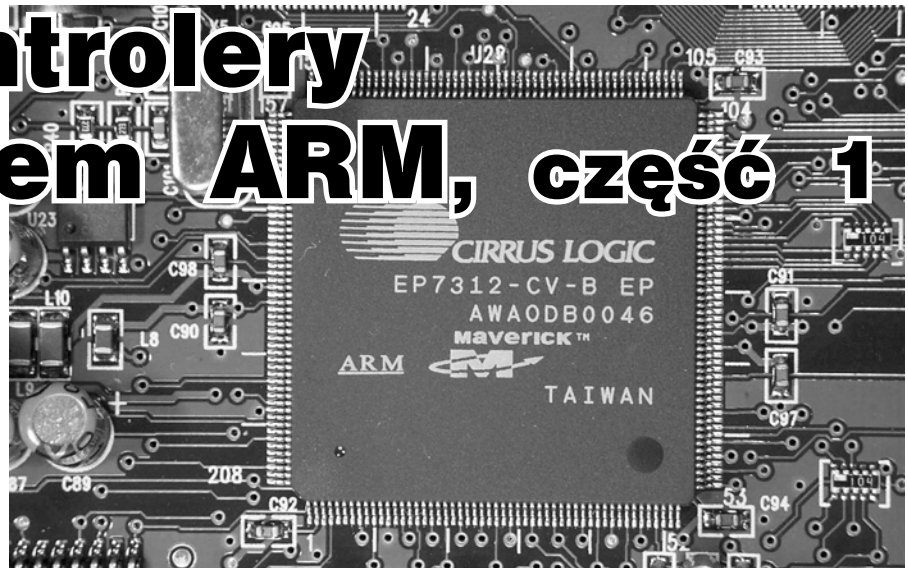
Mikrokontrolery z rdzeniem ARM, część 1



W większości obecnie prezentowanych (nie tylko w EP) projektów królują mikrokontrolery 8–bitowe. Dawniej była to nieśmiertelna rodzina 8051, obecnie panuje moda na AVR–y. Dotychczas użycie 32–bitowych mikrokontrolerów pozostawało w sferze marzeń przeciętnego konstruktora ze względu na wysoką cenę, skomplikowane rozwiązania układowe wymagające prowadzenia na płytkach 32–bitowych magistral systemowych oraz konieczność stosowania dodatkowych układów zewnętrznych. Sytuacja uległa diametralnej zmianie w momencie pojawienia się na rynku mikrokontrolerów z rdzeniem ARM i wbudowaną pamięcią Flash, najpierw firmy Philips LPC21xx i wkrótce później Atmela AT91SAM7, ATM – rodzina STR700 i wielu innych producentów.

Wymienione mikrokontrolery ARM zawierają w sobie cały 32–bitowy system mikroprocesorowy z pamięcią Flash oraz RAM, zwalniający konstruktora od prowadzenia skomplikowanych połączeń magistralowych. Ceny mikrokontrolerów są bardzo zachęcające, np. mikrokontroler LPC2131 (32 kB Flash, 8 kB RAM) kosztuje mniej więcej tyle samo, co ATmega16. Do dyspozycji mamy bardzo dużą moc obliczeniową, jaką daje 32–bitowy rdzeń taktowany z częstotliwością do 60 MHz. Mikrokontrolery LPC21xx są wyposażone w jednostkę centralną ARM7TDMI–S opracowaną w firmie ARM. Rdzeń ARM7TDMI–S staje się standardem wśród mikrokontrolerów 32–bitowych, podobnie jak rdzeń 8051 był i jest standardem wśród mikrokontrolerów 8–bitowych.

