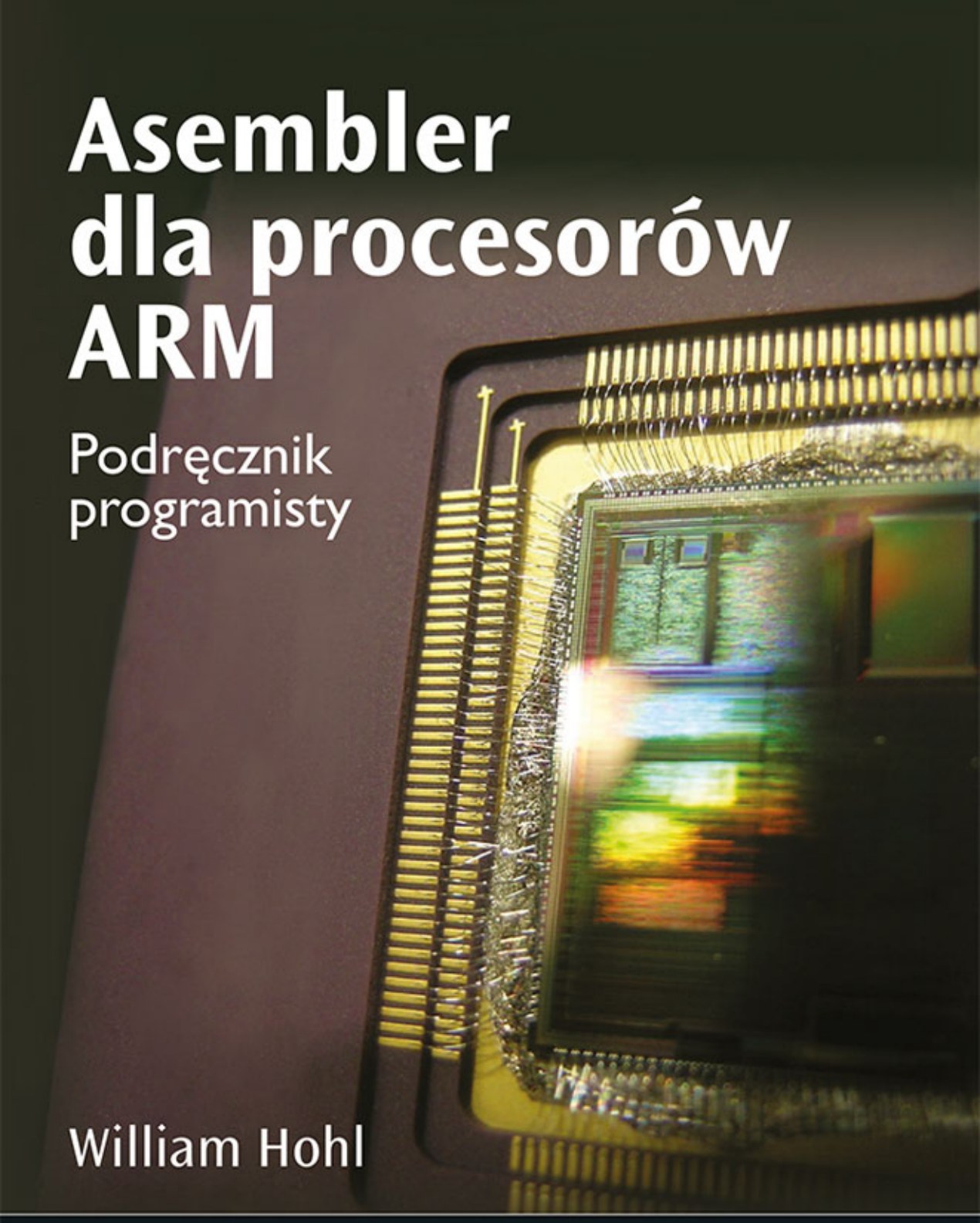
Asembler dla procesorów ARM

Podręcznik programisty



William Hohl

CRC Press
Taylor & Francis Group

Helion

Tytuł oryginału: ARM Assembly Language: Fundamentals and Techniques

Tłumaczenie: Paweł Gonera

ISBN: 978-83-246-9319-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/asarpp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 7

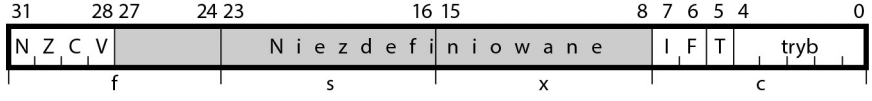
Operacje logiczne i arytmetyczne

7.1. WSTĘP

Jest to długi rozdział, ale są ku temu powody. Operacje arytmetyczne są prawdopodobnie jednymi z najczęściej używanych typów instrukcji, szczególnie jeżeli pisane oprogramowanie manipuluje dużymi porcjami danych, na przykład przychodzącym sygnałem audio. Algorytmy graficzne, algorytmy przetwarzania mowy, kontrolery cyfrowe oraz algorytmy przetwarzanie dźwięku wymagają dużej liczby obliczeń, więc ważne jest, aby poznać dostępne typy danych oraz sposoby wykonania operacji w możliwie najkrótszym czasie i (lub) z wykorzystaniem najmniejszej ilości pamięci. Zaczniemy od przedstawienia znaczników, zaprezentujemy podstawowe instrukcje arytmetyczne, a rozdział zakończymy przeglądem arytmetyki wykorzystującej ułamki. Gdy dobrze zrozumiesz koncepcje znajdujące zastosowanie w cyfrowej arytmetyce, będziesz w stanie przejść do bardziej zaawansowanych funkcji, takich jak funkcje trygonometryczne, wykładnicze oraz pierwiastki.

7.2. ZNACZNIKI I ICH WYKORZYSTYWANIE

Jak pamiętasz z rozdziału 2., rejestr Current Program Status Register zawiera znaczniki, dane o bieżącym stanie komputera oraz trybie pracy przedstawione jako wartości bitów pokazane na rysunku 7.1. W górnej części rejestru dostępne są cztery bity pozwalające określić, czy instrukcja powinna być wykonana warunkowo, czy nie. Znaczniki są ustawiane i kasowane na podstawie dwóch operacji: albo instrukcji używanej specjalnie do ustawienia lub wyczyszczenia znaczników, albo instrukcji, która została „poinformowana", przez dołączenie do jej kodu mnemonicznego litery „S", o tym, że na podstawie wyniku jej wykonania mają być ustawione bity. Na przykład `EORS` oblicza różnicę symetryczną, a następnie ustawia znaczniki, ponieważ w tej instrukcji ustawiony jest bit S. Bitu tego można użyć we wszystkich instrukcjach ALU (jednostki arytmetyczno-logicznej), więc możemy sterować tym, czy znaczniki są ustawiane, czy nie. Przyjrzyjmy się każdemu znacznikowi z osobna — niektóre są bardzo proste, a niektóre wymagają przemyślenia.



RYSUNEK 7.1. Rejestr Current Program Status Register

7.2.1. Znacznik N

Znacznik ten jest używany do sprawdzania wyniku ujemnego. Co to znaczy ujemnego? Definicja wyniku ujemnego bazuje na systemie liczb z uzupełnieniem do dwójki, a jak widziałeś w ostatnich kilku rozdziałach, liczba z dopełnieniem do dwójki jest uważana za ujemną, jeżeli jest ustawiony jej najbardziej znaczący bit. Należy jednak zachować ostrożność, ponieważ możemy dodać do siebie dwie liczby dodatnie i otrzymamy wartość z ustawionym najbardziej znaczącym bitem.

Przykład 7.1.

Dodanie –1 do –2 jest bardzo łatwe, a wynik ma ustawiony najbardziej znaczący bit, jak się tego spodziewaliśmy. W notacji z dopełnieniem do dwójki operacja ta jest reprezentowana jako:

```
  FFFFFFFF
+ FFFFFFFE
----------
  FFFFFFFD
```

Przykład 7.2.

Jeżeli dodamy poniższe wartości, to pomimo że składniki są dodatnie w zapisie z dopełnieniem do dwójki, suma jest ujemna:

```
  7B000000
+ 30000000
----------
  AB000000
```

co oznacza, że czasami można się pomylić. Na początek zwróć uwagę, że ponieważ najbardziej znaczący bit jest ustawiony, konieczne jest ustawienie bitu N, jeżeli nasza instrukcja `ADD` powinna ustawić znaczniki (jak pamiętamy, nie musi tego robić). Po drugie, jeśli nie korzystamy z liczb z dopełnieniem do dwójki, to prawdopodobnie nie musimy przejmować się wartością bitu N. Na koniec warto pamiętać, że w reprezentacji z dopełnieniem do dwójki suma dwóch liczb dodatnich daje większą liczbę dodatnią, ale powyższy wynik pokazuje, że ta suma liczb dodatnich nie może być reprezentowana za pomocą 32 bitów, więc w efekcie powstaje przepełnienie ponad dostępną precyzję liczb. Z tego powodu do pracy z wartościami ze znakiem potrzebujemy więcej znaczników.

7.2.2. Znacznik V

Jeżeli przy wykonywaniu takich operacji jak dodawanie lub odejmowanie określimy wartość znacznika V jako operację `XOR` bitu przeniesienia wchodzącego do najbardziej znaczącego bitu wyniku z bitem przeniesienia wychodzącym z najbardziej znaczącego bitu, to znacznik V

w precyzyjny sposób wskaże nam przepełnienie przy operacjach ze znakiem. Przepełnienie następuje, gdy wynik dodawania, odejmowania lub porównywania jest większy lub równy 2^{31} lub mniejszy niż -2^{31}.

Przykład 7.3.

Dodajmy do siebie dwie wartości ze znakiem, reprezentowane w postaci dopełnienia do dwójki:

```
  A1234567
+ B0000000
----------
 151234567
```

Wynik nie mieści się w 32 bitach. Co ważniejsze, ponieważ liczby są zapisane w postaci dopełnienia do dwójki, a dodajemy do siebie dwie dosyć duże liczby ujemne, to przy przepełnieniu najbardziej znaczący bit 32-bitowego wyniku jest czyszczony (zwróć uwagę na cyfrę 5 w najbardziej znaczącym bajcie wyniku).

