



Dariusz Makowski

**Katedra Mikroelektroniki i Technik
Informatycznych**

tel. 631 2720

dmakow@dmcs.pl

<http://neo.dmcs.p.lodz.pl/swcr>



Zakres przedmiotu

- ◆ Systemy mikroprocesorowe, systemy wbudowane
- ◆ Rodzina procesorów ARM
- ◆ **Asembler**
- ◆ Urządzenia peryferyjne
- ◆ Pamięci i dekodery adresowe
- ◆ Programy wbudowane na przykładzie procesorów ARM
- ◆ Metodyki projektowania systemów wbudowanych
- ◆ Interfejsy w systemach wbudowanych
- ◆ Systemy czasu rzeczywistego

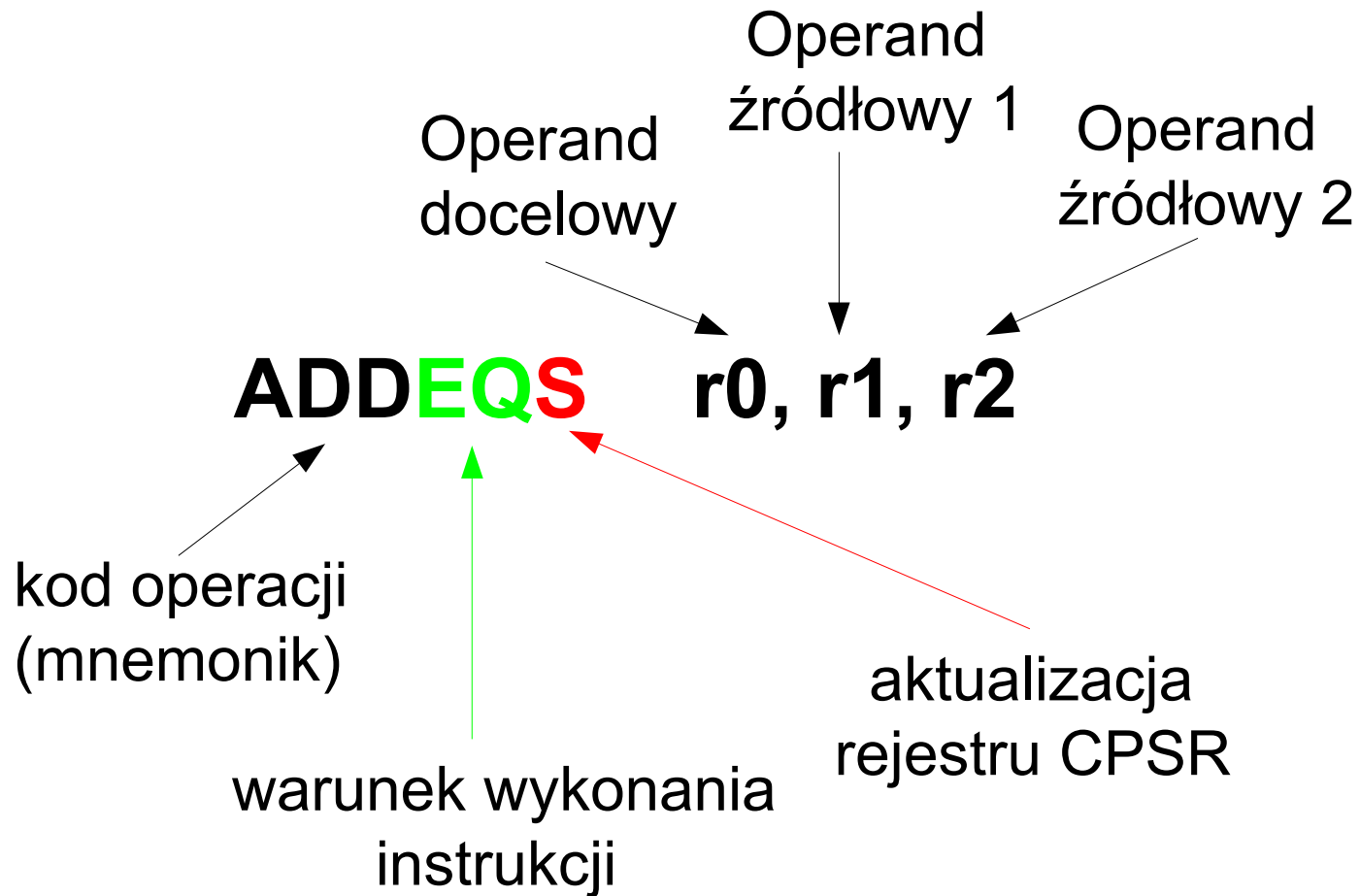


Zajęcia odwołane

- 14.04.2014 – Zajęcia odwołane, odróbka w maju



Asembler procesora ARM (1)





Model programowy – rejestry procesora

Procesor ARM posiada łącznie 37 rejestrów (wszystkie są 32 bitowe):

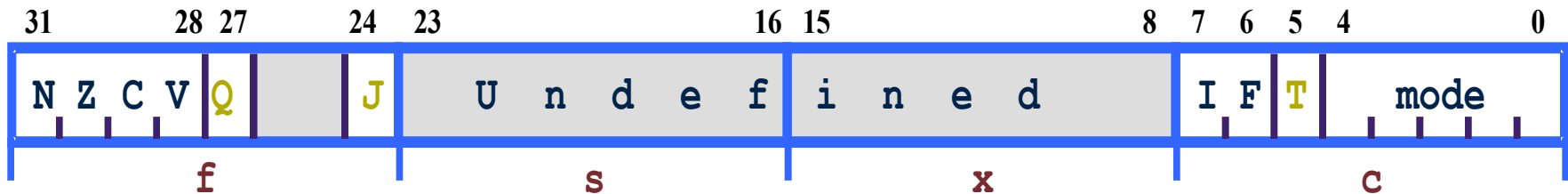
- **PC (r15)** – licznik programu (Program Counter)
- **CPSR** – rejestr statusowy, obecny status (Current Program Status Register)
- **SPSR** – rejestr statusowy, dostępne w różnych trybach uprzywilejowania (Saved Program Status Register)
- **LR (r14)** – rejestr powrotu (Link Register), wykorzystywany 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 różnych trybach uprzywilejowania procesora



Rejestr statusowy CPSR/SPSR procesora AMR9



Wskaźniki stanu

- V – przepełnienie podczas operacji ALU (oVerflow)
- C – przeniesienie/pożyczka podczas operacji ALU
- Z – zerowy wynik podczas operacji ALU
- N – ujemny wynik operacji ALU lub „mniejszy niż”

Wskaźniki dostępne jedynie dla architektury 5TE/J

- J – Procesor w trybie Jazelle
- Q – Sticky Overflow – wskaźnik nasycenia podczas operacji ALU (QADD, QDADD, QSUB or QDSUB, lub rezultat operacji SMLAxy or SMLAWx przekracza 32-bity)

Przerwania

- I=1 Przerwania IRQ wyłączone
- F=1 Przerwania FIQ wyłączone

Wskaźniki dostępne dla arch. xT

- T=0 Tryb pracy ARM
- T=1 Tryb pracy Thumb

Tryb pracy procesora

- Definiują jeden z 7 trybów



Asembler procesora ARM (2)

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



Flexible Operand

Operand źródłowy drugi (tzw. Flexible Operand 2) może zostać podany w postaci rejestru, wartości stałej lub wartości skalowanej:

- ◆ Rx , np. R8
- ◆ #wartość_stała , np. #5
- ◆ Rx, operacja_skalowania , np. LSR #5, np:

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

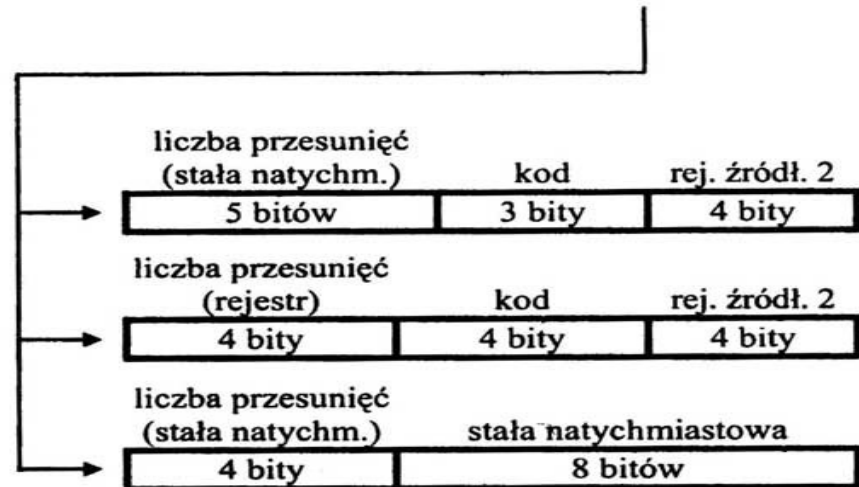
Immediate value	#<imm8m>
Register, optionally shifted by constant (see below)	Rm {, <opsh>}
Register, logical shift left by register	Rm, LSL Rs
Register, logical shift right by register	Rm, LSR Rs
Register, arithmetic shift right by register	Rm, ASR Rs
Register, rotate right by register	Rm, ROR Rs



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

warunek	kod grupy r.	kod rozkazu	uaktualnienie CPSR	rejestr docelowy	pierwszy operand	pole Operand_2
4 bity	3 bity	4 bit	1 bit	4 bity	4 bity	12 bitów





Podział instrukcji procesora ARM

- 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.

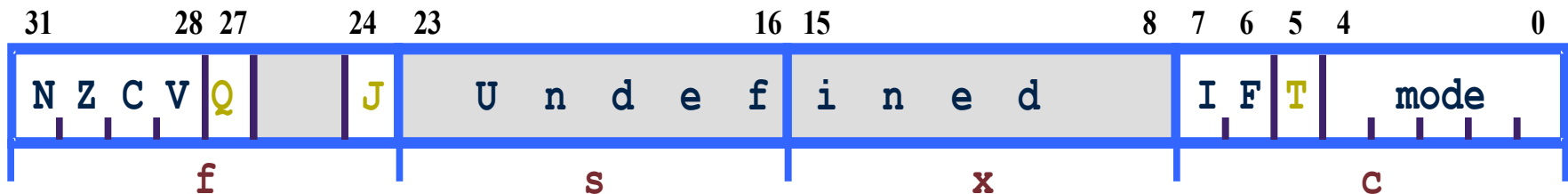


Instrukcje procesora a kod maszynowy

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data processing immediate shift	cond [1]	0	0	0	opcode				S	Rn				Rd				shift amount				shift		0	Rm								
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x x x x x x x x x x x x x x x x x x																0	x x x x						
Data processing register shift [2]	cond [1]	0	0	0	opcode				S	Rn				Rd				Rs				0	shift		1	Rm							
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x x x x x x x x x x x x x x x x x x																0	x x		1	x x x x			
Multiplies: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0	0	0	x	x	x	x	x	x x x x x x x x x x x x x x x x x x																1	x x		1	x x x x			
Data processing immediate [2]	cond [1]	0	0	1	opcode				S	Rn				Rd				rotate				immediate											
Undefined instruction	cond [1]	0	0	1	1	0	x	0	0	x x x x x x x x x x x x x x x x x x x x x x x x x x x x																							
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask				SBO				rotate				immediate											
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn				Rd				immediate															
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn				Rd				shift amount				shift		0	Rm								
Media instructions [4]: See Figure A3-2	cond [1]	0	1	1	x	x	x	x	x	x x x x x x x x x x x x x x x x x x x x																1	x x x x						
Architecturally undefined	cond [1]	0	1	1	1	1	1	1	1	x x x x x x x x x x x x x x x x												1	1	1	1	x x x x							
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn				register list																			
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																											
Coprocessor load/store and double register transfers	cond [3]	1	1	0	P	U	N	W	L	Rn				CRd				cp_num				8-bit offset											
Coprocessor data processing	cond [3]	1	1	1	0	opcode1				CRn				CRd				cp_num				opcode2		0	CRm								
Coprocessor register transfers	cond [3]	1	1	1	0	opcode1				L	CRn				Rd				cp_num				opcode2		1	CRm							
Software interrupt	cond [1]	1	1	1	1	swi number																											
Unconditional instructions: See Figure A3-6	1	1	1	1	x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x																												



Rejestr stanu i flagi statusu



Condition Code	Meaning
N	Negative condition code, set to 1 if result is negative
Z	Zero condition code, set to 1 if the result of the instruction is 0
C	Carry condition code, set to 1 if the instruction results in a carry condition
V	Overflow condition code, set to 1 if the instruction results in an overflow condition.

* rejestr stanu xSR dla architektury instrukcji powyżej ARMv5 może zawierać dodatkowo bity statusu dla instrukcji SIMD, bit kontrolujący kolejność bitów w pamięci dla instrukcji Load/Store oraz bit Imprecise Abort Mask



Model programowy – rejestry dostępne w trybie User oraz System

Current Visible Registers

User Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr

Banked out Registers

(20 rejestrów)

FIQ

IRQ

SVC

Undef

Abort

r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr



Model programowy – dostępne tryby pracy procesora

Tryb pracy procesora (operating mode) - określa jakie zasoby procesora są dostępne, np. dostępne rejestry, obszary pamięci, urządzenia peryferyjne.

