NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (3)

Przerwania, timery i obsługa wyświetlaczy

Do tej pory wspólnymi siłami udało nam się wbudować w pełni funkcjonalny system mikroprocesorowy, który odbierał i generował cyfrowe sygnały (powiedzmy dumnie, że przetwarzał sygnały cyfrowe!). Kilkakrotnie wspominałem przy tej okazji, że stosowanie opóźnień jest rozwiązaniem nagannym, jednak dotychczas nie mieliśmy alternatywy – czas ją poznać i wzbogacić system oraz swoją wiedzę o timery i system przerwań.

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Abyśmy mogli przystąpić do nauki, musimy mieć solidną bazę, czyli projekt zawierający porty *GPIO* (*PIO*) do sterowania wyświetlaczami 7-segmentowymi oraz mieć informacje na temat metody ich sterowania. Aby zbędnie nie przedłużać tej części (i ufając, że każdy opracował swoje zadanie), przedstawię swoją wersję wymaganego projektu i bardzo krótko ją omówię.

Dodano porty wyjściowe o szerokości odpowiednio 4 i 8 bitów – do sterowania włączaniem odpowiednio kolejnych wyświetlaczy i segmentów (7 oraz kropka). Port sterujący wyświetlaczami został wyposażony w funkcje ustawiania/kasowania pojedynczych bitów.

W programie dodano folder *7SEG* na pliki z funkcjami obsługującymi sterowanie wyświetlaczami. Klikamy PPM na nazwę projektu, potem *New* → *Folder* (**rysunek 1**).

1. Dodano pliki 7SEG.c i 7SEG.h w utworzonym folderze. Podobnie jak wcześniej, klikamy PPM na nazwę projektu i potem *New* → *File*.

2. Dodano folder *7SEG* do ścieżki wyszukiwania plików *Include*:
   a. otwieramy *Project → Properties*,
   b. wybieramy *Nios II Application Properties → Nios II Application Paths*,
   c. obok okna *Application include directories* klikamy na *Add…*,
   d. w nowym oknie lokalizujemy nasz folder. Wszędzie klikamy *OK/Yes* (**rysunek 2**).
5. Zdefiniowano kombinacje segmentów do zapalenia dla każdej cyfry z systemu heksadecymalnego oraz znaku „–”, funkcję ustawiającą odpowiednią kombinację na wyprowadzeniach połączonych z segmentam oraz funkcję wyłączającą wszystkie wyświetlacze lub włączającą jeden z nich.
6. Napisano program multipleksujący wyświetlacze (**listing 1**). Na jego przykładzie powtórzymy też w skrócie ideę multipleksowania wyświetlaczy:
   a. wyłączamy wszystkie wyświetlacze,
   b. ustawiamy kombinację segmentów dla kolejnego wyświetlacza,
   c. włączamy ten wyświetlacz,
   d. czekamy trochę czasu (tu zaraz pozbędziemy się instrukcji opóźniającej),
   e. powtarzamy dla kolejnego wyświetlacza.

UWAGA. Często popełnianym błędem jest zapominanie o instrukcji wyłączającej wszystkie wyświetlacze, przed zmianą kombinacji segmentów – powoduje to powstanie bardzo nieestetycznych duchów na wyświetlaczach, jeśli chcecie, możecie to sprawdzić, usuwając tę instrukcję.