Przeanalizujmy ponownie przykład 7.2. Gdy dodaliśmy 0x7B000000 do 0x30000000, wynik zmieścił się w 32 bitach. Jednak wynik może być interpretowany jako ujemny, kiedy dodamy dwie liczby dodatnie, więc czy jest to przypadek przepełnienia? Odpowiedź brzmi: tak.

7.2.3. Znacznik Z

Jest to jeden z najłatwiejszych do zrozumienia znaczników, ponieważ informuje nas, że wynikiem operacji było zero, czyli wszystkie 32 bity były ustawione na zero.

7.2.4. Znacznik C

Znacznik przeniesienia jest ustawiany, jeżeli wynik dodawania jest większy lub równy 2^{32}, jeśli wynik odejmowania jest dodatni lub w wyniku operacji na cyklicznym rejestrze przesuwnym w instrukcji kopiowania albo logicznej. Przeniesienie jest przydatnym znacznikiem, ponieważ pozwala budować operacje o większej dokładności, na przykład dodawania liczb 64-bitowych, co pokażemy nieco dalej.

7.3. INSTRUKCJE PORÓWNANIA

Oprócz użycia bitu S w instrukcji do ustawiania znaczników służą cztery instrukcje, które nie robią *nic poza* ustawianiem kodów warunku lub sprawdzaniem określonych bitów w rejestrze. Są to:

- `CMP` — porównanie. `CMP` odejmuje wartość rejestru lub literał od wartości rejestru i aktualizuje kody warunku. Za pomocą `CMP` możemy szybko porównać zawartość rejestru z określoną wartością, na przykład na początku lub na końcu pętli.
- `CMN` — porównanie ujemne. `CMN` dodaje wartość rejestru lub literał do wartości rejestru i aktualizuje kody warunku. `CMN` pozwala również szybko sprawdzić zawartość rejestru. Instrukcja ta jest odwrotnością `CMP`, a asembler będzie zastępował instrukcje `CMP`, jeżeli jest to właściwe. Na przykład gdy wpiszemy polecenie:

```
CMP r0, #-20
```

to asembler wygeneruje

```
CMN r0, #0x14
```

- TST — test. TST wykonuje logiczną operację AND z wartością rejestru i aktualizuje kody warunku bez wpływania na znacznik V. Za pomocą TST możemy sprawdzić, czy określone bity rejestru są wyzerowane lub czy co najmniej jeden bit rejestru jest ustawiony.
- TEQ — test odpowiedniości. TEQ wykonuje logiczną operację XOR z wartością rejestru i aktualizuje kody warunku bez wpływania na znacznik V. Za pomocą TEQ możemy sprawdzić, czy dwie wartości są takie same.

Składnia tych funkcji jest następująca:

instrukcja{<*warunek*>} <Rn>, <*operand2*>

gdzie {<*warunek*>} jest jednym z opcjonalnych warunków przedstawionych w rozdziale 8., a <*operand2*> może być rejestrem z opcjonalnym przesunięciem lub wartością. Typowe instrukcje mogą wyglądać następująco:

7

```
CMP    r8, #0          ; r8==0?
BEQ    routine         ; tak, więc przejdź do routine

TEQ    r9, r4, LSL #3
```

Jak pamiętasz, znaczniki kodów warunku są przechowywane w rejestrze Current Program Status Register, razem z informacją o trybie i stanie procesora. Możemy skorzystać z instrukcji MRS (przenieś PSR do rejestru ogólnego) do odczytania znaczników w CPSR i dowolnym SPSR, ponieważ instrukcja ta ładuje kopię całego rejestru stanu procesora do rejestru ogólnego przeznaczenia. Na przykład poniższe dwie instrukcje

```
MRS r0, CPSR
MRS r1, SPSR
```

ładują zawartość CPSR i SPSR do rejestrów odpowiednio r0 i r1. Dzięki temu możemy w dowolny sposób sprawdzać stan znaczników. Nie można jednak wykorzystać rejestru r15 jako docelowego ani nie można odwoływać się do SPSR w trybie User, ponieważ w tym przypadku on nie istnieje. W dokumencie ARM *Architectural Reference Manual* (ARM 2007b) zdefiniowano wynik tej operacji jako UNPREDICTABLE. Trzeba pamiętać, że powinniśmy korzystać z kodów warunku razem z instrukcją skoku (B) lub innymi instrukcjami do tworzenia pętli i warunkowych podprogramów asemblera, bez konieczności odczytywania znaczników, np. tak jak w pokazanej powyżej instrukcji BEQ. Temat ten przedstawiamy dokładniej w rozdziale 8.

7.4. OPERACJE PRZETWARZANIA DANYCH

Jak można oczekiwać, każdy mikroprocesor, nawet ten najprostszy, udostępnia podstawowe operacje, takie jak dodawanie, odejmowanie i przesunięcia, na podstawie których można budować bardziej zaawansowane operacje, jak dzielenie, mnożenie i pierwiastkowanie. Mikroprocesory ARM są zaprojektowane do wykorzystania w zastosowaniach wbudowanych, w których bardzo ważne jest zużycie energii i wielkość układu. W idealnej sytuacji procesor powinien oferować szeroką gamę instrukcji przetwarzania bez użycia zbyt wielu bramek logicznych

i bez zwiększania powierzchni układu. Dzięki połączeniu cyklicznego rejestru przesuwnego, 32-bitowej jednostki arytmetyczno-logicznej (ALU) oraz sprzętowego układu mnożenia procesor ARM7TDMI oferuje bogaty zestaw instrukcji przy niewielkim zużyciu energii.

Przykładowa operacja przetwarzania danych, ADD, wygląda następująco:

```
ADDS{<warunek>} r0, r1, <operand2>
```

gdzie {*<warunek>*} jest jednym z opcjonalnych warunków przedstawionych w rozdziale 8. Drugi operand, *<operand2>*, może być bezpośrednią wartością, rejestrem lub rejestrem ze skojarzonym z nim przesunięciem albo rotacją. Ostatnia opcja okazuje się bardzo wygodna, co pokażemy w kolejnych punktach. Warto też pamiętać, że niedawno składnia została zmodyfikowana na potrzeby UAL (*Unified Assembly Language*) — w starej wersji kodów mnemonicznych tę instrukcję zapisujemy jako ADD{*<warunek>*}S[1].

7.4.1. Operacje logiczne

ARM obsługuje operacje logiki Boolowskiej wykorzystujące dwa operandy zamieszczone w tabeli 7.1. Choć w poprzednich rozdziałach spotkałeś się już z instrukcją MOV, to warto wiedzieć, że instrukcja MOVN może być użyta do odwrócenia wszystkich bitów w rejestrze, ponieważ jako operandu oczekuje negacji z dopełnieniem do jedynki. Bardzo szybkim sposobem na załadowanie reprezentacji liczby –1 w dopełnieniu do dwójki jest logiczne odwrócenie zera, ponieważ 32-bitowa wartość 0xFFFFFFFF to właśnie –1 w notacji z dopełnieniem do dwójki, co można zapisać jako:

```
MOVN r5, #0          ; r5 = –1 w notacji z dopełnieniem do dwójki
```

TABELA 7.1. Operacje logiczne

Instrukcja	Komentarz
AND	Logiczna operacja AND na dwóch operandach.
ORR	Logiczna operacja OR na dwóch operandach.
EOR	Logiczna operacja XOR na dwóch operandach.
MOVN	Przeniesienie z odwróceniem — logiczna operacja NOT na wszystkich bitach.
BIC	Wyczyszczenie bitów — wyczyszczenie wybranych bitów w rejestrze.

Przykłady zastosowania tych operatorów są przedstawione poniżej:

```
AND r1, r2, r3      ; r1 = r2 AND r3
ORR r1, r2, r3      ; r1 = r2 OR r3
EOR r1, r2, r3      ; r1 = r2 XOR r3
BIC r1, r2, r3      ; r1 = r2 AND NOT r3
```

Pierwsze trzy instrukcje są dosyć proste — AND, OR i XOR to podstawowe funkcje logiczne. Czwarta instrukcja jest operacją czyszczenia bitów, która może być używana do wyczyszczenia wybranych bitów w rejestrze. Dla każdego bitu w drugim operandzie wartość 1 czyści odpowiedni

[1] Ta książka została wydana w czasie przechodzenia na UAL; najprawdopodobniej spotkasz się z obydwoma formatami instrukcji.

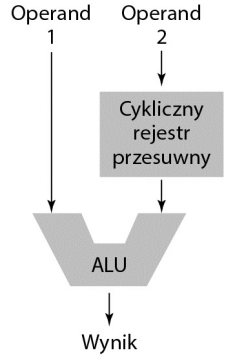
bit pierwszego operandu (rejestr), a 0 pozostawia go bez zmian. Zgodnie z formatem instrukcji przetwarzania danych dla drugiego operandu możemy użyć bezpośrednio wartości. Na przykład:

```
BIC r2, r3, #0xFF000000
```

powoduje wyczyszczenie górnego bajta rejestru r3 i przeniesienie wyniku do rejestru r2.