Procesory ARM mogą pracować w jednym z siedmiu trybów pracy:

- ◆ **User** – tryb użytkownika (nieuprzywilejowany) przeznaczony do wykonywania programów użytkownika,
- ◆ **FIQ** – tryb obsługujący przerwania i wyjątki o wysokich priorytetach (szybki)
- ◆ **IRQ** – obsługa przerwania z niskim priorytetem (low/normal priority)
- ◆ **Supervisor** – tryb pracy superużytkownika, dostęp do wszystkich zasobów procesora (dostępny po resecie lub przerwanie programowe)
- ◆ **Abort** – obsługa wyjątków związanych z pamięcią (memory access violations)
- ◆ **Undef** – obsługa nieznanych/błędnych rozkazów
- ◆ **System** – tryb pracy superużytkownika, dostęp do rejestrów takich jak w trybie User jednak możliwy dostęp do różnych obszarów pamięci

Mode	Abbreviation	Privileged	Mode[4:0]
<i>Abort</i>	abt	yes	10111
<i>Fast interrupt request</i>	fiq	yes	10001
<i>Interrupt request</i>	irq	yes	10010
<i>Supervisor</i>	svc	yes	10011
<i>System</i>	sys	yes	11111
<i>Undefined</i>	und	yes	11011
<i>User</i>	usr	no	10000



Model programowy – rejestry dostępne w trybie User oraz System

Current Visible Registers

User Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr

Banked out Registers

FIQ

IRQ

SVC

Undef

Abort

r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr



Model programowy – rejestry dostępne w trybie FIQ

Current Visible Registers

FIQ Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

IRQ

r13 (sp)
r14 (lr)

SVC

r13 (sp)
r14 (lr)

Undef

r13 (sp)
r14 (lr)

Abort

r13 (sp)
r14 (lr)

spsr

spsr

spsr

spsr



Model programowy – rejestry dostępne w trybie IRQ

Current Visible Registers

IRQ Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User

FIQ

SVC

Undef

Abort

r13 (sp)
r14 (lr)

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

spsr

spsr

spsr

spsr



Model programowy – rejestry dostępne w trybie Supervisor

Current Visible Registers

SVC Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User

FIQ

IRQ

Undef

Abort

r13 (sp)
r14 (lr)

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

spsr

spsr

spsr

spsr



Model programowy – rejestry dostępne w trybie Abort

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr



Model programowy – rejestry dostępne w trybie Undef

Current Visible Registers

Undef Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User

FIQ

IRQ

SVC

Abort

r13 (sp)
r14 (lr)

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

spsr

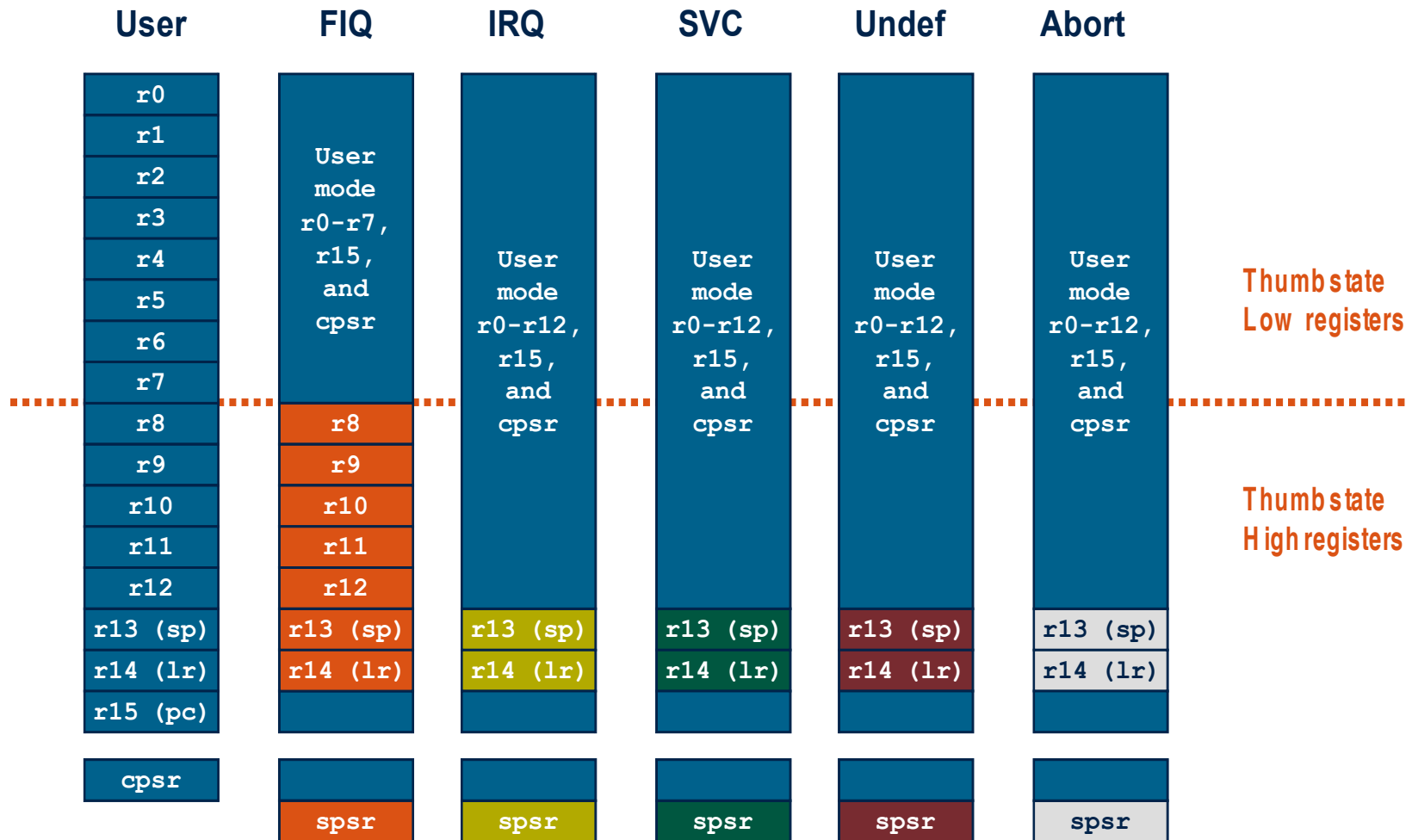
spsr

spsr

spsr



Model programowy – podsumowanie dostępnych rejestrów



Note: System mode uses the User mode register set



Przykładowy program

/ here you can write comments */*

```
main:                                /* label for main function */
    MOV    r0, #10                    /* set up parameters */
    MOV    r1, #3
    ADD    r0, r0, r1                /* r0 = r0 + r1 */

stop:
    LDR    r5, =0xffff.f200          /* base address for PIO_B */
    ADD    r0, r0, #1                /* increment r0 */
    STR    r0, [r5]                  /* write r0 to PIO_B */
    SWI    #0x10                     /* software interrupt */

while:
    B       while                    /* while (1) to avoid exception */
    END                                /* mark end of file */
```



Podstawowe instrukcje procesora ARM

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



Instrukcje asemblera procesora ARM (ARM and Thumb-2 Instruction Set Quick Reference Card)



Gdzie szukać pomocy ?

- J. Augustyn, „Projektowanie systemów wbudowanych na przykładzie rodziny SAM7S z rdzeniem ARM7TDMI”
- Lista instrukcji wspieranych przez rodzinę ARM
 - ❖ **ARM and Thumb-2 Instruction Set Quick Reference Card**
 - ❖ Thumb 16-bit Instruction Set Quick Reference Card
- Opis architektury rdzenia procesora ARM:
 - ❖ **ARMv5 Architecture Reference Manual**
 - ❖ ARMv6-M Architecture Reference Manual
 - ❖ ARMv7-AR Architecture Reference Manual
 - ❖ ARMv7-M Architecture Reference Manual
 - ❖ ARMv7-M Architecture Application Level Reference Manual
 - ❖ ARM v7-M Architecture Application Level Reference Manual Errata
- Materiały dostępne na stronie firmy ARM (<https://login.arm.com>)



Wspierana lista instrukcji asemblera

ARM architecture versions	
n	ARM architecture version n and above
nT, nJ	T or J variants of ARM architecture version n and above
5E	ARM v5E, and 6 and above
T2	All Thumb-2 versions of ARM v6 and above
6K	ARMv6K and above for ARM instructions, ARMv7 for Thumb
Z	All Security extension versions of ARMv6 and above
RM	ARMv7-R and ARMv7-M only
XS	XScale coprocessor instruction

Numer rdzenia ARM oraz numer wersji instrukcji:

- ARM7TDMI → ARMv4
- ARM9TDMI-E-JS → ARMv5
- ARM Cortex → ARMv7



Notacja (1)

Key to Tables

Rm {, <opsh>}	See Table Register, optionally shifted by constant
<Operand2>	See Table Flexible Operand 2 . Shift and rotate are only available as part of Operand2.
<fields>	See Table PSR fields .
<PSR>	Either CPSR (Current Processor Status Register) or SPSR (Saved Processor Status Register)
C*, V*	Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later.
<Rs sh>	Can be Rs or an immediate shift value. The values allowed for each shift type are the same as those shown in Table Register, optionally shifted by constant .
x, y	B meaning half-register [15:0], or T meaning [31:16].
<imm8m>	ARM: a 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits. Thumb: a 32-bit constant, formed by left-shifting an 8-bit value by any number of bits, or a bit pattern of one of the forms 0xXYXYXYXY, 0x00XY00XY or 0xXY00XY00.
<prefix>	See Table Prefixes for Parallel instructions
{IA IB DA DB}	Increment After, Increment Before, Decrement After, or Decrement Before. IB and DA are not available in Thumb state. If omitted, defaults to IA.
<size>	B, SB, H, or SH, meaning Byte, Signed Byte, Halfword, and Signed Halfword respectively. SB and SH are not available in STR instructions.



Notacja (2)

Key to Tables

<reglist>	A comma-separated list of registers, enclosed in braces { and }.
<reglist-PC>	As <reglist>, must not include the PC.
<reglist+PC>	As <reglist>, including the PC.
<flags>	Either nzcvg (ALU flags PSR[31:27]) or g (SIMD GE flags PSR[19:16])
§	See Table ARM architecture versions .
+/-	+ or -. (+ may be omitted.)
<iflags>	Interrupt flags. One or more of a, i, f (abort, interrupt, fast interrupt).
<p_mode>	See Table Processor Modes
SPm	SP for the processor mode specified by <p_mode>
<lsb>	Least significant bit of bitfield.
<width>	Width of bitfield. <width> + <lsb> must be ≤ 32.
{X}	RsX is Rs rotated 16 bits if X present. Otherwise, RsX is Rs.
{!}	Updates base register after data transfer if ! present (pre-indexed).
{S}	Updates condition flags if S present.
{T}	User mode privilege if T present.
{R}	Rounds result to nearest if R present, otherwise truncates result.



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

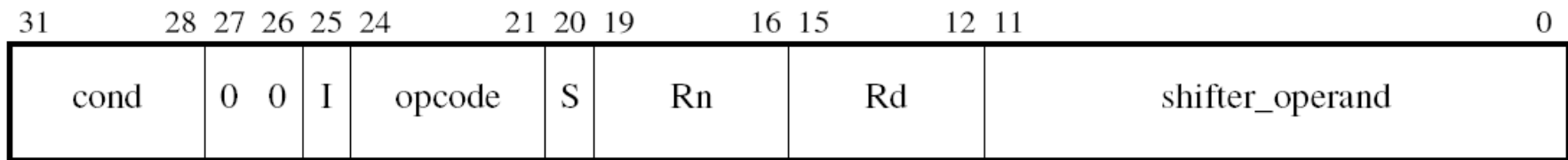
◆ Drugi operand przesyłany jest do ALU z wykorzystaniem rejestru przesuwne (ang. barrel shifter)

- ◆ Rozkazy 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 // $r0 = r1 + r2 + \text{carry}$

Odejmowanie

SUB r0, r1, r2 // $r0 = r1 - r2$

Odejmowanie z uwzględnieniem bitu przeniesienia

SBC r0, r1, r2 // $r0 = r1 - r2 - \text{NOT}(\text{carry})$

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 // $r3 = r2 - r1 - \text{NOT}(\text{carry})$



Ćwiczenia przy tablicy

Sumowanie liczb dłuższych niż 32 bity

- R1, R2 – liczba 64 bitowa (np. 0x1234.5678.ABCD.EF00)
- R3, R4 – liczba 64 bitowa (np. 0xAAAA.BBBB.CCCC.DDDD)
- $R0 = (R1|R2) + (R3|R4)$
- Przykładowy program ?
- Jaki będzie wynik ?
- Jak długi może być wynik ?

Odejmowanie liczb dłuższych niż 32 bity

- R7, R8 – liczba 64 bitowa (np. 0x1234.5678.ABCD.EF00)
- R10, R12 – liczba 64 bitowa (np. 0xAAAA.BBBB.CCCC.DDDD)
- $R14 = (R7|R8) - (R10|R12)$
- Przykładowy program ?
- Jaki będzie wynik ?
- Jak długi może być wynik ?



Instrukcje arytmetyczne – aktualizacja flag rejestru stanu

Operation		§	Assembler	S updates
Add	Add		ADD{S} Rd, Rn, <Operand2>	N Z C V
	with carry		ADC{S} Rd, Rn, <Operand2>	N Z C V
	wide	T2	ADD Rd, Rn, #<imm12>	
	saturating {doubled}	5E	Q{D}ADD Rd, Rm, Rn	
Address	Form PC-relative address		ADR Rd, <label>	
Subtract	Subtract		SUB{S} Rd, Rn, <Operand2>	N Z C V
	with carry		SBC{S} Rd, Rn, <Operand2>	N Z C V
	wide	T2	SUB Rd, Rn, #<imm12>	N Z C V
	reverse subtract		RSB{S} Rd, Rn, <Operand2>	N Z C V
	reverse subtract with carry		RSC{S} Rd, Rn, <Operand2>	N Z C V
	saturating {doubled}	5E	Q{D}SUB Rd, Rm, Rn	
	Exception return without stack		SUBS PC, LR, #<imm8>	



Instrukcje logiczne (1)

Lista podstawowych instrukcji:

Logiczne: **AND** **ORR** **EOR** **BIC**

AND – operacja iloczynu logicznego, np.

$0x12 \text{ AND } 0xF0 \rightarrow 0x10$

ORR – operacja sumy logicznej, np.

$0xAC \text{ ORR } 0x10 \rightarrow 0xBC$

EOR – operacja różnicy symetrycznej, np.

$0x12 \text{ XOR } 0xF0 \rightarrow 0xD0$

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

$0x12 \text{ AND } (\text{NOT } 0xF0) \rightarrow 0x02$



Instrukcje logiczne (2)

Lista podstawowych instrukcji:

Logiczne: **AND** **ORR** **EOR** **BIC**

```
LDR    R0, =0xDEADBEEF
```

```
AND     R0, R0, #0x00FFFF00
```

(gdb) p/x \$r0

\$0 = 0xdeadbeef

```
44    AND        r0, r0, #0x00FFFF00
```

```
1: x/i $pc 0x10010 <MAIN+12>: and r0,r0,#0x00ffff00
```

(gdb) s

(gdb) p/x \$r0

Jaki będzie wynik operacji ?