Listing 1. Program multipleksujący wyświetlacz

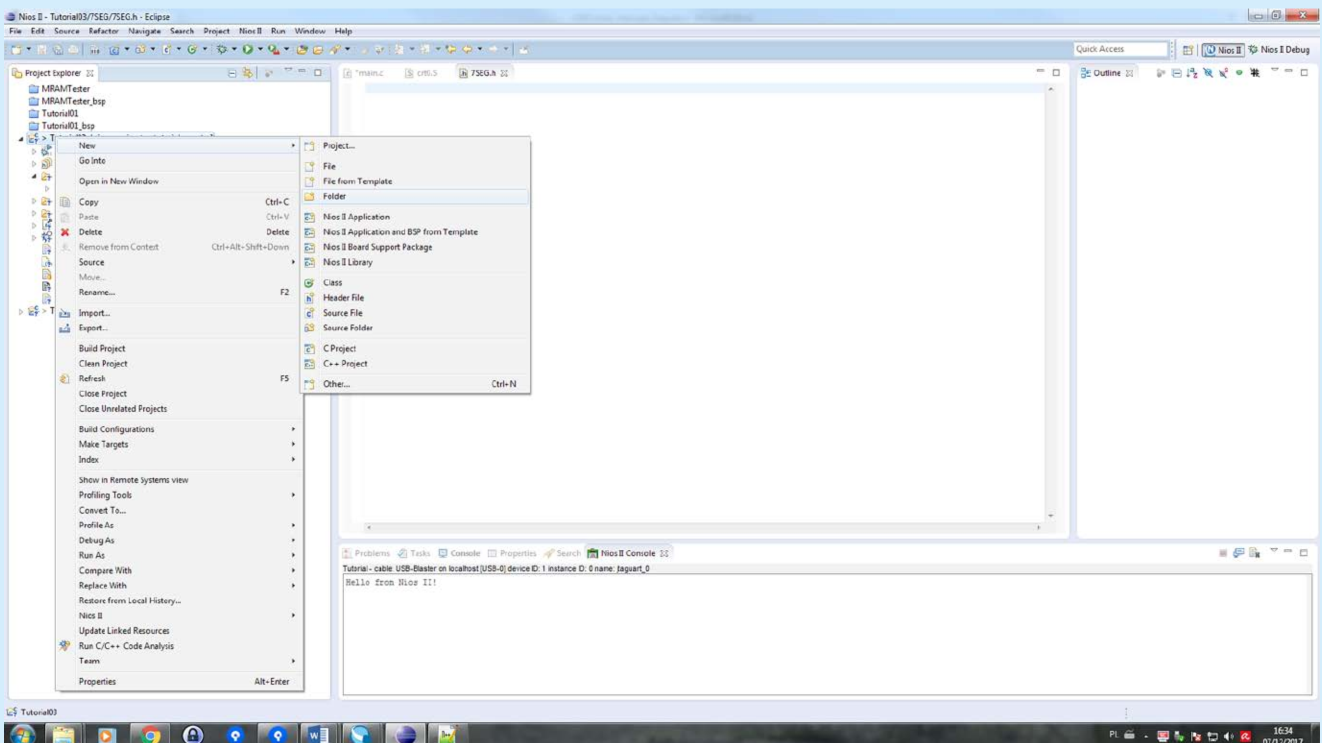
```
while (1) {
    for (int i=0 ; i<4 ; i++){
        setDisplay(DSOFF);
        setDigit(i,0);
        setDisplay(i);
        ALT_USLEEP(1000);
    }
}
```

Kody źródłowe zadania domowego w mojej interpretacji umieszczone zostały w pliku *Tutorial03_start.zip* i do nich będę się w dalszym ciągu naszego spotkania odwoływał, jeśli jednak przygotowaliście swoje wersje, to jeśli działają one poprawnie, zachęcam do kontynuowania prac nad nimi. Tak czy inaczej zachęcam do zapoznania się z moją propozycją i porównania swoich autorskich programów.

Timer i przerwania

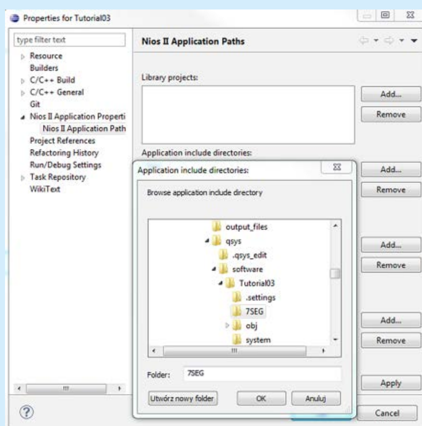
W wyżej przedstawionym programie znalazła się instrukcja, o której stosowaniu (a raczej prawie zakazie jej stosowania) już wspominałem. Chodzi o instrukcję generującą opóźnienie. Ta, której używaliśmy dotychczas, bazuje na fakcie wykonywania „pustych” instrukcji na rdzeniu procesora (instrukcje *NOP*). Takie rozwiązanie ma dwie wady. Pierwszą z nich jest to, że dokładne wygenerowanie opóźnienia jest dosyć kłopotliwe. Z jednej strony umieszczenie odpowiedniej liczby operacji *NOP* da dokładne opóźnienie, ale kosztem dużej zajętości pamięci (np. 1 ms przy zegarze 50 MHz to 50 tysięcy operacji, zakładając, że jedna operacja wykonuje się w jednym cyklu zegarowym). Z drugiej strony, zwykle instrukcje niskiego poziomu używane przy tworzeniu pętli nie mają jednoznacznie określonego czasu wykonywania (np. instrukcja porównania z zerem na pewnych procesorach może wykonać się w 1 lub 2 taktach, w zależności od tego, czy warunek jest spełniony, czy też nie). Druga poważna wada to fakt, że w czasie odmierzania opóźnienia procesor nie może robić nic innego i po prostu marnuje czas. Wyobraźmy sobie, że nasz system ma nie tylko wyświetlać dane na wyświetlaczu, ale jednocześnie np. zliczać impulsy, obsługiwać prostą klawiaturę i jakieś menu użytkownika… Już niejeden raz widziałem „arcydzieła” mogące konkurować z obrazami mistrza Picasso zawierające setki instrukcji warunkowych poprzeplatanych z maleńkimi kwantami opóźnień…