7.4.2. Przesunięcia i rotacje

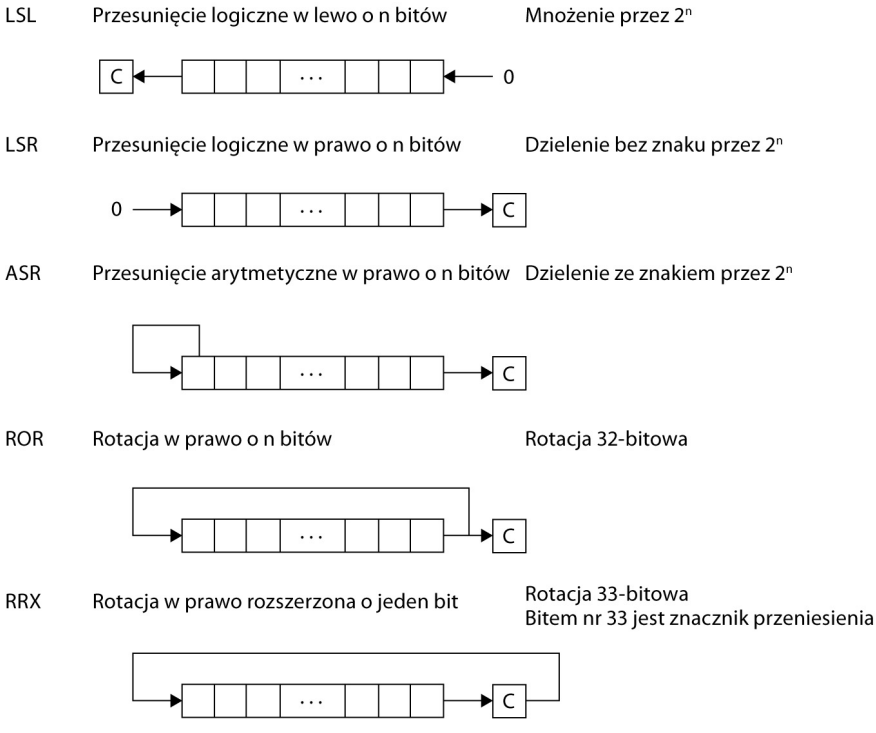
Na rysunku 7.2 jest pokazana część wewnętrznej ścieżki danych procesora ARM7TDMI, w której dane dla instrukcji trafiają na dwie magistrale prowadzące do głównej jednostki arytmetyczno-logicznej. Tylko jedna z tych magistral przechodzi przez cykliczny rejestr przesuwny, który jest dedykowanym blokiem sprzętowym pozwalającym na wykonywanie rotacji lub przesunięć danych w prawo albo w lewo. Z powodu tej asymetrii możemy rotować lub przesuwać tylko jeden operand instrukcji, ale jest to zazwyczaj wystarczające. Przy użyciu kilku dodatkowych instrukcji można ominąć to ograniczenie projektowe. W rzeczywistości pomysł umieszczenia cyklicznego rejestru przesuwnego pomiędzy bankiem rejestrów (termin opisujący fizyczne rejestry od r0 do r15) a głównym ALU pozwala na wykorzystanie stałych 32-bitowych w instrukcjach ALU oraz `MOV`, pomimo że sama instrukcja ma wielkość tylko 32 bitów. Pokazaliśmy to w rozdziale 6. na przykładzie literałów i stałych.



RYSUNEK 7.2. Cykliczny rejestr przesuwny procesora ARM7TDMI

Typy przesunięć i rotacji możliwych do wykonania przez procesory ARM są przedstawione na rysunku 7.3. Dostępne są dwa typy przesunięć logicznych, w których dane są traktowane jako liczby bez znaku, przesunięcie arytmetyczne, w którym dane są traktowane jako liczby ze znakiem, oraz dwa typy rotacji. Brak rotacji w lewo można wytłumaczyć tym, że rotacja w lewo o *m* bitów jest tym samym co rotacja w prawo o (32–*m*) bitów (poza efektem ubocznym zmiany bitu przeniesienia), więc może być realizowana za pomocą tej samej instrukcji. Inną instrukcją, której pozornie brakuje, jest `ASL`, czyli arytmetyczne przesunięcie w lewo. Jednak szybko staje się jasne, że nie będziemy nigdy potrzebować tej instrukcji, ponieważ przesunięcia arytmetyczne muszą zachowywać bit znaku, a przesunięcie danych ze znakiem w lewo pozwoli na to, o ile liczba nie ulegnie przepełnieniu. Na przykład liczba –1 w notacji z dopełnieniem do dwójki ma postać 0xFFFFFFFF, a po wykonaniu przesunięcia w lewo

otrzymujemy 0xFFFFFFFE, czyli –2, co jest prawidłowym wynikiem. Liczba 32-bitowa, na przykład 0x8000ABCD, przesunięta w lewo spowoduje przepełnienie, a w wyniku otrzymamy 0x0001579A, czyli liczbę dodatnią.



RYSUNEK 7.3. Przesunięcia i rotacje

Jeżeli chcesz przenieść dane bez wykonywania dodatkowej operacji, na przykład dodawania, to należy skorzystać z `MOV`. Jak pamiętamy z rozdziału 6., instrukcja `MOV` pozwala przesyłać dane pomiędzy rejestrami, więc dodanie opcjonalnej operacji przesunięcia pozwala nieco ją rozbudować.

Przykład 7.4.

Poniższe instrukcje ilustrują, jak łatwo pisze się przesunięcia i rotacje.

```
MOV r4, r6, LSL #4     ; r4 = r6 << 4 bity
MOV r4, r6, LSL r3     ; r4 = r6 << liczba określona w r3
MOV r4, r6, ROR #12    ; r4 = r6 rotowany w prawo o 12 bitów
          ; r4 = r6 rotowany w lewo o 20 bitów
```

Wszystkie operacje przesunięcia są wykonywane w jednym cyklu. Wyjątkiem są przesunięcia definiowane w rejestrach, które wymagają dodatkowego cyklu, ponieważ w banku rejestrów dostępne są tylko dwa porty odczytu, a wymagany jest dodatkowy odczyt. Przy

wykonywaniu przesunięć licznik przesunięcia może być albo 5-bitową wartością bez znaku, czyli liczbą od 0 do 31, jak w pierwszym przykładzie, albo dolnym bajtem w rejestrze, jak w drugim przykładzie.

Przykład 7.5.

Operacje przesunięcia i logiczne mogą być również używane do przenoszenia danych z jednego bajta do innego. Załóżmy, że musimy przenieść górny bajt z rejestru r2 i umieścić go w dolnym bajcie rejestru r3. Na początek należy przesunąć w lewo zawartość rejestru r3 o 8 bitów. Do tego celu można użyć dwóch instrukcji:

```
MOV r0, r2, LSR #24      ; pobranie górnego bajta z R2 do R0
ORR r3, r0, r3, LSL #8   ; przesunięcie w górę r3 i wstawienie do r0
```

Przykład 7.6. Kody Hamminga

W latach 40. ubiegłego wieku matematyk Richard Hamming opracował i formalnie zdefiniował sposoby nie tylko wykrywania błędów w strumieniu bitów, ale również ich korygowania. Na przykład jeżeli chcemy przesłać 8 bitów danych z komputera poprzez kanał (którym może być kawałek przewodu, łącze optyczne, a nawet interfejs bezprzewodowy) do odbiorcy, mamy nadzieję, że wartość wysłana zostanie odebrana w dokładnie takiej samej postaci. Jeżeli wystąpią błędy, to musimy o tym wiedzieć. Co bardziej interesujące, jeśli istnieje sposób na korektę błędu bitu, to ta część informacji nie musi być ponownie wysyłana. Od lat 40. zainteresowanie kodami korekcji błędów znacznie wzrosło, a nowoczesne schematy, takie jak kod Reeda-Solomona, Binary Golay czy BCH, są opisane w publikacji Rotha (2006). Zagadnienia teoretyczne związane z tym problemem są dosyć skomplikowane, ale prosty kod Hamminga jest dosyć łatwy do napisania, więc przedstawimy tu algorytmy pozwalające wykryć do dwóch błędnych bitów w wartości 8-bitowej. Algorytm ten pozwala również skorygować jeden błędny bit.

Przeanalizujmy pomysł dodania do wartości bitu nazywanego sumą kontrolną, który wskazuje na *parzystość* bitów w tej wartości. Na przykład jeżeli mamy liczbę 7-bitową:

```
1010111
```

i policzymy jedynki w wartości, w naszym przypadku 5, to musimy na początku wartości dodać 1, aby ich liczba była *parzysta*. Naszą nową wartością będzie:

```
11010111
```

Jeżeli dana zostanie przesłana w ten sposób, odbiorca może wykryć błąd, o ile jeden z bitów danych zostanie zmieniony, ponieważ nie będzie się zgadzała *parzystość*. Zwróć uwagę, że jeżeli dwa bity ulegną zmianie, to nie możemy wykryć błędu, ponieważ parzystość zostanie zachowana.

Jeden z typów kodów Hamminga może być utworzony przez użycie czterech sum kontrolnych umieszczonych w strategicznych lokalizacjach. Jeżeli wartość 12-bitowa jest utworzona z 8 bitów danych i czterech sum kontrolnych w pokazany poniżej sposób, to możemy użyć bitów sumy kontrolnej do wykrycia do dwóch błędów w danych i skorygować jeden błędny bit.

Oryginalna wartość 8-bitowa

d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

Zmodyfikowana wartość 8-bitowa

11	10	9	8	7	6	5	4	3	2	1	0
d_7	d_6	d_5	d_4	c_3	d_3	d_2	d_1	c_2	d_0	c_1	c_0

Bity sumy kontrolnej, c_3, c_2, c_1 i c_0, są obliczane w następujący sposób:

- Bit sumy kontrolnej c_0 powinien tworzyć parzystość dla bitów 0, 2, 4, 6, 8 i 10. Inaczej mówiąc, sprawdzamy bit, opuszczamy bit, sprawdzamy bit itd.
- Bit sumy kontrolnej c_1 powinien tworzyć parzystość dla bitów 1, 2, 5, 6, 9 i 10. Inaczej mówiąc, sprawdzamy dwa bity, opuszczamy dwa bity, sprawdzamy dwa bity itd.
- Bit sumy kontrolnej c_2 powinien tworzyć parzystość dla bitów 3, 4, 5, 6 i 11. Teraz sprawdzamy cztery bity, pomijamy cztery bity itd.
- Bit sumy kontrolnej c_3 powinien tworzyć parzystość dla bitów 7, 8, 9, 10 i 11.

Dla przykładu wygenerujmy sumy kontrolne dla wartości binarnej 10101100. Pierwszy bit sumy kontrolnej będzie miał wartość 1, ponieważ musi on zapewnić parzystość dla bitów 0, 0, 1, 0 i 0. Korzystając z tej samej metody, ustalamy, że pozostałe bity sumy kontrolnej będą miały wartości:

$$c_1 = 1$$
$$c_2 = 1$$
$$c_3 = 0$$

co daje w wyniku wartość 12-bitową 101001101011.

