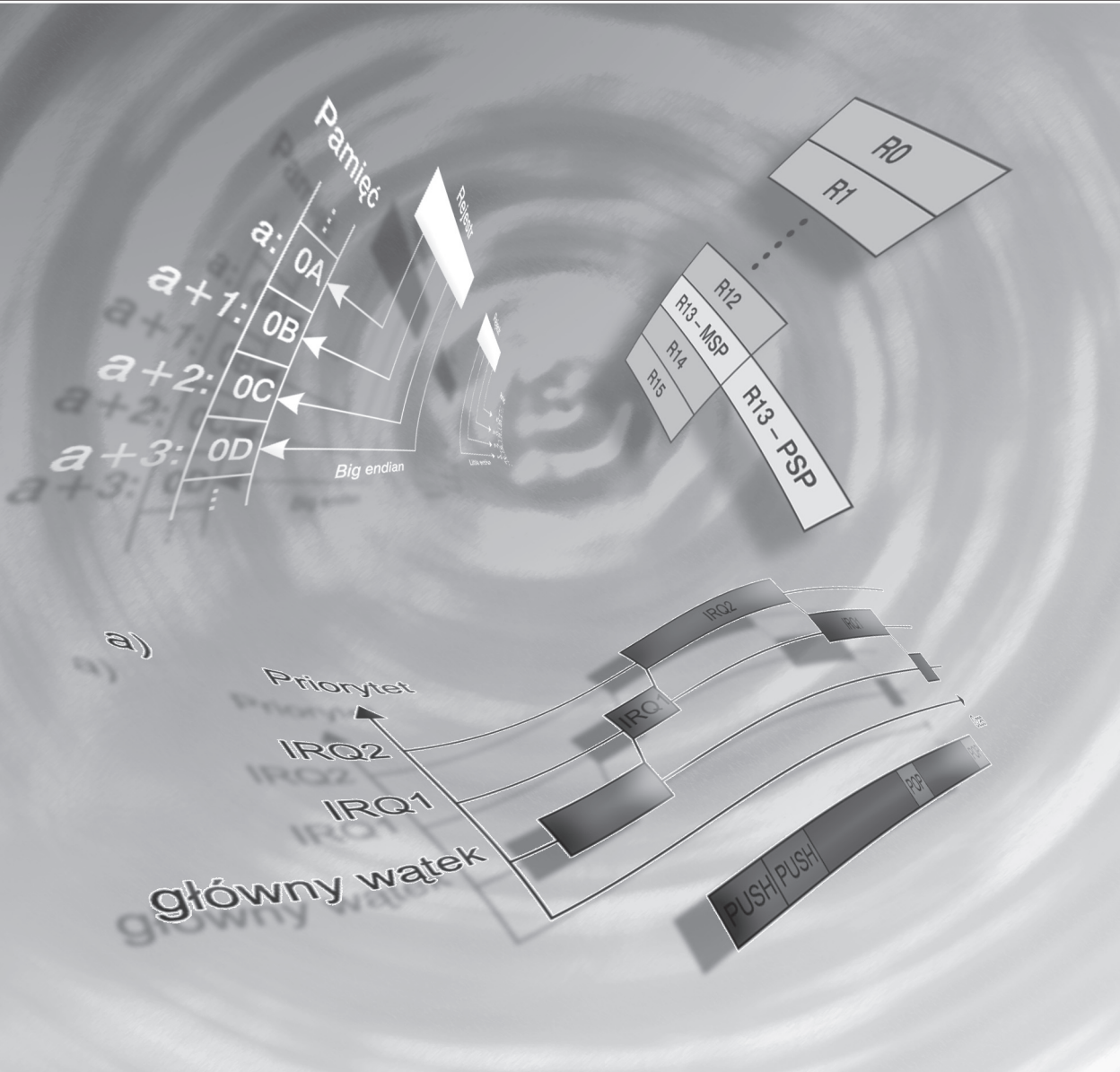


# 1

## Rdzeń Cortex-M3



Od kilku lat na rynku mikrokontrolerów obserwujemy nowe, interesujące zjawisko. O ile jeszcze kilka lat temu każdy większy producent półprzewodników miał w ofercie produkcyjnej mikrokontrolery wyposażone w rdzenie będące własnymi opracowaniami lub mikrokontrolery, w których zastosowano jeden z „klasycznych” rdzeni opracowanych przez rynkowych gigantów (przede wszystkim Intela i Motorolę), to od kilku lat sytuacja ulega zasadniczej zmianie. Obecnie większość liczących się na rynku producentów mikrokontrolerów oferuje lub wprowadza do swojej oferty mikrokontrolery wyposażone w rdzenie opracowane przez firmę ARM. Podbój rynku mikrokontrolerów przez rdzenie opracowane przez ARM rozpoczął się od opracowanego w roku 1993 rdzenia ARM7TDMI. Jego rynkowa obecność niedługo się zapewne zakończy za sprawą znacznie młodszego (z roku 2004) opracowania firmy ARM: rdzenia Cortex-M3, który zastosowano m.in. w mikrokontrolerach STM32 produkowanych przez firmę STMicroelectronics.