Poznając architekturę LPC21xx możemy w łatwy sposób przejść do programowania mikrokontrolerów innych producentów opartych na rdzeniu ARM7TDMI–S, np. Atmel albo STM, a gdy moc obliczeniowa stanie się niewystarczająca możemy pomyśleć o architekturze ARM9 albo ARM11. Gdy do dyspozycji mamy trochę większy mikrokontroler oparty na rdzeniu ARM wyposażony w jednostkę MMU (zarządzania pamięcią), możemy uruchamiać systemy operacyjne, takie jak Linux czy Windows CE. Programowanie mikrokontrolerów LPC21xx nie jest dużo bardziej skomplikowane od programowania AVR–ów, a do dyspozycji mamy analogiczne narzędzia, takie jak darmowy kompilator gcc (*arm–gcc*), pozwalający na pisanie programów w języku C oraz C++. Programowanie pamięci Flash mikrokontrolera może odbywać się w docelowym systemie poprzez port RS232 i oprogramowanie *LPC2000 Flash Utility*.

Rdzeń ARM7TDMI

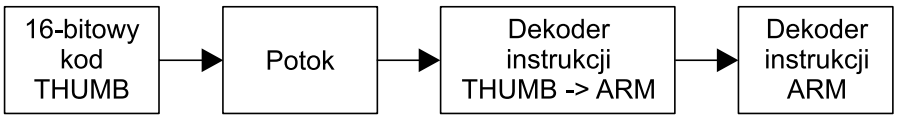
Sercem mikrokontrolerów rodziny LPC21xx jest jednostka centralna ARM7TDMI–S. Jest ona częścią dużej rodziny 32–bitowych mikroprocesorów ARM ogólnego przeznaczenia, które charakteryzują się bardzo małym poborem mocy, oraz prostotą budowy. Historia powstania mikroprocesorów ARM wywodzi się z lat 80–tych, kiedy to grupa inżynierów pod kierownictwem Rogera Wilsona i Steave`a Ferbera, pracująca dla firmy Acorn, rozpoczęła projektowanie rdzenia będącego rozwinięciem znanego mikroprocesora 6502 firmy MOS Technology. Firma Acorn budowała komputery w oparciu o ten procesor, więc celem było opracowanie nowego, wydajniejszego procesora, który miał się charakteryzować podobną architekturą do 6502. Było to duże wyzwanie ponieważ rok wcześniej zespół ten zajmował się projektowaniem mikroprocesorów 8–bitowych z pamięcią programu o pojemności 32 kB. Pierwsza wersja testowa mikroprocesora ujrzała światło dzienne w 1985 roku, a już rok później ukończono wersję produkcyjną ARM2, który był 32–bitowym mikroprocesorem z 26–bitową przestrzenią adresową. W późnych latach 80–tych firma Acorn rozpoczęła współpracę z firmą Apple w celu opracowania udoskonalonego rdzenia ARM. W tym samym czasie z firmy Acorn wydzieliła się firma ARM Ltd. Efektem współpracy było powstanie mikroprocesora ARM6 zastosowanego w palmtopie Apple Newton. Obecnie firma ARM zajmuje się sprzedażą licencji na aplikowanie rdzeni rodziny ARM.

Instrukcja	ADDS R0, R4, R8	ADCS R1, R5, R9	ADCS R2, R6, R10	ADCS R3, R7, R11	ORR R0, R0, #3	ORR R2, R2, #3
Pobranie		ADDS R0, R4, R8	ADDS R0, R4, R8	ADCS R2, R6, R10	ADCS R3, R7, R11	ORR R0, R0, #3
Dekodowanie			ADDS R0, R4, R8	ADDS R0, R4, R8	ADCS R2, R6, R10	ADCS R3, R7, R11
Wykonanie				ADDS R0, R4, R8	ADDS R0, R4, R8	ADCS R2, R6, R10

Rys. 1. Sposób wykonywania rozkazów przez procesory z rdzeniem ARM

W krainie Flashy ISP

W artykule skupiamy się na mikrokontrolerach z rodziny LPC21xx, ale przedstawione informacje odnoszą się także do innych rodzin mikrokontrolerów z rdzeniem ARM, jak np.: TMS470, ADuC7000, STR700 (i pochodne), AT91SAM7 (i pochodne), a także wielu innych wyposażonych w rdzeń ARM7TDMI–S.