\$1 = 0x00ADBE00



Instrukcje logiczne (3)

Lista podstawowych instrukcji:

Logiczne: **AND** **ORR** **EOR** **BIC**

```
LDR    R1, =0xDEAD.BEEF
```

```
ORR     R0, R1, #0x00FF.FF00
```

```
(gdb) p/x $r1
```

```
$0 = 0xdeadbeef
```

```
44    ORR      r0, r1, #0x00FFFF00
```

```
1: x/i $pc 0x10010 <MAIN+12>: orr r0,r1,#0x00ffff00
```

```
(gdb) s
```

```
(gdb) p/x $r0
```

Jaki będzie wynik operacji ?

```
$1 = 0xDEFF.FFEF
```



Instrukcje logiczne (4)

Lista podstawowych instrukcji:

Logiczne: **AND** **ORR** **EOR** **BIC**

```
LDR    R2, =0xABCD.EF12
```

```
ORR     R1, R2, #0xFF00.000F
```

```
(gdb) p/x $r2
```

```
$0 = 0xdeadbeef
```

```
44    EOR        r1, r2, #0x00FFFF00
```

```
1: x/i $pc 0x10010 <MAIN+12>: eor r1,r2,#0x00ffff00
```

```
(gdb) s
```

```
(gdb) p/x $r1
```

Jaki będzie wynik operacji ?

```
$1 = 0x54cdef1d
```



Instrukcje logiczne (5)

Lista podstawowych instrukcji:

Logiczne: **AND** **ORR** **EOR** **BIC**

```
LDR      R2, =0xABCD.EF12
```

```
BIC      R0, R2, #0xFF00.00FF      */ AND NOT /*
```

```
(gdb) p/x $r2
```

```
$0 = 0xdeadbeef
```

```
44      BIC          r0, r2, #0x00FFFF00
```

```
1: x/i $pc 0x10010 <MAIN+12>: bic r0,r2,#0x00ffff00
```

```
(gdb) s
```

```
(gdb) p/x $r0
```

Jaki będzie wynik operacji ?

```
$1 = 0x00cd.ef00
```



Instrukcje logiczne (1)

♦ 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 assembler 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)



Operacje porównania – flagi rejestru stanu

Count leading zeros		§	Assembler	S updates
Compare	Compare		CMP Rn, <Operand2>	N Z C V
	negative		CMN Rn, <Operand2>	N Z C V
Logical	Test		TST Rn, <Operand2>	N Z C
	Test equivalence		TEQ Rn, <Operand2>	N Z C
	AND		AND{S} Rd, Rn, <Operand2>	N Z C
	EOR		EOR{S} Rd, Rn, <Operand2>	N Z C
	ORR		ORR{S} Rd, Rn, <Operand2>	N Z C
	ORN	T2	ORN{S} Rd, Rn, <Operand2>	N Z C
	Bit Clear		BIC{S} Rd, Rn, <Operand2>	N Z C



Instrukcje logiczne (3)

Przykłady użycia instrukcji porównujących

CMP	r2, r9	porównanie rejestrów r2 i r9 (zmienia wszystkie flagi)
CMN	r0, #6400	porównanie rejestru r0 i stałej (zmienia wszystkie flagi)
CMP	GT r13, r7, LSL #2	test wykonywany warunkowo jeżeli flaga Z=0 i oraz N=V, porównanie rejestru r13 i wartości będącej wynikiem operacji $r7 \ll 2$
TST	r0, #0x3F8	test bitowy rejestru r0 i stałej #0x3f8
TEQ	EQ r10, r9	test bitowy rejestrów r0 i r9 (zmienia flagi Z/N)
TST	NE r1, r5, ASR r1	test wykonywany warunkowo jeżeli flaga Z nie jest ustawiona (zmienia wszystkie flagi) operand_2 = $r5 \gg r1$



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

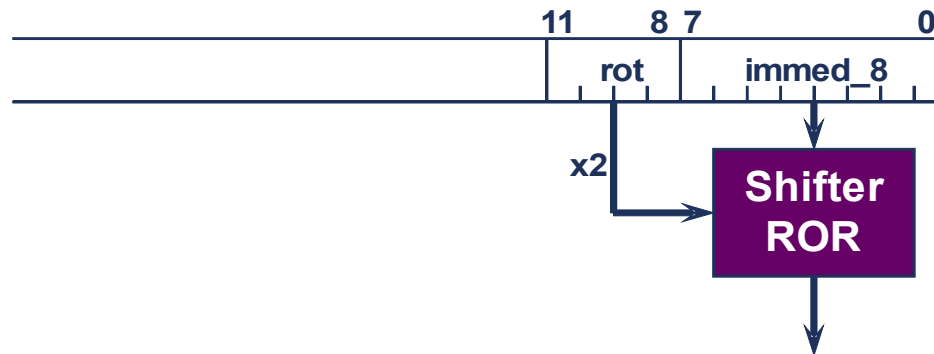
```
ADDS    r5, r2, r0      // r5 = r2 + r0  
ADDC    r6, r3, r1      // r6 = r3 + r1 + carry
```

Negation of 64-bit value

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

Operacje z wykorzystaniem stałych liczb (1)

- Żadna z instrukcji assemblera 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”



Operacje z wykorzystaniem stałych liczb (2)

- 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 ?

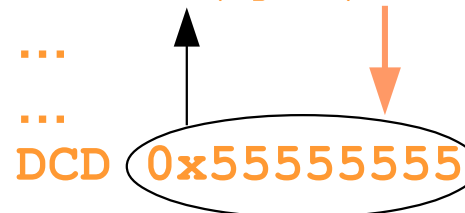
- Do wygodnego posługiwania się pełnym zakresem liczb 32 bitowych wykorzystuje się pseudoinstrukcje asemblera:

LDR rd, =const

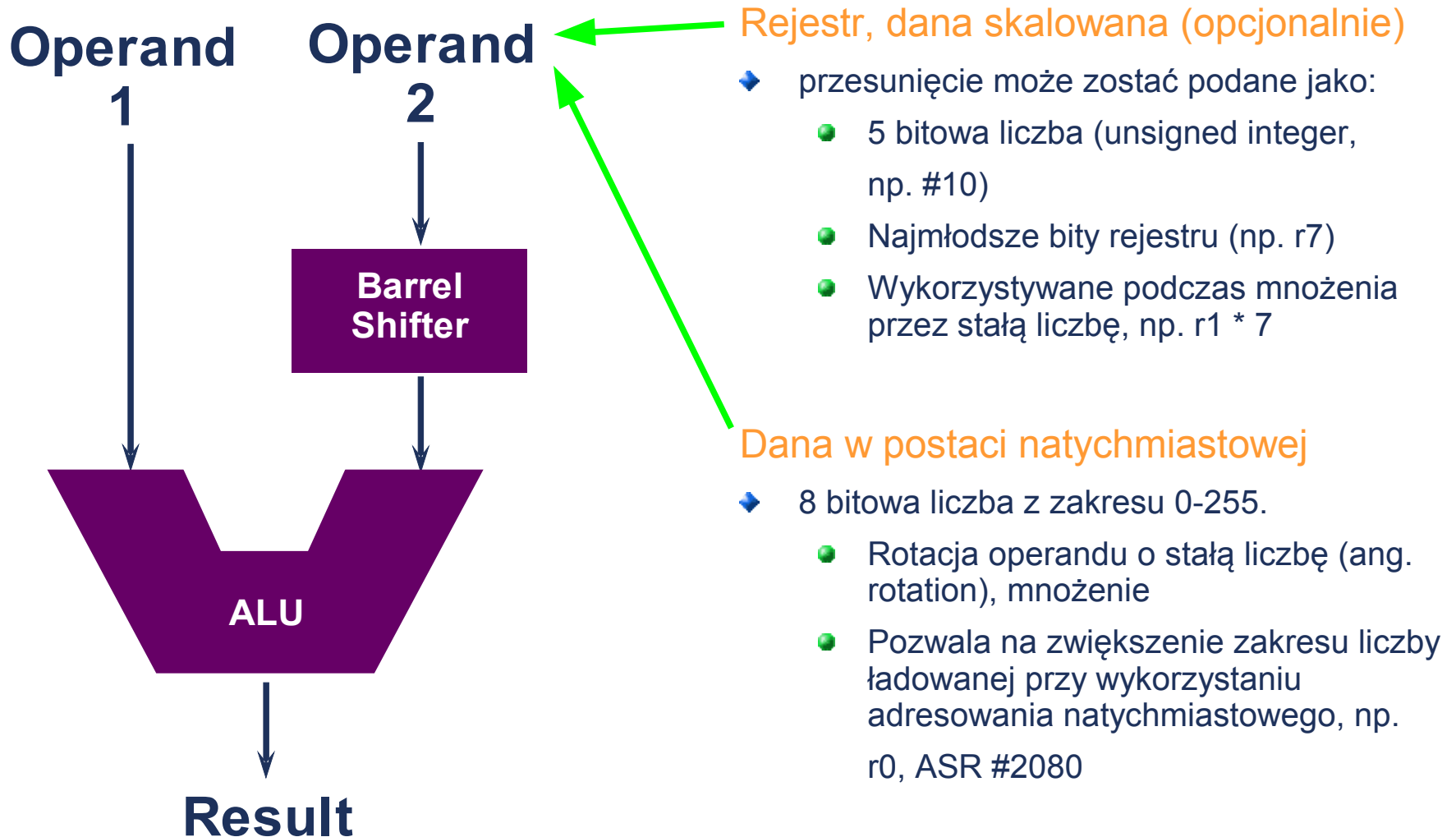
- W takim przypadku asembler posłuży się instrukcjami pozwalającymi na wygenerowanie stałej w jednym cyklu maszynowym (o ile to możliwe), np.
 - Użycie instrukcji MOV lub MVN,
 - Użycie przesuwnika bitowego ROR,
 - Lub użycie stałej umieszczonej w pamięci programu.

LDR r0, =0xFF => **MOV r0, #0xFF**

LDR r0, =0x55555555 => **LDR r0, [PC, #Imm12]**



Przesuwnik bitowy (1)



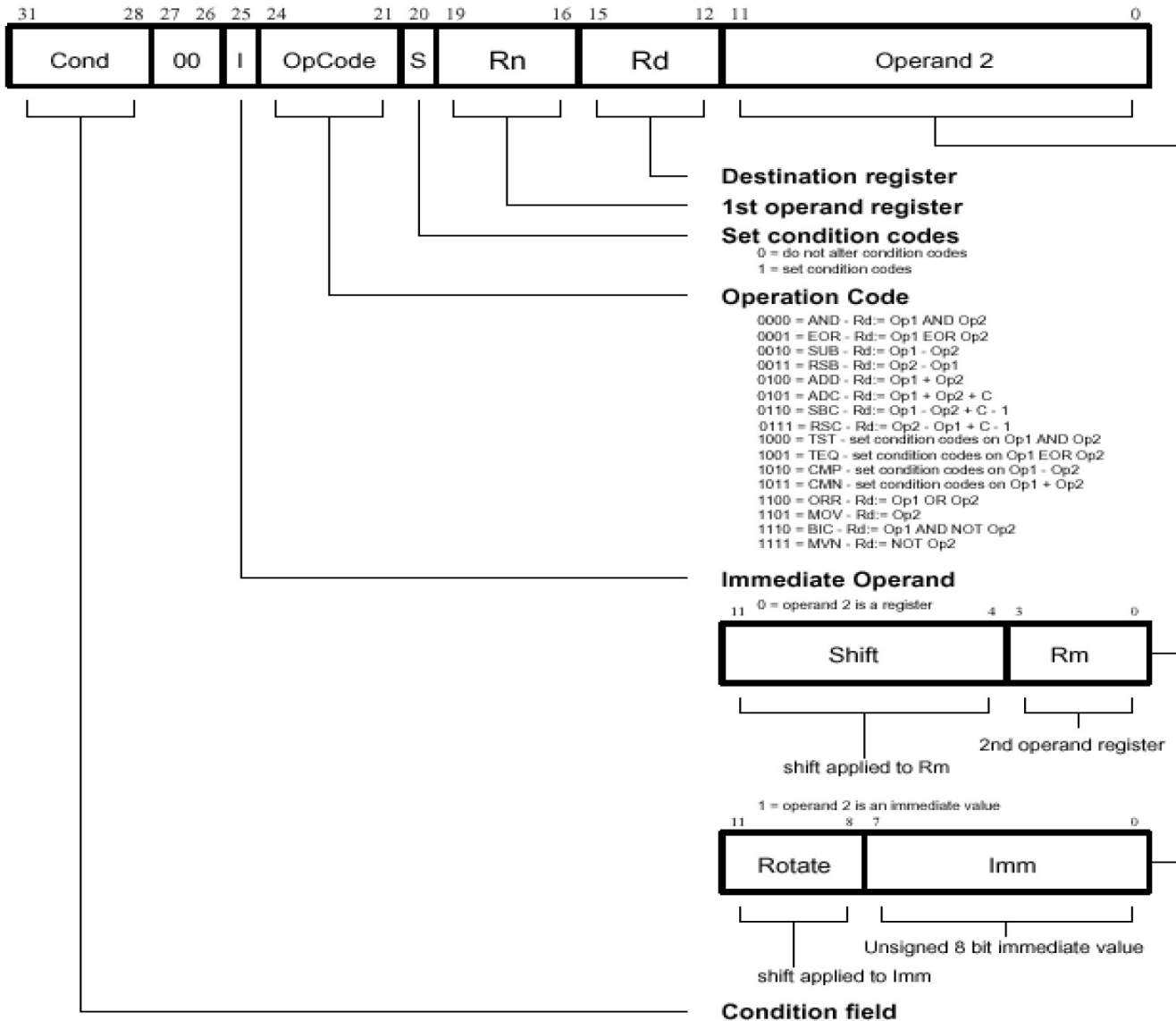
Przesuwnik bitowy (2)

Możliwe operacje z wykorzystaniem przesuwnika bitowego:

- LSL (Logical Shift Left) – przesunięcie logiczne w lewo (0 - 31),
- LSR (Logical Shift Right) – przesunięcie logiczne w prawo (1 - 32),
- ASR (Arithmetic Shift Right) – przesunięcie arytmetyczne w prawo (1 - 32),
- ROR (Rotate Right) – rotacja w prawo (1 - 31),
- RRX (Rotate Right Extend) – rotacja w prawo z uwzględnieniem flagi C.

Rn,	Rm,	LSL	#imm5 lub Rs
		LSR	
		ASR	
		ROR	
		RRX	
#imm8_4			

- #imm5 – pięciobitowy argument określający liczbę rotacji/przesunięć podawany z użyciem adresowania natychmiastowego,
- Rs - argument określający liczbę rotacji/przesunięć podawany z użyciem adresowania rejestrowego bezpośredniego (podczas operacji wykorzystywane jest 8 najmłodszych bitów rejestru),
- #imm8_4 – 8 bitowa stała natychmiastowa poddana parzystej liczbie rotacji w prawo (1-31 = 4 bity), co pozwala na konstrukcję liczby: 0-255 ze skokiem o 1, 0 – 1020 ze skokiem co cztery, 0- 4080 ze skokiem co 16, itd... (używane do konstrukcji liczb będących potęgą liczby 2, maski bitowe, przesunięcia tablicy itd...).

