Czy ARM zawsze znaczy szybki?

Badanie szybkości pracy portów mikrokontrolera LPC2114 firmy Philips

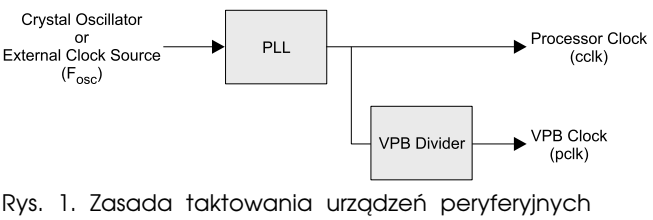
„ARM równa się szybkość". To krótkie zdanie wydaje się oczywiste – wydajny 32-bitowy rdzeń RISC taktowany sygnałem zegarowym o maksymalnej częstotliwości wielu dziesiątek megaherców pozwala oczekiwać, że coraz bardziej popularne mikrokontrolery z rdzeniem ARM w każdej dziedzinie biją na głowę swoich 8-bitowych kolegów. Okazuje się jednak, że rzeczywistość nie jest aż tak różowa o czym przekonało mnie praktyczne zapoznanie się z mikrokontrolerem ATM7TDMI-S typu LPC2114 firmy Philips.

Okazało się, że istnieje ważna z punktu widzenia elektronika dziedzina w której ARM-y nie tylko nie zachwycają, ale wręcz rozczarowują wydajnością. Dziedziną tą jest szybkość dostępu do portów mikrokontrolera. Niniejszy artykuł zawiera opis praktycznych prób i pomiarów szybkości, opis potoku procesorów ARM7, a także oparte o dokumentację procesora LPC2114 i rdzenia ARM7 wyjaśnienie zachodzących zjawisk, które do beczki miodu z napisem ARM dodają (niemałą) łyżkę dziegciu.

Nadzieje

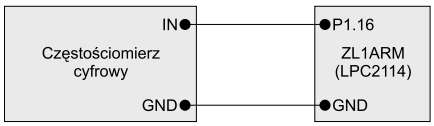
Muszę przyznać, że od kiedy oczywiste stało się iż ARM-y trafią pod strzechy nie mogłem doczekać się wypróbowania procesora, który niemal natychmiast otoczony został mgiełką ogromnej wydajności. Ogromne możliwości szybkościowe sprawiały, że w mojej głowie zrodziło się tysiąc pomysłów na ich wykorzystanie zanim w ogóle miałem okazję napisać choćby linijkę kodu. Tym bardziej ucieszyłem się, gdy okazja taka się nadarzyła. Dał mi ją zestaw laboratoryjny ZL1ARM firmy BTC zawierający na pokładzie mikrokontroler LPC2114 firmy Philips, obwody zasilania napięciami 1,8 V, 3,3 V i 5 V, kwarc 12 MHz, wyświetlacz alfanumeryczny LCD, 8 diod LED oraz dwa porty szeregowe i złącze interfejsu JTAG. Na płycie CD dostarczanej wraz z zestawem znajduje się kompletne oprogramowanie niezbędne do rozpoczęcia przygody z LPC2114.

Gdy bierzemy do ręki nieznany mikrokontroler pierwszą rzeczą jaką robimy jest napisanie prostego programiku, który w jakikolwiek sposób manifestuje swoje działanie. Chcemy zobaczyć, że „to naprawdę działa", później wszystko idzie już łatwo. Ja zawsze piszę program migający diodą LED, ale nie chce mi się pisać funkcji opóźniających i to miganie wychodzi całkiem szybkie – zachowanie portu obserwuję mierząc występującą na nim częstotliwość. Dzięki temu mam też pewien pogląd na szybkość pracy procesora. Tym razem jednak zabawę z zestawem ZL1ARM zacząłem od uruchomienia programu przykładowego dostarczonego wraz z nim na płycie CD. Po zainstalowaniu kompilatora i IDE firmy Keil oraz narzędzia *LPC2100 Flash Utility* program ruszył bez problemu i już po chwili ARM „ożył". Pobrane ze strony internetowej firmy Keil inne przykłady także zadziałały, choć wymagały wcześniej zmiany nazw rejestrów SFR tak, aby dokładnie zgadzały się z plikiem nagłówkowym *LPC21xx.h* i z dokumentacją mikrokontrolera. Zachęcony tym sukcesem (trzeba bowiem pamiętać, że nader często firmowe przykłady zawodzą) postanowiłem wrócić do swojego przyzwyczajenia i zobaczyć ile da się wycisnąć z ARM-a!