My jednak pozostaniemy z daleka od takich artystycznych rozwiązań i postaramy się wykonać zadanie możliwie zgodnie ze sztuką. Zatem zacznijmy od przedstawienia głównego bohatera

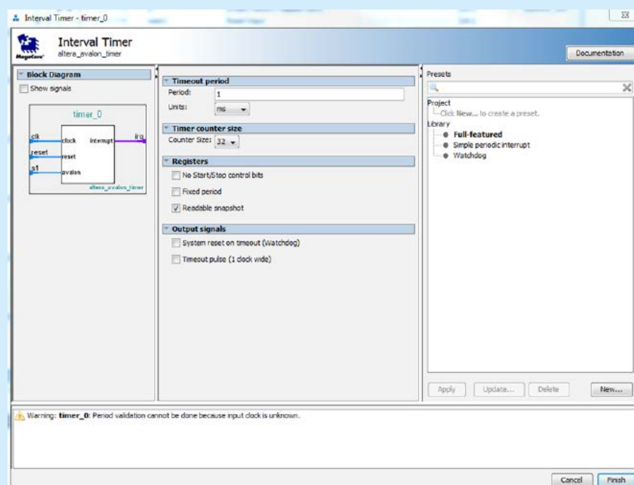


Rysunek 1. Dodawanie nowego folderu do projektu

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl



Rysunek 2. Dodajemy nowo utworzony folder jako Include Directory



Rysunek 3. Dodawanie komponentu Interval Timer i jego możliwe ustawienia

naszego spotkania, czyli *timera* (licznika). Jak sama nazwa wskazuje, układ ten liczy impulsy, w naszym przypadku będą to impulsy sygnału zegarowego. Otwórzmy teraz nasz projekt *Qsys* i wyszukajmy *Interval timer*, a następnie rozpocznijmy proces dodawania go do naszego systemu (**rysunek 3**).

TIMER Licznik dostarczony nam przez producenta ma możliwość zliczania tylko w dół, czyli jego wartość zmniejsza się, aż do osiągnięcia zera. W tym momencie nasz *timer* może wygenerować sygnał dla procesora (o czym powiemy za chwilę) i zatrzymać się lub przyjąć określoną wartość i znów rozpocząć zliczanie do zera, w zależności od ustawienia.

Omówmy teraz w skrócie parametry, jakie możemy ustawić z poziomu *Qsys*:

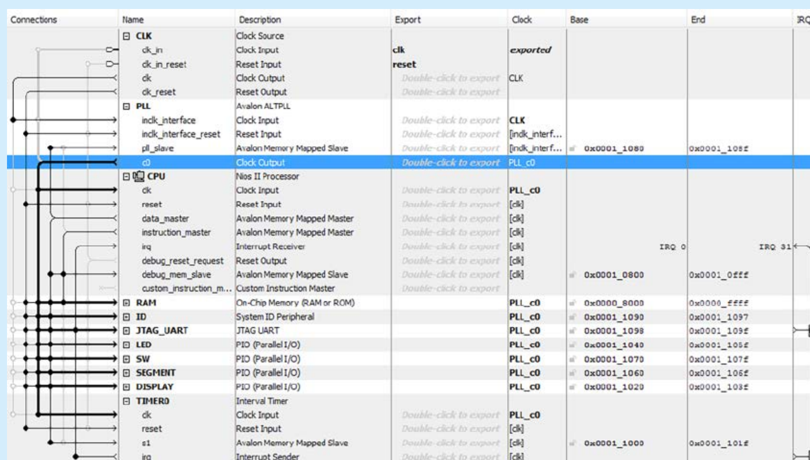
- *Timeout Period: Period, Units* – definiujemy tu wartość, jaka ma być ładowania do *timera* po zakończeniu odliczania, oraz jednostki, w jakich jest ona wyrażona (jeśli wszystko wykonujemy poprawnie, *Qsys* wie, jaka jest częstotliwość zegara i jest w stanie przeliczyć czas na liczbę cykli zegara – to bardzo wygodne). W skrócie definiujemy tu okres, z jakim licznik będzie nas powiadamiał o zakończeniu odmierzania czasu.
- *Counter Size* – tu definiowana jest liczba bitów, jaką miał będzie nasz licznik. Jak nietrudno zauważyć, zależy od niej maksymalny czas, jaki możemy odmierzyć. Oczywiście w danej liczbie bitów musi zmieścić się liczba cykli zegara potrzebna do odmierzenia danego czasu, a nie tenże czas podany np. w sekundach.
- *No Start/Stop control bits* – opcja ta pozwala na pozbawienie naszego licznika możliwości uruchamiania go i zatrzymywania z poziomu programu (napisanego na procesor NIOS II).
- *Fixed period* – to ustawienie pozwala z kolei na pozbawienie licznika możliwości zmiany jego okresu (patrz punkt 1) z poziomu programu.
- *Readable snapshot* – zaznaczenie tej opcji pozwala na odczyt stanu licznika (aktualnej wartości) w dowolnym momencie z poziomu programu.
- *System reset on timeout (Watchdog)* – dodaje do naszego timera wyjście sygnału resetującego, którym możemy zresetować cały system po zliczeniu do 0. Dzięki temu możemy stworzyć, jak sama nazwa wskazuje, układ *Watchdog*, czyli nadzorujący pracę układu i mogący zrestartować go w wypadku błędu (normalnie program powinien okresowo zerować układ *Watchdog*, jeśli do takiego resetu nie dojdzie to wtedy układ resetuje cały system).
- *Timeout pulse* – dodaje dodatkowe wyjście z timera, które generuje impuls po zliczeniu do 0. Możemy ten sygnał wykorzystać w dowolny sposób.

