Instrukcje procesora ARM można podzielić na sześć grup:

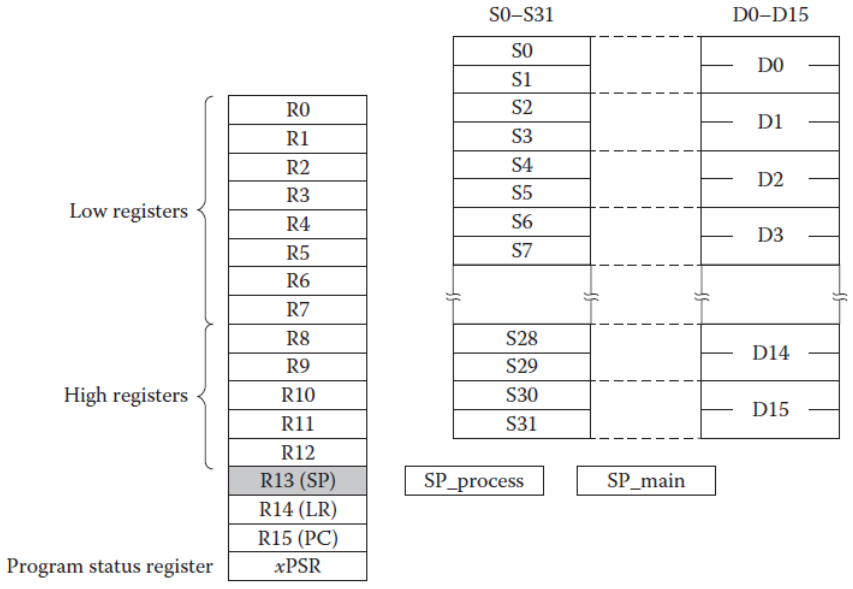
- Instrukcje przetwarzające dane,
  - Arytmetyczne/logiczne, porównujące, instrukcje mnożące (dzielące),
  - SIMD (Single Instruction Multiple Data) – instrukcje wykonujące podwójne lub poczwórne operacje,
  - Instrukcje modyfikujące PC (rozgałęzienie programu),
- Instrukcje skoków,
  - Skoki bezwarunkowe/warunkowe,
  - Skoki do podprogramów,
  - Zmiana trybu pracy (ARM/THUMB/Jazelle),
- Instrukcje operujące na pamięci,
  - Zapis/odczyt danych z pamięci (obsługa wielu rejestrów),
  - Operacje atomowe do obsługi semaforów,
- Instrukcje obsługujące rejestr stanu,
  - Modyfikacja oraz odczyt bitów rejestrów CPSR/SPSR,
- Instrukcje wykorzystywane przez koprocesor,
  - Wymiana danych pomiędzy rejestrami ALU a rejestrami koprocesora,
- Instrukcje generujące wyjątki,
  - Programowe przerwania,
  - Programowe pułapki.

Ogólne własności architektury

Jednolita przestrzeń adresowa – architektura typu Princeton (von Neumanna)
Osobne szyny do pamięci danych i programu – organizacja typu Hardward
Rdzenie Cortex-M praktycznie nie wymagają pisania czegokolwiek w asemblerze

Rejestry w Cortex-M

R0 do R12 – rejestry ogólnego przeznaczenia
SP (R13, MSP, PSP) – wskaźnik stosu
LR (R14) – adres powrotu
PC (R15) – licznik programu
PSR (APSR, IPSR, EPSR) – rejestr znaczników
PRIMASK, FAULTMASK, BASEPRI – rejestry maskujące przerwania
CONTROL – rejestr sterujący trybami pracy rdzenia



Procesor ARM posiada łącznie 37 rejestrów (wszystkie są 32 bitowe):
PC (r15) – licznik programu (Program Counter)
CPSR – rejestr statusowy, obecny status (Current Program Status Register)
SPSR – rejestr statusowy, dostępne w rożnych trybach uprzywilejowania (Saved Program Status Register)
LR (r14) – rejestr powrotu (Link Register), wykorzystywany podczas tworzenia ramki stosu (instrukcje skoku do funkcji)
SP (r13) – zwykle używany jako wskaźnik stosu (Stack Pointer)
r0 - r12 – rejestry ogólnego przeznaczenia
Uwaga :
Nie wszystkie rejestry są dostępne w rożnych trybach uprzywilejowania procesora

Przykład prostego programu:

```
        AREA   Prog1,  CODE,  READONLY     ; deklaracja bloku kodu
        ENTRY                              ; punkt wejścia do programu
        MOV    r0,  #0x11                  ; załadowanie wartości początkowej
        MOV    r1,  r0,  LSL #1            ; przesunięcie o 1 bit w lewo
        MOV    r2,  r0,  LSL #1            ; przesunięcie o 1 bit w lewo

stop    B      stop                        ; zatrzymanie programu
        END                                ; koniec programu
```

Komentarz rozpoczyna się od znaku średnik (;).