Rys. 2. Sposób dekodowania rozkazów THUMB

Tradycyjne mikrokontrolery (budowane w architekturze CISC) posiadają rozbudowaną listę rozkazów, których wykonanie zajmuje wiele cykli maszynowych. Badania wykazały, że tylko niewielka część rozkazów jest wykorzystywana w przeciętnym programie, a reszta pozostaje nieużywana. Długa lista rozkazów powoduje skomplikowanie budowy dekodera instrukcji, oraz komplikację budowy rdzenia. Rejestry procesora CISC mają zazwyczaj ściśle określone przeznaczenie, np. akumulator służący do wykonywania operacji arytmetycznych i logicznych, rejestry indeksowe służące do adresowania pamięci itp. W programie musi więc znajdować się dużo przesłań rejestr – rejestr, ponieważ argumentami rozkazów mogą być tylko określone rejestry. W procesorach CISC do dyspozycji mamy zazwyczaj niewiele rejestrów (np. 8086 posiada 8 rejestrów), dlatego na liście rozkazów znajduje się wiele instrukcji potrafiących wykonywać operację bezpośrednio na pamięci. Biorąc pod uwagę wszystkie wady architektury CISC oraz to, że tylko niewielka liczba instrukcji jest wykorzystywana w typowych programach opracowano architekturę RISC (*Reduced Instruction Set Computers*). Mikroprocesory RISC potrafią wykonywać najczęściej kilkadziesiąt rozkazów, podczas, gdy np. popularny mikrokontroler 8051 ma ich 111. Posiadają zdecydowanie mniej trybów adresowania dzięki czemu kody rozkazów są dużo prostsze. Wszystkie operacje wykonywane są na rejestrach. Brak jest rozkazów operujących bezpośrednio na pamięci. Jedynymi rozkazami mającymi dostęp do niej to rozkazy LOAD (załaduj zawartość pamięci do rejestru) oraz STORE (zapisz zawartość rejestru do pamięci) Wykonanie operacji na danych znajdujących się w pamięci odbywa się według schematu: załaduj daną z pamięci do rejestru, wykonaj operację na rejestrze, prześlij daną powrotem do pamięci. Do dyspozycji mamy także większą liczbę rejestrów, które są rejestrami ogólnego przeznaczenia i nie mają ściśle określonych funkcji. Instrukcję przetwarzane są w potoku dzięki czemu większość instrukcji wykonywana jest w jednym cyklu zegarowym. Dzięki swojej prostocie procesory RISC posiadają bardzo prostą budowę, dzięki czemu (zazwyczaj) mogą pracować szybciej niż ich odpowiedniki CISC. Krótka lista instrukcji upraszcza ponadto tworzenie kompilatorów oraz zapewnia lepszą optymalizację kodu.

ARM7TDMI–S jest 32–bitowym rdzeniem mikroprocesora bazującym na architekturze RISC. Wyposażono go w 31 równouprawnionych, 32–bitowych rejestrów, na których można wykonywać wszystkie operacje z listy rozkazów. Jeden z tych rejestrów (R15) jest licznikiem rozkazów więc procesor potrafi zaadresować do 4 GB pamięci.

Przetwarzanie instrukcji odbywa się z zastosowaniem trójpoziomowego potoku. CPU w tym samym czasie, gdy wykonuje dany rozkaz, dekoduje już kolejny rozkaz i dodatkowo w tym samym czasie pobiera jeszcze kolejny rozkaz. Sposób przetwarzania rozkazów przez procesor ARM przedstawiono na **rys. 1**.