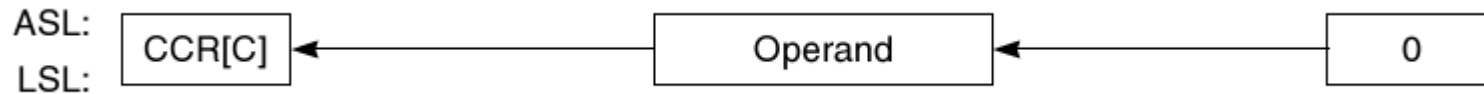




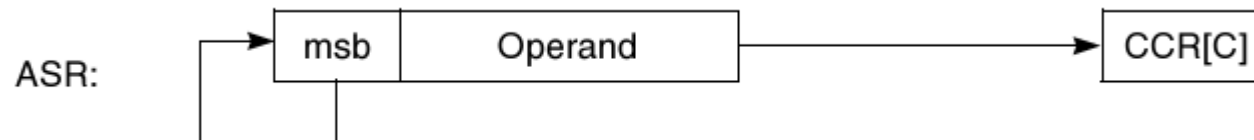
Przesuwnik bitowy - przesunięcie logiczne vs arytmetyczne (3)

- Podczas wykonywania operacji przesunięcia logicznego bity uzupełniane są zerami
- Przesunięcie arytmetyczne zachowuje znak operacji

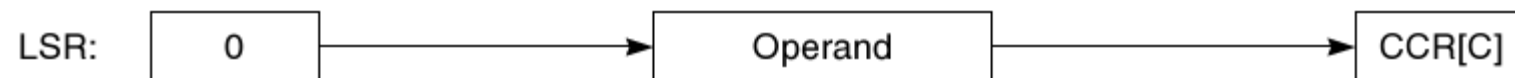
Przesunięcie logiczne/arytmetyczne w lewo:



Przesunięcie arytmetyczne w prawo (zachowany znak operandu):



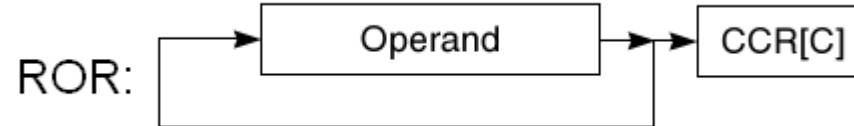
Przesunięcie logiczne w prawo:



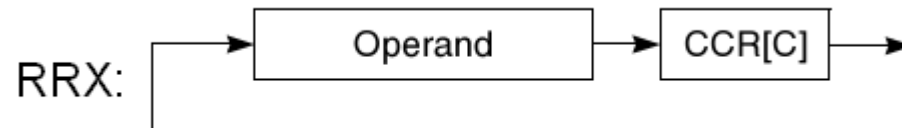
Przesuwnik bitowy - przesunięcie vs rotacja(4)

- Podczas wykonywania operacji rotacji bity „wysuwane” z rejestru wprowadzane są na drugi jego koniec, np. lsb \rightarrow msb,
- Możliwość realizacji operacji rotacji z uwzględnieniem flagi przeniesienia C,
- Procesor ARM udostępnia rozkazy umożliwiające wykonanie rotacji w prawo.

Operacja rotacji w prawo:



Operacja rotacji w prawo z uwzględnieniem flagi C:





Przesuwnik bitowy (5)

Przykłady:

SUB r4, r5, **r6, ASR #2**

Opis: przesunięcie arytmetyczne operandu w rej. r6 o dwa bity w prawo, odjęcie od rejestru r5 wyniku przesunięcia i zapisanie rezultatu operacji w rejestrze r4.

MOV r4, #0

MOV r5, #100

MOV r6, #16

SUB r4, r5, r6, ASR #2

Wynik: 96



Przesuwnik bitowy (5)

Przykłady:

ADD r3, r5, **r7, LSL r2**

Opis: przesunięcie arytmetyczne w lewo operandu w rej. r7 o liczbę bitów kreśloną w rejestrze r2, dodanie rezultatu operacji przesunięcia do rejestru r5 zapisanie rezultatu operacji w rejestrze r3.

MOV r3, #0

MOV r5, #10

MOV r7, #32

MOV r2, #3

ADD r3, r5, r7, LSL r2

Wynik: 266



Przesuwnik bitowy (5)

Przykłady:

ADD r5, r5, **r1, LSR #4**

Opis: przesunięcie arytmetyczne w prawo operandu w rej. r1 o 4, dodanie rezultatu operacji przesunięcia do rejestru r5 zapisanie rezultatu operacji w rejestrze r5.

MOV r5, #10

MOV r1, #1024

MOV r2, #3

ADD r5, r5, r1, LSR #3

Wynik: 138

SUB r2, r3, **r8, ROR #3** // r3=1000, r8=1 r2=1.073.742.824

ADD r2, r7, **r1, RRX** // r7=1000, r1=1 r2=1000



Przesuwnik bitowy – podsumowanie (6)

Flexible Operand 2		
Immediate value	#<immed_8r>	
Logical shift left immediate	Rm, LSL #<shift>	Allowed shifts 0-31
Logical shift right immediate	Rm, LSR #<shift>	Allowed shifts 1-32
Arithmetic shift right immediate	Rm, ASR #<shift>	Allowed shifts 1-32
Rotate right immediate	Rm, ROR #<shift>	Allowed shifts 1-31
Register	Rm	
Rotate right extended	Rm, RRX	
Logical shift left register	Rm, LSL Rs	
Logical shift right register	Rm, LSR Rs	
Arithmetic shift right register	Rm, ASR Rs	
Rotate right register	Rm, ROR Rs	



Przesuwnik bitowy – problemy do rozwiązania (7)

1. Napisać funkcję ABS (Absolute Value) operująca na rejestrze r7 przy użyciu tylko 2 instrukcji asemblera

```
MOVS      r7, r7           // set flags
RSBMI     r7, r7, #0       // if negative, r7 = 0 - r7
```

2. Wykonać efektywną operację mnożenia rejestru r10 przez 35 (mnożenie powinno zostać wykonane w 2 cyklach maszynowych)

```
ADD       r9, r8, r8, LSL #2    // r9 = (r8*4 + r8)
RSB       r10, r9, r9, LSL #3   // r10 = (r9*8 - r9)
```




Warunkowe wykonywanie instrukcji (1)

- Instrukcje procesora ARM mogą być wykonywane warunkowo,
- Warunek instrukcji określany jest przez dodanie odpowiedniego sufiksu do instrukcji,
- Warunkowe wykonanie instrukcji nie wydłuża czasu potrzebnego na ich realizację,
- Taki mechanizm umożliwia przyspieszenie wykonywania instrukcji, zmniejsza wielkość generowanego kodu oraz zmniejsza liczbę kosztownych operacji skoku (synchronizacja potoku).

Przykład:

```
CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
ADDNE  r0,r1,r2
```

loop

...

```
SUBS  r1,r1,#1
```

```
BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch



Tablica warunków

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

- ❖ Prawie wszystkie instrukcje ARM lub Thumb-2 mogą zostać wykonane warunkowo.
- ❖ W przypadku rodziny rozkazów Thumb tylko instrukcja B może zostać wykonana warunkowo.
- ❖ Brak warunku oznacza bezwarunkowe wykonanie instrukcji (AL).



Utworzenie nowej grupy lab.

- | | |
|------------------------|--------|
| 1. Jakub Macioch | 164402 |
| 2. Krzysztof Krotosz | 164383 |
| 3. Dawid Szubert | 164467 |
| 4. Rafał Szmit | 152487 |
| 5. Maciej Kociołek | 152436 |
| 6. Piotr Sadowski | 164456 |
| 7. Anna Marciniak | 164405 |
| 8. Kevin Karolak (IFE) | |

→ poniedziałek >14.00

→ piątek (cały dzień)



Warunkowe wykonywanie instrukcji - porównanie

C source code

```
if (r0 == 0)
{
    r1 = r1 + 1;
}
else
{
    r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
CMP r0, #0
BNE else
ADD r1, r1, #1
B end
else
    ADD r2, r2, #1
end
...
```

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

- 3 instructions
- 3 words
- 3 cycles



Przykład użycia instrukcji warunkowych (1)

- ◆ Kod w języku C dla algorytmu Euklidesa GDC (ang. Greatest Common Divider) zwracającego największy dodatni wspólny całkowity dzielnik liczby a oraz b.

- ◆ Przykład implementacji:

```
int GDC (int a, int b) {  
    int c;  
    while (b != 0)  
    {  
        c = a % b;  
        a = b;  
        b = c;  
    }  
    return a;  
}
```

GDC(1071,1029) => GDC(1029,1071 mod 1029 = 42)

GDC(1029,42) => GDC(42, 1029 mod 42 = 21)

GDC(42,21) => GDC(21,42 mod 21 = 0) => GDC(1071,1029) = 21



Przykład użycia instrukcji warunkowych (2)

- Kod programu w języku C

```
int GDC (int a, int b) {  
    while (a<>b) {  
        if (a>b) then  
            a=a-b;  
        else  
            b=b-a;  
    }  
    return a;  
}
```

- Kod asemblera bez użycia instrukcji warunkowych

```
GDC:  
    CMP     r0, r1  
    BEQ     End  
    BLT     Less  
    SUB     r0, r0, r1  
    BAL     GDC  
  
Less:  
    SUB     r1, r1, r0  
    BAL     GDC  
  
End:
```

- Kod asemblera z wykorzystaniem instrukcji warunkowych

```
GDC:  
    CMP     r0,r1  
    SUBGT   r0, r0, r1  
    SUBLT   r1, r1, r0  
    BNE     GDC  
  
End:
```

Instrukcje rozgałęzienia programu (1)

- Instrukcja skoku (rozgałęzienie programu)

Branch: **B{<cond>}** label

- Instrukcja skoku z zachowaniem adresu powrotu (wywołania funkcji)

Branch with Link: **BL{<cond>}** subroutine_label



- Podczas obliczania adresu skoku rdzeń procesora przesuw offset o 2 bity w lewo, rozwija przesunięcie z zachowaniem znaku do 32 bitów i dodaje do licznika programu PC (r15)
- Umożliwia to wykonanie skoków względnych w zakresie ± 32 MB pamięci,
- Instrukcje skoków umożliwiają pisanie programów relekownalnych



Struktura warunkowa SWITCH w asemblerze (1)

```
char    TEST;                TEST:        .byte 0x01
...
...
switch (TEST){
    case 0:
        ....
        break;

    case 1:
        ....
        break;

    case 2:
        ....
        break;

    default:
        ....
        break;
}
```

TEST: .byte 0x01
.align 4



Ćwiczenie przy tablicy

- Proszę napisać program realizujący strukturę warunkową SWITCH posługując się asemblerem

```
CASE:  
    LDR        r5, =TEST
```

```
.data  
TEST: .byte 0x00
```



Struktura warunkowa SWITCH w assemblerze (2)

```
CASE:
    LDR        r5, =TEST
    LDR        r0, [r5]                | check TEST variable
    BEQ        ACT1
    SUBS       #1, r0
    BEQ        ACT2
    SUBS       #1, r0
    BEQ        ACT3
OTHERS:
    ADD        r7, #0                | default condition
    BRA        EXIT

ACT1:
    ADD        r7, #100                | case if TEST=0
    BRA        EXIT

ACT2:
    ADD        r7, #200                | case if TEST=1
    BRA        EXIT

ACT3:
    ADD        r7, #300                | case if TEST=2
    BRA        EXIT

EXIT:
.data
TEST: .byte 0x00
```

Proszę wskazać problemy w powyższym programie



Struktura warunkowa SWITCH w assemblerze – poprawione błędy