Często używane dyrektywy:

AREA	Definiuje blok danych lub kodu.
RN	Przydziela nazwę rejestrowi.
EQU	Przydziela nazwę stałej numerycznej.
ENTRY	Deklaruje punkt wejścia do programu.
DCB	Przydziela jeden lub więcej bajtów pamięci. Pozwala również określić początkową zawartość pamięci.
DCW	Przydziela jedno lub więcej półsłów pamięci. Pozwala również określić początkową zawartość pamięci.
DCD	Przydziela jedno lub więcej słów pamięci. Pozwala również określić początkową zawartość pamięci.
ALIGN	Wyrównuje dane lub kod do określonej granicy pamięci.
SPACE	Rezerwuje wyczyszczony blok pamięci o dowolnej wielkości.
LTORG	Przypisuje punkt startowy puli literałów.
END	Oznacza koniec pliku źródłowego.

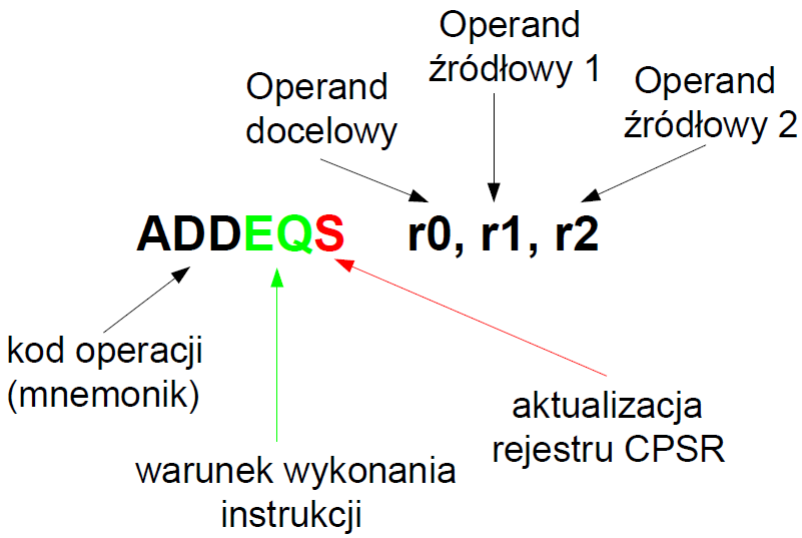
Instrukcje ładowania i zapisywania.

Instrukcje ładowania pobierają jedną wartość z pamięci i zapisują ją do rejestru roboczego. Instrukcje zapisu odczytują wartość z rejestru i umieszczają ją w pamięci.

Ładowanie	Zapisywanie	Rozmiar i typ
LDR	STR	Słowo (32 bity)
LDRB	STRB	Bajt (8 bitów)
LDRH	STRH	Półsłowo (16 bitów)
LDRSB	-	Bajt ze znakiem
LDRSH	-	Półsłowo ze znakiem
LDM	STM	Wiele słów

Dla większości instrukcji ARM ogólny format instrukcji jest następujący:

instrukcja cel, źródło, źródło



Składnia assemblera:

A = B + C ;Wynik <= Argument 1 operacja Argument 2)

```
SUB     Rd, Rs, Operand_2
SUB     R1, R2, R3              ; R1 = R2 - R3
```

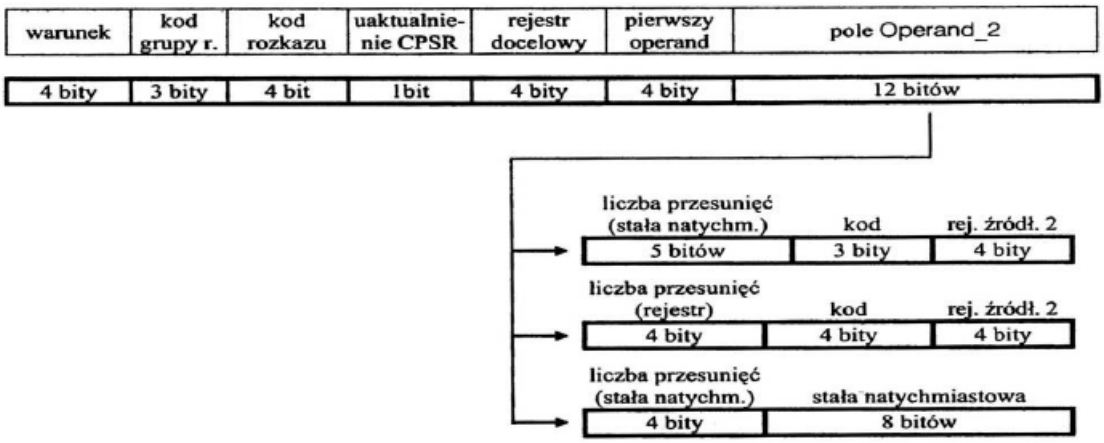
- Jako operand docelowy oraz pierwszy operand źródłowy zawsze należy używać rejestru
- Operand źródłowy drugi może zostać podany w postaci rejestru, wartości stałej lub wartości skalowanej:
  - Rx , np. R8
  - #wartość_stała , np. #5
  - Rx, operacja_skalowania , np. LSR #5

```
AND     R1, R2, R1              ; R1 = R2 + R1
ADC     R5, R7, #255            ; R5 = R7 + 255
ADD     R5, R7, R8, ROR #3      ; R5 = R7 + (R8>>3)
```

Flexible Operand 2	
Immediate value	`#<imm8m>`
Register, optionally shifted by constant (see below)	`Rm {, <opsh>}`
Register, logical shift left by register	`Rm, LSL Rs`
Register, logical shift right by register	`Rm, LSR Rs`
Register, arithmetic shift right by register	`Rm, ASR Rs`
Register, rotate right by register	`Rm, ROR Rs`