Efektem równoległego przetwarzania trzech rozkazów w różnych fazach jest wykonywanie większości instrukcji w jednym cyklu zegarowym. Potok działa najefektywniej, gdy program wykonywany jest sekwencyjnie i nie występują rozgałęzienia. W przypadku wystąpienia rozgałęzienia, potok musi zostać oczyszczony z niepotrzebnych instrukcji występujących za rozgałęzieniem i wypełniony nowymi instrukcjami, co powoduje wydłużenie czasu wykonania instrukcji. W odróżnieniu od innych powszechnie znanych mikroprocesorów, w ARM––ach prawie każda instrukcja może być wykonywana warunkowo, w zależności od stanu flag zmienionych przez poprzednią instrukcję przetwarzającą dane. Znakomicie ułatwia to

Jak to jest z *endianami*

Big–endian to sposób zapisu wielobajtowych danych, w którym najbardziej znaczący bajt zapisuje się pod najniższym adresem w pamięci. W formacie *little–endian* najmniej znaczący bajt zapisuje się pod najniższym adresem.

Tab. 1. Identyfikatory rejestrów prywatnych w trybach ochrony

Tryb ochrony	Identyfikator
User	Usr
Fast Interrupt	Fiq
Interrupt	Irq
Supervisor	Svc
Abort	Abt
System	Sys
Undefined	Und

tworzenie kodu bez rozgałęzień, zapobiegając stracie czasu na ponowne zapełnienie potoku. Licznik rozkazów PC (R15) wskazuje na osiem bajtów w przód w stosunku do bieżącej instrukcji, czyli na instrukcje aktualnie pobieraną. Należy o tym pamiętać przy adresowaniu względnym odnoszącym się do licznika rozkazów. Rdzeń ARM7TDMI–S posiada wspólną przestrzeń adresową dla rozkazów i danych. Dostęp do pamięci możliwy jest poprzez instrukcję ładowania, zapisu oraz zamiany. Pamięć może być adresowana w postaci 8, 16 lub 32 słów danych, przy czym procesor zapisuje dane w porządku *big–endian* lub *little–endian*.

W przypadku mikrokontrolerów LPC21xx dane są przechowywane zawsze w formacie *little–endian*. Ponieważ najszybsze pamięci Flash osiągają prędkości rzędu 20 MHz, natomiast rdzeń ARM7TDMI–S może wykonywać rozkazy ze znacznie większą prędkością, najczęściej stosuje się pamięć pośredniczącą *cache* lub przepisuje się zawartość pamięci Flash do RAM (w momencie uruchomienia mikrokontrolera), z której następnie program jest wykonywany. Zastosowanie pamięci *cache* lub dodatkowego RAM–u powoduje konieczność użycia dodatkowych zasobów oraz komplikuje układ. W mikrokontrolerach LPC21xx zastosowano inną, ale równie skuteczną metodę polegającą na odczytywaniu z pamięci Flash w jednym cyklu 4 rozkazów naraz, co umożliwia uruchamianie programu z pamięci Flash z pełną prędkością pracy rdzenia. Za zarządzanie pamięcią odpowiada kontroler MAM (*Memory Acceleration Module*).

Rdzeń może wykonywać dwa podzbiory rozkazów:
– 32–bitowe instrukcje ARM,
– 16–bitowe instrukcje THUMB.

Kod 16–bitowy charakteryzuje się wolniejszym wykonywaniem instrukcji 32–bitowych oraz większą gęstością upakowania kodu. Natomiast kod 32–bitowy zapewnia większą wydajność podczas działaniu na danych 32–bitowych oraz lepsze zarządzanie dużymi obszarami pamięci. Normalnie mikroprocesory 32–bitowe pracujące w trybie 16–bitowym, posługują się 16–bitowymi instrukcjami oraz rejestrami o długości 16 bitów. Tryb THUMB zapewnia natomiast 16 bitową listę instrukcji, operującą na 32–bitowych danych i 32–bitowej przestrzeni adresowej. Zbiór instrukcji THUMB jest podzbiorem listy instrukcji ARM i ma on swój odpowiednik w 32–bitowej liście instrukcji ARM. Podczas wykonania programu 16–bitowe instrukcje THUMB są dekodowane „w locie" do pełnych, 32–bitowych instrukcji ARM bez istotnej utraty wydajności wykonania. Na **rys. 2** przedstawiono sposób, w jaki rdzeń dekoduje i wykonuje rozkazy THUMB.