```
CASE:
    LDR        r5, =TEST
    LDRBS    r0, [r5]                // check TEST variable
    MOVS    r0, r0
    BEQ     ACT1
    SUBS     r0, r0, #1
    BEQ     ACT2
    SUBS     r0, r0, #1
    BEQ     ACT3
OTHERS:
    ADD        r7, r7, #0            // default condition
    BAL    EXIT
ACT1:
    ADD        r7, r7, #100          // case if TEST=0
    BAL    EXIT
ACT2:
    ADD        r7, r7, #200          // case if TEST=1
    BAL    EXIT
ACT3:
    ADD        r7, r7, #300          // case if TEST=2
```

EXIT:

Kolorem czerwonym zaznaczono błędy (za wyjątkiem modyfikatora „S”)



Konwersja HEX na ASCII (1)

Proszę napisać program konwertujący liczby podane w postaci szesnastkowej do kodów ASCII.

HEX -> ASCII

0 -> ?

1 -> ?

... -> ...

A -> ?

... -> ...

F -> ?

R1 - licznik 0-15

R0 - tymczasowy wynik ASCII

.data

.align 4

ASCII: .byte 0x00, 0x01, 0x02...

Wynik

ASCII: .byte 0x30, 0x31, 0x32...

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del



Konwersja HEX na ASCII (2)

LDR r9, =ASCII

....

MOV r1, #0

Next:

LDRB r0, [r9, r1]
ADD r0, r0, #0x30
CMP r0, #0x39
ADDGT r0, #0x7

STRB r0, [r9, r1]
ADD r1, r1, #1
CMP r1, #16
BNE Next

B Return

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del



Operacje mnożenia (1)

- Dostępne są dwie grupy instrukcji generujące wynik 32 lub 64 bitowy,
- Instrukcje umożliwiające operacje typu dodaj i sumuj (MLA),
- Instrukcje wykonują się ciągu 2-5 cykli (ARM7TDMI).

Instrukcje generujące 32-bitowy wynik mnożenia (młodsza część wyniku)

- **MUL r0, r1, r2** // $r0 = r1 * r2$
- **MLA r0, r1, r2, r3** // $r0 = (r1 * r2) + r3$

Instrukcje generujące 64-bitowy wynik mnożenia (wymagają dwóch rejestrów docelowych)

- **[U|S]MULL r4, r5, r2, r3** // $r5/r4 = r2 * r3$
- **[U|S]MLAL r4, r5, r2, r3** // $r5/r4 = (r2 * r3) + r5/r4$

Cześć rdzeni ARM nie dostarcza sprzętowych instrukcji dzielenia, dzielenie realizowane jest przez dostępne biblioteki (ANSI C library), podobnie jak obliczenia zmiennoprzecinkowe.



Operacje mnożenia (32 bit) – przykłady (2)

MUL Rd, Rz, Rs // mnożenie $Rd = Rz * Rs$

MLA Rd, Rz, Rs, Rn // mnożenie $Rd = Rn + Rz * Rs$

❖ Nie można używać rejestru PC jako rejestru docelowego oraz jako operandu

❖ Rozkazy mogą być wykonywane warunkowo

MULEQ r0, r1, r2 // if Z=1 then $r0 = r1 * r2$

MULS r3, r0, r2 // $r3 = r0 * r2$, update CPSR

MLANES r3, r0, r2, r7 // if Z=0 then $r3 = r7 + (r0 * r2)$, update CPSR



Operacje mnożenia (64 bit) – przykłady (3)

UMULL	Rdl, Rdh, Rm, Rs	// Rdh/Rdl = Rm * Rs (unsigned)
SMULL	Rdl, Rdh, Rm, Rs	// Rdh/Rdl = Rm * Rs (signed)
UMLAL	Rdl, Rdh, Rm, Rs	// Rdh/Rdl = Rdh/Rdl + (Rm * Rs) (unsigned)
SMLAL	Rdl, Rdh, Rm, Rs	// Rdh/Rdl = Rdh/Rdl + (Rm * Rs) (signed)

❖ Nie można używać rejestru PC jako rejestru docelowego oraz jako operandu

❖ Rozkazy mogą być wykonywane warunkowo

SMULLVSS r0, r1, r2, r3 // if OV=1 then r1/r0 = r2 * r3, update CPRS

UMLALALS r4, r8, r3, r2 // r8/r4 = r8/r4 + (r3 * r2), update CPSR

❖ Czas wykonywania instrukcji MULL/MLAL jest o jeden cykl zegara dłuższy niż MUL/MLA



Operacje mnożenia (4)

Operation		§	Assembler	S updates
Multiply	Multiply		MUL{S} Rd, Rm, Rs	N Z C*
	and accumulate		MLA{S} Rd, Rm, Rs, Rn	N Z C*
	and subtract	T2	MLS Rd, Rm, Rs, Rn	
	unsigned long		UMULL{S} RdLo, RdHi, Rm, Rs	N Z C* V*
	unsigned accumulate long		UMLAL{S} RdLo, RdHi, Rm, Rs	N Z C* V*
	unsigned double accumulate long	6	UMAAL RdLo, RdHi, Rm, Rs	
	Signed multiply long		SMULL{S} RdLo, RdHi, Rm, Rs	N Z C* V*
	and accumulate long		SMLAL{S} RdLo, RdHi, Rm, Rs	N Z C* V*
	16 * 16 bit	5E	SMULxy Rd, Rm, Rs	
	32 * 16 bit	5E	SMULWy Rd, Rm, Rs	
	16 * 16 bit and accumulate	5E	SMLAxy Rd, Rm, Rs, Rn	
	32 * 16 bit and accumulate	5E	SMLAWy Rd, Rm, Rs, Rn	
	16 * 16 bit and accumulate long	5E	SMLALxy RdLo, RdHi, Rm, Rs	
	Dual signed multiply, add	6	SMUAD{X} Rd, Rm, Rs	
	and accumulate	6	SMLAD{X} Rd, Rm, Rs, Rn	
	and accumulate long	6	SMLALD{X} RdLo, RdHi, Rm, Rs	
	Dual signed multiply, subtract	6	SMUSD{X} Rd, Rm, Rs	
	and accumulate	6	SMLSD{X} Rd, Rm, Rs, Rn	
	and accumulate long	6	SMLSLD{X} RdLo, RdHi, Rm, Rs	
	Signed top word multiply	6	SMMUL{R} Rd, Rm, Rs	
	and accumulate	6	SMMLA{R} Rd, Rm, Rs, Rn	
	and subtract	6	SMMLS{R} Rd, Rm, Rs, Rn	
	with internal 40-bit accumulate	XS	MIA Ac, Rm, Rs	
	packed halfword	XS	MIAPH Ac, Rm, Rs	
	halfword	XS	MIAXy Ac, Rm, Rs	



Najprostszy algorytm dzielenia

Dzielenie polega na odejmowaniu od liczby dzielonej liczby przez którą dzielimy w pętli, dopóki wartość ta jest dodatnia. Po zakończeniu obliczeń licznik pętli wskazuje wynik, reszta z dzielenia znajduje się w rejestrze od którego dokonywano odejmowania.

Przykładowy program:

```
MOV    r1, #128 // divide R1
MOV    r2, #4    // by R2
MOV    r0, #0    // initialise counter
```

subtract:

```
SUBS   r1, r1, r2 // subtract R2 from R1 and store result back in R1 setting flags
ADD    r0, r0, #1 // add 1 to counter, NOT setting flags
BHI    subtract // branch to start of loop on condition Higher, i.e. R1 is still greater
                // than R2
```

// Answer now in R0, remainder in R1

Powyższy algorytm jest nieefektywny dla znacznie różniących się liczb (ze względu na znaczną liczbę operacji odejmowania jaką należy wykonać)



Dzielenie z przesunięciem (1)

Algorytm polega na zwiększeniu liczby przez, którą dzielimy przesuwając ją w lewo, dopóki jest ona mniejsza od liczby dzielonej. Następnie uzyskana liczba odejmowana jest od liczby dzielonej (znacznie mniej operacji odejmowania), co daje pierwszą (najbardziej znaczącą liczbę wyniku dzielenia). W kolejnym kroku następuje przesunięcie liczby przez, którą dzielimy w prawo i rozpoczęcie kolejnego dzielenia. Operację powtarzamy do momentu kiedy liczba przez którą dzielimy będzie równa zeru.

Przykład dzielenia $128/4$ (wynik 32):

- Przesuń 4 o jedną pozycję w lewo, co daje 40.
- Przesuń 40 o jedną pozycję w lewo, co daje 400. Liczba jest większa od 128, zbyt duża wartość. Wracamy do liczby 40.
- Wykonujemy odejmowanie w celu obliczenia pierwszej cyfry (najbardziej znaczącej):
 - ♦ $128 - 40 = 88$ (licznik = 1),
 - ♦ $88 - 40 = 48$ (licznik = 2),
 - ♦ $48 - 40 = 8$ (licznik = 3),
- Nie da się już więcej odjąć, pierwsza cyfra wyniku dzielenia to 3,
- Przesuwamy liczbę, którą odejmujemy o 1 pozycję w prawo, co daje 4.
- Wykonujemy odejmowanie w celu obliczenia drugiej cyfry:
 - ♦ $8 - 4 = 4$ (licznik = 1),
 - ♦ $4 - 4 = 0$ (licznik = 2),
- ♦ Nie ma możliwości przesunięcia liczby, którą dzieliliśmy o kolejną pozycję w prawo, kończymy obliczenia.
- ♦ Wynik: 32, reszta: 0.



Dzielenie z przesunięciem (2)

Przykładowa funkcja wykonująca dzielenie liczb dodatnich, arytmetyka dwójkowa.

// Enter with numbers in Ra and Rb. Rcnt = 32 bit variable

// Ra = 50, Rb 10 => Rc = 50/10 = 5, Ra = 0

```
MOV          Rcnt, #1          // Bit to control the division.
Div1:  CMP          Rb, #0x80000000 // Do not work with negative
                                           // numbers
        CMPCC       Rb, Ra      // Move Rb until greater than
                                           // Ra., C=0 (lower)
        MOVCC       Rb, Rb, ASL#1
        MOVCC       Rcnt, Rcnt, ASL#1
        BCC         Div1
Div2:  MOV          Rc, #0
        CMP         Ra, Rb      // Test for possible subtraction.
        SUBCS       Ra, Ra, Rb  // Subtract if ok, C=1 (higher or
                                           // same)
        ADDCS       Rc, Rc, Rcnt // put relevant bit into result
        MOVS        Rcnt, Rcnt, LSR#1 // shift control bit
        MOVNE       Rb, Rb, LSR#1 // halve unless finished.
        BNE         Div2       // Divide result in Rc,
                                           // remainder in Ra.
```

1 krok (Div1):

50d = 11.0010b
10d = 1010b
10<<1 = 1.0100b
10<<2 = 10.1000b
10<<3 = 101.0000b
Rcnt = 1000b (8d)

2 krok (Div2):

Pierwsza cyfra:
Rcnt = 1000b
11.0010b – 10.1000b =
1010b, Rc = 100b
Druga cyfra:
1010b – 1.0100b wynik
ujemny, Rc = 100b
Trzecia cyfra:
1010b – 1010b = 0,
Rc=101b, Ra = 0



Tablica warunków

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

- ❖ Prawie wszystkie instrukcje ARM lub Thumb-2 mogą zostać wykonane warunkowo.
- ❖ W przypadku rodziny rozkazów Thumb tylko instrukcja B może zostać wykonana warunkowo.
- ❖ Brak warunku oznacza bezwarunkowe wykonanie instrukcji (AL).



Instrukcje transferu danych

Architektura ARM udostępnia dwie instrukcje umożliwiające dostęp do pamięci:

- ♦ LDR – odczyt danej z pamięci (load)
- ♦ STR – zapis danej do pamięci (store)
 - Syntaks:

LDR{<size>}{<cond>} Rd, <address>

STR{<size>}{<cond>} Rd, <address>

Instrukcje LDR/STR umożliwiają dostęp do pamięci dla argumentów o długości 8, 16 i 32 bitów ze znakiem lub bez znaku (signed/unsigned):

LDR STR Word (32 bit)

LDRB STRB Byte (8 bit)

LDRH STRH Halfword (16 bit)

LDRSB Signed byte load

LDRSH Signed halfword load

Instrukcje mogą być realizowane warunkowo, np. **LDRSBEQ**, **STRBMI**



Instrukcje transferu danych

◆ Syntaks:

LDR{<size>}{<cond>} Rd, <address>

STR{<size>}{<cond>} Rd, <address>

Operand <address> określa tryb adresowania i może być zbudowany:

- ◆ Adresu bazowego znajdującego się w rejestrze (rejestr bazy). Rejestrem bazy może być każdy rejestr, również r15 (PC). W przypadku użycia rejestru PC jest to adresowanie względne.
- ◆ Przesunięcia względem adresu bazowego (dodatnie lub ujemne). Przesunięcie może zostać podane w postaci:
 - ➔ Wartości natychmiastowej (szerokość zależy od rodzaju rozkazu). Maksymalna wartość +/- 4095 bajtów (dla transferów 32-bitowych lub 8-bitów bez znaku) lub +/- 255 bajtów (dla transferów 16-bitowych i bajtów ze znakiem),
 - ➔ Wartości rejestru,
 - ➔ Skalowanej wartości rejestru (nie dotyczy transferów 16-bitowych i 8-bitów ze znakiem).

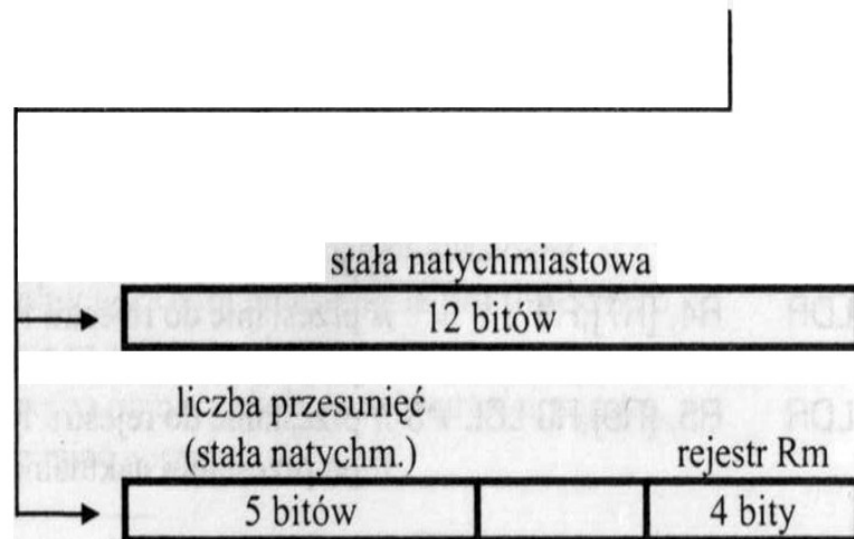


Format instrukcji (1)

- Format instrukcji dla transferów 32-bitowych oraz bajtów bez znaku.

waru- nek	kod grupy r.	LDR/ STR	word / bajt	post-idx / pre-idx	rej. źródło- wy/docelowy	rejestr bazy Rb	uaktual- nienie Rb	±	Przesunięcie
--------------	-----------------	-------------	----------------	-----------------------	-----------------------------	--------------------	-----------------------	---	--------------

4 bity	3 bity	1 bit	1 bit	1 bit	4 bity	4 bity	1 bit	1 bit	12 bitów
--------	--------	-------	-------	-------	--------	--------	-------	-------	----------

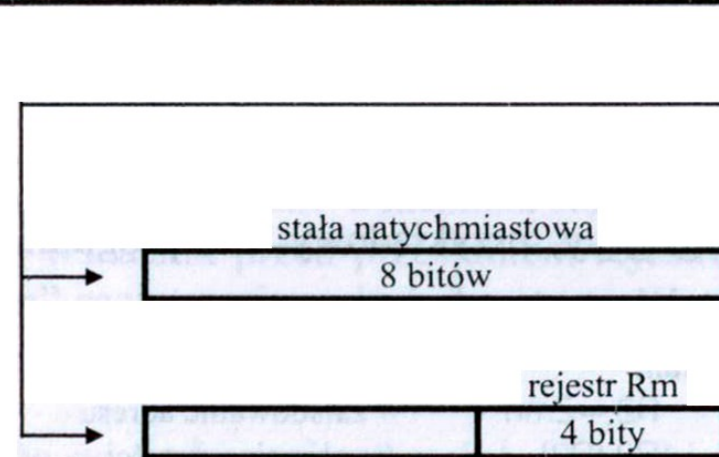


- Load and Store Word or Byte
 - LDR / STR / LDRB / STRB

Format instrukcji (2)

- Format instrukcji dla transferów 16-bitowych (liczby bez znaku: zapis i odczyt, liczby ze znakiem: odczyt) oraz bajtów ze znakiem (odczyt).

warunek	kod grupy r.	kod pod grupy r.	LDR/STR	H / SH / SB	post-idx / pre-idx	rej. źródłowy/docelowy	rejestr bazy Rb	uaktualnienie Rb	±	przesunięcie
4 bity	3 bity	3bity	1 bit	2 bity	1 bit	4 bity	4 bity	1 bit	1 bit	8 bitów



- Load and Store Halfword
 - LDRH / STRH
- Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
 - LDRSB / LDRSH

Adresowanie pośrednie rejestrowe

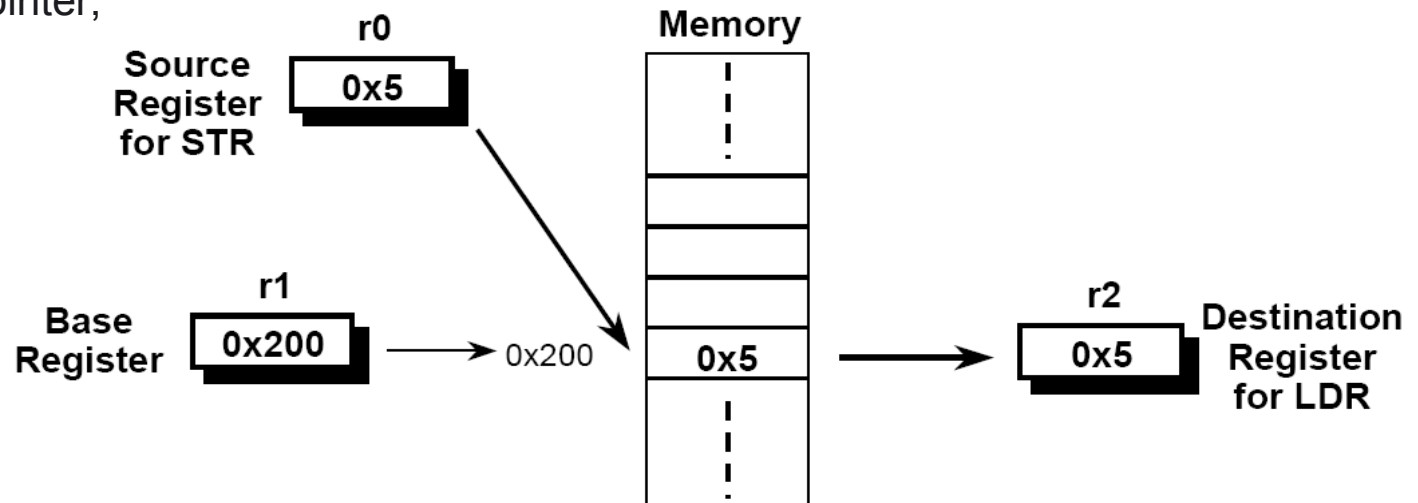
- Odpowiednik wskaźnika w języku C, np.