Budowa instrukcji procesora ARM

- Wszystkie instrukcje mają długość 32 bit (instrukcje należy wyrównać do granicy 32 bit.)
- Odwołanie do pamięci z wykorzystaniem techniki RMW (Read-Modify-Write) – instrukcje Load – Store
- Ortogonalne argumenty (argument docelowy-źródłowy)
- Możliwość użycia jednego z 16 rejestrów (r0-r15)
- Regularna budowa instrukcji (kodu maszynowego) – uproszczenie dekodera instrukcji



Podstawowe instrukcje procesora ARM

Grupa	Mnemonik	Rozwinięcie mnemonika	Opis
Arytmetyczne	ADD ADC	*add / add with carry*	dodawanie / dodaw. z uwzględnieniem bitu carry
	SUB SBC	*substract / substr. with carry*	odejmowanie / odejm. z uwzględnieniem bitu carry
	RSB RSC	*revers substract / rewers substract with carry*	odejmowanie w odwrotnej kolejności / / odejm. w odwr. kol. z uwzględnieniem bitu carry
	CMP CMN	*compare / compare negative*	porównanie / porówn. ze zmienionym znakiem arg2
Logiczne	AND BIC	*and / bit clear (and not)*	iloczyn logiczny / zerowanie bitów
	ORR EOR	*or / exor*	suma logiczna / różnica symetryczna
	TST TEQ	*test / test equivalence*	test / test identyczności
Mnożenia	MUL MLA	*multiply / multiply-accumulate*	mnożenie / mnożenie z dodawaniem
	UMULL SMULL	*unsigned / signed multiply*	mnożenie bez znaku / mnożenie ze znakiem
	UMLAL SMLAL	*unsigned / signed multiply-accumulate*	mnożenie z dodawaniem bez znaku / moż. z dodaw. ze znakiem
Skoki	B	*branch*	rozgałęzienie (skok)
	BL	*branch with link*	rozgałęzienie (skok) z zachowaniem rej. PC
	BX	*branch and exchange*	rozgałęzienie (skok) ze zmianą trybu ARM/Thumb
Przesłań	MOV MVN	*move / move not*	przesłania pomiędzy rejestrami oraz ładowanie stałych do rejestrów / j.w. z negacją
	LDR STR	*load / store register*	przesłania rejestru z/do pamięci
	LDM STM	*load / store multiply register*	przesłania wielu rejestrów z/do pamięci
	SWP	*swap register and memory*	wymiana zawartości rejestru i pamięci
	MRS MSR	*move xPSR to / from register*	przesłania pomiędzy rejestrami statusowymi a rejestrami
Pozostałe	SWI	*software interrupt*	przerwanie programowe
Rozkazy opcjonalnych koprocesorów	MRC MCR	*move coprocessor – register*	przesłania rdzeń – koprocesor
	LDC STC	*move coprocessor – memory*	przesłania pamięć – koprocesor
	CDP	*coprocessor data operation*	instrukcja koprocesora

Operacje matematyczne i logiczne

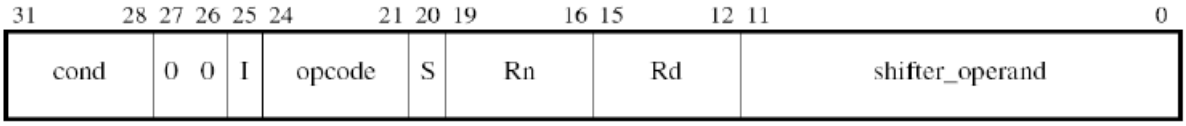
- Lista podstawowych instrukcji:
  - Arytmetyczne: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logiczne: `AND` `ORR` `EOR` `BIC`
  - Porównujące: `CMP` `CMN` `TST` `TEQ`
  - Operacje na danych: `MOV` `MVN`
- Instrukcje operują wyłącznie na rejestrach (brak odwołania do pamięci).
- Składnia:
  - Rozkazy trójargumentowe:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Rozkazy dwuargumentowe:
    - Operacje porównujące nie korzystają z rejestru Rd
    - Operacje na danych nie korzystają z rejestru Rn
- Drugi operand przesyłany jest do ALU z wykorzystaniem rejestru przesuwnego (ang. barrel shifter)
  - Rozkazu porównawcze zawsze aktualizują flagi rejestru CPSR, pozostałe w zależności od preferencji programisty

Kodowanie instrukcji matematycznych i logicznych

```
<opcode1>{<cond>}{S}  <Rd>, <shifter_operand>
<opcode1> := MOV | MVN
<opcode2>{<cond>}  <Rn>, <shifter_operand>
<opcode2> := CMP | CMN | TST | TEQ
<opcode3>{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
```



- Bit I – służy do rozróżniania liczby przesunięć drugiego argumentu (adresowanie natychmiastowe lub rejestrowe bezpośrednie)
- Bit S =1 – aktualizacja CPSR po wykonaniu instrukcji
- Rn – pierwszy operand źródłowy
- Rd – operand docelowy
- shifter_operand - drugi operand źródłowy (Flexible Operand 2)

Tabela kodów dla instrukcji matematycznych i logicznych

- Kod instrukcji kodowany jest przy pomocy 4 bitów (OpCode)

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Przykłady instrukcji arytmetycznych