Rys. 1. Zasada taktowania urządzeń peryferyjnych

Pobieżne przejrzenie dokumentacji zdradziło, że aby uzyskać maksymalną szybkość pracy portów należy oprócz ustawienia maksymalnej częstotliwości taktowania rdzenia odpowiednio skonfigurować rejestr VPBDIV (**rys. 1**). Dwa najmniej znaczące bity tego rejestru decydują o częstotliwości z jaką pracują urządzenia peryferyjne, do których należą porty GPIO mikrokontrolera (*General Purpose Input Output*). Do tych bitów rejestru VPDIV wpisałem (w kodzie programu) wartości 01 co sprawia, że częstotliwość ta jest taka sama jak częstotliwość pracy rdzenia. Tę drugą ustawiłem na 60 MHz wybierając mnożnik PLL równy 5. Środowisko uVision3 uwalnia nas od konieczności żmudnego ustawiania odpowiednich rejestrów związanych z PLL – wyboru mnożnika dokonuje się wybierając żądaną wartość z listy rozwijanej, do której mamy dostęp dwukrotnie klikając na plik *startup.s* w drzewie plików projektu. Mamy tam także możliwość włączenia pełnego, włączenia częściowego lub wyłączenia modułu MAM (*Memory Acceleration Module)*, który poprawia szybkość odczytu instrukcji z pamięci Flash. Oczywiście włączyłem pełną funkcjonalność modułu MAM. Asemblerowy plik *startup.s* został automatycznie zmodyfikowany zgodnie z tym co zostało wybrane.



Rys. 2. Schemat układu pomiarowego

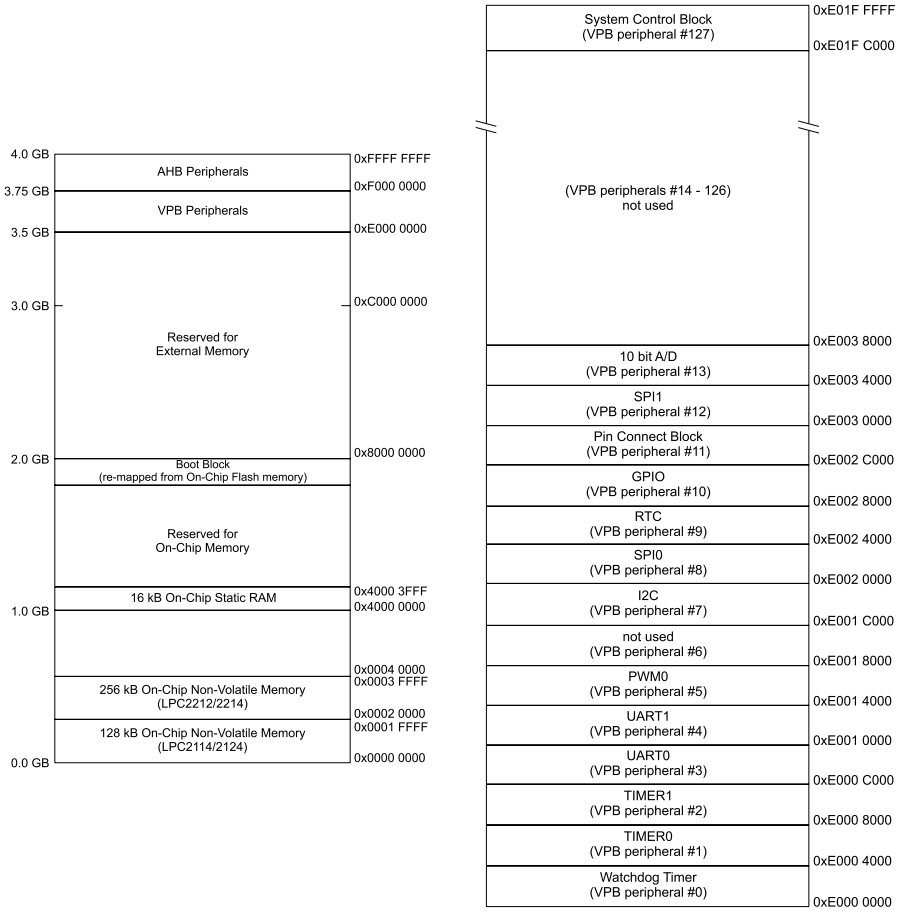
W funkcji *main* wpisałem prostą pętelkę, która najszybciej jak tylko można zmienia stan linii P1.23...P1.16. Wszystko było gotowe do próby. Podłączyłem częstościomierz (**rys. 2**) i uruchomiłem program.

Jakież było moje zdziwienie, gdy na wyświetlaczu zamiast oczekiwanych ok. 10...20 MHz otrzymałem wynik nieco ponad 3,5 MHz! Postanowiłem zajrzeć do pliku **.lst* który zawiera wynik działania kompilatora C w postaci programu asemblerowego. W tym celu w oknie *Project -> Options for Target ... -> Listing* zaznaczyłem pole *Assembler Listing* a pod nim pola *High Level Source* i *Assembly Code*. Dzięki temu w pliku **.lst* widoczne są zakomentowane fragmenty kodu źródłowego w języku C wraz z odpowiadającymi im fragmentami asemblerowymi wytworzonymi przez kompilator. Analiza pliku **.lst* z pomocą listy instrukcji ARM7 ujawniła dwie rzeczy. Po pierwsze, kompilator działał idealnie i niska częstotliwość nie była wynikiem jego nieoptymalnej pracy (to akurat mnie nie zdziwiło). Po drugie, procesor odwołując się do przestrzeni adresowej zarówno portów jak i pamięci RAM może używać jedynie adresowania pośredniego, z adresem podanym w rejestrze. Nie jest możliwe przesłanie stałej bezpośrednio na port lub do pamięci, nie są też dozwolone transfery pamięć-pamięć. Zdradza to architekturę zwaną *Load-Store*, co nie wróży dobrze szybkości współpracy z portami, jednak nadal nie wyjaśnia aż tak niskiej częstotliwości ich pracy.