Z perspektywy konstruktorów systemów mikroprocesorowych taki stan rzeczy ma pozytywny wpływ na łatwość projektowania: konkurencja pomiędzy producentami mikrokontrolerów jest duża, podobnie jak łatwość zastąpienia mikrokontrolerów pochodzących od jednego producenta przez inne, znacznie prościej można przenosić aplikacje pomiędzy mikrokontrolerami pochodzącymi od różnych producentów. Na pewno w krótkim czasie wzrośnie także liczba procedur programowych dostępnych bezpłatnie. Niebagatelne znaczenie ma także możliwość doboru mikrokontrolera optymalnego pod względem wyposażenia i możliwości w stosunku do ceny. Dotychczas konieczne były – czasami dość bolesne – kompromisy.

## 1.1. Firma ARM i jej wyroby

Firma ARM ma swoje początki w roku 1990, kiedy to z porozumienia kilku przedsiębiorstw (Apple Computer, Acorn Computer Group, VLSI Technology) utworzono firmę o nazwie Advanced RISC Machines. Miało to miejsce w Cambridge w Wielkiej Brytanii. Dopiero w roku 1998 nazwę zmieniono i aktualnie firma nazywa się ARM Holdings.

Jednym z czynników, które wpłynęły na zmianę nazwy, była fonetyczna dwuznaczność słowa RISC w języku angielskim. W języku Shakespeare’a brzmi to jak *risk*, czyli ryzyko. Specjaliści od wizerunku doszli do wniosku, skądinąd słusznego, że może to się źle kojarzyć. O ile w inżynierskim świecie pierwotna nazwa nie miała większego znaczenia, o tyle na rynkach finansowych, gdzie ludzkie emocje grają ogromną rolę, dwuznaczna nazwa mogła być przyczyną niepowodzeń, poza tym warto też było skrócić dotychczasową, nieco przydługą, nazwę. I tak powstała, znana obecnie na całym naszym globie firma ARM Holdings.

Firma ARM zajmuje się projektowaniem układów cyfrowych, przy czym nie jest producentem półprzewodników, opracowuje i sprzedaje tzw. bloki własności intelektualnej IP (*Intellectual Property*). Najbardziej znanym wyrobem tej firmy są rdzenie mikroprocesorów i mikrokontrolerów, pośród których szczególną popularność zdobyły m.in. rodziny ARM7 i ARM9, obecnie zastępowane przez rdzenie z rodziny Cortex. Klientami ARM Holdings są największe na świecie firmy produkujące układy scalone (np. STMicroelectronics, Texas Instruments, Freescale, Toshiba, Zilog, Maxim



Założyciele firmy Advanced RISC Machines

itd.). W konsekwencji ARM nie produkując fizycznie żadnych układów ma ogromny wpływ na rozwój całego przemysłu półprzewodnikowego zajmującego się zaawansowanymi układami cyfrowymi.

Odwiedzając stronę internetową firmy ARM ([www.arm.com](http://www.arm.com)) i przeglądając dokumentację rdzeni można się nieco pogubić. Producenci mikrokontrolerów w swoich materiałach podają nazwy zastosowanych rdzeni, natomiast ARM często posługuje się nazwą zastosowanej w nich architektury. Dla przykładu: w rdzeniu ARM7TDMI zastosowano architekturę ARMv4T, natomiast w rdzeniach Cortex firma zastosowała architekturę ARMv7M. Na **rysunku 1.1** przedstawiono kilka podobnych przypadków.

ARM7	→	ARMv4
ARM9	→	ARMv5
ARM11	→	ARMv6
Cortex	→	ARMv7

**Rys. 1.1.** Nazwy wybranych rodzin rdzeni opracowanych przez ARM Holdings (z lewej strony) i nazwy zastosowanych w nich architektur (z prawej strony)

## 1.2. Rodzina rdzeni Cortex

Rodzina Cortex składa się z trzech podrodzin, o budowie zoptymalizowanej pod kątem różnych aplikacji:

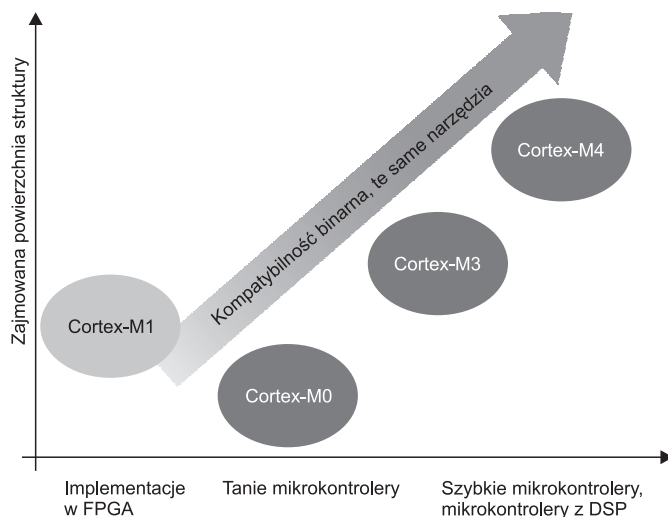
- Cortex-Ax – jest przeznaczona dla wymagających aplikacji z systemami operacyjnymi, takimi jak Symbian, Linux i Windows Embedded, które wymagają dużej mocy obliczeniowych, obsługi pamięci wirtualnej z MMU (*Memory Management Unit*) lub implementacji wirtualnej maszyny Javy,
- Cortex-Rx – przeznaczona do implementowania w systemach, w których krytyczny jest czas reakcji (np. ABS i inne aplikacje samochodowe),
- Cortex-Mx – podrodzina zoptymalizowana pod kątem minimalizacji ceny przy zachowaniu dużej wydajności, przeznaczona do zastosowań konsumenckich i przemysłowych.