Instrukcja dodawania

```
ADD     r0, r1, r2          // r0 = r1 + r2
ADD     r0, r1, #200        // r0 = r1 + 200
```

Dodawanie z uwzględnieniem bitu przeniesienia

```
ADC     r0, r1, r2          //  r0 = r1 + r2 + carry
```

Odejmowanie

```
SUB     r0, r1, r2          // r0 = r1 – r2
```

Odejmowanie z uwzględnieniem bitu przeniesienia

```
SBC     r0, r1, r2          // r0 = r1 - r2 - NOT(carry)
```

Odejmowanie, odwrócona kolejność argumentów

```
RSB     r1, r2, #0          // r1 = #0 - r2 (r1 w kodzie U2)
```

Odejmowanie, odwrócona kolejność arg. z uwzględnieniem bitu przeniesienia

```
RSC     r3, r1, r2          // r3 = r2 - r1 - NOT(carry)
```

Instrukcje logiczne i porównania

- Lista podstawowych instrukcji:
  - Logiczne: `AND` `ORR` `EOR` `BIC`

AND – operacja iloczynu logicznego, np. 0x12 AND 0xF0 → 0x10

ORR – operacja sumy logicznej, np. 0xAC ORR 0x10 → 0xBC

EOR – operacja różnicy symetrycznej,np. 0x12 XOR 0xF0 → 0xD0

BIC – operacja iloczynu z negacją (AND NOT), np.

0x12 AND (NOT 0xF0) → 0x02

- Lista podstawowych instrukcji:
  - Porównujące: `CMP` `CMN` `TST` `TEQ`
  - Instrukcje oddziałują na flagi rejestru stanu (CPSR)

CMP – instrukcja porównująca dwa operandy, np.

CPSR = status po operacji (Rz – Operand_2)

CMN – instrukcja porównująca dwa operandy z negacją (czasami asembler potrafi podmienić instrukcję CMP na CMN w celu optymalizacji szybkości wyk. instr.), np.

CPSR = status po operacji (Rz + Operand_2)

TST – instrukcja testująca bity rejestrów (odpowiednik ANDS jednak rezultat operacji jest tracony), np.

CPSR = status po operacji (Rz AND Operand_2)

TEQ – instrukcja porównująca bity rejestrów (nie modyfikuje flag C i V, odpowiednik EORS jednak rezultat operacji jest tracony), np.

CPSR = status po operacji (Rz EOR Operand_2)

Aktualizacja rejestru statusu

- Domyślnie instrukcje nie aktualizują bitów rejestru stanu (nie dotyczy instrukcji porównujących).
- W celu aktualizacji flag rejestru stanu należy posłużyć się sufiksem “S”.

```
loop
    ...
    SUBS    r1, r1, #1      // r1 = r1 -1, update CPRS
    BNE loop                // jump if r1<>0
```

Addition of 64-bit values

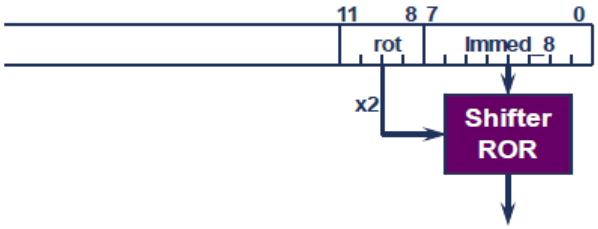
```
    ADDS    r5, r2, r0      // r5 = r2 + r0
    ADDC    r6, r3, r1      // r6 = r3 + r1 + carry
```

Negation of 64-bit value

```
    RSBS    r2, r0, #0      // r2 = - r0
    RSC     r3, r1, #0      // r3 = - r1 - NOT(carry)
```

Operacje z wykorzystaniem stałych liczb

- Żadna z instrukcji asemblera ARM nie może operować na 32-bitowych stałych.
- Ze względu na użytą metodykę kodowania instrukcji pozostało 12 bitów do generowania liczb stałych (argument 2), np. ADD r0, r1, #10000 ?
- Jednostka ALU wykorzystuje dodatkowy układ przesuwnika bitowego (ang. inline barrel shifter) do zwiększenia zakresu generatorowych liczb stałych.
- Daje to możliwość użycia liczby stałej z zakresu 0 - 255, które jest następnie skalowana.
- Możliwość przeskalowania liczby z wykorzystaniem rotacji w zakresie od 0 do 30 (krok co 2) – 4 bitowa wartość pomnożona przez 2.



"8-bitowa wartość przeskalowana przez parzystą liczbę przesunięć w lewo"

- Stałe często wykorzystywane są jako maski bitowe lub wartości przesunięcia:
  - Poprawne wartości:
    - 0xFF, 0x104, 0xFF0, 0xFF00, 0xFF00.0000, 0xF000.000F,
  - Niepoprawne wartości:
    - 0x101, 0x102, 0xFF1, 0xFF04, 0xFF003, 0xF000.001F, (0xFFFF.FFFF),
  - Wpisanie niepoprawnej wartości zwykle kończy się komunikatem:
    - Błąd asemblacji

Przykład użycia przesuwnika bitowego:

- MOV r4, #476 // pole bitowe 0x77 obrócone o wartość 0xF, 476d = 111011100b rot 30
- MOV r7, #2080 // pole bitowe 0x82 obrócone o wartość 0xE, 2080d = 0x820 rot 28
- MOV r4, #473 // błąd podczas asemblacji

Jak załadować pełną liczbę 32-bitową?