Przedstawiony poniżej kod asemblera pozwalający zbudować 12-bitowy kod Hamminga wykorzystuje cykliczny rejestr przesuwny w czasie operacji logicznych.

```
AREA HAMMING, CODE
ENTRY
; Używane rejestry:
; R0 — tymczasowy
; R1 — używany do przechowywania adresu danych
; R2 — przechowuje wartość do wysłania
; R4 — tymczasowy
main
  MOV r2, #0                    ; czyszczenie rejestru transmisji
  ADR r1, arraya                ; początek stałych
  LDRB r0, [r1]
  ;
  ; obliczenie c0 z użyciem bitów 76543210
  ;                             * ** **
  ; parzystość, więc wynik XOR jest wartością c0
  ;
  MOV r4, r0                    ; wykonanie kopii
  EOR r4, r4, r0, ROR #1        ; 1 XOR 0
  EOR r4, r4, r0, ROR #3        ; 3 XOR 1 XOR 0
  EOR r4, r4, r0, ROR #4        ; 4 XOR 3 XOR 1 XOR 0
  EOR r4, r4, r0, ROR #6        ; 6 XOR 4 XOR 3 XOR 1 XOR 0
  AND r2, r4, #1                ; tworzenie c0 -> R2
  ;
  ; obliczenie c1 za pomocą bitów 76543210
```

```
  ;                              ** ** *
  MOV r4, r0
  EOR r4, r4, r0, ROR #2          ; 2 XOR 0
  EOR r4, r4, r0, ROR #3          ; 3 XOR 2 XOR 0
  EOR r4, r4, r0, ROR #5          ; 5 XOR 3 XOR 2 XOR 0
  EOR r4, r4, r0, ROR #6          ; 6 XOR 5 XOR 3 XOR 2 XOR 0
  AND r4, r4, #1                  ; izolowanie bitów
  ORR r2, r2, r4, LSL #1          ; 7 6 5 4 3 2 c1 c0
  ;
  ; obliczenie c2 za pomocą bitów 76543210
  ;                             *  ***
  MOV r4, r0, ROR #1              ; pobranie bitu 1
  EOR r4, r4, r0, ROR #2          ; 2 XOR 1
  EOR r4, r4, r0, ROR #3          ; 3 XOR 2 XOR 1
  EOR r4, r4, r0, ROR #7          ; 7 XOR 3 XOR 2 XOR 1
  AND r4, r4, #1                  ; izolowanie bitów
  ORR r2, r2, r4, ROR #29         ; 7 6 5 4 c2 2 c1 c0
  ;
  ; obliczenie c3 za pomocą bitów 76543210
  ;                              ****
  MOV r4, r0, ROR #4              ; pobranie bitu 4
  EOR r4, r4, r0, ROR #5          ; 5 XOR 4
  EOR r4, r4, r0, ROR #6          ; 6 XOR 5 XOR 4
  EOR r4, r4, r0, ROR #7          ; 7 XOR 6 XOR 5 XOR 4
  AND r4, r4, #1
  ;
  ; budowanie końcowego 12-bitowego wyniku
  ;
  ORR r2, r2, r4, ROR #25         ; rotacja w lewo o 7 bitów
  AND r4, r0, #1                  ; pobranie bitu 0 z oryginału
  ORR r2, r2, r4, LSL #2          ; dodanie bitu 0 do wyniku
  BIC r4, r0, #0xF1               ; pobranie bitów 3, 2, 1
  ORR r2, r2, r4, LSL #3          ; dodanie bitów 3, 2, 1 do wyniku
  BIC r4, r0, #0x0F               ; pobranie górnej czwórki
  ORR r2, r2, r4, LSL #4          ; r2 zawiera teraz 12 bitów
                                  ; z sumą kontrolną
done   B    done

  ALIGN
arraya
  DCB 0xB5
  DCB 0xAA
  DCB 0x55
  DCB 0xAA

END
```

Nasza początkowa wartość 8-bitowa znajduje się w pamięci w lokalizacji `arraya`, skąd jest ładowana do rejestru r0. Szybkim sposobem na wygenerowanie bitu parzystości jest użycie wyniku operacji `XOR` jako bitu sumy kontrolnej, np. gdy pobierzemy nieparzystą liczbę jedynek i wykonamy na nich operację `XOR`, wynikiem będzie 1, więc suma kontrolna powinna mieć wartość 1, aby utworzyć parzystą liczbę jedynek. Pierwsza suma kontrolna jest generowana za pomocą kolejnych instrukcji `EOR`, na oryginalnych danych, wszystkie bity poza 0 są ignorowane. Zwróć uwagę, że pierwsza instrukcja logiczna:

```
EOR r4, r4, r0, ROR #1
```

pobiera oryginalne dane i wykonuje operację XOR bitu 0 z bitem 1 skopiowanych danych przy użyciu jednej instrukcji. Kolejne instrukcje EOR pobierają skopiowane dane i rotują je o odpowiednią liczbę bitów aż do bitu 0. W końcu jesteśmy zainteresowani tylko bitem 0, więc wykonujemy logiczną operację AND końcowego wyniku z 1, aby wyczyścić wszystkie bity poza najmniej znaczącym, ponieważ wykonanie operacji AND z wartością 0 daje 0, a AND z wartością 1 daje oryginalną wartość.

Pozostałe sumy kontrolne są obliczane w taki sam sposób, odpowiednie bity są przesuwane w kierunku bitu 0, a następnie jest wykonywana instrukcja EOR z tymczasowym wynikiem. Końcowa 12-bitowa wartość jest tworzona na podstawie oryginalnej 8-bitowej wartości i czterech sum kontrolnych za pomocą funkcji logicznych. Zwróć uwagę, że rotacja w lewo jest realizowana przy użyciu instrukcji ROR, ponieważ rotacja w lewo o *n* bitów jest tym samym co rotacja w prawo o (32–*n*) bitów, dlatego nie ma instrukcji ROL. Końcowa wartość jest umieszczona w rejestrze r2.

Aby wykryć błąd w przesłanej wartości, należy sprawdzić cztery bity sumy kontrolnej, c_3, c_2, c_1 i c_0. Jeżeli okaże się, że jeden z bitów sumy kontrolnej jest nieprawidłowy (można to zweryfikować przez sprawdzenie bitów 12-bitowej danej), to sama suma kontrolna jest nieprawidłowa. Jeżeli dwa bity sumy kontrolnej są nieprawidłowe, na przykład c_n i c_m, to pozycja nieprawidłowego bitu, *j*, może być określona jako:

$$j = (2^n+2^m)-1$$

Na przykład jeżeli bity sumy kontrolnej c_3 i c_2 są nieprawidłowe, to błąd znajduje się w bicie 11. Ponieważ jest to jedyny błąd, może być poprawiony.

7.4.3. Dodawanie i odejmowanie

Instrukcje arytmetyczne w zbiorze instrukcji ARM obejmują operacje dodawania, odejmowania oraz odwrotnego odejmowania, z których wszystkie są w wersjach z przeniesieniem i bez.

```
ADD     r1, r2, r3      ; r1 = r2+r3
ADC     r1, r2, r3      ; r1 = r2+r3+C
SUB     r1, r2, r3      ; r1 = r2–r3
SUBC    r1, r2, r3      ; r1 = r2–r3+C–1
RSB     r1, r2, r3      ; r1 = r3–r2
RSC     r1, r2, r3      ; r1 = r3–r2+C–1
```

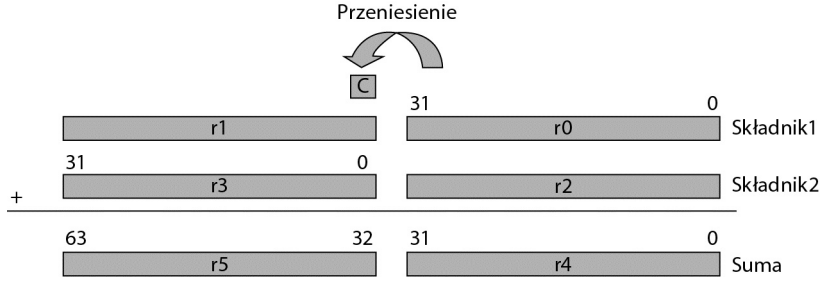
Z fragmentu na temat znaczników pamiętamy, że znacznik przeniesienia może być używany do wskazania, że operacja wygenerowała przeniesienie na najbardziej znaczącym bicie wyniku. Operacje ADC, SUBC oraz RSC wykorzystują ten znacznik przez dodanie znacznika przeniesienia do operacji. Załóżmy, że chcemy wykonać dodawanie 64-bitowe. Ponieważ rejestry mają wielkość 32 bitów, musimy zapisać każdy ze składników w dwóch rejestrach, suma również będzie przechowywana w dwóch rejestrach.

Przykład 7.7.

Poniższe dwie instrukcje dodają 64-bitową liczbę całkowitą zapisaną w rejestrach r2 i r3 do innej 64-bitowej liczby całkowitej zapisanej w rejestrach r0 i r1, a wynik umieszczają w rejestrach r4 i r5:

```
ADDS    r4, r0, r2      ; dodawanie mniej znaczących słów
ADC     r5, r1, r3      ; dodawanie bardziej znaczących słów
```

Jak możesz zobaczyć na rysunku 7.4, przeniesienie z dolnej 32-bitowej sumy jest dodawane do górnej 32-bitowej sumy, co daje końcowy 64-bitowy wynik.



RYSUNEK 7.4. Dodawanie 64-bitowe

Przykład 7.8.