Dokładne przewertowanie dostarczonej dokumentacji nie przyniosło jednoznacznej odpowiedzi, choć znajdujący się tam schemat blokowy LPC2114 mógł sugerować przyczynę – więcej w dalszej części tego artykułu. Postanowiłem zmierzyć czasy wykonania instrukcji przesłań (*ldr* i *str*) przy dostępie do portów i pamięci RAM, a także czasy wykonywania instrukcji działających jedynie na rejestrach.

Pomiary

Pomysł jak zmierzyć czasy wykonania określonych instrukcji jest prosty – wystarczy dodać je do pętli w której naprzemiennie zmieniamy stan portu, a okres tych zmian wydłuży się o czas ich wykonania (sposób w jaki procesor wykonuje kolejne instrukcje został bardziej szczegółowo przedstawiony dalej – w opisie potoku). Napisałem prosty program



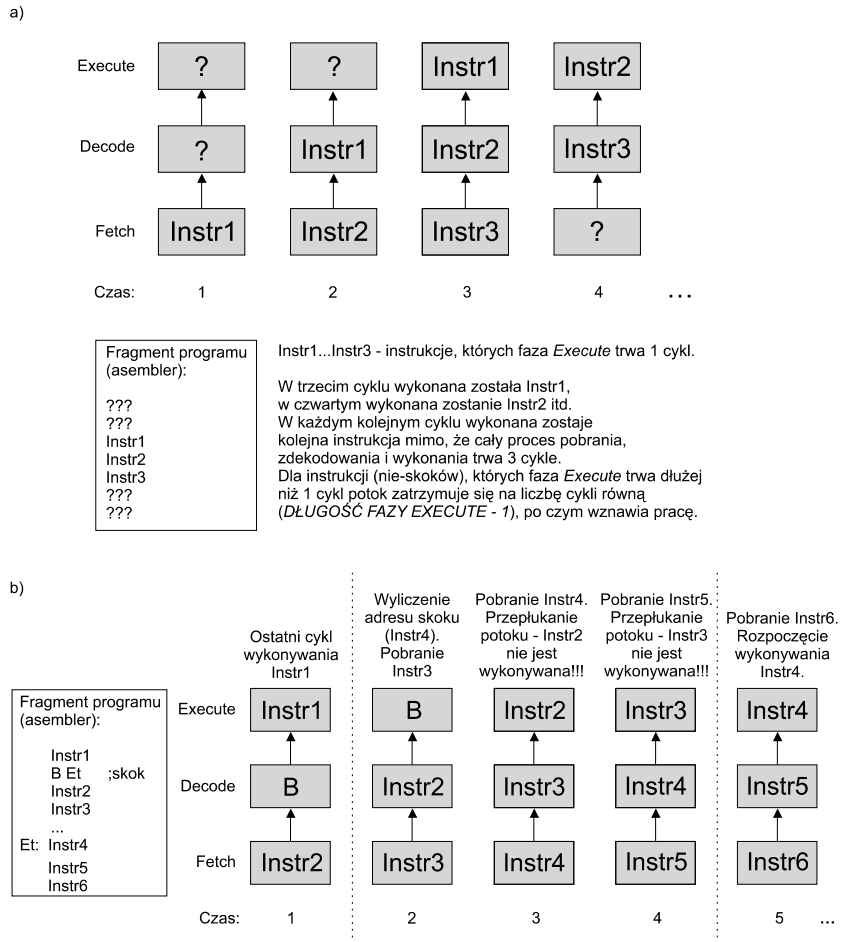
Rys. 3. Mapa pamięci mikrokontrolera LPC2114

hybrydowy, w którym główna część (odpowiedzialna za konfigurację procesora) napisana jest w języku C, zaś właściwa pętla pomiarowa zaimplementowana została w asemblerze ARM7. **List. 1** przedstawia część napisaną w C. Jest to króciutki programik, w którym najpierw odpowiednio konfigurujemy szybkość taktowania peryferiów (wspomniany rejestr VPBDIV), następnie określamy kierunki linii we/wy po czym wywołujemy napisaną w asemblerze funkcję *toggle()*. Została ona zadeklarowana z dyrektywą *extern* po to, aby jej ciało można było umieścić w innym pliku źródłowym (*toggle.s*). Kompilatora nie interesuje przy tym w jakim języku zostanie zaimplementowana – wystarczy aby linker miał informację o tym gdzie jej szukać. Na **list. 2** znajduje się kod funkcji *toggle()*. Dyrektywa *.global toggle* pozwala wywołać ją z innych plików projektu, zaś zgodność nazw funkcji w plikach **.c* i **.s* daje linkerowi jednoznaczną informację gdzie się ona znajduje.