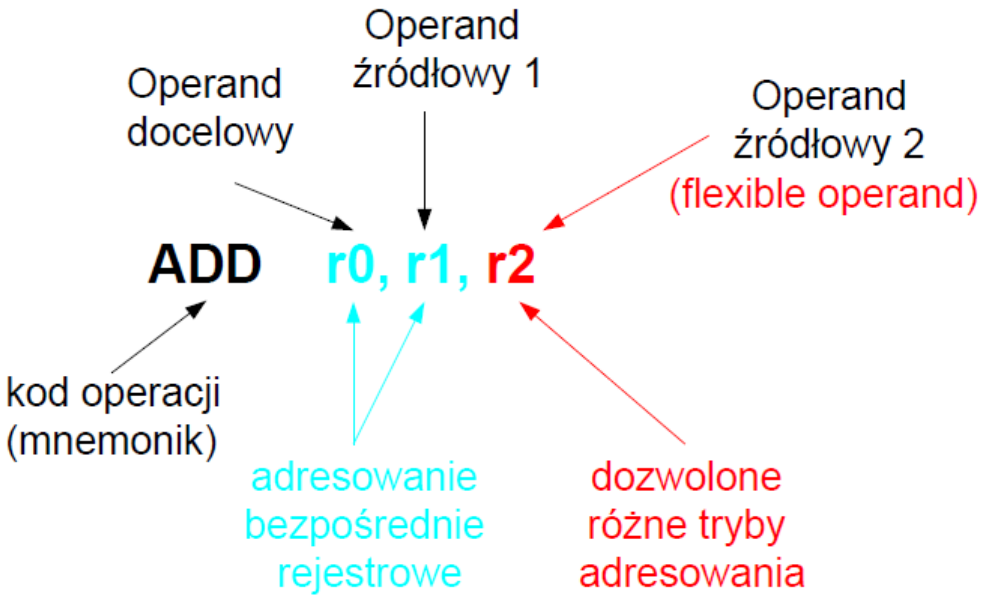
- Do wygodnego posługiwania się pełnym zakresem liczb 32 bitowych wykorzystuje się pseudoinstrukcje asemblera:

```
LDR rd, =const
```

- W takim przypadku asembler posłuży się instrukcjami pozwalającymi na wygenerowanie stałej w jednym cyklu maszynowym (o ile to możliwe), np.
  - Użycie instrukcji MOV lub MVN,
  - Użycie przesuwnika bitowego ROR,
  - Lub użyje stałej umieszczonej w pamięci programu.

```
LDR r0,=0xFF          =>    MOV r0,#0xFF
LDR r0,=0x55555555    =>    LDR r0,[PC,#Imm12]
                            ...
                            ...
                            DCD 0x55555555
```

Tryby adresowania dla procesora z rodziny ARM



Adresowanie natychmiastowe

- Pozwala na zapisanie wartości podanej w postaci natychmiastowej bezpośrednio w rejestrze procesora (r0-r15),

- Przykład:

```
LDR     r0, #-100
LDR     r15, #0xFF00.0000
```

- Jaki tryb adresowani użyty jest w poniższym przykładzie ?

```
LDR     r10, =237685
```

Jest to pseudoinstrukcja i tryb adresowania nie jest znany na etapie pisania programu. Asembler może zastąpić instrukcję pojedynczą instrukcją lub kilkoma instrukcjami w zależności od wartości stałej. Dla dużych wartości (jeżeli nie da się „zbudować" liczby przy użyciu przesuwnika bitowego) liczba pobierana jest z pamięci – adresowanie pośrednie rejestrowe.

Adresowanie bezpośrednie rejestrowe

- Adresowanie bezpośrednie rejestrowe pozwala na bezpośredni dostęp do głównych rejestrów procesora, np. r0-r15.
- Przykład:

  LDR r0, #-100

  LDR r15, #0xFF00.0000
- Tryb adresowania bezpośredniego rejestrowego może zostać dodatkowo zmodyfikowany po wprowadzeniu stałego przesunięcia podawanego w postaci natychmiastowej lub w postaci rejestrowej, np:

  SUB r0, r0, r1, ASR #2

  ADD r1, r2, r3, ASR R4

  W przypadku procesorów ARM, prawie zawsze, wymagane jest użycie bezpośredniego adresowania rejestrowego do przekazywania operandu 1 lub 2 (operand 2 lub 3 może zostać przekazany przy wykorzystaniu innego trybu adresowania).