Na razie zostawmy domyślne ustawienia, dzięki czemu dostępne będziemy mieli prawie wszystkie (poza dwoma ostatnimi) opcje timera. Chciałbym tu jednak zwrócić uwagę na ważną rzecz – mianowicie optymalizację. Na etapie budowania systemu powinniśmy znać już założenia i wymagania stawiane przez oprogramowanie. I tak na przykład do prostego multipleksowania wyświetlaczy powinniśmy usunąć opcje zatrzymywania i uruchamiania timera, zmiany jego okresu czy odczytu jego wartości. Dzięki temu zaoszczędzimy cenne elementy logiczne w układzie.

Teraz doprowadzimy sygnał zegarowy, tak jak poprzednio (wyjście z pętli PLL), sygnał zerowania (*reset*), magistralę *Avalon* do *Data Master* naszego CPU oraz tworzymy połączenie od *irq*. Wybierzmy jeszcze, mimo braku błędów, ale dla nabrania dobrych nawyków *System* → *Assign Base Addresses*. Warto zmienić nazwę nowo dodanego komponentu na np. *TIMER0*.

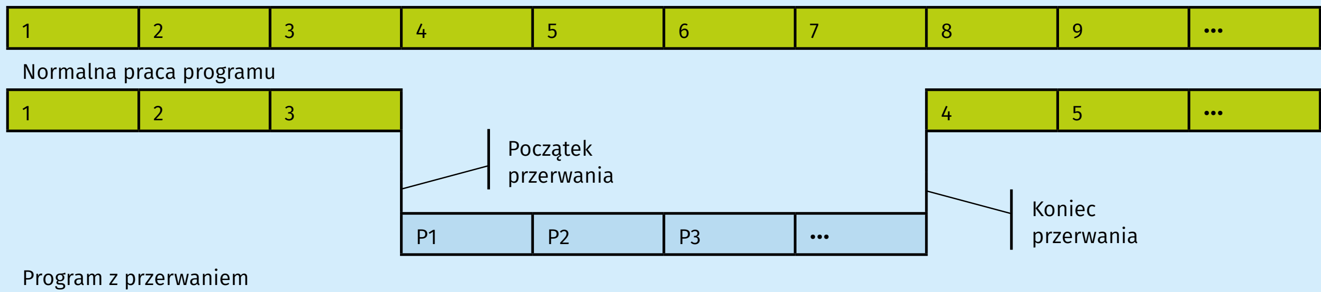
System (po zwinięciu mniej istotnych teraz elementów) powinien wyglądać jak na **rysunku 4**.

PRZERWANIA Właśnie podłączyliśmy z naszego timera sygnał przerwania, jestem jednak w tym momencie winny kilka wyjaśnień, czym tak naprawdę są przerwania i jakie mamy możliwości ich konfiguracji w *Qsys*.



Rysunek 4. Widok systemu po dodaniu i podłączeniu timera

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl



Rysunek 5. Praca programu: normalna (góra) i z przerwaniem (dół). Zielonym kolorem oznaczono instrukcje głównego programu, a niebieskim – programu przerwania

Normalnie program w naszym procesorze jest wykonywany krok po kroku, a instrukcje są realizowane po kolei (za wyjątkiem skoków związanych np. z pętlami czy instrukcjami warunkowymi). Czasem jednak jest wymagane obsłużenie nagłego zdarzenia (np. wciśnięcie przycisku bezpieczeństwa, odebranie danych przesłanych jakimś interfejsem). W tym właśnie celu używa się przerwań, które w określonej sytuacji powodują przerwanie (jak sama nazwa wskazuje) wykonywania głównego programu i wykonanie innych, „priorytetowych" zadań. Po tym wszystkim zwykle następuje powrót do wykonywania głównego programu od miejsca, w którym został on przerwany. Przedstawia to schematycznie **rysunek 5**.