Pierwszą część funkcji stanowi

List. 1. Program testowy

```
/*****************************************************************************/
/*  Test szybkosci pracy portow GPIO procesora LPC2114 firmy Philips (ARM7TDMI)       */
/*  Zestaw laboratoryjny: ZL1ARM firmy BTC                                           */
/*  Autor:          Arkadiusz Antoniak, 2005                                         */
/*****************************************************************************/

#include <LPC21xx.H>

extern void toggle(void);     //Funkcja toggle zaimplementowana w asemblerze ARM7
                              //w pliku toggle.s

int main (void)
{
   VPBDIV&=0xFFFFFFFC;
   VPBDIV|=0x01;              //Ustawienie maksymalnej czestotliwosci pracy urzadzen
                              //peryferyjnych, rownej czestotliwosci taktowania rdzenia

   IODIR0 = 0xFF000000;       //P0.24 ... P0.31 - wyjscia, reszta - wejscia
   IODIR1 = 0x00FF0000;       //P1.16 ... P1.23 - wyjscia, reszta - wejscia

   IOPIN1 = 0x00000000;       //Poczatkowo P1 wygaszony

   toggle();                  //Wywolanie funkcji toggle

   while(1);                  //Profilaktyczna petla terminujaca
}
////////////////////////////////////////////////////////////////////////////////
```



Rys. 4. Działanie potoku (pipeline) procesora ARM7

inicjalizacja rejestrów przechowujących adresy portów i pamięci RAM oraz wartości, które będą tam przesyłane. Do R6 wpisywana jest wartość 0x0xE0028000 będąca adresem rejestru IOPIN0, który związany jest z liniami portu 0. R3 przechowuje adres portu 1 (0xE0028010), zaś R5 - adres miejsca w pamięci RAM dokładnie na początku drugiego kilobajta (0x40000400). Rejestry R1 i R2 przechowują wartości, które wpisywane będą do IOPIN1 powodując odpowiednio wygaszenie lub zapalenia linii P1.23...P1.16. W zrozumieniu pochodzenia podanych liczb pomocny będzie **rys. 3**, na którym przedstawiona została mapa przestrzeni adresowej procesora LPC2114 (4 GB robią wrażenie!) oraz bardziej szczegółowa mapa pamięci urządzeń peryferyjnych. Wyjaśnienia wymaga dziwny sposób przesyłania stałych do rejestrów – nie są one wpisywane „za jednym zamachem", zamiast tego do adresu „bazowego" dodawane są odpowiednie (okrągłe) wartości. Jest tak dlatego, że asembler akceptuje jedynie takie stałe 32-bitowe, z których da się otrzymać stałe 8-bitowe po rotacji o parzystą liczbę miejsc (0, 2, 4, ..., 30). Na przykład - nie da się przesłać do rejestru stałej 0x1234. Da się natomiast wpisać tam stałą 0xC000003F, gdyż po rotacji o 2 miejsca w lewo otrzymujemy z niej 8-bitową liczbę 0xFF. Po fazie inicjalizacji następuje właściwa pętla pomiarowa *toggle_loop*. Stanowią ją dwie instrukcje *str* wpisujące do rejestru IOPIN1 zawartość R1 i R2. Nazwijmy je STR1 i STR2 – oznaczenia te będą ważne później, przy opisie wyjaśnienia zachodzących zjawisk. Po nich występuje instrukcja mierzona, a następnie skok zamykający pętlę. Należy zauważyć, że jedną z instrukcji mierzonych jest instrukcja oznaczona STR3, która także dokonuje wpisu do IOPIN1 (chodzi w końcu o to, aby zmierzyć jak długo to robi). Jednak wpisuje ona tam to samo co STR2, a więc wpływa na okres przebiegu wyjściowego jedynie czasem wykonania.

List. 2. Implementacja funkcji toggle w asemblerze ARM7

```
      .global   toggle          @ funkcja toggle dostepna z innych plikow projektu
toggle:
      mov     R6,#0xE0000000
      add     R6,R6,#0x28000      @ R6 = 0xE0028000 (IOPIN0)

      mov     R3,#0xE0000000
      add     R3,R3,#0x28000
      add     R3,R3,#0x10         @ R3 = 0xE0028010 (IOPIN1)

      mov     R5,#0x40000000
      add     R5,R5,#0x400        @ R5 = 0x40000400 (wewnetrzny RAM)

      mov     R1,#0
      mov     R2,#0x00FF0000

toggle_loop:
      str     R1,[R3]             @ STR1 - P1.16 ... P1.23 = 00000000b
      str     R2,[R3]             @ STR2 - P1.16 ... P1.23 = 11111111b

      @ -- mierzona instrukcja --
      @ldr      R4,[R6]           @ pobranie slowa z P0 (IOPIN0)
      @str      R2,[R5]           @ STR4 - zapis slowa do wewnetrznej pamieci RAM
      @str      R2,[R3]           @ STR3 - zapis slowa do P1 (IOPIN1)
      @mov      R4,R4             @ przykladowa operacja na rejestrach

      b       toggle_loop         @ B - kontynuacja petli
```