W każdym przypadku litera x oznacza liczbę, która precyzyjnie określa wersję rdzenia z danej podrodziny.

Poza wymienionymi różnicami, rdzenie z grupy M obsługują wyłącznie polecenia zgodne z listą instrukcji Thumb-2, w odróżnieniu od pozostałych rdzeni Cortex, które są przystosowane także do dekodowania klasycznych instrukcji ARM.

Najnowszy w rodzinie Cortex-M4 jest jednym z najszybszych wariantów w rodzinie rdzeni mikrokontrolerowych („M”), do tego wyposażonym w bloki sprzętowe pozwalające realizować obliczenia charakterystyczne dla techniki DSP. W chwili przygotowywania drugiego wydania książki firma STMicroelectronics nie ujawniała informacji o planach wdrożenia do produkcji mikrokontrolerów wyposażonych w ten rdzeń, ale można się spodziewać, że takie się niebawem pojawią.

Rdzeń Cortex-M3 jest historycznie pierwszym opracowaniem z rodziny Cortex’ów opracowanych przez firmę ARM Holdings. W chwili wydania książki są dostępne cztery wersje rdzenia Cortex-M: M0, M1, M3 oraz M4 (**rysunek 1.2**). Przedostatni



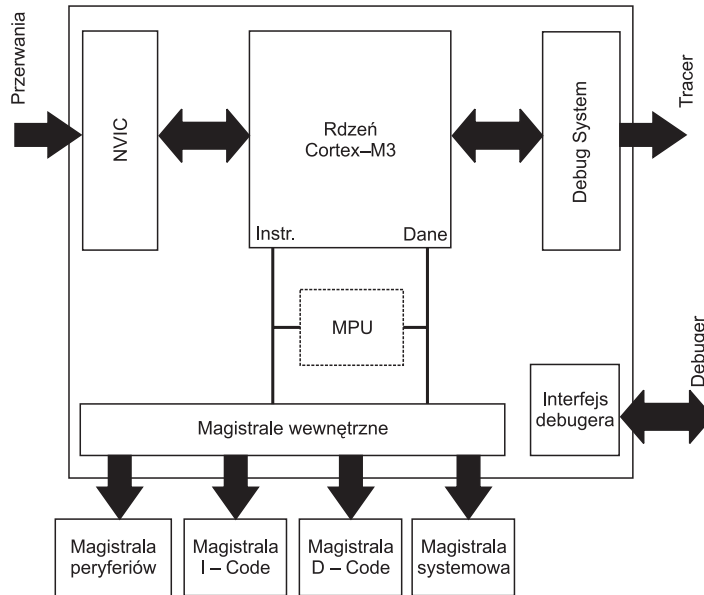
**Rys. 1.2.** Porównanie najważniejszych cech rdzeni z rodziny Cortex



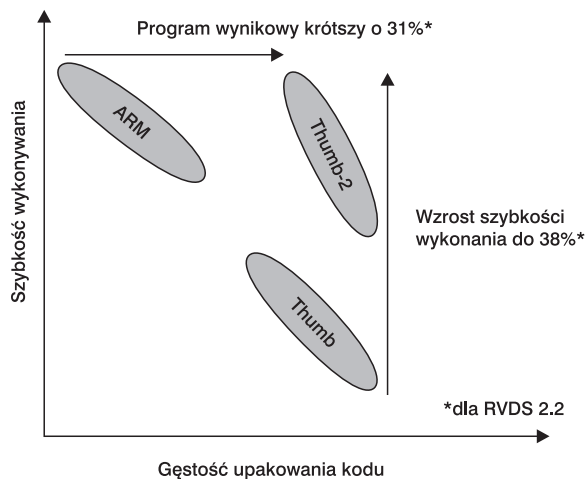
mapę poleceń assemblerowych „rozumianych” przez poszczególne rdzenie z rodziny Cortex-M.