```
int IntData, r2Data;
```

```
int r1Pointer = &IntData;
```

```
*r1Data = 5;
```

```
r2Data = *r1Pointer;
```



```
STR    r0, [r1]    // store r0 to memory pointed by r1
LDR    r2, [r1]    // load r2 with data pointed by r1
```



Instrukcje transferu danych - przykłady

Przykłady użycia adresu z przesunięciem podanym w postaci:

- ◆ Natychmiastowej (liczba 12-bitowa bez znaku, 0-4095 bajtów)

LDR r0, [r1, #8] // adresowanie pośrednie rej. z przesunięciem w postaci
// natychmiastowej

- ◆ Rejestru (r0-r15), opcjonalnie przesuniętego przez stałą wartość:

LDR r0, [r1, r2] // adr. pośrednie rejestrowe z przesunięciem. w postaci rej.
LDR r0, [r1, r2, LSL#2] // adr. pośrednie rejestrowe z przesunięciem i indeksem

- ◆ Wartość przesunięcia może zostać dodana lub odjęta od rejestru bazowego:

LDR r0, [r1, #-8]
LDR r0, [r1, -r2, LSL#2]

- ◆ Dla transferów 16-bitowych i oraz 8-bitowych ze znakiem przesunięcie jest ograniczone do 0-255 (8 bitów), brak możliwości użycia skalowania:

LDRH r0, [r1, #-255] //
LDRH r0, [r1, -r2] //

- ◆ Użycie bardziej złożonych trybów adresowania:

LDR r0, [r1, #-8]! // adresowanie pośrednie rejestrowe z przesunięciem i
// preindeksowaniem



Instrukcje transferu danych - zastosowanie

- Przesunięcia względem adresu bazowego (dodatnie lub ujemne).
Przesunięcie może zostać podane w postaci:
 - Wartości natychmiastowej:
 - ◆ Dostęp do struktur, rejestrów, urządzeń I/O, elementów na stosie (znajdujących się w odległości od niezmiennego rejestru odniesienia, np. frame pointer). W szczególnym przypadku przesunięcia może być równe 0,
 - Wartości rejestru:
 - ◆ Indeksowanie kolejnych wartości tablic, struktur lub bloków danych,
 - Skalowanej wartości rejestru:
 - ◆ Indeksowanie kolejnych elementów tablic lub struktur, których numery kolejnych elementów są skalowane rozmiarem tego elementów, np. przeglądanie tablicy słów 64-bitowych → skalowanie 8 bajtów (LDRH r0, [r1, -r2, LSL#3]).



Przykład użycia instrukcji Load/Store

Zadeklarowano tablicę array składającą się z 25 słów 32-bitowych. Kompilator języka C przypisał rejestr r1 do zmiennej y typu int oraz umieścił adres pierwszego elementu tablicy w rejestrze r2 . Proszę napisać program w assemblerze odpowiadający linii poniżej:

```
int array[25];  
  
array[10] = array[5] + y;
```

Ćwiczenie przy tablicy

```
LDR    r2, =array  
  
LDR    r3, [r2, #20]    // r3 = array[5], r2 = pointer to array  
ADD    r3, r3, r1       // r3 = array[5] + y  
STR    r3, [r2, #40]    // array[10] = array[5] + y
```



Tryby adresowania

- ◆ Architektura ARM udostępnia 11 podstawowych trybów adresowania:
 - ◆ Adresowanie natychmiastowe (ang. immediate) #<immediate>, np. #13,
 - ◆ Adresowanie rejestrowe bezpośrednie (ang. register) <Rm>, np. r7,
 - ◆ Adresowanie rejestrowe z przesunięciem (ang. register with offset) <Rm>, rot #<shift_imm>, np. r0, LSL #4,
 - ◆ Adresowanie rejestrowe pośrednie (ang. register indirect),
 - ◆ Adresowanie rejestrowe pośrednie z indeksowaniem (ang. register indirect pre-indexed with no write-back),
 - ◆ Adresowanie rejestrowe pośrednie z preindeksowaniem (ang. register indirect pre-indexed with write-back),
 - ◆ Adresowanie rejestrowe pośrednie z postindeksowaniem (ang. register indirect post-indexed with write-back),
 - ◆ Adresowanie względem licznika programu (ang. Program Counter register indirect).
 - ◆ Pośrednie względem PC,
 - ◆ Pośrednie względem PC z indeksowaniem,
 - ◆ Pośrednie względem PC z preindeksowaniem,
 - ◆ Pośrednie względem PC z postindeksowaniem.



Instrukcja NOP

- ▶ Na liście instrukcji procesora ARM nie ma instrukcji NOP,
- ▶ Instrukcja NOP nie wykonuje żadnej czynności, powoduje wygenerowanie opóźnienia równego pojedynczemu cyklowi zegara,
- ▶ W przypadku procesorów ARM instrukcja NOP realizowana jest jako tzw. pseudoinstrukcja,
- ▶ Na etapie kompilacji kompilator wstawia instrukcję, która w danym momencie nie powoduje modyfikacji ALU ani nie odwołuje się do pamięci, np. wykonanie instrukcji warunkowej, dla której warunek jest zawsze nieprawdziwy.



Operacje na kilku rejestrach danych

Składnia instrukcji:

- **<LDM|STM>****<addressing_mode>**{**<cond>**} Rb{!}, <register list>
- Dostępne są 4 tryby adresowania pośredniego rejestrowego:
 - **LDMIA / STMIA** Postinkrementacja (ang. increment after)
 - **LDMIB / STMIB** Preinkrementacja (ang. increment before)
 - **LMDA / STMDA** Postdekrementacja (ang. decrement after)
 - **LDMDB / STMDB** Predekrementacja (ang. decrement before)

LDMIA r10!, {r0,r1,r4} ; Adresowanie pośrednie rejestrowe z postinkrementacją

STMDB r10!, {r0,r1,r4} ; Adresowanie pośrednie rejestrowe z predekrementacją

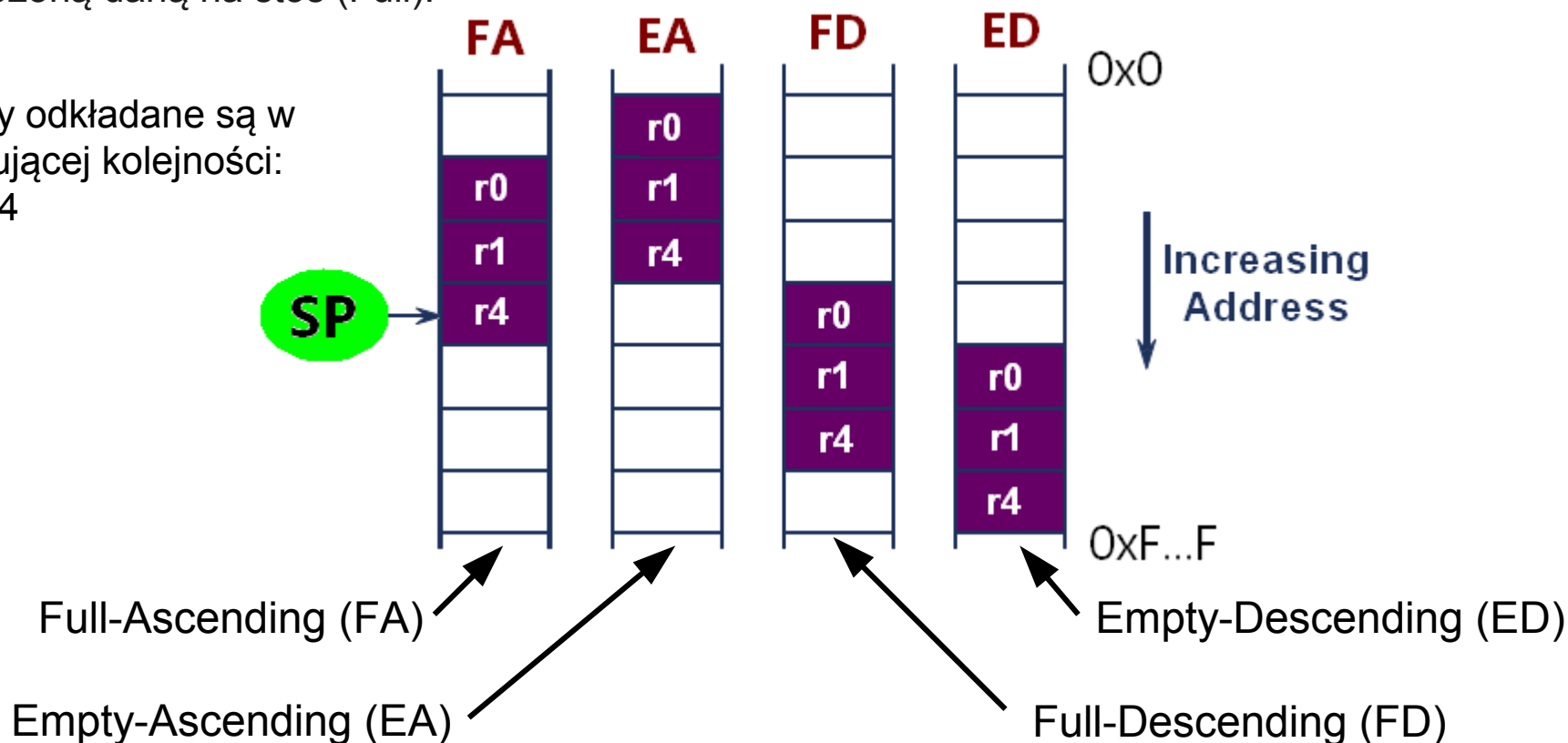
Other memory operations		§	Assembler
Load multiple	Block data load		LDM{ IA IB DA DB} Rn{!}, <reglist-PC>
	return (and exchange)		LDM{ IA IB DA DB} Rn{!}, <reglist+PC>
	and restore CPSR		LDM{ IA IB DA DB} Rn{!}, <reglist+PC>^
	User mode registers		LDM{ IA IB DA DB} Rn, <reglist-PC>^



Operacje na stosie

- Wyróżniamy 4 rodzaje stosu:
- Rośnie w górę (Ascending) lub w dół (Descending),
- Wskaźnik stosu pokazuje na pierwszą wolną komórkę (Empty) pamięci lub na ostatnio odłożoną daną na stos (Full).

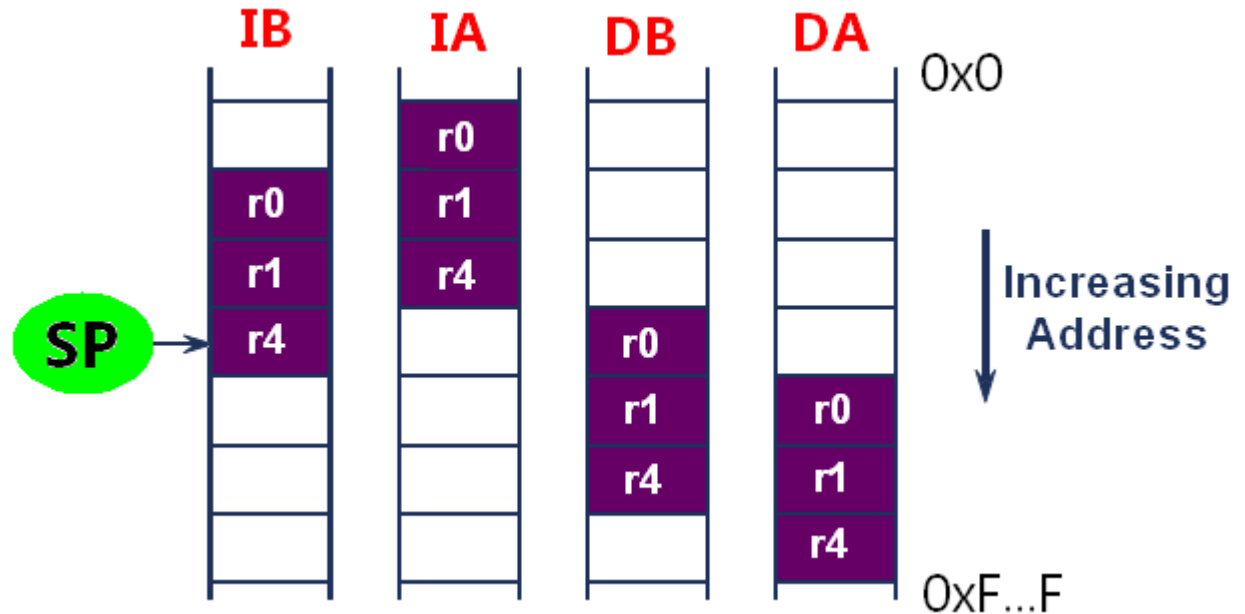
Rejestry odkładane są w następującej kolejności:
r0, r1, r4





Operacje na stosie

STMDB r10!, {r0,r1,r4}
Adresowanie pośrednie
rejestrów z
predekrementacją
Odłożenie danych na stos



Operacje na stosie (1)

W przypadku procesorów ARM zwykle stosuje się stos typu „pełny schodzący” (ang. Full-Descending, np. język C):

- Rosnący w kierunku malejących adresów
- Wskaźnik stosu (r13) pokazuje na ostatnio odłożony element

➤ Do obsługi stosu używa się instrukcji:

- **LDMIA** - zdjęcie danej lub danych ze stosu
- **STMDB** - odłożenie danej lub danych na stos

➤ W celu uniknięcia pomyłek podczas operacji na stosie wykorzystuje się następujące aliasy:

- **LDMFD** → **LDMIA**
- **STMFD** → **STMDB**, np.

Odłożenie rejestrów na stos: **STMFD** sp!, {r4,r6-r11,pc}

Zdjęcie rejestrów ze stosu: **LDMFD** sp!, {r4,r6-r11,pc}



Operacje na stosie (2)

- Do obsługi stosu używa się następujących instrukcji:
 - LDMXX Rx!, {Reg list}** - odczytanie wielokrotnych danych z pamięci (LoaD multiple Data)
 - STMXX Rx!, {Reg list}** - zapisanie wielokrotnych danych w pamięci (STore multiple Data)

STMEA	r13!, {r0-r2}	; Push data onto an Empty Ascending Stack
LDMEA	r13!, {r0-r2}	; Pop data off an Empty Ascending Stack
STMED	r13!, {r0-r2}	; Push data onto an Empty Descending Stack
LDMED	r13!, {r0-r2}	; Pop data off an Empty Descending Stack
STMFA	r13!, {r0-r2}	; Push data onto a Full Ascending Stack
LDMFA	r13!, {r0-r2}	; Pop data off a Full Ascending Stack
STMFD	r13!, {r0-r2}	; Push data onto a Full Descending Stack
LDMFD	r13!, {r0-r2}	; Pop data off a Full Descending Stack





Przykłady (1)

LDMIA	r8, {r0,r2,r9}	
STMDB	r1!, {r3-r6,r11,r12}	
STMFD	r13!, {r0,r4-r7,LR}	// Push registers including the stack pointer
LDMFD	r13!, {r0,r4-r7,PC}	// Pop the same registers and return from subroutine ;

Rozkazy niepoprawne:

STMIA	r5!, {r5,r4,r9}	// value stored for r5 unpredictable
LDMDA	r2, {}	// must be at least one register in list



Przykłady (2)

irq_handler:

```
/*- Manage Exception Entry */
sub    lr, lr, #4                /*- Adjust and save LR_irq in IRQ stack */
stmfd  sp!, {lr}
/*- Save r0 and SPSR in IRQ stack */
mrs    r14, SPSR
stmfd  sp!, {r0,r14}
/*- Write in the IVR to support Protect Mode */
/*- No effect in Normal Mode */
/*- De-assert the NIRQ and clear the source in Protect Mode */
ldr    r14, =AT91C_BASE_AIC
ldr    r0, [r14, #AIC_IVR]
str    r14, [r14, #AIC_IVR]
/*- Enable Interrupt and Switch in User Mode */
msr    CPSR_c, #ARM_MODE_SVC
stmfd  sp!, {r1-r3, r12, r14}    /*- Save scratch/used registers and LR in User Stack */
mov    r14, pc                  /*- Branch to the routine pointed by the AIC_IVR */
bx     r0
/*- Restore scratch/used registers and LR from User Stack */
ldmia  sp!, {r1-r3, r12, r14}
/*- Disable Interrupt and switch back in IRQ mode */
msr    CPSR_c, #ARM_MODE_IRQ | I_BIT
ldr    r14, =AT91C_BASE_AIC      /*- Mark the End of Interrupt on the AIC */
str    r14, [r14, #AIC_EOICR]
ldmia  sp!, {r0,r14}             /*- Restore SPSR_irq and r0 from IRQ stack */
msr    SPSR_cxsf, r14

/*- Restore adjusted LR_irq from IRQ stack directly in the PC */
ldmia  sp!, {pc}^
```



Podprogramy



Wywołanie funkcji

- Zapisywanie rejestrów na stosie w funkcjach
- Powrót z funkcji

BL Function1

Functions1:

```
STMFD sp!, {r0-r12, lr}      ; stack all registers
.....                      ; and the return address
.....
LDMFD sp!, {r0-r12, pc}      ; load all the registers
                              ; and return automatically
```

Instrukcje rozgałęzienia programu (1)

- Instrukcja skoku (rozgałęzienie programu)

Branch: **B{<cond>}** **label**

- Instrukcja skoku z zachowaniem adresu powrotu (wywołania funkcji)

Branch with Link: **BL{<cond>}** **subroutine_label**



- Podczas obliczania adresu skoku rdzeń procesora przesuw offset o 2 bity w lewo, rozwija przesunięcie z zachowaniem znaku do 32 bitów i dodaje do licznika programu PC (r15)
- Umożliwia to wykonanie skoków względnych w zakresie ± 32 MB pamięci,
- Instrukcje skoków umożliwiają pisanie programów relokowalnych



Programy relokowalne

```
.org 0x1000
```

```
MOV    r0, #0x1400
```

```
MOV    r1, #1200
```

```
B      0x1200
```

```
...
```

InputData:

DC.W 0xABCD, 0x1234,...

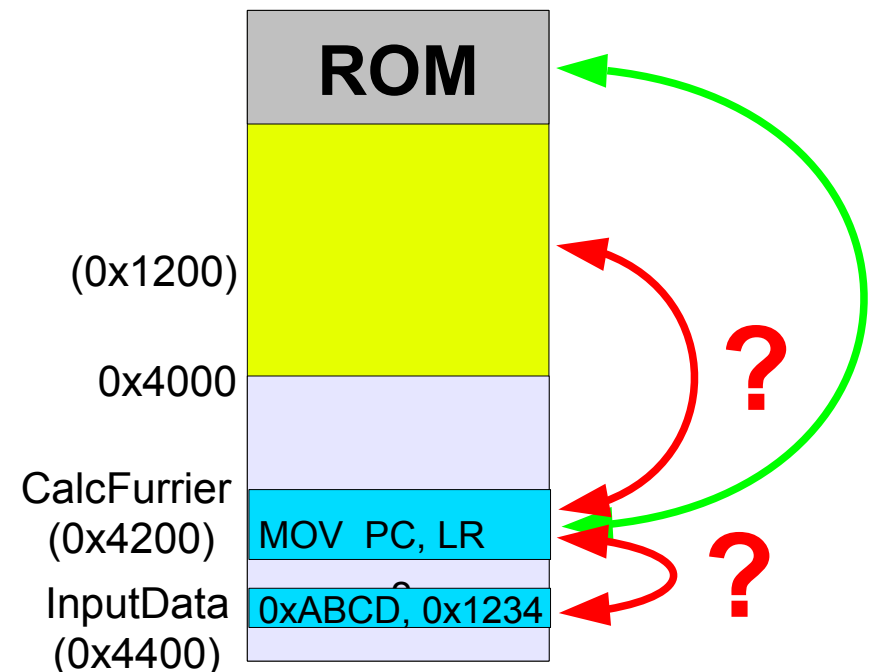
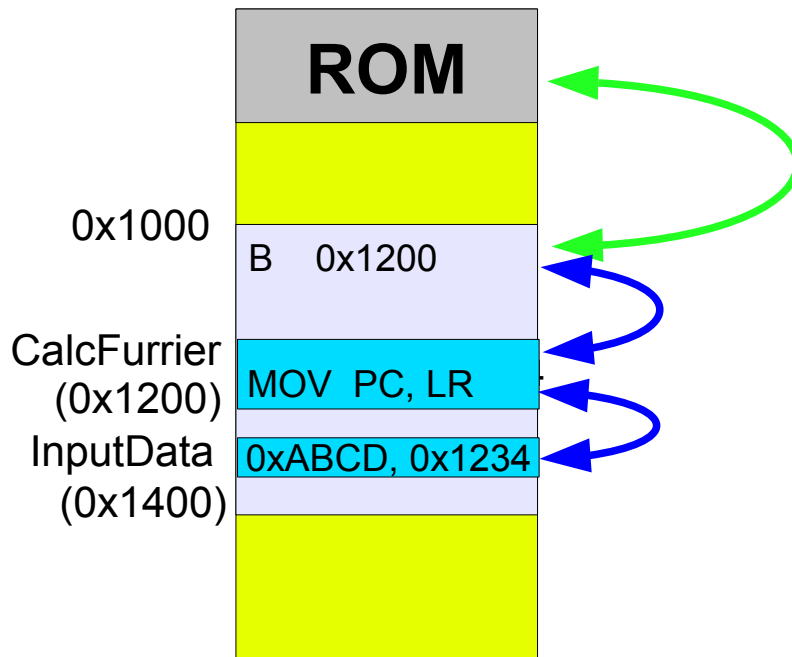
Asembler

Program relokowalny – adresy
liczone względem PC