Adresowanie pośrednie rejestrowe

- W przypadku adresowania pośredniego, rejestr wskazuje na adres w pamięci, w którym znajduje się dana, na której wykonywana jest operacja. Adresowanie pośrednie wykorzystywane jest podczas zapisywania lub odczytywania danej lub danych z pamięci procesora (operacje na stosie)

  Przykłady:
- Adresowanie pośrednie rejestrowe:

  LDRB R4, [R7] zapisanie w rej. R4 danej znajdującej się w pamięci pod adresem wskazywanym przez rejestr R7
- Adresowanie pośrednie rejestrowe z przesunięciem w postaci natychmiastowej:

  STRB R4, [R7, #6] zapisanie danej znajdującej się w rejestrze R4 pod adresem pamięci wskazywanym przez rejestr R7 zwiększonym o 6 bajtów
- Adresowanie pośrednie rejestrowe z przesunięciem w postaci rejestrowej:

  LDR R4, [R7, R0] zapisanie w rej. R4 danej znajdującej się w pamięci pod adresem wskazywanym przez sumę rejestru bazowego R7 i rejestru pomocniczego R0
- Adresowanie rejestrowe pośrednie z indeksowaniem:

  LDR R4, [R7, R0, LSL #2] zapisanie w rej. R4 danej znajdującej się w pamięci pod adresem wskazywanym przez sumę rejestru bazowego R7 oraz przeskalowanego rejestru pomocniczego R0

Adresowanie pośrednie rejestrowe preindeksowane

- W przypadku adresowania pośredniego z preindeksowaniem, przed wykonaniem operacji na danej obliczany jest adres pod którym znajduje się dana. Rejestr bazowy uaktualniany jest przed wykonaniem operacji na danej. Tryb preindeksowany oznaczany jest symbolem wykrzyknika występującym za rejestrem bazowym.

  Przykłady:

- Adresowanie pośrednie rejestrowe preindeksowane wartością w postaci natychmiastowej:

  LDRB R4, [R7, #-36]! obliczenie adresu efektywnego będącego sumą zawartości R7 i stałej natychmiastowej 36, zapisanie wyniku w rej. bazowym R7. Przesłanie zawartości w R4 do pamięci wskazywanej przez nową zawartość rej. R7

- Adresowanie pośrednie rejestrowe preindeksowane rejestrem:

  LDRB R4, [R7, -R6]! obliczenie adresu efektywnego będącego sumą zawartości R7 i rejestru -R6, zapisanie wyniku w rej. bazowym R7. Przesłanie zawartości w R4 do pamięci wskazywanej przez nową zawartość rej. R7

- Adresowanie pośrednie rejestrowe preindeksowane rejestrem z indeksemj:

  LDRB R4, [R7, -R6, LSL #2]! obliczenie adresu efektywnego będącego sumą zawartości R7 i przeskalowanego rejestru -R6, zapisanie wyniku w rej. bazowym R7. Przesłanie zawartości w R4 do pamięci wskazywanej przez nową zawartość rej. R7

Adresowanie pośrednie rejestrowe postindeksowane

- W przypadku adresowania pośredniego z postindeksowaniem, operacja dostępu do pamięci wykonywana z użyciem rejestru bazowego. Po wykonaniu operacji na danej obliczany jest nowy adres pod którym znajduje się dana. Rejestr bazowy uaktualniany jest po wykonaniu operacji na danej. Tryb postindeksowany oznaczany jest przez zapisanie przesunięci poza nawiasem wskazującym adresowanie pośrednie.

  Przykłady:

- Adresowanie pośrednie rejestrowe postindeksowane wartością w postaci natychmiastowej:

  LDRB R4, [R7], #-36 Przesłanie zawartości w R4 do pamięci wskazywanej przez rejestr bazowy R7. Po wykonaniu transferu danej obliczany jest adres efektywny będącego sumą zawartości R7 i stałej natychmiastowej 36, zapisanie wyniku w rej. bazowym R7.

- Adresowanie pośrednie rejestrowe postindeksowane rejestrem:

  LDRB R4, [R7], -R6 Przesłanie zawartości w R4 do pamięci wskazywanej przez rejestr bazowy R7. Po wykonaniu transferu danej obliczany jest adres efektywny będący sumą zawartości R7 i rejestru -R6, zapisanie wyniku w rej. bazowym R7.

- Adresowanie pośrednie rejestrowe postindeksowane rejestrem z indeksem:

  LDRB R4, [R7], -R6, LSL #3 Przesłanie zawartości w R4 do pamięci wskazywanej przez rejestr bazowy R7. Przesłanie zawartości w R4 do pamięci wskazywanej przez rejestr bazowy R7 oraz przeskalowany rejestr R6. Po wykonaniu transferu danej obliczany jest adres efektywny będący sumą zawartości R7

Adresowanie względem licznika rozkazów

- Podczas obliczania adresu efektywnego możliwe jest użycie licznika programów jako rejestru bazowego. W takim przypadku dane przesyłane są w miejsce pamięci zależne od aktualnego położenia programu w pamięci procesora (programy relokowalne) – adresowanie względne z użycie licznika programu.
- Przykłady:

```
LDR     r0, [PC, -#16]
LDR     r0, [PC, R1]
LDR     r0, [PC, R0, ASL #2]
LDR     r0, [PC, -#16] !
LDR     r0, [PC, R1] !
LDR     r0, [PC, R0, ASL #2] !
LDR     r0, [PC], -#16
LDR     r0, [PC], R1
LDR     r0, [PC], R0, ASL #2
```

Rozkazy skoków i pętli.

W procesorze ARM7TDMI dostępne są trzy typy instrukcji skoku:

- B – skok. Jest to najprostsza forma skoku, w której można użyć kodów warunku do podjęcia decyzji, czy wykonać skok do nowego adresu w kodzie, czy też nie.
- BX – skok i wymiana. Oprócz możliwości wykonania bezpośredniego skoku za pomocą zarejestrowanej wartości instrukcja ta dostarcza mechanizmu przełączania się z 32-bitowych instrukcji ARM do 16-bitowych instrukcji THUMB.
- BL – skok i złączenie. W tym przypadku wykorzystywany jest rejestr łączący (R14) do przechowywania adresu powrotu do lokalizacji bezpośrednio po instrukcji skoku, więc jeżeli chcemy wykonać procedurę i wrócić do głównego programu, to procesor musi tylko umieścić zawartość rejestru łączącego do licznika programu.