### 1.3. Ogólne spojrzenie na architekturę rdzenia Cortex-M3

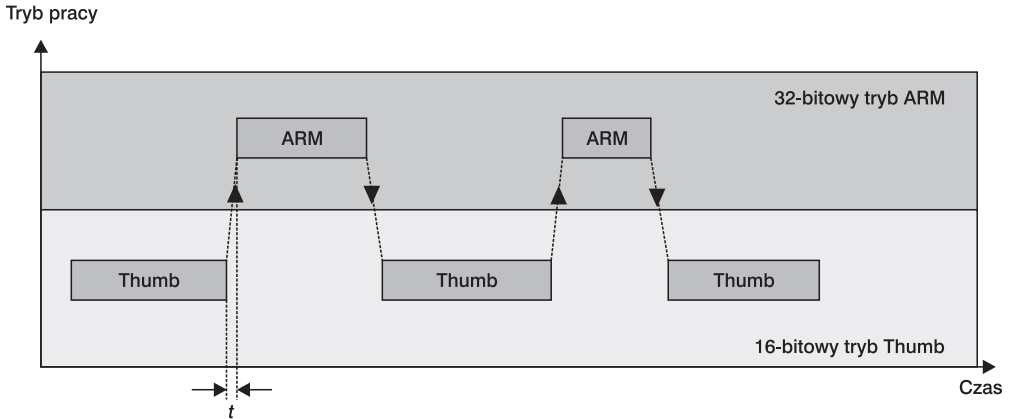
Na **rysunku 1.4** przedstawiono w uproszczeniu budowę rdzenia Cortex-M3. Jak wspomniano, obsługują one listę instrukcji Thumb-2, która zawiera polecenia umożliwiające operacje zarówno na danych 16- jak i 32-bitowych. Zaletą tej listy jest większa, w porównaniu do rozkazów ARM, gęstość upakowania poleceń (zaj-



Rys. 1.4. Budowa rdzenia Cortex-M3



Rys. 1.5. Porównanie cech list instrukcji ARM, Thumb i Thumb-2



Rys. 1.6. Przełączanie pomiędzy trybami ARM i Thumb w rdzeniach ARM7 i ARM9

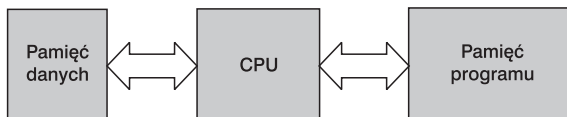
mują mniej miejsca w pamięci Flash) i szybsze działanie programów (nawet do 1,25 DMIPS/MHz) – niż miało to miejsce w przypadku znanej ze starszych rozwiązań listy Thumb – zapisanych w postaci listy poleceń Thumb-2 (rysunek 1.5).

Nie ma zatem potrzeby, tak jak to było w przypadku rdzeni ARM7 i ARM9, przełączania się pomiędzy 16-bitowym trybem Thumb a 32-bitowym ARM, co było typowym zabiegiem programistów, pozwalającym oszczędzać pamięć programu tam, gdzie prędkość wykonywania programów nie była krytyczna. Na rysunku 1.6 symbolicznie zaznaczono czas  $t$ , dla podkreślenia, że przełączanie trybów pracy wprowadza pewne opóźnienie w wykonywaniu programu.

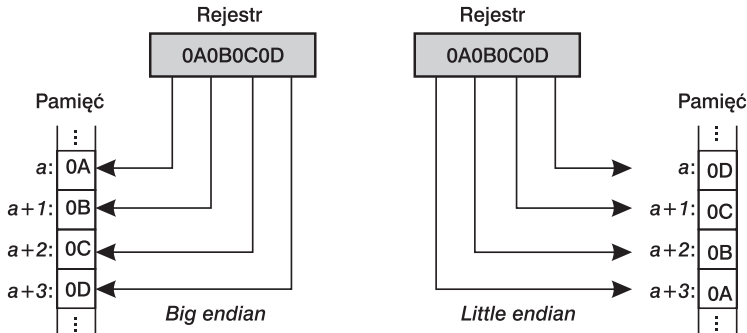
Lista Thumb-2 ma pewne rozszerzenia w stosunku do listy ARM (np. sprzętowe dzielenie), problemem jest to, że rdzenie Cortex-M3 nie są kompatybilne ze swoimi poprzednikami. Niestety oznacza to, że nie można bezpośrednio uruchomić programu napisanego np. dla ARM7 na układzie wyposażonym w rdzeń Cortex-M3.

Rdzeń Cortex-M3 zaprojektowano z wykorzystaniem architektury harwardzkiej, co oznacza, że magistrale: pamięci programu i pamięci danych są od siebie oddzielone, co zilustrowano na rysunku 1.7.

Takie podejście pozwala na dostęp do pamięci programu i pamięci danych w tym samym czasie, pamięci te współdzielą tę samą przestrzeń adresową, a więc nie można zaadresować, jak mogłoby się wydawać do 8 GB, a „tylko” do 4 GB w każdym obszarze. Zastosowanie architektury harwardzkiej pozwoliło na zwiększenie wydajności jednostki centralnej, ponieważ dostęp do pamięci danych może nie mieć dużego wpływu na wykonywanie programu. Może, ale nie musi. Bolączką pamięci Flash wbudowanych w mikrokontrolery STM32F103 jest ich relatywnie



Rys. 1.7. Istotnym elementem architektury harwardzkiej (zastosowanej w rdzeniu Cortex-M3) jest rozdzielenie magistrali zapewniających dostęp do pamięci programu i pamięci danych



Rys. 1.8. Sposoby zapisu danych w formatach *big endian* i *little endian*

niska częstotliwość maksymalnego sygnału zegarowego, przynajmniej w odniesieniu do maksymalnej częstotliwości rdzenia. Pamięć Flash nie może być taktowana szybciej niż sygnałem 24 MHz, a rdzeń może pracować z prędkością do 72 MHz. Gdy kolejno po sobie następujące instrukcje wykonują operacje na pamięci Flash, to wtedy wykonywanie programu może zostać opóźnione ze względu na wolniejszą pracę pamięci. Więcej informacji na temat współpracy rdzenia mikrokontrolera i wewnętrznej pamięci Flash zamieszczono w rozdziale 3.