Operacja 64-bitowego odejmowania może być zbudowana podobnie do operacji pokazanego powyżej dodawania. Poniższy kod odejmuje dolne połówki wartości 64-bitowych, aktualizując znacznik przeniesienia, a następnie odejmuje górne połówki, uwzględniając przeniesienie:

```
sub64   SUBS    r0, r0, r2      ; odejmowanie dolnych połówek,
                                ; ustawienie znacznika przeniesienia
        SBC     r1, r1, r3      ; odejmowanie górnych połówek i przeniesienia
```

Użyliśmy tu dwóch rzadko wykorzystywanych, ale użytecznych instrukcji: `RSB` i `RSC`, realizujących odwrotne odejmowanie. Instrukcja odwrotnego odejmowania wynika z posiadania cyklicznego rejestru przesuwnego tylko na jednej magistrali pomiędzy bankiem rejestrów a głównym ALU, co jest pokazane na rysunku 7.2. Załóżmy, że chcemy wykonać następującą operację:

```
SUB     r0, r2, r3, LSL #2      ; r0 = r2–r3*4
```

Możemy to zrobić za pomocą tej jednej instrukcji. Co w przypadku, gdy chcemy zmodyfikować (przesunąć) r2, a nie r3 przed wykonaniem odejmowania? Ponieważ odejmowanie nie jest operacją przemienną, czyli:

$$x–y \neq y–x, \qquad \text{gdzie } x, y \neq 0$$

to rejestr r2 musi być w jakiś sposób umieszczony na magistrali zawierającej rejestr przesuwny. Zapewnia to operacja odwrotnego odejmowania, w której instrukcje mogą być zapisywane w następujący sposób:

```
RSB     r0, r3, r2, LSL #2      ; r0 = r2*4–r3
```

Ta sama instrukcja może być użyta dla uzyskania jeszcze większego efektu, ponieważ drugi operand również może być stałą, więc można odejmować wartość rejestru *od* stałej, a nie odwrotnie.

Przykład 7.9.

Napiszmy kod asemblera ARM obliczający wartość bezwzględną. Rejestr r0 będzie zawierać wartość początkową, a r1 wartość bezwzględną. Pseudoinstrukcja będzie wyglądała tak jak poniżej:

```
ABS r1, r0
```

Spróbuj użyć tylko dwóch instrukcji (nie licząc instrukcji do zakończenia programu ani dyrektyw).

Rozwiązanie

Jak wiemy, wartość bezwzględna jest funkcją zwracającą dodatnią wartość argumentu, więc $f(x) = |x|$ zmienia znak argumentu, jeżeli wartość jest ujemna. Potrzebujemy jednego porównania i jednej instrukcji zmieniającej znak:

```
        AREA Prog7a, CODE, READONLY
        ENTRY

        CMP         r1, #0
        RSBLT       r0, r1, #0

done    B           done
        END
```

Program wykonuje porównanie z zerem w celu sprawdzenia, czy cokolwiek jest do zrobienia. Jeżeli argument jest zerem, wynikiem jest zero. Jeżeli argument jest ujemny (LT oznacza mniejszy niż zero, co opiszemy dokładniej w rozdziale 8.), to za pomocą odwrotnego odejmowania odejmujemy r1 *od* zera, w efekcie zmieniając znak. Zwróć uwagę na warunkowe wykonanie instrukcji RSB, ponieważ wartości dodatnie nie spełnią warunku.

7.4.4. Mnożenie

Mnożenie binarne jest realizowane obecnie w niemal każdym procesorze, ale jest obarczone pewnym kosztem. Jako operacja jest dosyć częste. Jako blok sprzętowy jest kosztowne, ponieważ taki blok zajmuje zwykle dosyć dużo miejsca i pobiera dużo prądu w stosunku do pozostałych części mikroprocesora. Starsze mikrokontrolery często korzystały z podprogramów sumujących i przesuwających w celu wykonania mnożenia, co pozwalało uniknąć budowania dużych macierzy mnożników, ale były to operacje dosyć powolne. Nowoczesne zwykle wykonują mnożenie w jednym lub dwóch cyklach, ale ponownie, z powodów związanych z zarządzaniem energią, jeżeli istnieje sposób, by uniknąć macierzy, kompilator ARM próbuje wytworzyć kod bez instrukcji mnożenia, co pokażemy nieco dalej. O wyborze mikroprocesorów i (lub) układów DSP często decydują możliwości szybkiego wykonywania mnożenia, szczególnie w przypadku przetwarzania mowy i sygnałów, analizy sygnału i sterowania adaptacyjnego.

W tabeli 7.2 są zamieszczone wszystkie instrukcje mnożenia dostępne w zbiorze instrukcji Version 4T. Instrukcje MUL i MLA pozwalają na wykonanie operacji mnożenia oraz mnożenia i akumulacji, które dają wyniki 32-bitowe. Instrukcja MUL mnoży wartości w dwóch rejestrach, obcina wynik do 32 bitów i zapisuje iloczyn w trzecim rejestrze. Instrukcja MLA mnoży zawartość dwóch rejestrów, obcina wynik do 32 bitów, dodaje wartość trzeciego rejestru do iloczynu i zapisuje wynik w czwartym rejestrze, na przykład:

```
MUL     r4, r2, r1          ; r4 = r2*r1
MULS    r4, r2, r1          ; r4 = r2*r1, następnie ustawia znaczniki
MLA     r7, r8, r9, r3      ; r7 = r8*r9+r3
```

TABELA 7.2. Instrukcje mnożenia oraz mnożenia i akumulacji

Instrukcja	Komentarz
MUL	Mnożenie 32 bitów przez 32 bity z 32-bitowym iloczynem.
MLA	Mnożenie 32 bitów przez 32 bity, wynik dodawany do 32-bitowej wartości akumulowanej.
SMULL	Mnożenie ze znakiem 32 bitów przez 32 bity z 64-bitowym iloczynem.
UMULL	Mnożenie bez znaku 32 bitów przez 32 bity z 64-bitowym iloczynem.
SMLAL	Mnożenie ze znakiem 32 bitów przez 32 bity, wynik dodawany do 64-bitowej wartości akumulowanej.
UMLAL	Mnożenie bez znaku 32 bitów przez 32 bity, wynik dodawany do 64-bitowej wartości akumulowanej.

Zarówno MUL, jak i MLA mogą opcjonalnie ustawiać znaczniki kodu warunku N i Z. Dla mnożenia dającego tylko 32-bitowy wynik nie istnieje rozróżnienie pomiędzy mnożeniem ze znakiem i bez znaku. Jedynie najmniej znaczące 32 bity wyniku są zapisywane w rejestrze docelowym, a znak operandów nie wpływa na wartość.

Długie instrukcje mnożenia generują wynik 64-bitowy. Pozwalają one na pomnożenie dwóch rejestrów i zapisanie 64-bitowego wyniku w trzecim i czwartym rejestrze. SMLL oraz UMULL są długimi instrukcjami mnożenia ze znakiem i bez znaku:

```
SMULL r4, r8, r2, r3        ; r4 = bity 31 – 0 z r2*r3
                            ; r8 = bity 63 – 32 z r2*r3
UMULL r6, r8, r0, r1        ; {r6, r8} = r0*r1
```

Poniższe instrukcje mnożą wartości dwóch rejestrów, dodają wartość 64-bitową z trzeciego i czwartego rejestru i zapisują 64-bitowy wynik w trzecim i czwartym rejestrze:

```
SMLAL r4, r8, r2, r3        ; r4 = bity 31 – 0 z r2*r3+{r4, r8}
                            ; r8 = bity 63 – 32 z r2*r3+{r4, r8}
UMLAL r5, r8, r0, r1        ; {r5, r8} = r0*r1+{r5, r8}
```

Wszystkie długie instrukcje mnożenia mogą opcjonalnie ustawiać znaczniki kodu warunku N i Z. Jeżeli jakikolwiek operand źródłowy jest ujemny, to wpływa to na najbardziej znaczące 32 bity wyniku.

7.4.5. Mnożenie przez stałą

W naszym omówieniu przesunięć i rotacji pokazaliśmy, że wewnętrzny cykliczny rejestr przesuwny w ścieżce danych procesora ARM7TDMI może być używany w połączeniu z innymi instrukcjami, takimi jak ADD lub SUB, co daje w efekcie darmowe mnożenie. Funkcja ta jest w pełni wykorzystywana w przypadkach, gdy określone mnożenia są realizowane za pomocą cyklicznego rejestru przesuwnego, a nie tablicy mnożników. Weźmy pod uwagę przypadek mnożenia liczby przez potęgę dwójki. Można to zrealizować za pomocą wyłącznie instrukcji MOV razem z przesunięciem, np.:

```
MOV r1, r0, LSL #2          ; r1 = r0*4
```

Co można zrobić, gdy chcemy pomnożyć dwie liczby, z których jedna nie jest potęgą dwójki, jak np. pięć? Przeanalizujmy poniższą instrukcję:

```
ADD r0, r1, r1, LSL #2      ; r0 = r1+r1*4
```

Jest to operacja analogiczna do pobrania wartości, przesunięcia jej w lewo o dwa bity (co daje mnożenie przez cztery), a następnie dodania oryginalnej wartości do iloczynu. Inaczej mówiąc, mnożenia liczby przez pięć. Dlaczego wykonujemy to w ten sposób? Weźmy pod uwagę rozmiar macierzy mnożników zaznaczonej na rysunku 7.5 w mikroprocesorze ARM10200 i zużycie energii. W zastosowaniach o niskim poborze prądu często wymagane jest wykorzystanie każdej możliwej sztuczki, aby oszczędzić energię: wyłączenie nieużywanej logiki, wyłączenie pamięci podręcznej lub całego procesora, gdy nie jest potrzebny, ograniczenie napięć i częstotliwości itd. Dzięki użyciu wyłącznie 32-bitowego sumatora i cyklicznego rejestru przesuwnego procesory ARM mogą generować mnożenia przez 2^n, 2^n–1 i 2^n+1 w jednym cyklu bez konieczności korzystania z macierzy mnożników. Potencjalnie pozwala to również na skrócenie czasu wykonania. Na przykład:

```
RSB r0, r2, r2, LSL #3      ; r0 = r2*7
```