Pomiary polegały na odkomentowaniu wybranej instrukcji i zmierzeniu częstotliwości na P1.16. Wyniki przedstawia **tab. 1**. Jak widać czas wykonywania instrukcji przesłań bardzo mocno zależy od miejsca w pamięci, do którego dane są wpisywane, lub z którego są pobierane. Choć przesłanie pomiędzy rejestrami (są one częścią rdzenia procesora) zajmuje tylko 1 cykl zegarowy a dostęp do przestrzeni adresowej wewnętrznej pamięci RAM trwa niewiele dłużej (2 cykle), to zapis do portu trwa 7 cykli a odczyt - aż 8 cykli! Najkrótszy okres jaki da się uzyskać to 17 cykli, co przy częstotliwości pracy wynoszącej 60 MHz daje częstotliwość 3,53 MHz.

Co zrobiłem źle?

Po dokonaniu pomiarów zadawałem sobie powyższe pytanie. Nie wierzyłem, że ARM aż tak powoli radzi sobie z portami GPIO i sądziłem że nie ustawiłem odpowiednio któregoś z rejestrów itp. Modyfikując jeden z przykładów pobrany ze strony firmy Keil (dotyczący przerwań od Timera0) sprawdziłem, czy częstotliwość pracy rdzenia rzeczywiście wynosi 60 MHz. Okazało się że tak. Jak wspomniałem dokumentacja milczała na temat szybkości GPIO. Pozostał Internet.

Znalazłem sporo ciekawych materiałów w tym dokumentację rdzenia ARM7, która pośrednio pomogła w wyjaśnieniu problemu [1]. Jednak nie znalazłem nic co bezpośrednio trakto-

wałoby o tym jak szybkie mogą być porty ARM-a. Wtedy spróbowałem poszukać grup dyskusyjnych poświęconych LPC21xx i ogólnie procesorom ARM. Okazało się (nie po raz pierwszy), że w niektórych przypadkach doświadczenie kolegów po fachu jest cenniejsze niż dokumentacja. Wiele osób pisało o problemie z szybkością GPIO w tym pracownik firmy Philips, który częściowo wyjaśnił sprawę. Poniżej prezentuję moje wyjaśnienie oparte o post owego pracownika, dokumentację ARM7 ([1] i [2]) oraz dokonane pomiary. Konieczne jest jednak wcześniejsze przedstawienie budowy potoku procesorów ARM7.

Tab. 1. Wyniki pomiarów (częstotliwość pracy rdzenia i układów peryferyjnych równa 60 MHz)

Instrukcja mierzona	Częstotliwość [kHz]	Okres T [ns]	Okres T [cykle]	ΔT=T-283,(3)ns [ns]	Liczba cykli wykonywania fazy Execute
- brak instrukcji	3530	283,(3)	17	0	-
mov R4,R4 rejestr <- rejestr	3334	300	18	16,(6)	1
str R2,[R5] rejestr -> RAM	3158	316,(6)	19	33,(3)	2
str R2,[R3] rejestr -> port	2500	400	24	116,(6)	7
ldr R4,[R6] rejestr <- port	2400	416,(6)	25	133,(3)	8

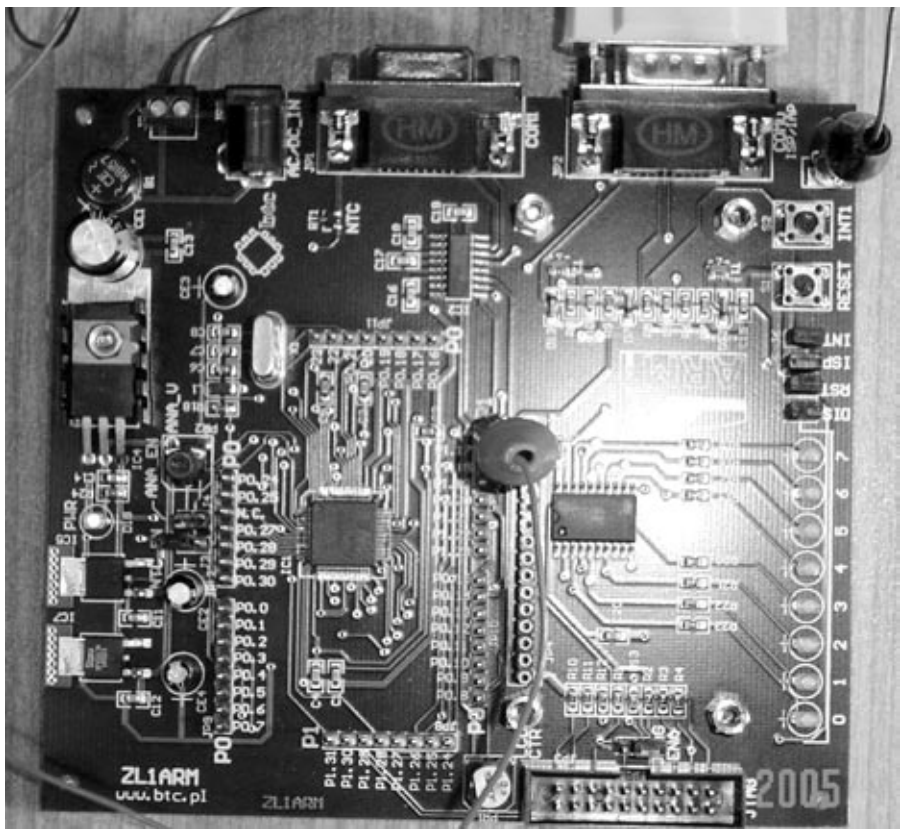
Działanie potoku (*pipeline*) procesora ARM7