Objętościowo kod THUMB zajmuje 65% objętości kodu ARM i jest wolniejszy o około 40% od kodu ARM. Podczas działania programu można łatwo przełączać tryby pracy, np. część programu może być napisana w trybie THUMB, natomiast czasowo krytyczne procedury mogą działać w trybie ARM. Przełączanie trybu odbywa się za pomocą instrukcji BX lub BLX (nie można tutaj wykorzystać manipulacji na rejestrze znaczników). Wszystkie wyjątki procesora są wykonywane w 32–bitowym trybie ARM i jeżeli podczas wystąpienia wyjątku procesor znajduje się w trybie THUMB zostaje automatycznie przełączony do trybu ARM. Po zakończeniu obsługi wyjątku procesor powraca do trybu THUMB.

Tryby ochrony

W mikroprocesorach 8/16–bitowych wykonanie niedozwolonej operacji, np. skok do nieistniejącego obszaru pamięci programu lub zablokowanie na stałe przyjmowania przerwań, powodowało najczęściej zawieszenie całego systemu. Przykładowo, pracując w znanym systemie MS–DOS, gdy źle działający program zapisał niedozwolony obszar pamięci będący częścią systemu operacyjnego cały system ulegał zawieszeniu, a jedynym sposobem na wyjście z tej sytuacji było jego ponowne uruchomienie. Aby uniknąć podobnych sytuacji w konstrukcjach mikroprocesorów 32–bitowych wprowadzono sprzętowe tryby ochrony. Tryby ochrony działają w ten sposób, że wprowadza się kilka dodatkowych trybów pracy mikroprocesora. W każdym z tych trybów wydziela się określone zasoby i obszar pamięci mikroprocesora, do których program ma dostęp. Najczęściej do dyspozycji mamy dwie kategorie trybów ochrony:
– tryb użytkownika (USER MODE),
– tryb uprzywilejowany (PRIVILEDGED MODE).

W trybie użytkownika, program ma dostęp tylko do ograniczonych zasobów i instrukcji, których zmiana lub wykonanie nie jest krytyczne dla pracy reszty systemu. Przykładowo brak jest możliwości zmiany ustawień przerwań, nie jest też możliwe programowe wyjście z tego trybu ochrony. Blokowane są także niedozwolone instrukcje – np. w procesorach 386 *in, out*, których wykonanie może zaburzyć pracę systemu.

W trybie uprzywilejowanym program natomiast ma dostęp do wszystkich zasobów i całego obszaru pamięci, tak więc można wykonywać dowolne operacje na wszystkich zasobach systemu. Jest to tryb ochrony wykorzystywany przez oprogramowanie systemowe (system operacyjny).

Dzięki wprowadzeniu mechanizmu trybów ochrony tylko aplikacja pracująca w warstwie systemu operacyjnego ma dostęp do krytycznych zasobów, aplikacje użytkownika mogą się do tych zasobów odwoływać tylko za pośrednictwem odwołań systemowych. Dzięki temu mamy pewność, że dostęp do tych zasobów został wykonany w sposób prawidłowy. Przykładem systemu operacyjnego, który zapewnia dobrą ochronę zasobów poprzez korzystanie z trybów ochrony mikroprocesora Intel 386 jest system operacyjny Windows NT oraz Linux. Zapisanie dowolnego obszaru pamięci znajdującego się poza obszarem aplikacji powoduje wygenerowanie odpowiedniego wyjątku, którego wynik działania możemy zaobserwować np. w postaci komunikatu *Segmentation Fault*. Procesor ARM7TDMI–S został wyposażony w 7 trybów ochrony, z których 6 jest trybami uprzywilejowanymi (PRIVILEDGED MODE) oraz jeden z nich jest trybem użytkownika (USER MODE). Do dyspozycji mamy:
– **Tryb użytkownika (User Mode) –** wykorzystywany podczas normalnego wykonywania programu. W tym trybie niektóre operację mogące zaburzyć pracę systemu są niedostępne. Np. zablokowana jest możliwość modyfikacji flag