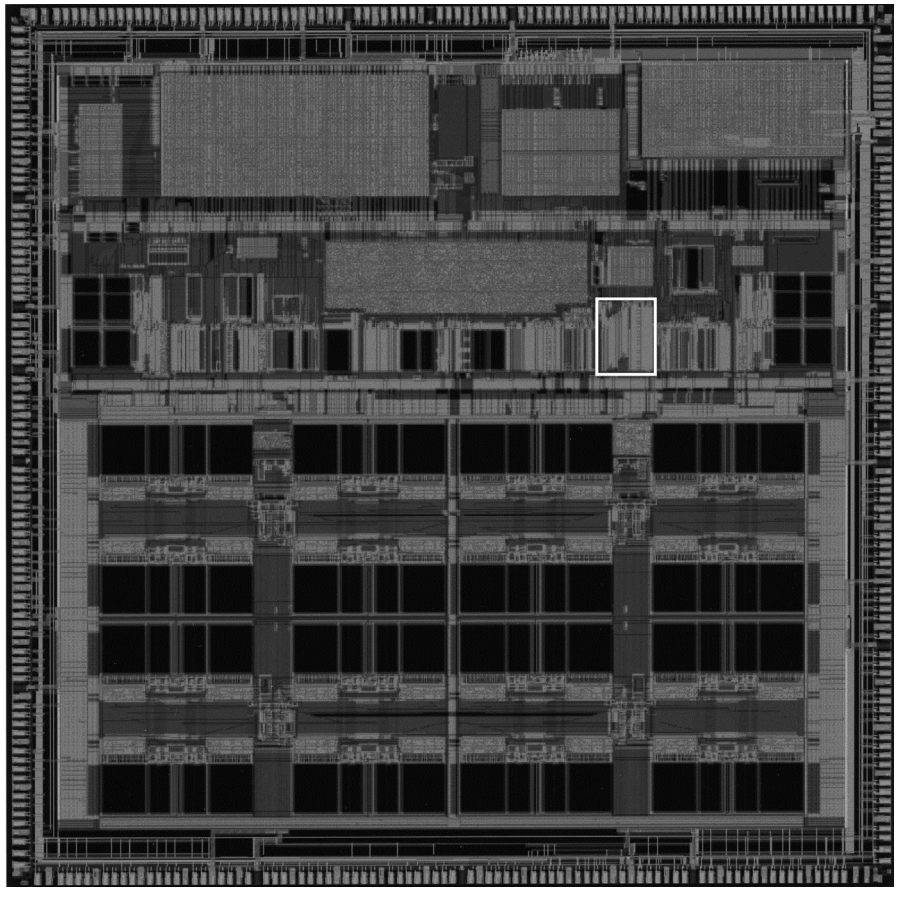
będzie realizować mnożenie przez 7 przez pobranie rejestru r2, przesunięcie w lewo o 3 bity, co daje mnożenie przez 8, a następnie odjęcie rejestru r2 od iloczynu. Zwróć uwagę, że użyta tu została instrukcja odwrotnego odejmowania, ponieważ zwykłe odejmowanie da niewłaściwy wynik. Przez połączenie operacji mnożenia, na przykład przez 5, a następnie przez 7, można tworzyć większe stałe. Spójrz na poniższy przykład, pokazujący, że możliwe jest mnożenie argumentów niebędących potęgami dwójki:

```
ADD r0, r1, r1, LSL #1      ; r0 = r1*3
SUB r0, r0, r1, LSL #4      ; r0 = (r1*3)–(r1*16) = r1*–13
ADD r0, r0, r1, LSL #7      ; r0 = (r1*–13)+(r1*128) = r1*115
```

7.4.6. Dzielenie

Dzielenie binarne jest tematem, który bardzo szybko staje się skomplikowany. Większość rdzeni ARM nie zawiera sprzętowego układu dzielenia całkowitego (Cortex-M3 taki oferuje), zwykle dlatego, że dzielenie jest rzadko używane (w związku z tym może być realizowane przez podprogramy). Ponadto układ dzielenia zajmuje zbyt dużo miejsca i wykorzystuje zbyt dużo energii, aby sensowne było umieszczanie go we wbudowanym procesorze, poza tym istnieją sposoby, aby całkowicie uniknąć dzielenia. Nie można powiedzieć, że brakuje dobrych podprogramów. Przy wybieraniu procedury dzielenia należy wziąć pod uwagę takie czynniki jak typ danych (dane ułamkowe czy całkowite), wymagana wydajność algorytmu oraz rozmiar kodu potrzebnego do realizacji algorytmu. Jeżeli chcesz zapoznać się dokładniej z tym tematem, polecamy pozycję Slossa, Symesa i Wrighta (2004). Biblioteki uruchomieniowe zawierają procedury dzielenia, więc jeżeli piszesz w C lub C++, kompilator zajmie się za Ciebie algorytmami dzielenia. My skupiamy się na asemblerze, więc przedstawimy tu przynajmniej najprostszy przypadek.

Kod przedstawiony na rysunku 7.6 jest odmianą algorytmu przesuwająco-odejmującego i może być używany do dzielenia dwóch wartości 32-bitowych bez znaku; w rejestrze Ra znajduje się dzielna, w rejestrze Rb dzielnik. Po wykonaniu procedury w Rc znajduje się iloraz, a w Ra reszta z dzielenia.



RYSUNEK 7.5. Zdjęcie układu ARM10200 z zaznaczoną macierzą mnożników

Przykład 7.10.

Korzystając z procedury z rysunku 7.6, możemy podzielić 150 przez 120. Ponieważ wartości są małe, do ich załadowania do rejestrów Ra i Rb możemy użyć dwóch instrukcji `MOV`:

```
MOV Ra, #0x96          ; ładowanie rejestru r1
MOV Rb, #0x78          ; ładowanie rejestru r2
```

Po uruchomieniu kodu rejestr r3 będzie zawierał wartość 1 (iloraz), a rejestr r1 wartość 0x1E (reszta z dzielenia).

7.5. NOTACJA UŁAMKOWA

Jednym z pierwszych problemów, jakie powstają przy nauce asemblera, jest wykorzystanie w programie takich wartości jak e lub π. Najprawdopodobniej napotkasz też w kodzie takie wartości jak $\sqrt{2}$, a przecież procesor ARM operuje tylko na 32-bitowych wartościach całkowitych, o ile nie jest dostępna jednostka zmiennoprzecinkowa. A może nie? Jak przedstawiliśmy

```
        AREA Prog7b, CODE, READONLY
Rcnt    RN      0                       ; przypisz R0 do Rcnt
Ra      RN      1                       ; przypisz R1 do Ra
Rb      RN      2                       ; przypisz R2 do Rb
Rc      RN      3                       ; przypisz R3 do Rc

        ENTRY

        ; umieść dzielną w Ra
        ; umieść dzielnik w Rb

        MOV         Rcnt, #1            ; bit sterujący
                                        ; dzieleniem
Div1    CMP         Rb, #0x80000000     ; przenoś Rb do momentu,
                                        ; gdy będzie większy niż Ra
        CMPCC       Rb, Ra
        MOVCC       Rb, Rb, LSL #1
        MOVCC       Rcnt, Rcnt, LSL #1
        BCC         Div1
        MOV         Rc, #0
Div2    CMP         Ra, Rb              ; test możliwości
                                        ; odejmowania
        SUBCS       Ra, Ra, Rb          ; odejmij, jeżeli OK
        ADDCS       Rc, Rc, Rcnt        ; umieść odpowiedni bit
                                        ; w wyniku
        MOVS        Rcnt, Rcnt, LSR #1  ; przesuń bit kontrolny
        MOVNE       Rb, Rb, LSR #1      ; podziel na pół, jeżeli
                                        ; nie koniec
        BNE         Div2                ; wynik dzielenia w Rc
                                        ; reszta w Ra
done    B           done
        END
```

RYSUNEK 7.6. Prosty algorytm dzielenia

w rozdziale 1., procesor operuje na danych, surowych liczbach, wzorcach bitowych. Te wzorce bitowe są interpretowane przez *programistę*, a nie przez procesor (o ile nie powiemy mu jasno, że ma to zrobić). Załóżmy, że w rejestrze r3 mamy wartość 32-bitową 0xF320ABCD. Czy jest to liczba dodatnia, czy ujemna? W normalnych warunkach po prostu tego nie wiemy. Możesz ją zinterpretować jako dziesiętną wartość −215 962 675, jeżeli zastosujesz reprezentację z dopełnieniem do dwójki. Możesz ją też zinterpretować jako 4 079 004 621, jeśli potraktujesz ją jako liczbę bez znaku. Decyzja często zależy od algorytmu i typu operacji arytmetycznych wykonywanych na tych liczbach. Normalnie program oczekuje danych w określonej postaci i odpowiednio do tego je interpretuje — jeżeli procedura filtra adaptacyjnego operuje na liczbach ze znakiem, to programista musi traktować wynik jako wartość ze znakiem.

Kontynuując tę argumentację, możemy zadać pytanie, gdzie jest przecinek dziesiętny w 0xF320ABCD? Inaczej mówiąc, gdzie zaczyna się część całkowita tej liczby, a gdzie jest część ułamkowa? Czy ta liczba ma część ułamkową? Również w tym przypadku odpowiedź

zależy od programisty. Użyteczną techniką jest przedstawianie problemu na małych wartościach, a następnie uogólnianie do liczb 32-bitowych.

Jeżeli liczba 1011 jest traktowana jako liczba bez znaku, to w postaci dziesiętnej ma wartość 8+2+1 = 11. Jeśli jednak założymy, że przecinek dziesiętny jest umieszczony przed ostatnią cyfrą, jak jest to pokazane na rysunku 7.7, to wartość 101,1 będzie równa 4+1+0,5 = 5,5. W tej reprezentacji mamy tylko dwie możliwe wartości części ułamkowej — 0 lub 0,5, co nie jest szczególnie użyteczne. Jeżeli przesuniemy przecinek dziesiętny jeszcze o jedno miejsce w lewo, liczba będzie miała postać 10,11, czyli 2+0,5+0,25 = 2,75 przy podstawie 10. Zwróć uwagę, że po prawej stronie przecinka dziesiętnego mamy dwa bity, co daje nam cztery możliwe wartości ułamkowe: 0, 0,25, 0,5 i 0,75. Teraz rozdzielczością ułamka jest 0,25, inaczej mówiąc, różnica pomiędzy dowolnymi dwiema wartościami ułamkowymi będzie nie mniejsza niż 0,25. Jeżeli przesuniemy przecinek dziesiętny całkowicie w lewo, otrzymamy liczbę 0,1011, czyli 0,5+0,125+0,0625 = 0,6875 przy podstawie 10. Możliwość tworzenia wartości ułamkowych jest nadal ograniczona, ale rozdzielczość naszego ułamka wynosi teraz 0,0635, co jest znacznym usprawnieniem.