Idea potoku opiera się na sprytnej sztuczce która pozwala wykonywać instrukcje nawet w 1 cyklu zegarowym, choć tak na prawdę cały proces od pobrania instrukcji do końca jej wykonania trwa dłużej. Sam potok to nic innego jak kilka zatrzasków (są one stopniami potoku) ustawionych jeden za drugim, które w każdym cyklu zegarowym przesuwają instrukcję do następnego stopnia. Obecność instrukcji w danym stopniu wiąże się z jej pobraniem, zdekodowaniem, odczytem/zapisem do pamięci itd. - zależnie od stopnia. W rezultacie w każdym cyklu kończy się wykonywanie 1 instrukcji (pod warunkiem, że faza jej wykonywania trwa 1 cykl) mimo, że od pobrania minęło tyle cykli, ile jest stopni potoku.

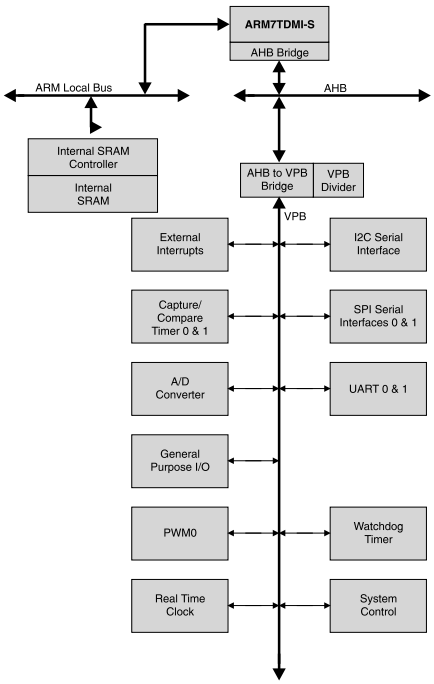
Istnieją różne implementacje potoków różniące się m.in. liczbą stopni. **Rys. 4a** przedstawia ogólną zasadę działania potoku procesorów ARM7. Jak widać w przypadku tej rodziny potok jest trzystopniowy (ARM9 posiada 5 stopni) i składa się z następujących stopni:

1. *Fetch* – faza pobrania instrukcji z pamięci programu;
2. *Decode* – faza dekodowania instrukcji. Po jej wykonaniu procesor „wie" z jaką instrukcja ma do czynienia;
3. Execute – faza wykonania instrukcji. W jej trakcie procesor dokonuje odpowiednich operacji zależnych od tego, co wykonywana instrukcja robi. Faza *Execute* niektórych instrukcji trwa dłużej niż 1 cykl.

W każdym cyklu zegarowym instrukcje przesuwają się do następnego stopnia. Jeśli faza *Execute* instrukcji znajdującej się w tym stopniu trwa 1 cykl, to potok działa bez zatrzymywania się i co cykl kończy się wykonywanie kolejnych instrukcji. Jednak niektóre z instrukcji wykonują się dłużej. Na przykład - jak napisano w [1] wykonanie instrukcji *str reg_src,[reg_dest,#offset]* trwa 2 cykle. W pierwszym cyklu rdzeń oblicza adres przeznaczenia - dodaje *offset* do wartości *reg_dest* (nie podanie wartości *offset* traktowane jest jak podanie wartości 0). W drugim dokonuje właściwego wpisu. Podczas drugiego cyklu fazy *Execute* tej instrukcji potok zatrzymuje się i kolejne instrukcje nie są pobierane. Inaczej jest gdy wykonywana jest instrukcja skoku. W ogóle skoki są największym utrapieniem procesorów z przetwarzaniem potokowym. Zanim procesor dobrze rozpozna skok i adres, do którego należy skoczyć, to w niższych stopniach potoku już znajdują się instrukcje położone w pamięci programu tuż za instrukcją skoku. Instrukcje te oczywiście nie powinny być wykonane - i nie zostaną - jednak wymaga to specjalnych zabiegów co zabiera czas. Ogólnie zjawisko to nazywa się **opóźnieniem skoku** (*branch penalty*). Inną sprawą jest występowanie w architekturach niektórych procesorów (np. SPARC) zjawiska **skoku opóźnionego** (*delayed branching*). Polega ono na jawnym założeniu, że instrukcja znajdująca się tuż za skokiem zostanie wykonana mimo wykonania skoku! Pisząc program należy o tym pamiętać. Często jednak asemblery takich procesorów są tak na prawdę meta-asemble-