A w naszym wypadku? Co da nam przerwanie? Odpowiedź na te pytania powoli nasuwa się sama. Jeśli procedura odświeżania naszego wyświetlacza ma być wykonywana co 1 ms, wstępnie ustawiliśmy timer na odmierzanie czasu właśnie 1 ms, to wystarczy umieścić instrukcje odpowiedzialne za multipleksowanie w przerwaniu i… W zasadzie zapominamy o tym, że musimy multipleksować wyświetlacz! A to dopiero początek możliwości, jakie otwiera przed nami użycie timerów i przerwań.

KONFIGUROWANIE PRZERWAŃ Przerwania musimy przede wszystkim skonfigurować w *Qsys*. I tak, patrząc na rysunek 4, widzimy kolumnę *IRQ*, w której są widoczne połączenia linii przerwań, które są zdublowane w kolumnie *Connections* wraz z zaznaczonymi numerami przerwań (0 dla *JTAG_UART* oraz 1 dla nowo dodanego *TIMER0*). W rdzeniu *Nios II Economy* mamy możliwość dodania 32 źródeł przerwania, z których każde musi mieć unikalny numer. W wypadku konfliktów możemy ręcznie zmienić numery przerwań lub posłużyć się opcją *System* à *Assign Interrupt Numbers*. Jeśli nasz system nie ma żadnych błędów, co sprawdzamy w panelu *Messages*, możemy kliknąć *Finish* i w dalszych krokach odpowiedzieć twierdząco na pytanie o ponowne wygenerowanie systemu. Następnie przeprowadzamy kompilowanie projektu.

Wracamy do programowania

Teraz możemy spokojnie uruchomić środowisko *Eclipse*, aby zacząć modyfikowanie naszego oprogramowania. W pierwszej kolejności oczywiście generujemy nowe BSP (dokładnie jak poprzednio). I zabieramy się do dzieła. Jednak pewnie zaraz padnie pytanie – gdzie najlepiej szukać informacji na temat obsługi poszczególnych komponentów? Odpowiem tak, jak zwykle w tej chwili odpowiadam – w dokumentacji dostarczonej przez producenta, czyli Intel FPGA (dawniej Altera). Padnie tu też zachęta do nauki języka angielskiego, którą możemy czasem „wspomóc" (oczywiście z rozwagą i pewną dozą nieufności) za pomocą dostępnych online narzędzi tłumaczących. Dlaczego tak? Po pierwsze warto jak najwcześniej przyzwyczajać się do czytania tego typu dokumentacji, po drugie znajdziemy tam zawsze informacje z pierwszej ręki i aktualne, po trzecie omówienie wszystkich komponentów systemu wraz z przykładami i przetłumaczenie

Documentation: Nios® II Processor

These handbooks serve as the primary documentation for the Nios® II processor:

- The Nios® II (Gen2) Processor Reference Guide (ver 2016.10.28) answers the question "What is the Nios II processor?" and is the primary reference for the Nios II processor architecture
- The Nios® II (Gen2) Software Developer's Handbook (ver 2017.05.08) answers the question "How do I write programs for the Nios II processor?" and is the primary reference for programming the Nios II processor
- The Embedded Design Handbook (ver 2017.11.06) describes how to most effectively use the Nios II Embedded Design Suite (EDS) tools and helps to increase the efficiency of developing, debugging, and optimizing Nios II processor-based embedded systems
- The Embedded Peripherals IP User Guide (ver 2017.11.06) describes our peripherals that work seamlessly with the Nios II processor and are included with the Intel Quartus® Prime software
- The Nios II Floating Point Hardware 2 Component User Guide (ver 2016.05.03) describes how to use the floating-point hardware 2 (FPH2) component with the Nios II processor. The FPH2 component is provided by Intel and included with the Intel Quartus Prime software.

Rysunek 6. Zrzut strony ze spisem dokumentacji dla systemu Nios II

Listing 2. Funkcja odpowiedzialna za pojedynczy cykl wyświetlania

```
void refreshDisplay(void){
    static uint8_t disp = 0;

    setDisplay(DSOFF);
    setDigit(disp, 0);
    setDisplay(disp);
    if(disp < 3) disp++;
    else disp = 0;
}
```

Listing 3. Funkcja obsługi przerwania

```
void timer0Interrupt(void* context){
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER0_BASE, 0);
    refreshDisplay();
}
```

całej dokumentacji na język polski byłoby zadaniem karkołomnym. Co można też skwitować powiedzeniem, że „lepiej nauczyć kogoś łowić ryby i dać mu wędkę, niż dać mu kilka ryb".