Część całkowita				Część ułamkowa			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	1				
	1	0	1	1			
		1	0	1	1		
			1	0	1	1	
				1	0	1	1

RYSUNEK 7.7. Reprezentacja binarna

Jak pamiętasz z rozdziału 1., jeżeli liczba *n* jest reprezentowana jako *m*-bitowa w reprezentacji z dopełnieniem do dwójki, gdzie *b* to wartości poszczególnych bitów, to wartość tej liczby przy podstawie 10 może być wyliczona jako:

$$n = -b_{m-1}2^{m-1} + \sum_{i=0}^{m-2} b_i 2^i$$

więc liczba 8-bitowa 10110010_2 zapisana w postaci dopełnienia do dwójki ma wartość:

$$-2^7+2^5+2^4+2^1 = -78$$

Jednak jak wspominaliśmy, pozycja przecinka dziesiętnego w dowolnej reprezentacji zależy od programisty. Załóżmy, że ta 8-bitowa liczba została podzielona przez 2^7. Wzorzec bitowy jest identyczny — interpretacja jest inna. Okazuje się, że przy podstawie 10 liczba *n* może być obliczona jako:

$$n = -1+2^{-2}+2^{-3}+2^{-6} = -0{,}609375$$

W zasadzie możemy podzielić każdą liczbę *m*-bitową przez 2^{m-1}, otrzymując tylko ułamkową wartość *n*, taką jak:

$$-1 \le n \le (1-2^{-(m-1)})$$

Wracając do naszego pytania na temat *e* i π, jeżeli chcemy reprezentować liczby zawierające tylko część ułamkową, możemy przeskalować taką liczbę przez mniej niż 2^{m-1}. Załóżmy,

że mamy 16 bitów i chcemy w obliczeniach użyć *e*. Wiemy, że do reprezentacji części całkowitej liczby potrzebujemy co najmniej dwóch bitów (ponieważ liczba ta wynosi 2,71828…), więc użyjemy co najwyżej 13 bitów po lewej stronie przecinka, zakładając, że najbardziej znaczący bit jest bitem znaku. W literaturze jest to czasami nazywane notacją Q, w której nasza liczba będzie nazywana liczbą Q13. Na szczęście zasady notacji Q są proste. Liczby dodawane i odejmowane w tej notacji powinny zawsze mieć wyrównane miejsca dziesiętne, więc Q*n*+Q*n* = Q*n*. Gdy dwie liczby są mnożone, Q*n*×Q*m*, to wynik będzie w formacie Q(*n*+*m*).

Aby zapisać *e* w notacji Q13, bierzemy wartość *e*, mnożymy ją przez 2^{13}, a następnie konwertujemy wynik na postać szesnastkową (przyda się kalkulator). Mamy więc:

$$e \times 2^{13} = 22\,268{,}1647 = 0x56FC$$

Zwróć uwagę, że na postać szesnastkową konwertujemy tylko część całkowitą. Jeżeli liczba jest interpretowana w notacji Q13, to wiemy, że po lewej stronie hipotetycznego przecinka binarnego mamy dwa bity, a po prawej stronie 13 bitów:

Bit znaku
↓
0x56FC = 0101011011111100
↑
Hipotetyczny przecinek binarny

Przykład 7.11.

Kontynuując, użyjemy teraz naszego komputera, aby pomnożyć *e* przez 2. Musimy skonwertować drugą liczbę na liczbę w notacji ułamkowej, więc ponownie skorzystamy z notacji Q13. Według tych samych zasad reprezentacją 2 w Q13 jest 0x2D41. Poniższy prosty blok kodu wykonuje mnożenie:

```
        AREA Prog7c, CODE, READONLY
        ENTRY
        LDR r3, =0x56FC                 ; e w notacji Q13
        LDR r2, =0x2D41                 ; sqrt(2) w notacji Q13
        MUL r5, r2, r3                  ; iloczyn jest w notacji Q26
done    B done
        END
```

Po uruchomieniu kodu powinieneś mieć w rejestrze r5 wartość 0xF6061FC. Ponieważ iloczyn jest w notacji Q26, musimy skonwertować ją na wartość dziesiętną, a następnie podzielić przez 2^{26}, aby uzyskać końcowy wynik, którym jest 3,84412378. Ten iloraz ma wartość 3,84423102, ale trzeba się spodziewać, że z powodu reprezentacji liczb w notacji o ograniczonej precyzji utracimy dokładność. Jeżeli mielibyśmy nieskończoną liczbę bitów do wykonania operacji, wynik byłby dokładny. Nie powiedzieliśmy jeszcze, co możemy zrobić z wartością umieszczoną w rejestrze. Ta liczba 32-bitowa może nadal reprezentować dużą liczbę dodatnią lub na przykład maskę dla rejestru konfiguracji w kontrolerze pamięci podręcznej! Procesor nie ma pojęcia, co tu robimy.

Przykład 7.12.

Wykonajmy następny przykład; tym razem operand będzie ujemny, ponieważ wprowadzi to kilka kolejnych utrudnień do obsługi tej notacji. Pomnożymy teraz dwie wartości, /4 i pewną wartość sygnału cyfrowego, załóżmy, –0,3872. Te dwie wartości powinny być zapisane w notacji Q15 reprezentowanej przez 16 bitów. Inaczej mówiąc, reprezentacja ta wygląda następująco:

$$s.f_{14}f_{13}f_{12}f_{11}f_{10}f_9f_8f_7f_6f_5f_4f_3f_2f_1f_0$$

Zawiera ona jeden bit znaku, jeden hipotetyczny przecinek binarny i 15 bitów wartości ułamkowej. Aby skonwertować /4 na reprezentację Q15, wykonujemy te same czynności co poprzednio, mnożąc wartość dziesiętną przez 2^{15}, co daje:

$/4 \times 2^{15} = 25\,735{,}927 = 0x6487$ (na postać szesnastkową konwertujemy tylko część dziesiętną)

Druga, ujemna wartość będzie wymagała nieco więcej myślenia. Najłatwiejszym sposobem obsługi liczb ujemnych jest ich konwersja na wartości dodatnie, a następnie zanegowanie wyniku. Aby uzyskać dodatnią wartość Q15, wykonujemy działania:

$$|-0{,}3872 \times 2^{15} = 12\,687\,7696 = 0x318F$$

Wynik 0x318F możemy zanegować ręcznie (niezalecane) lub przy użyciu kalkulatora albo komputera do wykonania negacji z uzupełnieniem do dwójki. Wynikiem jest wartość 16-bitowa z ustawionym najbardziej znaczącym bitem — lepiej, żeby tam był, ponieważ w innym przypadku wartość nie będzie ujemna. Negacja 0x318F daje w wyniku 0xCE71 (warto wiedzieć, że niektóre kalkulatory rozszerzają tę wartość o znak — trzeba pamiętać, że do reprezentacji tej liczby używamy tylko 16 bitów!). Dla sprawdzenia możemy spojrzeć na tę wartość w następujący sposób:

s	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f
1	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	

↑
Hipotetyczny przecinek binarny

Jak powiedzieliśmy wcześniej, wynik ten możemy też przedstawić jako:

$$\begin{gathered} -1 + \text{ wszystkich bitów ułamka} = \\ -1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-6} + \ldots = \\ -1 + 0{,}6128 = -0{,}3871765 \end{gathered}$$

więc otrzymaliśmy oczekiwaną wartość. Aby pomnożyć wartości w kodzie, wykonujemy te same operacje co poprzednio, musimy jedynie rozszerzyć ujemne operandy do 32 bitów. Dlaczego? Mnożnik w ARM7TDMI oczekuje dwóch operandów 32-bitowych i mnoży je przez siebie (zwracając dolne 32 bity w przypadku operacji `MUL`), więc jeżeli umieścimy ujemną wartość 16-bitową w jednym z rejestrów, wynik będzie nieprawidłowy, ponieważ technicznie jest to dodatnia wartość w dopełnieniu do dwójki przy użyciu 32 bitów — przynajmniej dla procesora. Jeżeli użylibyśmy procesora ARM9E z nowszymi instrukcjami pobierającymi wartości 16-bitowe, które za nas rozszerza o znak, to nie musielibyśmy wykonywać tej operacji. Jednak nasze symulacje wykorzystują zestaw instrukcji Version 4T, więc niezbędne jest wykonanie tych operacji w nieco staroświecki sposób. Odpowiedni kod będzie wyglądać następująco:

```
        AREA Prog7d, CODE, READONLY

        ENTRY

        LDR r3, =0x6487          ; pi/4 w notacji Q15
        LDR r2, =0xFFFFCE71      ; –0,3872 w notacji Q15

        MUL r5, r2, r3           ; iloczyn jest w notacji Q30
        MOV r5, r5, LSL #1       ; przesunięcie o jeden bit w lewo
done    B done
        END
```

W rejestrze r5 otrzymamy wynik 0xD914032E. Ponieważ wiemy, że liczba jest ujemna, aby ją zinterpretować, najłatwiej ją zanegować (ponownie korzystając z kalkulatora), co daje 0x26EBFCD2. Jest to również reprezentacja Q31, więc zapisujemy ją w notacji o podstawie dziesiętnej i dzielimy przez 2^{31}, co daje 0,3041. Dlaczego na końcu jest wykonywane dodatkowe przesunięcie? Należy pamiętać, że mnożenie dwóch liczb Q15 daje iloczyn Q30, jednak wynik ma 32 bity, co oznacza, że mamy nadmiarowy bit znaku w najbardziej znaczącym bicie. Aby ponownie wyrównać przecinki binarne, przesuwamy wszystko o jeden bit w lewo. Końcowy wynik może być pobrany z górnego półsłowa (16 bitów) rejestru r5, co daje kolejną liczbę Q15. Nie wykonywaliśmy tego przesunięcia w pierwszym przykładzie, ponieważ operandy były dodatnie (więc nie był ustawiony bit znaku) i nie wykonywaliśmy wyrównania — zatrzymaliśmy się na dodatniej liczbie Q26.