Mikrokontrolery wyposażone w rdzenie Cortex-M3 mogą używać kodowania zarówno *big endian* jak i *little endian*, co oznacza różne kolejności zapisu bajtów do pamięci. Na **rysunku 1.8** przedstawiono sposób zapisu bajtów w obydwu standardach: jak widać *big endian* polega na tym, że najbardziej znaczący bajt jest zapisywany jako pierwszy, natomiast w *little endian* jako pierwszy zapisywany jest mniej znaczący bajt. Dzięki dostępności obydwu trybów zapisu do pamięci można projektowaną aplikację dostosować do własnych wymagań, przykładowo aplikacje wykorzystujące sieć Ethernet mogą od razu pracować w kodowaniu *big endian*, przez co nie jest potrzebna zmiana kolejności nadchodzących bajtów.

W przypadku bardziej zaawansowanego i wymagającego systemu rdzenie z rodziny Cortex mogą być wyposażone w jednostkę ochrony pamięci (MPU – *Memory Protection Unit*). Architektura Cortex'ów zawiera w swej strukturze już zaimplementowane bloki do debugowania z obsługą pułapek (*breakpoint*) i punktów podglądu (*watchpoint*).

Rdzenie Cortex obsługują dwa tryby pracy o różnych stopniach uprzywilejowania:

- tryb uprzywilejowany,
- tryb użytkownika.

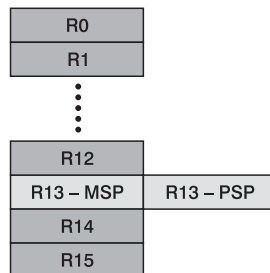
Po zerowaniu jednostka centralna zawsze uruchamia się w trybie uprzywilejowanym, a do jego zmiany służy najmłodszy bit rejestru specjalnego CONTROL. Praca w tym trybie umożliwia nieograniczony dostęp do zasobów rdzenia. Gdy program jest wykonywany w trybie użytkownika, niektóre zasoby rdzenia nie są dostępne, dzięki czemu aplikacja użytkownika nie ma dostępu do kluczowych – ze względu na niezawodność – elementów systemu, takich jak m.in. niektóre obszary pamięci. Dzięki takiemu rozwiązaniu łatwiej jest tworzyć na platformie Cortex niezawodnie działające systemy operacyjne, ponieważ *kernel* pracujący w trybie uprzywilejowa-



nym ma dostęp do wszystkich zasobów, natomiast aplikacja uruchamiana w systemie pracuje w bezpiecznym trybie nieuprzywilejowanym (użytkownika).

## 1.4. Rejestry podstawowe

Rdzeń Cortex-M3 wyposażono w szesnaście rejestrów podstawowych (R0 do R15), przy czym trzynaście z nich (od R0 do R12) jest rejestrami ogólnego przeznaczenia (*General Purpose Register*) – **rysunek 1.9**. Trzeba zaznaczyć, że większość instrukcji 16-bitowych może operować tylko na ośmiu młodszych rejestrach z zakresu R0 do R7, a tylko niektóre z instrukcji 16-bitowych mogą pracować na rejestrach „starszych”.



**Rys. 1.9.** Rejestry rdzenia Cortex-M3

Rejestr R13 jest wskaźnikiem stosu. Podzielono go na dwa oddzielne rejestry bankowane, co oznacza, że w danym momencie jest widoczny tylko jeden. Rejestr R13 składa się z:

- MSP – *Main Stack Pointer* – domyślnego rejestru używanego przez przerwania i jądra systemów operacyjnych, pracujących w trybie uprzywilejowanym,
- PSP – *Process Stack Pointer* – używanego przez program użytkownika uruchomionym „pod skrzydłami” systemu operacyjnego.

Model wykorzystujący dwa stosy umożliwia tworzenie aplikacji o wyższym poziomie bezpieczeństwa, ponieważ program użytkownika nie ma dostępu do stosu systemu operacyjnego, a co za tym idzie ma ograniczony negatywny wpływ na stabilność systemu.

Pozostałe rejestry: R14 – tzw. *Link Register* – zawiera w sobie adres powrotu, a R15 – licznik rozkazów – zawiera adres aktualnie wykonywanej instrukcji. Może być zapisywany w celu sterowania wykonywaniem programu.