Fot. 5. Zestaw ZL1ARM podczas pomiarów



Rys. 6. Schemat blokowy mikrokontrolera LPC2114 (fragment)

Fot. 7. Maksymalna częstotliwość pracy portu

rami i ukrywają tę właściwość przed programistą. Uwaga! Pojęć „opóźnienie skoku" i „skok opóźniony" nie należy mylić ze sobą – są to dwie zupełnie różne rzeczy. Skok opóźniony jest jednym ze sposobów na radzenie sobie z opóźnieniem skoku.

Wróćmy do rodziny ARM7. Prosty przykład wykonania fragmentu programu w którym znajduję się skok przedstawiony został na **rys. 4b**. Prześledźmy go szczegółowo.

W cyklu numer 1 instrukcja B (skok) jest dekodowana i rdzeń nie wie jeszcze, że ma nastąpić skok – jakby nigdy nic pobiera pierwszą po skoku instrukcję Instr2. **Cykle 2...4 to wykonanie instrukcji B**.

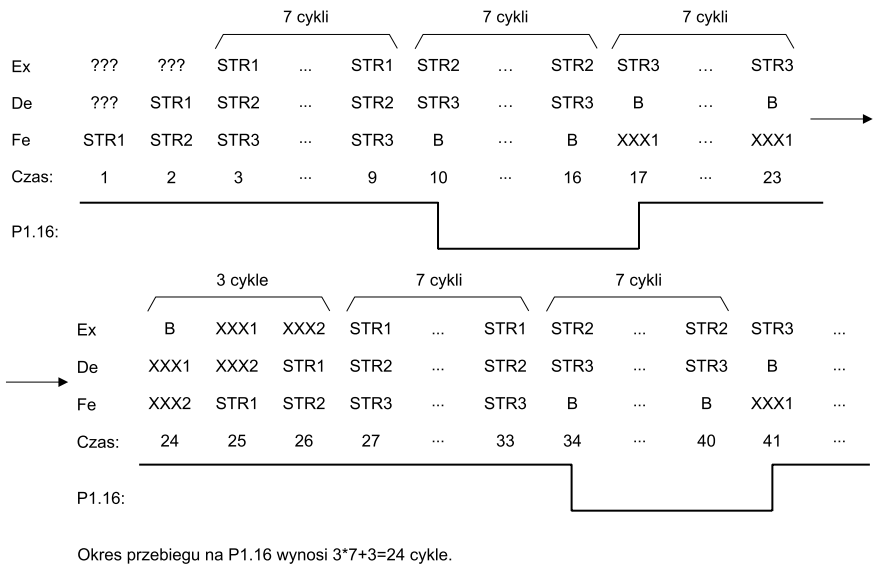
W cyklu numer 2 wyliczany jest adres skoku. Ponieważ w tym cyklu instrukcja B weszła do stopnia *Execute* to automatycznie w stopniu *Fetch* znajduje się już kolejna instrukcja z pamięci programu, czyli Instr3. Nic się na to poradzić nie da.

W cyklu numer 3 znany jest już adres do którego należy skoczyć (etykieta Et) i pobierana jest stamtąd instrukcja Instr4. Jednocześnie do stopnia *Execute* wsuwa się Instr2 jednak dzięki specjalnym mechanizmom nie jest wykonywana – po prostu przechodzi i znika w niepamięci.

W cyklu numer 4 zachodzi identyczne zjawisko jak w cyklu 3 z tym, że pobierana jest kolejna instrukcja z docelowego miejsca skoku, czyli Instr5.

Cykl numer 5 jest pierwszym cyklem wykonywania instrukcji Instr4. Skok do etykiety Et został wykonany. Jak widać wykonanie instrukcji B trwa 3 cykle.

Uzbrojeni w elementarną wiedzę o potoku procesora ARM7 możemy wreszcie wyjaśnić dlaczego mimo tak dużej szybkości pracy rdzenia (instrukcje przesłań rejestr-rejestr wykonują się w 1 cyklu) pozwala on na tak mizerną pod względem szybkości pracę z portami GPIO.