Argument rozkazu skoku jest 24-bitowy, co po uwzględnieniu, że adresowanie pamięci programu dotyczy słów 32-bitowych pozwala przenosić sterowanie o ±32MB. Jeśli chcemy wykonać skok w dalszy obszar pamięci można skorzystać wpisania do licznika programu zawartości rejestru 32-bitowego, np. BX R4.

Również zamiast tradycyjnej instrukcji powrotu z podprogramu
MOV PC, Lr
można skorzystać z instrukcji
BX Lr

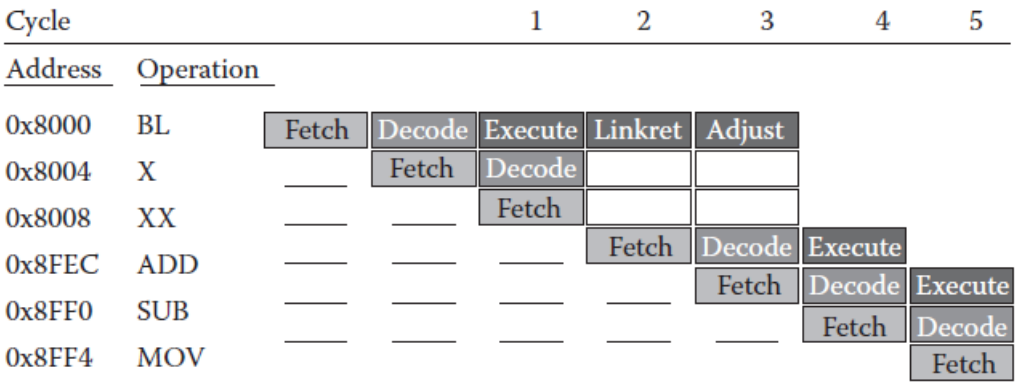


Diagram potoku ARM7TDMI

Wykonanie skoku powoduje problemy w pracy potokowej gdyż w momencie wykonania skoku (faza wykonania instrukcji) w potoku znajdują się już kolejna instrukcja pobrana i instrukcja zdekodowana. Nie będą one wykonane i należy je usunąć z potoku.

Mnemonik	Flagi warunków	Działanie
EQ	Z=1	=
NE	Z=0	!=
CS	C=1	>= (liczba bez znaku)
CC	C=0	< (liczba bez znaku)
MI	N=1	Ujemny
PL	N=0	Dodatni lub 0
VS	V=1	Przepełnienie
VC	V=0	Brak przepełnienia
HI	C=1 i Z=0	> (liczba bez znaku)
LS	C=0 i Z=1	<= (liczba bez znaku)
GE	N=V	>=
LT	N!=V	<
GT	Z=0 i (N=Y)	>
LE	Z=1 lub (N!=V)	<=
AL.	Bez znaczenia	Zawsze

Lista mnemoników warunków w ARM7TDMI

Mnemoniki te mogą być również wykorzystane do warunkowego wykonania dowolnej instrukcji gdyż zapisywane są jako bity 31-28 kodu instrukcji.

Instrukcje pętli

Pętla WHILE

Jest to pętla z nieznaną z góry liczbą powtórzeń. Może być zbudowana w następujący sposób:

```
        B       Test
Loop    …       ; instrukcje
        …
Test    …       ; wartościowanie warunku
        BNE Loop
```

Pętla FOR

Wykonywana jest z góry znaną liczbę razy.

```
for (i = 0; i < 10; i++)
        { instrukcje w pętli }
```

W asemblerze taki kod będzie wyglądał następująco:

```
        MOV r1,  #0             ; i = 0
LOOP    CMP  r1,  #10           ; i < 0?
        BGE  DONE               ; jeśli i >= 10 to koniec
        …..                     ; instrukcje w pętli
        ADD  r1,  #1            ; i ++
        B  LOOP
DONE    …..
```

Znacznie lepszym sposobem jest odliczanie w dół, a nie w górę:

```
        MOV  r1, #10            ; i = 10

LOOP

        ….                      ; instrukcje w pętli

        SUBS  r1, r1, #1        ; i = i-1

        BNE  LOOP

DONE    ….
```

Wykorzystywany jest tylko jeden rozkaz skoku.

Pętla DO … WHILE

Poniżej przedstawione jest ciało pętli wykonywane przed wartościowaniem warunku:

```
LOOP   ….              ; ciało pętli

       ….

       BNE  LOOP

EXIT
```

Operacje na stosie, procedury

Procesory ARM mają wskaźnik stosu w rejestrze r13, który zawiera adres albo następnego pustego elementu, albo ostatniego wypełnionego miejsca w kolejce w zależności od używanego typu stosu.

Instrukcje LDM i STM.