blokady przerwań.
- **Tryb przerwania szybkiego (Fast Interrupt) FIQ** – wykorzystywany do wykonywania przerwań krytycznych czasowo.
- **Tryb przerwania (Interrupt) IRQ** – wykorzystywany do obsługi zwykłych przerwań.
- **Tryb nadzorcy (Supervisor) –** wykorzystywany przez system operacyjny.
- **Abort Mode** – tryb ten jest uruchamiany, np. gdy procesor napotka instrukcję odwołująca się do nieistniejącego obszaru pamięci. W mikrokontrolerach i systemach zawierających blok zarządzania pamięcią MMU, może być wykorzystywany do realizacji pamięci wirtualnej, znanej doskonale z systemów takich jak Windows czy Linux.
- **Tryb systemu operacyjnego (System Mode)** – uprzywilejowany tryb wykorzystywany przez system operacyjny.
- **Undefined Mode** – procesor wchodzi w ten tryb, gdy napotka nieznaną instrukcję.

Zmiana trybu ochrony następuje w momencie wystąpienia sytuacji wyjątkowej. Przykładowo gdy zgłaszane jest przerwanie procesor zmienia tryb na IRQ. Tryb ochrony można także zmienić w trybach uprzywilejowanych na drodze programowej poprzez modyfikację słowa stanu procesora. Natomiast, gdy jesteśmy w trybie użytkownika trybu nie można zmienić w sposób programowy. Gdyby tak nie było wówczas dowolny program użytkownika mógłby zmienić tryb na tryb uprzywilejowany i przejąć pełną kontrolę nad procesorem, co podważało by sens stosowania trybów chronionych. W procesorach ARM stosuje się mechanizm bankowania rejestrów. Polega on na tym, że gdy procesor zmienia tryb ochrony zestaw niektórych rejestrów jest podmieniany na inny. W efekcie tego każdy tryb ochrony posiada swój własny rejestr, który jest widoczny tylko w tym trybie. Mechanizm bankowania jest świetnie znany wszystkim programistom mikrokontrolera 8051, gdzie do dyspozycji mamy 4 banki rejestrów R0...R7 przełączane za pomocą bitów RS0 RS1 w rejestrze PSW. W ARM–ach sytuacja jest bardzo podobna, z tym, że rejestry podmieniane są automatycznie przy zmianie trybu ochrony i nie jest bankowany cały zestaw rejestrów a jedynie ich część. To, które rejestry są bankowane, zależy od trybu ochrony. Zastosowanie mechanizmu bankowania przy zmianie trybu umożliwia np. zrealizowanie osobnego stosu dla każdego trybu ochrony, co dodatkowo podnosi bezpieczeństwo. Jeżeli mielibyśmy tylko jeden stos dla wszystkich trybów, wówczas program użytkownika mógłby np. nie zdjąć ze stosu zawartości jakiegoś rejestru, i działanie systemu operacyjnego mogło by się załamać na skutek zdjęcia przez system ze stosu nieprawidłowych wartości. Tutaj nie ma takiego problemu, ponieważ każdy tryb ochrony posiada swój własny stos i nawet jak program w trybie użytkownika nie zdejmie zawartości jakiegoś rejestru ze stosu, nie będzie to miało wpływu na pracę systemu operacyjnego.

Lucjan Bryndza SQ7FGB
lucjan.bryndza@ep.com.pl