Oprócz wyżej wymienionych rejestrów rdzenie Cortex-M3 wyposażono także w rejestry specjalne: *Program Status Register*, *Interrupt Mask Register*, *Control Register*. Służą one do sterowania pracą rdzenia i sposobem wykonywania programu, a ich zawartość może być modyfikowana tylko za pomocą instrukcji specjalnych, nie mogą być modyfikowane podczas normalnej pracy rdzenia.

## 1.5. Przestrzeń adresowa

Przestrzeń adresowa obsługiwana przez rdzenie Cortex-M3, wynosząca w sumie 4 GB, jest podzielona na segmenty, m.in.: segment pamięci programu, pamięci SRAM i pamięci zewnętrznej RAM, urządzeń peryferyjnych itd. Mapę pamięci przedstawiono na **rysunku 1.10**.

Mimo, że przestrzeń adresową zdefiniował producent, istnieje możliwość odmiennego jej skonfigurowania. Przykładem może być np. umieszczenie programu w pamięci SRAM, zewnętrznej pamięci RAM, jak również np. umieszczenie danych w przestrzeni adresowej pamięci programu. Tego typu rozwiązania są mało efektywne, ale podczas pisania i testowania aplikacji czasem bardzo wygodne.

Obszar systemowy <b>0,5 GB</b>	0xFFFFFFFF
Prywatne peryferia (NVIC, MPU itp.)	0xE0000000
	0xDFFFFFFF
Urządzenia zewnętrzne <b>1 GB</b>	
	0xA0000000
	0x9FFFFFFF
Zewnętrzna RAM <b>1 GB</b>	
	0x60000000
	0x5FFFFFFF
Peryferia <b>0,5 GB</b>	
	0x40000000
	0x3FFFFFFF
SRAM <b>0,5 GB</b>	
	0x20000000
	0x1FFFFFFF
Program <b>0,5 GB</b>	
	0x00000000

Rys. 1.10. Mapa pamięci obsługiwana przez rdzeń Cortex-M3

### 1.5.1. **Bit-band, czyli obszary o dostępie atomowym**

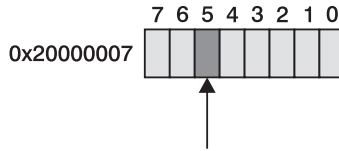
Podtytuł z pewnością wymaga wyjaśnienia: co to bowiem znaczy „obszar o dostępie atomowym”? Kiedyś uważano, że atom jest najmniejszą, niepodzielną cząstką materii. Rozwój nauki wykazał błędność tego założenia, ale w celu wytłumaczenia określenia *dostęp atomowy*, trzymajmy się starszej wersji poglądu na budowę materii.

Analogią do atomu w systemach cyfrowych jest bit. Po tym porównaniu łatwo już zrozumieć istotę obszarów o *dostępie atomowym*. Są to obszary pamięci, gdzie jest możliwy dostęp do pojedynczych bitów, a nie jak to jest zazwyczaj do całych bajtów.

W przestrzeni adresowej rdzenia Cortex M3 znajdują się dwa obszary *bit-band*. Pierwszy w regionie pamięci RAM, a drugi w regionie urządzeń peryferyjnych. Dla pamięci RAM obszar, w którym jest możliwy *dostęp atomowy* rozpoczyna się od adresu 0x20000000, natomiast dla urządzeń peryferyjnych jest to adres 0x40000000.

Pomysł obszarów *bit-band* jest efektem dążenia do maksymalnej optymalizacji pracy rdzenia. Często zdarza się, że aplikacja musi zmienić stan tylko jednego bitu w danym miejscu w pamięci. Standardowo odbywa się to za pomocą trzech elementarnych operacji:

- odczytu komórki pamięci do rejestru,
- ustawienia interesującego bitu,
- przepisania zmodyfikowanej wartości z rejestru do komórki pamięci.



**Rys. 1.11.** Bit do którego wyznaczamy w przykładzie *dostęp atomowy*

Wykorzystanie *dostępu atomowego* znacznie upraszcza sprawę, należy wykonać tylko operację zapisu (lub odczytu) pod odpowiedni adres. Zapis musi się odbyć na adres obliczony na podstawie adresu interesującego słowa, adresu bazowego regionu *bit-band*, numeru bajtu oraz pozycji bitu w bajcie. Obliczenia adresu dostępu atomowego przedstawiono poniżej.

Założmy, że chcemy ustawić bit 5 komórki (bajtu) o adresie 0x20000007 (zaznaczony na **rysunku 1.11**) – czyli jest to obszar *bit-band* w pamięci RAM. Adres tego bitu jest obliczany z zależności:

$$adres\_bitu = poczatek\_regionu\_bit\_band + (przesuniecie\_bajtu * 32) + (numer\_bitu * 4),$$

gdzie:

*adres\_bitu* – na ten adres należy dokonać zapisu/odczytu,

*poczatek\_regionu\_bit\_band* – dla obszaru pamięci RAM jest to 0x22000000,

*przesuniecie\_bajtu* – *offset* w stosunku do początku regionu *bit-band*, w omawianym przypadku ma on wartość 0x07,