Instrukcje te przesyłają jedno lub więcej słów korzystając z rejestrów i wskaźników do pamięci. Rejestry te są nazywane rejestrami bazowymi. Instrukcje LDM i STM pozwalają zapamiętać stan rejestrów procesora w trakcie obsługi wyjątku lub wywołania procedury, a następnie odtworzyć je. Pojedyncza instrukcja LDM może załadować do 16 rejestrów z pamięci za pomocą jednej instrukcji. Jej składnia jest następująca:

LDM {<warunek>} <tryb_adresowania> <Rn> {!} , <lista_rejestrów> {^}

gdzie {<warunek>} jest opcjonalnym kodem warunku; <tryb_adresowania> określa tryb adresowania instrukcji, który mówi nam, kiedy i jak zmienić rejestr bazowy; <Rn> jest rejestrem bazowym dla operacji ładowania, natomiast <lista_rejestrów> jest rozdzielaną przecinkami listą nazw symbolicznych rejestrów oraz zakresów rejestrów ujętych w nawiasy klamrowe. Kolejność wymienienia adresów na liście nie ma znaczenia, niższe numery rejestrów zawsze ładowane są z niższych adresów pamięci. Rejestr bazowy nie ulega zmianie podczas wykonywania tej operacji chyba że wymusimy jego aktualizację za pomocą opcji {!}

Tryb adresowania określa kiedy i w którym kierunku zmienia się wartość z adresu bazowego:

- IA – postinkrementacja,
- IB – preinkrementacja,
- DA – postdekrementacja,
- DB – predekrementacja.

Podobną składnię ma instrukcja STM, która służy do zapisywania zawartości rejestrów do pamięci pod wskazanym adresem:

STM {<warunek>} <tryb_adresowania> <Rn> {!} , <lista_rejestrów> {^}

Przykłady użycia instrukcji:

STMIA r9, {r0-r3, r12}

LDMIA r9, {r0-r3, r12}

Operacje na stosach są łatwe do zaimplementowania z wykorzystaniem instrukcji LDM i STM. Rejestrem bazowym jest wtedy r13 (wskaźnik stosu). Dostępne są następujące opcje:

- Rosnący lub malejący – stos rośnie w dół, zaczyna się od wyższych adresów i jest kontynuowany w kierunku niższych (stos malejący), lub rośnie w górę, a więc zaczyna się od niższych adresów i jest kontynuowany w kierunku wyższych (stos rosnący).
- Pełny lub pusty – wskaźnik stosu pokazuje na ostatni element stosu (pełny stos) lub następne wolne miejsce na stosie (pusty stos).

W celu ułatwienia programowania operacji na stosie można używać przyrostków zorientowanych na stosy:

Typ stosu	Położenie	Zdejmowanie
Pełny malejący	STMFD (STMDB)	LDMFD (LDMIA)
Pełny rosnący	STMFA (STMIB)	LDMFA (LDMDA)
Pusty malejący	STMED (STMDA)	LDMED (LDMIB)
Pusty rosnący	STMEA (STMIA)	LDMEA (LDMDB)

Podprogramy

Można wywoływać za pomocą instrukcji BL (skok i złączenie), która przenosi adres początkowy podprogramu do licznika programu, jak również przesyła adres powrotu do rejestru łączącego r14, dzięki czemu podprogram może wrócić do wywołującego go programu. Gdy wywołany podprogram chciałby wywołać inny, to należałoby zachować rejestr łączący na stosie wraz z innymi rejestrami. Można wtedy wykonać powrót za pomocą jednej instrukcji LDM. W przeciwnym razie nastąpi przepisanie jego zawartości innymi danymi i brak możliwości powrotu do programu głównego.

Przekazywanie parametrów do podprogramów.

- Przekazywanie przez rejestry.
- Przekazywanie przez umieszczenie w pamięci i przekazanie referencji przez rejestr.
- Przekazywanie parametrów przez stos z wykorzystaniem rejestru r13.

Standard ARM APCS

Dla architektury ARM zdefiniowany jest standard o nazwie ARM Application Procedure Call Standard (AAPCS), wchodzący w skład Application Binary Interface (ABI), który definiuje sposób pisania podprogramów, ich osobnej kompilacji i asemblacji, aby podprogramy mogły ze sobą współpracować. Opisuje on kontrakt pomiędzy wywołującym a wywoływanym podprogramem:

- obowiązki wywołującego w zakresie utworzenia stanu programu, w którym wywołana procedura może zacząć działać,
- obowiązki wywoływanej procedury w zakresie zachowania stanu programu,
- prawa wywoływanej procedury w zakresie zmiany stanu programu.

Dokument ten opisuje procedury pisania kodu, jak również definiuje użycie rejestrów:

- Pierwsze cztery rejestry r0-r3 są używane do przekazywania wartości argumentów do podprogramów i do zwracania wartości z funkcji. Mogą być również wykorzystane do przechowywania wartości roboczych w podprogramie.
- Rejestr r12 (IP) może być używany przez linker jako rejestr roboczy zapewniający połączenie między głównym programem a wywołaną przez niego procedurą. Może być również wykorzystywany wewnątrz procedury do przechowywania wartości tymczasowych pomiędzy wywołaniami podprogramów.
- Zwykle rejestry r4-r8, r10 i r11 są używane do przechowywania zmiennych lokalnych w procedurze.
- Podprogram musi zachowywać zawartość rejestrów r4-r8, r10, r11 i SP.
- Stosy są wyrównywane do ośmiu bajtów, a kompilatory ARM i THUMB c i C++ zawsze używają pełnego stosu malejącego.