Rys. 8. Praca potoku przy instrukcji mierzonej str R2, (R3) (oznaczonej STR3 – przesłanie rejestr -> port)

Wyjaśnienie

Okazało się, że przyczyną wolnego dostępu do portów jest umieszczenie ich w przestrzeni adresowej urządzeń peryferyjnych, które nie są bezpośrednio dostępne rdzeniowi (RAM jest) i konsekwentne zaimplementowanie architektury *Load-Store*. Niestety, w samej dokumentacji procesora zabrakło informacji na ten temat co jest bardzo przykre. Na **rys. 6** znajduje się fragment schematu blokowego LPC2114. Przestrzeń adresowa portów (podobnie jak innych peryferiów) dostępna jest rdzeniowi za pośrednictwem mostu szyny AHB (*Advanced High-performance Bus*) i dalej poprzez most pośredniczący pomiędzy szynami AHB i VPB (*VLSI Peripheral Bus*). Trzeba też wiedzieć, że istnieje kilka rodzajów cykli fazy *Execute* wszystkich instrukcji. Niektóre z nich (*Non sequential cycles*) mają to do siebie, że most AHB/VPB dodaje do nich 1 cykl oczekiwania. Same operacje na szynie VPB też zabierają kilka cykli. W rezultacie faza *Execute* instrukcji *ldr* lub *str* odwołującej się do przestrzeni VPB wykonuje się odpowiednio 8 i 7 cykli. Ta sama instrukcja *str* która wpisuje dane do wewnętrznej pamięci RAM wykonywana jest przez zaledwie 2 cykle – na przykład instrukcja STR4 z list. 2 (wynik w tab. 1).

Rys. 8 przedstawia pracę potoku podczas pomiaru czasu trwania instrukcji STR3 z list. 2 (przesłanie rejestr->port) wraz z pokazaniem jak zmienia się stan na linii P1.16 (tak na prawdę na liniach P1.23... P1.16). Instrukcje oznaczone XXX1 i XXX2 to „instrukcje" znajdujące

się w pamięci *Flash* tuż za instrukcją B. Ze względu na to, że nie ma tam żadnych prawdziwych instrukcji są to jakieś słowa o długości równej *opcode*-owi instrukcji ARM7. Instrukcje oznaczone ??? mogą reprezentować różne instrukcje programu. Jeśli jest to pierwszy przebieg pętli są to ostatnie instrukcje z fazy inicjalizacji rejestrów. Jeśli zaś są to kolejne przebiegi pętli – instrukcje ??? stanowią to samo co instrukcje XXX1 i XXX2 (wtedy cykl numer 25 jest taki sam jak cykl numer 1, a cykl numer 26 jest taki sam jak cykl numer 2). Okres przebiegu na P1.16 wynosi tu 24 cykle przy długości pętli równej tylko 4 instrukcje (wliczając skok). Nawet przy taktowaniu 60 MHz 24 cykle odpowiadają częstotliwości zaledwie 2,5 MHz.

Podsumowanie

Trzy i pół megaherca na porcie to w ogólności nie jest mało, ale jak na procesor, którego rdzeń taktowany jest z częstotliwością 60 MHz jest to bardzo mało. Najłatwiej zdać sobie z tego sprawę analizując następujące porównanie. Procesor z rdzeniem AVR serii ATMega taktowany z maksymalną częstotliwością 16 MHz potrafi zmieniać stan portu z częstotliwością 4 MHz, a więc szybciej niż ARM7!!! Nie powiem już jak szybko może to robić 8051 o zmodyfikowanym 1-taktowym rdzeniu taktowany zegarem o częstotliwości kilkudziesięciu MHz (obecna instrukcja *CPL Port.bit*).

Nie znaczy to oczywiście, że uprawnione jest stwierdzenie iż AVR jest szybszy niż ARM. Oczywiście, że nie! 32-bitowa szybka arytmetyka z instrukcjami arytmetycznymi, logicznymi i innymi wykonywanymi na rejestrach (inaczej się nie da) sprawia, że ARM-y na prawdę biją na głowę AVR-y i im podobne mikrokontrolery – tyle, że nie w każdej dziedzinie. Procesory z rdzeniem ARM7 i nowszymi rdzeniami z tej rodziny doskonale sprawdzą się w aplikacjach, gdzie relatywnie rzadko potrzebne są dostępy do portów, zaś bardzo istotna jest szybkość dokonywania obliczeń. W wielu przypadkach zastąpią DSP! Trzeba jednak zdawać sobie sprawę, że ustępują szybkim 8-bitowcom jeśli chodzi o szybkość pracy portów GPIO, co dla nas – elektroników – często ma kluczowe znaczenie.

Arkadiusz Antoniak
arkadiusz.antoniak@ep.com.pl

Literatura:

[1] „ARM7TDMI-S Technical Reference Manual (Rev 4)”, ARM Limited 2001 [pdf]
[2] „LPC2114/2124/2212/2214 User Manual”, Philips Semiconductors 2004 [pdf]
[3] Fora dyskusyjne http://groups.yahoo.com/group/lpc2000/ *i http://www.embeddedrelated.com/groups/lpc2000/*