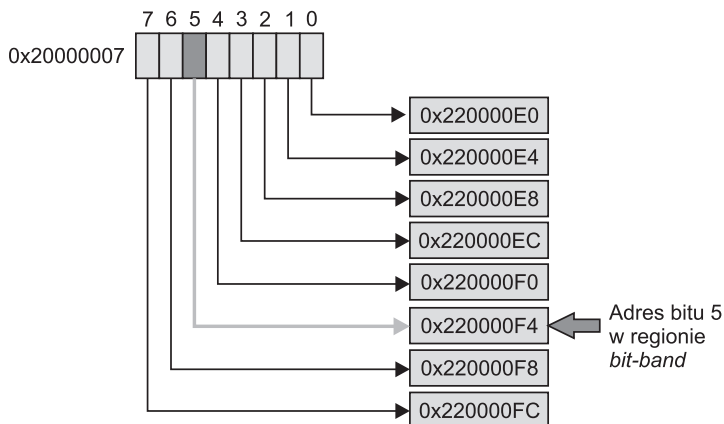
*numer\_bitu* – pozycja bitu, do której chcemy uzyskać *dostęp atomowy*.

Wykorzystując powyższe informacje podstawiamy dane:

$$adres\_bitu = 0x22000000 + (0x07 * 0x20) + (0x05 * 0x04)$$

Wszystkie liczby są w systemie heksadecymalnym. Po wykonaniu obliczeń otrzymujemy:

$$adres\_bitu = 0x220000F4$$



**Rys. 1.12.** Mapowanie obszarów *bit-band*

Zapisując na ten adres wartość 0 lub 1 dokonamy (odpowiednio) skasowania lub ustawienia bitu 5 w bajcie o adresie 0x20000007. Z powyższych informacji i obliczeń wynika, że każdy bit z obszaru o *dostępie atomowym* ma przypisany zupełnie inny adres – dla pierwszego bitu spod adresu 0x20000000 jest to adres 0x22000000. Dla omawianego przypadku zilustrowano to na **rysunku 1.12**.

## 1.6. Sterownik przerw NVIC

Jednym z istotniejszych zadań większości systemów mikroprocesorowych jest reakcja na zdarzenia ze świata zewnętrznego. Konstrukcje typowych mikrokontrolerów są zoptymalizowane pod względem obsługi zdarzeń i przerw. W rdzeniu Cortex-M3 wbudowano sprzętowy kontroler przerw – NVIC (*Nested Vectored Interrupt Controller*) i jest on nieodłącznym elementem tej architektury.

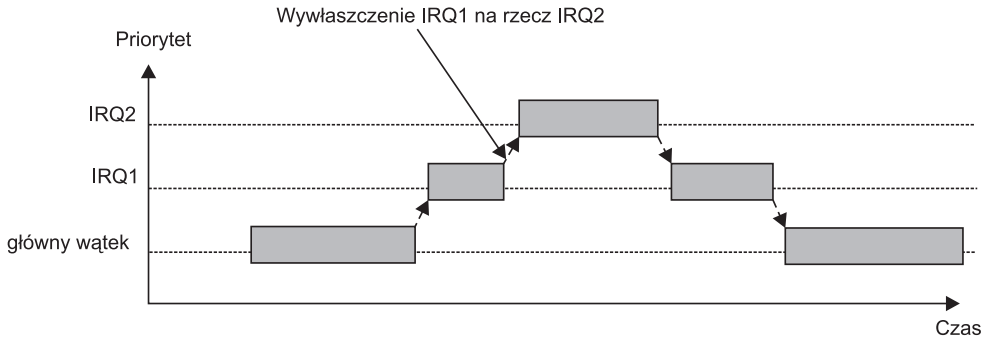
Celem stosowania NVIC jest – po pierwsze – uproszczenie obsługi (zagnieżdżonych) przerw, a po drugie – skrócenie opóźnień w obsłudze przerw. Jak wiadomo, w rdzeniach ARM7/ARM9 dostępne były dwa rodzaje przerw (IRQ i FIQ), a ich obsługa była dość skomplikowana. Inżynierowie z firmy ARM zauważyli to i w rdzeniach Cortex mamy do dyspozycji NVIC, który znacznie upraszcza obsługę przerw.

Rdzeń Cortex-M3 obsługuje do piętnastu przerw (wyjątków) systemowych i do 240 przerw zewnętrznych – to ile tych przerw będzie faktycznie obsługiwanych przez fizyczny układ zależy od producenta mikrokontrolera. Większość wyjątków systemowych i wszystkie przerwy zewnętrzne mogą mieć programowo ustalany priorytet, co przedstawiono na **rysunku 1.13**.

Na uwagę zasługują trzy pierwsze wyjątki: ich priorytety mają wartość ujemną dla podkreślenia tego, że to one są w systemie najważniejsze, przy czym -3 jest priorytetem najwyższym z możliwych – jest to zerowanie systemu.