W naszym konkretnym przypadku informacji szukać należy (na chwilę pisania tego tekstu) na stronie ***https://goo.gl/7M2Jg3*** (Documentation: Nios II Procesor, **rysunek 6**).

W chwili obecnej interesować będą nas najbardziej dokumenty:

- *Software Developer's Handbook*, który zawiera opis wszelkich funkcji wyższego poziomu przygotowanych przez producenta do obsługi rdzenia i innych modułów (tzw. warstwa *HAL – Hardware Abstraction Layer* – warstwa abstrakcji sprzętu, dzięki której nie musimy „znać rejestrów procesora", aby coś zrobić, np. wysłać tekst przez nasz JTAG do konsoli na komputerze). Tu znajdziemy m.in. informacje o funkcjach ułatwiających obsługę i kontrolę przerwań.
- *Embedded Peripherials IP User Guide*, który zawiera opis dostarczonych przez producenta modułów i opcji ich konfiguracji, a także ich rejestrów.

Modyfikujemy nasz program

Na samym początku w pliku *7SEG.c* utwórzmy funkcję, odpowiedzialną za pojedynczy cykl odświeżania wyświetlacza, oraz dodamy odpowiednią deklarację do pliku nagłówkowego (**listing 2**). Funkcja ta to w pewnym sensie „rozwinięta" pętla for z pierwotnego programu. Kluczową rolę odgrywa tutaj słowo

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

static, które powoduje, że zmienna *disp* zachowuje swoją wartość pomiędzy kolejnymi wywołaniami funkcji.

W pliku głównym, przed funkcją *main* definiujemy funkcję, która będzie funkcją obsługi przerwania – pokazano ją na **listingu 3**.

Pierwsza z instrukcji powoduje skasowanie (potwierdzenie) przerwania, w przypadku braku takiego potwierdzenia przerwanie od naszego timera byłoby ciągle aktywne i procesor zawiesiłby się wykonując tylko i wyłącznie instrukcje przypisane do tego przerwania. Informacje o tym znajdujemy oczywiście w… odpowiednim dokumencie, który nawet ostrzej informuje o możliwym nieprzewidywalnym działaniu systemu w takiej sytuacji. Druga instrukcja nie wymaga wyjaśnienia, za to kilku słów wymaga z pewnością sam wygląd funkcji. O ile pierwsze *void* nas nie dziwi, o tyle argument tejże funkcji może zastanawiać. Spieszę z wyjaśnieniem, że ma on taką postać, aby z jednej strony był ogólny, a z drugiej pozwalał na przekazanie do naszej funkcji dowolnych argumentów. W jakim jednak celu? Choćby po to, aby móc tę samą funkcję wykorzystać do obsługi kilku przerwań, które będą powodowały jej wywołanie z różnymi argumentami. Dzięki temu mamy 1 ciało funkcji, ale dzięki argumentowi mogące rozróżnić, skąd zostało „wezwane do działania".

Sama instrukcja (a właściwie makro) zapisu wartości 0 do rejestru statusu (bo w ten właśnie sposób jest zdefiniowane potwierdzenie przerwania) jest bliźniaczo podobna do tych, których używaliśmy w przypadku pracy z portami *PIO*. Wszystkie tego typu makra znajdziemy w pliku altera_avalon_timer_regs.h.

Spójrzmy teraz pokrótce na rejestry dostępne w naszym 32-bitowym timerze (timer 64-bitowy ma dodatkowe rejestry *period* i *snap*). Pokazano je na **rysunku 7**. Wszystkie nieopisane bity mogą przy odczycie przyjmować niezdefiniowane wartości, a zapisywane powinny być zawsze zerami:

Dlaczego? Wyobraźmy sobie zegar, z którego jednocześnie możemy odczytać tylko albo minuty albo godziny, a odczyt prowadzimy co minutę. Zegar wskazuję godzinę 19:59. Odczytujemy minuty i zapisujemy: 59. Kolejny odczyt przeprowadzamy po minucie, zegar wskazuje teraz 20:00, odczytujemy 20 i łączymy odczytane dane: 20:59… Coś poszło nie tak… Jest to tak zwany problem atomowości dostępu do danych czy wykonywania instrukcji. W naszym wypadku odczyt nie był atomowy, czyli nie odbył się w sposób gwarantujący, że odczytane dane nie zostaną zmienione w jego trakcie. Jeśli zaś przed rozpoczęciem odczytu skopiowalibyśmy równocześnie minuty i godziny do dodatkowego bufora, i potem prowadzili odczyt z bufora to dostaniemy „nieuszkodzone" dane z momentu ich kopiowania.