```
MOV    r0, =InputData
```

```
MOV    r1, #parameter
```

```
B      CalcFourrier
```





Instrukcje skoków w zakresie pełnej przestrzeni adresowej

- Instrukcje asemblera procesora ARM umożliwiają wykonanie skoku względnego w ograniczonym zakresie

Jak wykonać skok w pełnym zakresie przestrzeni adresowej (4 GB) ?

- Posługując się trybem adresowania natychmiastowego ustawiamy licznik programu na adres naszej funkcji (jeżeli jest to skok do funkcji należy wykonać kopię licznika bieżącej wartości licznika programu)

MOV lr, pc ; kopia PC

LDR pc, =dest ; skok do podprogramu, adres pamiętany w zmiennej dest

Linker automatycznie rozpoznaje takie sytuacje, oblicza adres funkcji oraz generuje odpowiedni kod maszynowy w zależności od zakresu w jakim wykonywany jest skok.



Branch and change mode

- ◆ Instrukcja pozwala na wykonanie skoku ze zmianą trybu pracy:

- ◆ Przejście z trybu ARM do Thumb lub Jazzele,
- ◆ Przejście z trybu Thumb lub Jazzele do trybu ARM.

- ◆ Składnia:

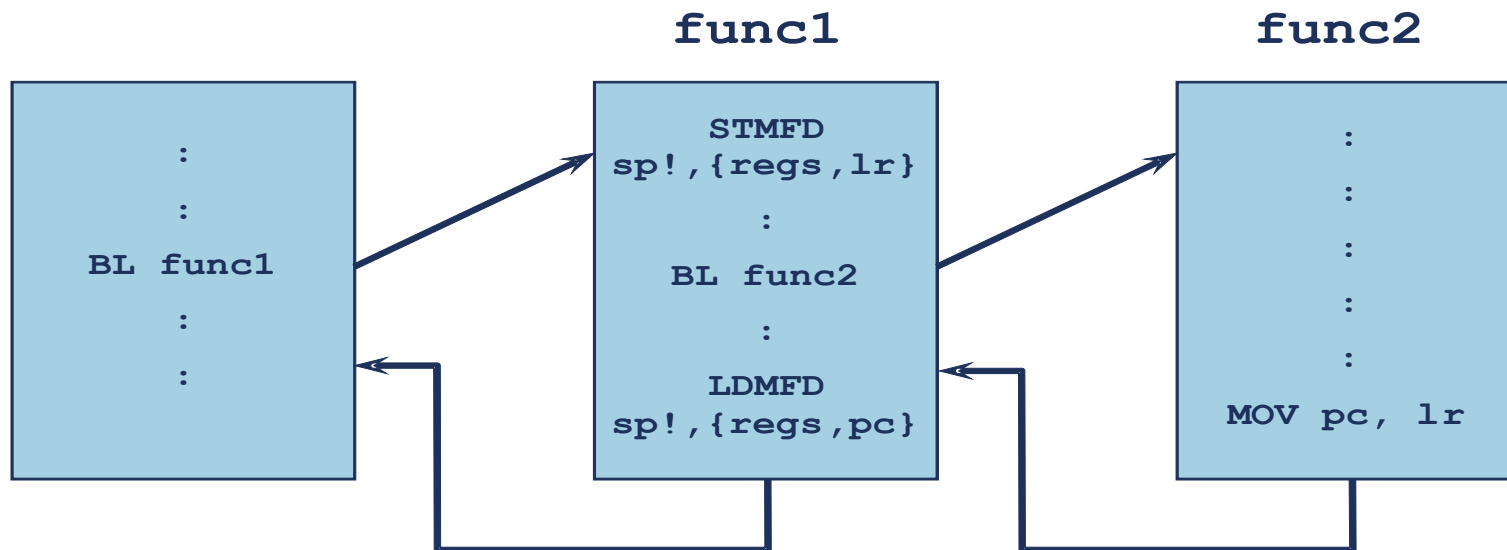
`BX{cond} Rm // Rm adres pod, który należy wykonać skok (bit zero
// odwołuje się do flagi T w rejestrze CPSR)`

- ◆ Przykład:

- ◆ `BX r7 // jump and change mode`
- ◆ `BXVS r0 // jump if overflow`

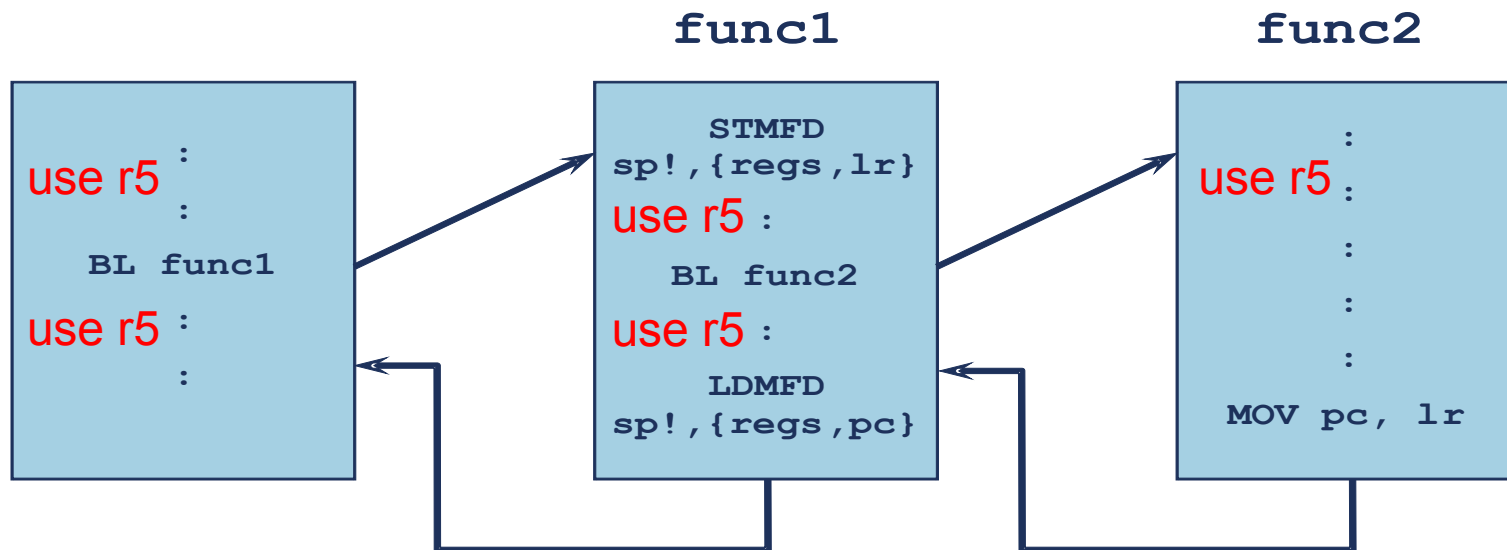
Skok do podprogramu (2)

- Skok do podprogramu możliwy jest przy wykorzystaniu instrukcji BL
- Instrukcja zachowuje adres powrotu w rejestrze r14 (Link Register)
- Jeżeli podprogram wykorzystuje pewne rejestry (np. r0-r7, r9, r13, r14) przed rozpoczęciem obliczeń należy wykonać ich kopię na stosie, Rejestry odtwarzane są podczas powrotu z funkcji.
- Programy zagnieżdżone wymagają również archiwizacji rejestru r14.
- Powrót z podprogramu: **MOV pc, lr** lub **LDMFD sp!, {r0-r7, r9, r13, pc}**



Skok do podprogramu (3)

- ♦ Jak przekazać argumenty do funkcji?
- ♦ Rejestry → jakie?
- ♦ Stos → kiedy odłożyć, w jakiej kolejności? Kiedy zdjąć ze stosu?





ARM Procedure Call Standard (ARM APCS)

ARM Ltd. dostarcza zestaw reguł określających parametry wejściowe i zwracane przez procedury

- Kod generowany przez różne kompilatory może być linkowany
- Procedury mogą być wywoływane pomiędzy językami niskiego i wysokiego poziomu

Specyfikacja APCS określa:

- Przeznaczenie rejestrów
- Użycie stosu
- Format struktur danych przechowywanych na stosie
- Metody przekazywania argumentów do funkcji



Przeznaczenie rejestrów (1)

Zgodnie za standardem APCS:

- Rejestry procesora mają przypisane konkretne znaczenie
- Rejestry r0 – r3 (a1-a4) przeznaczone są do przekazywania 4 pierwszych parametrów do procedur lub zwracania rezultatów funkcji (pozostałe wartości mogą być przekazywane przez stos), rejestry zapisywane przez program wywołujący
- Rejestry r4-r10 przeznaczone są do przechowywania zmiennych lokalnych procedur, rejestry zapisywane przez program wywołany
- Rejestry r12-r15 mają zawsze przypisane specjalne przeznaczenie: IP, SP, LR, PC
- Stos powinien być wyrównany do granicy 8 bajtów



Przeznaczenie rejestrów (2)

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter



APCS – Przekazanie parametrów

Parametry przekazywane do podprogramu

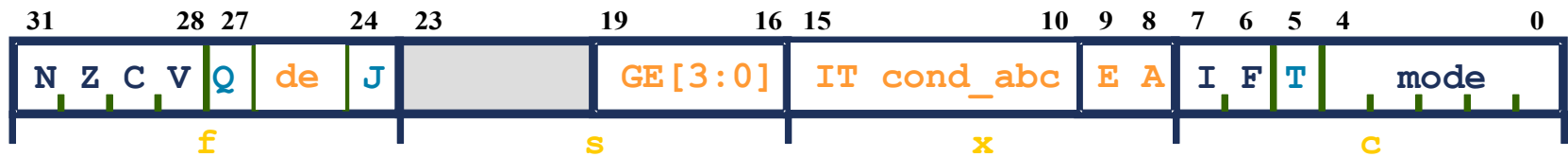
- Pierwsze 4 parametry przekazywane są przez rejestry R0-R3
- Pozostałe parametry przekazywane są przez stos (odkładane w odwrotnej kolejności)
- Procedury z liczbą parametrów ≤ 4 wykonują się szybciej ze względu na brak potrzeby odwołania do stosu przy przekazywaniu parametrów

Parametry zwracane z podprogramu

- Pojedyncza zmienna (typu word) – rejestr R0
- Złożone z kilku słów (2-4 word) odpowiednio R0/R1, R0/R1/R2, R0/R1/R2/R3



Operacje na rejestrach stanu



- Instrukcje MRS and MSR umożliwiają wykonanie operacji na rejestrach stanu CPSR / SPSR. Zawartość rejestrów stanu może zostać zapisana lub odtworzona z rejestru ogólnego przeznaczenia.
- Instrukcja MSR pozwala na aktualizację całego lub części rejestru stanu, n.p. włączenie lub wyłączenie przerw, zmiana trybu pracy. Modyfikacja flag rejestru stanu wymaga użycia mechanizmu read-modify-write:

MRS **r0, CPSR** // read CPSR into r0

BIC **r0, r0, #0x80** // clear bit 7 to enable IRQ

MSR **CPSR_c, r0** // write modified value to 'c' byte only

- W trybie użytkownika rejestr stanu może zostać odczytany, natomiast tylko bajt f (CPSR_f) może zostać zmieniony.



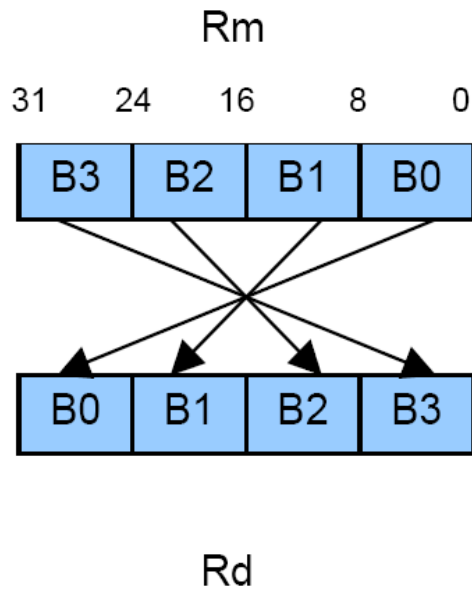
Operacje na rejestrach stanu

Operation		§	Assembler
Move to or from PSR	PSR to register		MRS Rd, <PSR>
	register to PSR		MSR <PSR>_<fields>, Rm
	immediate to PSR		MSR <PSR>_<fields>, #<imm8m>
Processor state change	Change processor state	6	CPSID <iflags> {, #<p_mode>}
		6	CPSIE <iflags> {, #<p_mode>}
	Change processor mode	6	CPS #<p_mode>
	Set endianness	6	SETEND <endianness>

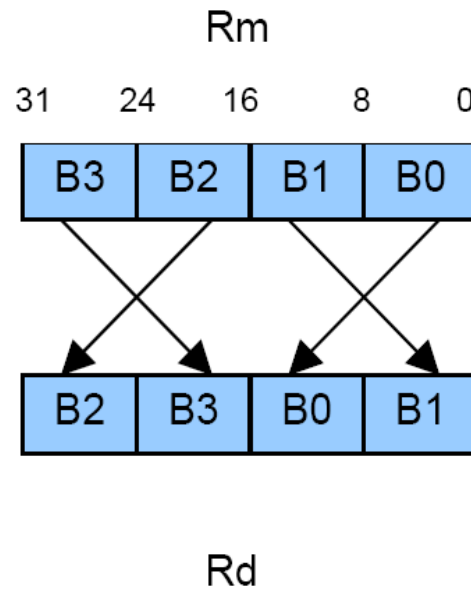


Instrukcje odwracające bajty

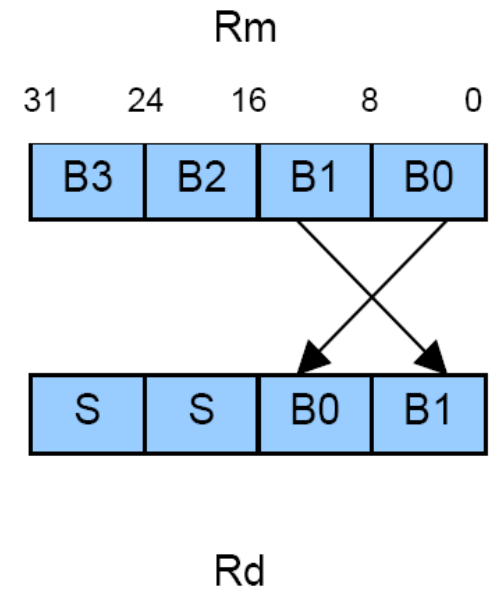
REV{<cond>} Rd, Rm



REV16{<cond>} Rd, Rm



REVSH{<cond>} Rd, Rm





Tryby adresowania

Egzamin

LETNIA SESJA EGZAMINACYJNA 24.06.2011 r. – 07.07.2011r.

Termin 1:	27.06.2013 (godz. 8.15 – 9.15)
Termin 2:	01.07.2013 (godz. 8.15 – 9.15)
Termin 3:	02.09.2013 (godz. 8.15 – 9.15)



Tryby adresowania

- Tryb adresowania - metoda wykorzystywana do określenia operandów źródłowych i docelowych rozkazów asemblera. Procesory CISC (Complex Instruction Set Computer) umożliwiają użycie dużej liczby trybów adresowania, natomiast liczba trybów adresowania dla procesorów RISC jest zwykle znacznie mniejsza, ze względu na uproszczenie budowy CPU.
- Tryb adresowania określa dostęp do danych umieszczonych w rejestrach (adresowanie rejestrowe) lub w pamięci procesora (adresowanie pośrednie).

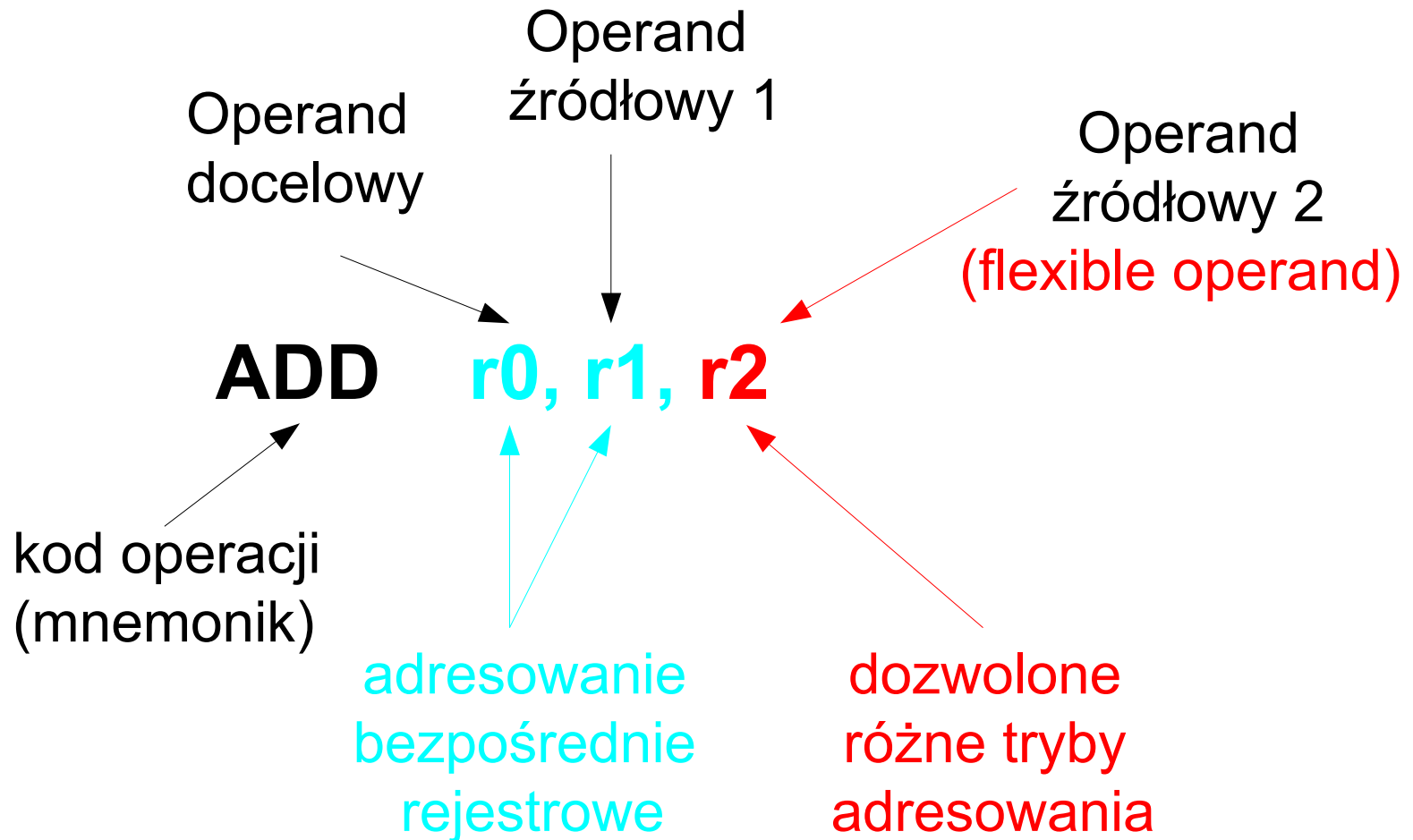
Przykład adresowania dla procesora CISC z rodziny M68000

- MOVE.L 8(PC, D2), -(A0)

Odwołanie do pamięci
Adresowanie pośrednie rejestrowe
z predekrementacją

Odwołanie do pamięci
Adresowanie pośrednie rejestrowe względem licznika programu
z przesunięciem i indeksem

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 adresowania 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 przesunięcia 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 będący 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 programu

- 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



Tryby adresowania

- ◆ Architektura ARM udostępnia 11 podstawowych trybów adresowania:
 - ◆ Adresowanie natychmiastowe (ang. immediate) #<immediate>, np. #13,
 - ◆ Adresowanie rejestrowe bezpośrednie (ang. direct register) <Rm>, np. r7,
 - ◆ Adresowanie rejestrowe z przesunięciem (ang. register with offset) <Rm>, rot #<shift_imm>, np. r0, LSL #4,
 - ◆ Adresowanie rejestrowe pośrednie (ang. register indirect),
 - ◆ Adresowanie rejestrowe pośrednie z indeksowaniem (ang. register indirect pre-indexed with no write-back),
 - ◆ Adresowanie rejestrowe pośrednie z preindeksowaniem (ang. register indirect pre-indexed with write-back),
 - ◆ Adresowanie rejestrowe pośrednie z postindeksowaniem (ang. register indirect post-indexed with write-back),
 - ◆ Adresowanie względem licznika programu (ang. Program Counter register indirect).
 - ◆ Pośrednie względem PC,
 - ◆ Pośrednie względem PC z indeksowaniem,
 - ◆ Pośrednie względem PC z preindeksowaniem,
 - ◆ Pośrednie względem PC z postindeksowaniem.