Numer	Wyjątek	Priorytet
1	Reset	-3
2	NMI	-2
3	Hard Fault	-1
4	MemManage Fault	Programowany
5	Bus Fault	Programowany
6	Usage Fault	Programowany
7...10	Reserved	-
11	SVCALL	Programowany
12	Debug Monitor	Programowany
13	Reserved	-
14	PendSV	Programowany
15	SYSTICK	Programowany
16...255	External Interrupt	Programowany

**Rys. 1.13.** Priorytety przerw rdzenia Cortex-M3



Rys. 1.14. Obsługa przerwań zagnieżdżonych

Kontroler NVIC umożliwia:

- Obsługę przerwań zagnieżdżonych – wszystkie zewnętrzne przerwania i większość przerwań systemowych mogą mieć ustawiane różne priorytety. W momencie wystąpienia przerwania kontroler NVIC porównuje priorytet tego przerwania z priorytetem obecnego zadania. Jeżeli priorytet nowego przerwania jest wyższy od dotychczasowego, wtedy nowe zadanie o wyższym priorytecie dokona wywłaszczenia poprzedniego i jest wykonywane w pierwszej kolejności. Sytuację taką przedstawiono na **rysunku 1.14**.
- Obsługę wektorów przerwań – NVIC oferuje również sprzętowe wsparcie dla wektorów przerwań. W momencie, kiedy przerwanie zostaje przyjęte do realizacji adres funkcji obsługi przerwania (ISR) jest pobierany z wektora w pamięci. Nie ma potrzeby programowego wyznaczania adresu ISR, dzięki temu czas potrzebny na obsłużenie przerwania jest krótszy.
- Dynamiczne zmiany priorytetów przerwań co oznacza, że priorytety przerwań mogą być zmieniane programowo podczas pracy mikrokontrolera. Przerwanie, które jest w danym momencie obsługiwane, zostaje zablokowane przed zmianą priorytetu dopóki nie opuści ISR, zatem nie występuje niebezpieczeństwo wielokrotnego obsłużenia tego samego przerwania podczas zmiany priorytetu.
- Zmniejszenie opóźnień związanych z przerwaniami – rdzeń Cortex-M3 zoptymalizowano pod kątem możliwie najkrótszych opóźnień czasowych obsługi przerwania. Jedną z podstawowych czynności mających na celu skrócenie tego czasu jest automatyczne zapisywanie i przywracanie kontekstu zadania, czyli zawartości kluczowych dla tego zadania rejestrów. W dalszej części tego rozdziału omówiono dokładniej poszczególne mechanizmy skracające opóźnienia w obsłudze przerwań (*Tail-Chaining*, *Late Arrival*, przerywanie operacji POP).
- Maskowanie przerwań – przerwania i wyjątki (*exceptions*) mogą być maskowane na podstawie ich priorytetów lub maskowane całkowicie poprzez odpowiednie użycie rejestrów maskujących. Ma to szczególne znaczenie dla zadań, w których wykonaniu czas jest parametrem krytycznym. W tym przypadku takie przerwania będą obsługiwane bez wywłaszczenia. Przykładowo można blokować przerwania od danego priorytetu w dół.

## 1.7. Lista rozkazów Thumb-2

Jedną z najważniejszych zalet rdzenia Cortex-M3 jest zdolność do obsługi instrukcji na danych 16- i 32-bitowych bez konieczności stosowania jakichkolwiek zabiegów (lista instrukcji Thumb-2). Takie rozwiązanie pozwoliło na zmniejszenie objętości wynikowego kodu oraz na zwiększenie wypadkowej prędkości wykonywania programu.

W celu możliwie jak najlepszego wykorzystania możliwości drzemiących w zestawie instrukcji Thumb-2, stworzono pewną dedykowaną do tego celu odmianę asemblera – *Unified Assembler Language* (UAL) – zunifikowany język asemblera. Dzięki temu operowanie instrukcjami 16- i 32-bitowymi stało się dużo prostsze i bardziej przejrzyste.

W zestawie instrukcji Thumb-2 niektóre operacje mogą być wykonywane zarówno jako 16- jak i 32-bitowe. Przykładem może być dodawanie wartości liczbowej do rejestru:

```
ADDS R0, #5      ;Zostanie użyta 16-bitowa instrukcja Thumb (domyślna, dla mniejszego
                  ;rozmiaru kodu)
ADDS.N R0, #5    ;Zostanie użyta również 16-bitowa instrukcja (N = Narrow)
ADDS.W R0, #5    ;Tutaj będzie zastosowana 32-bitowa instrukcja Thumb - 2 (W = Wide)
```

Bez użycia sufiksu *W* (*wide*) lub *N* (*narrow*) kompilator wybierze domyślny rodzaj instrukcji, z reguły będzie to 16-bitowa instrukcja z zestawu Thumb-2.

Z przedstawionych informacji widać, że zarówno twórcy kompilatorów jak i programiści, którzy podejmują się pisania części aplikacji w asemblerze, mają spore pole do popisu. Możliwości optymalizacji kodu są ogromne, choć jest to oczywiście praco- i czasochłonne. Tak więc decyzje dotyczące optymalizacji sposobu pisania kodu aplikacji powinny być podejmowane przy braniu pod uwagę zarówno czynników czasowych jak i ekonomicznych.