W zależności od zastosowania i wymaganej dokładności algorytmu dane mogą być w pewnym momencie obcinane. W przypadku danych graficznych, dla których wszystkie wartości są z zakresu od 0 do 0xFF, gdy algorytm utworzy wynik, cała część ułamkowa może być obcięta przed zapisaniem wyniku. Przy danych dźwiękowych lub zmiennych w kontrolerze cyfrowym możemy zachować część ułamka lub cały ułamek przed wysłaniem go do konwertera cyfrowo-analogowego (C/A). Zastosowanie ma ogromny wpływ na sposób obsługi danych.

7.6. ĆWICZENIA

1. Znajdź błąd w poniższych instrukcjach. Możesz skorzystać z dodatku A zawierającego pełne opisy instrukcji i ich ograniczeń.
   a) `ADD r3, r7, #1023`
   b) `SUB r11, r12, r3, LSL #32`
   c) `RSCLES r0, r15, r0, LSL r4`
   d) `EORS r0, r15, r3, ROR r6`
2. Nie korzystając z `MUL`, podaj instrukcje mnożące rejestr r4 przez:
   a) 135
   b) 255
   c) 18
   d) 16 384

3. Napisz procedurę porównującą dwie wartości 64-bitowe, składającą się z dwóch instrukcji (podpowiedź: druga instrukcja jest wykonywana warunkowo na podstawie pierwszego porównania).
4. Napisz instrukcje przesunięcia pozwalające na wykonanie przesunięcia arytmetycznego wartości 64-bitowej zapisanej w dwóch rejestrach. Procedura powinna przesuwać operand w lewo lub w prawo o jeden bit.
5. Zapisz poniższe wartości dziesiętne w notacji Q15:
   a) 0,3487
   b) –0,1234
   c) –0,1111
   d) 0,7574
6. Zapisz poniższe wartości Q8 ze znakiem w postaci dopełnienia do dwójki w formacie dziesiętnym:
   a) 0xFE32
   b) 0x9834
   c) 0xE800
   d) 0xF000
7. Napisz kod asemblera potrzebny do wykrywania błędów w 12-bitowym kodzie Hamminga, w którym kod testuje cztery bity sumy kontrolnej, c3, c2, c1 i c0. Zapisz w pamięci uszkodzone dane. Zakładając, że w danych występuje jeden błąd, umieść skorygowaną wartość w rejestrze r6.
8. Napisz program obliczający $\pi \times 48{,}9$ w notacji Q10.
9. Zapisz reprezentację sin(82°) w notacji Q15.
10. Zapisz reprezentację sin(193°) w notacji Q15.
11. Do konwersji temperatury ze stopni Celsjusza na stopnie Fahrenheita można użyć wzoru:

$$C = \frac{5}{9}(F - 32)$$

    gdzie C i F są wyrażone w stopniach. Napisz program konwertujący wartość w stopniach Celsjusza w rejestrze r0 na wartość w stopniach Fahrenheita. Skonwertuj ułamek na reprezentację Q15 i w procedurze użyj mnożenia zamiast dzielenia. Załaduj wartość testową z lokalizacji pamięci o nazwie `CELS` i zapisz wynik w pamięci o nazwie `FAHR`. Pamiętaj, że konieczne jest określenie adresu początkowego pamięci RAM dla mikrokontrolera używanego w symulacji. Na przykład mikrokontroler LPC2132 ma SRAM od adresu 0x40000000.
12. Napisz program zliczający jedynki w wartości 32-bitowej. Zapisz wynik w rejestrze r3.
13. Korzystając z dodatku A (lub narzędzi Keil), określ wzorzec bitów dla poniższych instrukcji:
    a) `RSB r0, r3, r2, LSL #2`
    b) `SMLAL r3, r8, r2, r4`
    c) `ADD r0, r0, r1, LSL #7`

14. Operacją często wykonywaną przez mikrokontrolery jest konwersja ASCII na postać binarną. Jeżeli naciśniesz cyfrę na klawiaturze, to procesor otrzyma reprezentację ASCII tej cyfry, a nie jej reprezentację binarną. Do konwersji danych na postać binarną używaną w operacjach arytmetycznych jest potrzebna pewna prosta procedura. Jeżeli spojrzysz na tabelę ASCII w dodatku C, zauważysz, że cyfry od 0 do 9 są reprezentowane przez kody ASCII od 0x30 do 0x39. Cyfry od A do F mają kody od 0x41 do 0x46. Ponieważ w zakresach występuje przerwa, konieczne jest wykonanie dwóch testów przy konwersji.

    Algorytm konwersji jest następujący:

    Za pomocą maski usuń bit parzystości (bit 7 w reprezentacji ASCII), ponieważ nie będziemy go potrzebować.

    Odejmij nadmiar od wartości ASCII.

    Sprawdź, czy cyfra jest z zakresu od 0 do 9.

    Jeżeli tak, operacja jest zakończona. W przeciwnym razie odejmij 7, aby określić wartość.

    Zapisz tę procedurę w asemblerze. Możesz założyć, że reprezentacja ASCII zawiera prawidłowe znaki od 0 do F.

15. Zapisz cztery różne instrukcje wypełniające rejestr r7 zerami.

16. Określ wzorzec bitowy, który asembler Keil wytworzy dla poniższej instrukcji. Wyjaśnij dlaczego taki.

```
BIC r6, r6, #0xFFFFFFFF
```

17. Napisz instrukcje ustawiające bity 0, 4 i 12 w rejestrze r6, pozostawiające pozostałe bity bez zmian.

18. Napisz program konwertujący wartość binarną z zakresu od 0 do 15 na jej reprezentację ASCII. Więcej informacji na ten temat znajduje się w ćwiczeniu 14.

19. Załóż, że nie jest dostępna instrukcja długiego mnożenia ze znakiem. Napisz program wykonujący mnożenie 32×32, dający wynik 64-bitowy, korzystając tylko z `UMULL` i operacji logicznych. Uruchom program, aby zweryfikować jego działanie.

20. Napisz program dodający do siebie liczby 128-bitowe, umieszczający wynik w rejestrach r0, r1, r2 i r3. Pierwszy operand powinien się znaleźć w rejestrach r4, r5, r6 i r7, a drugi w rejestrach r8, r9, r10 i r11.

21. Napisz program, który pobiera dane od „a” do „z” i zwraca te znaki zapisane wielkimi literami.

22. Podaj trzy różne metody sprawdzenia odpowiedniości dwóch wartości zapisanych w rejestrach r0 i r1.

23. Napisz kod asemblera realizujący następujące dzielenie ze znakiem:

```
r1 = r0/16
```

24. Pomnóż 0xFFFFFFFF (–1 w reprezentacji z uzupełnieniem do dwójki) i 0x80000000 (największa 32-bitowa liczba ujemna w reprezentacji z uzupełnieniem do dwójki). Użyj instrukcji `MUL`. Jaką wartość otrzymałeś? Czy ta liczba ma sens? Dlaczego lub dlaczego nie?

25. Kod Graya jest sposobem porządkowania liczb binarnych 2^n, aby przy przechodzeniu z jednej pozycji do drugiej zmieniał się tylko jeden bit. Przykładem 2-bitowego kodu Graya jest b10 11 01 00. Odstępy w tym przykładzie są użyte dla poprawienia czytelności. Napisz asembler ARM zamieniający 2-bitowy kod Graya w rejestrze r1 na 3-bitowy kod Graya w rejestrze r2. Zwróć uwagę, że 2-bitowy kod Graya zajmuje tylko bity [7:0] w rejestrze r1, a 3-bitowy kod Graya zajmuje bity [23:0] w rejestrze r2. Możesz zignorować początkowe zera. Jednym ze sposobów na zbudowanie *n*-bitowego kodu Graya na podstawie (*n*–1) kodu Graya jest prefiksowanie zerem każdego (*n*–1) elementu kodu. Następnie tworzone są dodatkowe n-bitowe elementy kodu przez pobranie każdego (*n*–1) kodu w odwrotnej kolejności i prefiksowanie ich jedynką. Na przykład pokazany wcześniej 2-bitowy kod Graya przyjmuje postać:

```
b010 011 001 000 100 101 111 110
```

26. Napisz program obliczający powierzchnię koła. Rejestr r0 będzie zawierał promień koła w notacji Q3. Znajdź reprezentację π w notacji Q10 i zapisz wynik w rejestrze r3 w notacji Q3.

Skorowidz

Wszystko o języku asembler dla procesorów ARM!

Jeszcze do niedawna mało kto zdawał sobie sprawę z istnienia takich rozwiązań jak procesory ARM. Ten stan rzeczy zmieniła inwazja urządzeń mobilnych: tabletów, smartfonów oraz platform typu Raspberry Pi. Przed profesjonalnymi programistami stanęło nowe wyzwanie — poznanie asemblera platformy ARM.

Jeżeli należysz do tej grupy, trafiłeś na świetną książkę poświęconą temu tematowi. W trakcie lektury zaznajomisz się ze sposobami reprezentacji liczb i znaków oraz modelem programowania ARM7TDMI. Następnie stworzysz swój pierwszy program z wykorzystaniem asemblera oraz poznasz dyrektywy tego języka. Dalsze rozdziały to kolejne elementy programowania w języku asembler. Adresowanie, ładowanie danych, operacje logiczne i arytmetyczne, pętle i instrukcje warunkowe to tylko niektóre z poruszanych zagadnień. Dzięki tej książce zdobędziesz też cenną wiedzę o urządzeniach peryferyjnych oraz obsłudze wyjątków. Książka jest doskonałą lekturą dla wszystkich programistów tworzących oprogramowanie dla procesorów ARM.

Dzięki tej książce:

- poznasz język asembler dla architektury ARM,
- opanujesz dyrektywy i zasady korzystania z asemblera,
- wykonasz typowe operacje logiczne i arytmetyczne,
- z łatwością wykorzystasz typowe konstrukcje języka.

Helion
Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9319-1
cena: 59,00 zł
9 788324 693191