Listing 4. Przykładowa funkcja wyświetlająca dane

```
void setDigit(uint8_t digit, uint8_t dp, uint8_t pos){
    if(pos < 4){
        if(digit > 16){  // dla wartości spoza zakresu gasimy wszystkie segmenty
            displayData[pos] = 0|(dp!=0?(1<<SEG_DP):0);
        }else{  // dla pozostałych wartości wyświetlamy odpowiednią liczbę
            displayData[pos] = digits[digit] | (dp!=0?(1<<SEG_DP):0);
        }
    }
}

void intDisplayDec(uint16_t number){
    for(uint8_t i = 0 ; i < 4 ; i++){
        setDigit(number % 10, 0, i);
        number /= 10;
    }
}
```

- *status* zawiera bity: *RUN* (przyjmuje 1, gdy licznik pracuje) oraz *TO* (przyjmujący 1, gdy sygnalizowane jest przerwanie). Zapis 0 do tego rejestru powoduje skasowanie przerwania.
- *control* zawiera bity *STOP* i *START* (zapisanie do jednego z nich 1 powoduje uruchomienie licznika, zaś do drugiego – jego zatrzymanie, nie wolno zapisywać 1 do obu tych bitów naraz), *CONT* (ustawienie na 1 powoduje, że timer działa cyklicznie, zaś 0 skutkuje jednokrotnym odmierzeniem zadanego czasu, po czym konieczny jest ręczny start timera za pomocą bitu *START*) oraz bit *ITO* (ustawienie go na 1 pozwala na generowanie przerwania).
- *periodh/l* – w 2 połówkach 32-bitowa wartość okresu timera. Zapis do któregokolwiek z tych rejestrów powoduje zatrzymanie timera.
- *snaph/l* – w 2 połowach aktualna wartość, jaką ma licznik. Aby odczytać wartość, należy najpierw dokonać zapisu dowolnej (ignorowanej) wartości do któregokolwiek z rejestrów, a dopiero potem przeprowadzić odczyt (patrz ramka „Dlaczego") z obu rejestrów.

Jakie są w świetle tych informacji nasze dalsze kroki? Po pierwsze musimy „przypisać" zdefiniowaną wcześniej funkcję do przerwania naszego timera. Robimy to w następujący sposób (pełna dokumentacja w *Software Developer's Handbook*):

alt_ic_isr_register(TIMER0_IRQ_INTERRUPT_CONTROLLER_ID, TIMER0_IRQ, timer0Interrupt, NULL, NULL);

Pierwszy argument, który moglibyśmy pominąć, to identyfikator kontrolera przerwań (tylko Nios II Fast ma możliwość podłączania dodatkowych kontrolerów przerwań celem zwiększenia ich liczby ponad 32), jednak dla porządku (a może kiedyś zechcemy przenieść nasz kod na ten lepszy rdzeń?) podajemy tam odpowiednią etykietę, a tych szukamy w pliku *system.h*, jak podczas „zabawy" z *PIO*. Kolejny argument to numer przerwania, do którego się odnosimy (pamiętacie, jak mówiłem o tym w czasie projektowania systemu w *Qsys*?) – i tu znów etykieta zamiast liczby – tak jest bezpieczniej i wygodniej. Wygodniej, bo od razu widzimy co to za przerwania, a bezpieczniej, bo gdy

Name	R/W	Description of Bits						
		15	...	4	3	2	1	0
status	RW	(1)					RUN	TO
control	RW	(1)			STOP	START	CONT	ITO
periodl	RW	Timeout Period – 1 (bits [15:0])						
periodh	RW	Timeout Period – 1 (bits [31:16])						
snapl	RW	Counter Snapshot (bits [15:0])						
snaph	RW	Counter Snapshot (bits [31:16])						

Rysunek 7. Rejestry timera w wersji 32-bitowej. Pełna dokumentacja w *Embedded Peripherials IP User Guide*

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

zmienimy coś w *Qsys* (nawet kliknięcie *Assign Interrupt Numbers* może czasem coś zamieszać) i zmienią się numery przerwań – nie musimy w programie karkołomnie wyszukiwać wszędzie wystąpienia zera czy jedynki (no, chyba że bliżej nam do mistrza Salvadora Dalego niż dobrego programisty). Dalej? No oczywiście – nazwa naszej funkcji, zaraz po niej miejsce na jej argumenty (tak, tak, to, co tu wpiszemy, zostanie przekazane jako argument funkcji), w naszym wypadu nie przekazujemy nic (podajemy pusty wskaźnik, czyli 0, zapisane tu jako *NULL*). Ostatni argument naszej funkcji, zgodnie z zaleceniami producenta, ma mieć także wartość *NULL*.

