcji trwa dokładnie jeden takt zegara.

W kontekście przerwań warto tu zwrócić uwagę, że co prawda sygnał potwierdzenia przyjęcia przerwania pojawia się już dwa takty później w stosunku do sygnału zgłoszenia przerwania (sytuacja zilustrowana na rys. 3), jednak faktycznie podprogram obsługi przerwania realizowany jest dopiero w kolejnym takcie. W tymże kolejnym takcie, po wycofaniu sygnału INTERRUPT_ACK, wykonywana jest instrukcja zawarta pod adresem 3FFh (wektor przerwania). Dlatego też rzeczywisty czas przyjęcia przerwania liczony od momentu zgłoszenia przerwania wynosi, w przypadku pokazanym na rys. 3, trzy takty zegara.

Z punktu widzenia czasu reakcji mikrokontrolera na przerwanie najbardziej niekorzystna sytuacja miałaby miejsce, gdy sygnał przerwania pojawiłby się takt zegara wcześniej niż w przypadku pokazanym na rys. 3. Czyli w pierwszym takcie podczas wykonywania operacji wejścia – wyjścia. Wówczas czas ten wyniósłby 4 takty zegara. W każdym innym przypadku mikrokontroler reaguje na przerwanie już w ciągu dwóch taktów zegara. Warto również zwrócić uwagę, że sam sygnał przerwania może trwać bardzo krótko – nawet krócej niż jeden takt zegara jeżeli spełnione są warunki związane z czasem ustalania (*setup time*) i przetrzymywania danych (*hold time*) przerzutnika synchronizującego sygnał zgłoszenia przerwania.

Podsumowanie

Dzięki zastosowaniu opisu behawioralnego mikrokontroler PBlazeZH w bardzo łatwy sposób może być poddany pewnym modyfikacjom zwiększającym jego funkcjonalność. W prosty sposób można na przykład zwiększyć ilość obsługiwanych przerwań i spowodować by każdemu z dodatkowych sygnałów zgłoszenia przerwania odpowiadał inny wektor przerwania – skok do innej lokalizacji.

Według raportu syntezy uzyskanego przy wykorzystaniu oprogramowania *Xilinx* WebPack ISE 10.1i, mikrokontroler pBlazeZH zaimplementowany w układach XC3S200 wymaga 166 bloków logicznych typu *slice*. Dla porównania oryginalny PicoBlaze zaimplementowany w tych samych układach zajmuje tylko 96 bloków logicznych. Jednak zaletą pPlazeZH jest to, że jest on dwukrotnie szybszy od PicoBlaze oraz może być wykorzystany do implementacji w układach programowalnych dowolnego producenta, a nie tylko układach *Xilinx*.

Maksymalna częstotliwość taktowania pBlazeZH zaimplementowanego w układach *Spartan 3* o klasie szybkości –4 wynosi, jak podaje raport syntezy, nieco ponad 90 MHz. Biorąc pod uwagę, ze pBlazeZH niemal wszystkie rozkazy wykonuje w ciągu jednego taktu zegara, oznacza to, że mikrokontroler ten dysponuje dość sporą mocą obliczeniową sięgającą blisko 90 MIPS (milionów instrukcji na sekundę).

Mikrokontroler pBlazeZH, oprócz szczegółowych badań symulacyjnych, testowany był również w wielu aplikacjach praktycznych. Przeprowadzone testy potwierdzają jego poprawne działanie oraz programową i sprzętową zgodność z PicoBlaze.

Zbigniew Hajduk
zhajduk@prz-rzeszow.pl