Co jeszcze pozostało nam zrobić? Jedynie włączyć timer, ustawić go w tryb ciągłej pracy i zezwolić na generowanie przerwania. Realizuje to następujący fragment kodu:

```
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER0_BASE,
ALTERA_AVALON_TIMER_CONTROL_START_MSK | ALTERA_
AVALON_TIMER_CONTROL_CONT_MSK | ALTERA_AVALON_
TIMER_CONTROL_ITO_MSK );
```

Po nim powinna znaleźć się tylko pusta pętla *while*, bez żadnych dodatkowych instrukcji. Program możemy skompilować i wgrać do procesora, pamiętając, aby wcześniej wgrać odpowiednią konfigurację układu FPGA i dokonać *Refresh connections*. Voilà! Nasz wyświetlacz działa dokładnie tak, jak poprzednio! Tym razem jednak nasza pętla główna jest pusta, a w programie nie ma ani jednej instrukcji generującej opóźnienie! Czas jednak, aby obsługa naszego wyświetlacza była nieco bardziej praktyczna i abyśmy mieli łatwą kontrolę nad tym, co jest na nim wyświetlane.

Nieco modyfikacji

Aby nasz wyświetlacz mógł zostać praktycznie wykorzystany musimy wprowadzić kilka modyfikacji. Po pierwsze, tworzymy w pliku *7SEG.c* tablicę volatile uint8_t displayData[4]={0, 0, 0, 0}; Przechowywać będzie ona kombinacje segmentów do wyświetlenia na kolejnych wyświetlaczach. Słowo poprzedzające typ danych (*volatile*) informuje kompilator, że dostęp do danych będzie miał miejsce zarówno z przerwań, jak i programu i nie może zostać dokonana optymalizacja dostępu do tych danych. Modyfikujemy także funkcję *refreshDisplay*, aby korzystała z dopiero utworzonej tablicy, a dane wysyłała bezpośrednio na odpowiedni port:

```
...
setDisplay(DSOFF);
IOWR _ ALTERA _ AVALON _ PIO _ DATA(SEGMENT _ BASE,
displayData[disp]);
setDisplay(disp);
...
```

Jeszcze pozostało tylko zmodyfikować jedną z funkcji i dodać przykładową funkcję wyświetlającą dane w formacie dziesiętnym na wyświetlaczu jak na **listingu 4**.

W pierwszej z nich dodano wskazanie pozycji (wyświetlacza), na którym ma zostać ustawiona dana liczba, wraz z kontrolą wartości (jest to ochrona przed tzw. wyciekiem pamięci – czyli zapisem danych w nieznane miejsce, np. jako piąty, nieistniejący, element naszej tablicy; wycieki pamięci to stosunkowo częsty i trudny do wykrycia błąd). Ponadto zapis danych na port zastąpiono zapisem do tablicy, którą wcześniej przygotowaliśmy.

Druga funkcja po prostu rozbija podaną liczbę na poszczególne cyfry w zapisie dziesiętnym i ustawia je na kolejnych wyświetlaczach. Po odpowiednim poprawieniu pliku nagłówkowego, możemy zaraz przed pustą pętlą *while* w funkcji *main* dodać in DisplayDec(1234); Po kompilacji programu i jego uruchomieniu powinniśmy zobaczyć prawidłowo wyświetlaną liczbę.

Podsumowanie i zadania

W czasie tego spotkania nasz system wzbogaciliśmy o timer, który może generować przerwania. Dowiedzieliśmy się o jego działaniu a także o działaniu systemu przerwań, po czym uruchomiliśmy odświeżanie wyświetlaczy LED działające właśnie w przerwaniu. Czas teraz na kolejną porcję zadań (programistycznych) do realizacji (bez zmian w *Qsys*):

- Zmieniając okres timera, spowolnić multipleksowanie do tak małej prędkości, aby można było naocznie przekonać się, jak to wszystko działa.
- W oparciu o dokładnie ten sam timer przygotować odmierzanie czasu w sekundach i wyświetlanie go na wyświetlaczach (tak, tak, 1 timer możemy wykorzystać do obsługi wielu zadań – dlatego nasza biblioteka *7SEG* nie zawiera obsługi timera, a jedynie ma wbudowaną funkcję, którą należy cyklicznie wywoływać).
- Napisać dla wprawki funkcje, które mogłyby wyświetlać dane w innych systemach liczbowych (10, 16), prezentować dane dodatnie i ujemne, nie wyświetlać nieznaczących nic zer (np. 0001), umożliwiać zapalenie kropki na wybranym wyświetlaczu (uwaga – NIE używamy zapisu zmiennoprzecinkowego – typu float, double itp. są zakazane!).

Powodzenia! W czasie kolejnego spotkania będziemy kontynuować nasze zmagania z przerwaniami i timerami, m.in. w celu skutecznej programowej eliminacji drgań styków.

Piotr Rzeszut, AGH

REKLAMA

ELEKTRONIKA PRAKTYCZNA NA KAŻDYM EKRANIE

www.ulubionykiosk.pl

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl