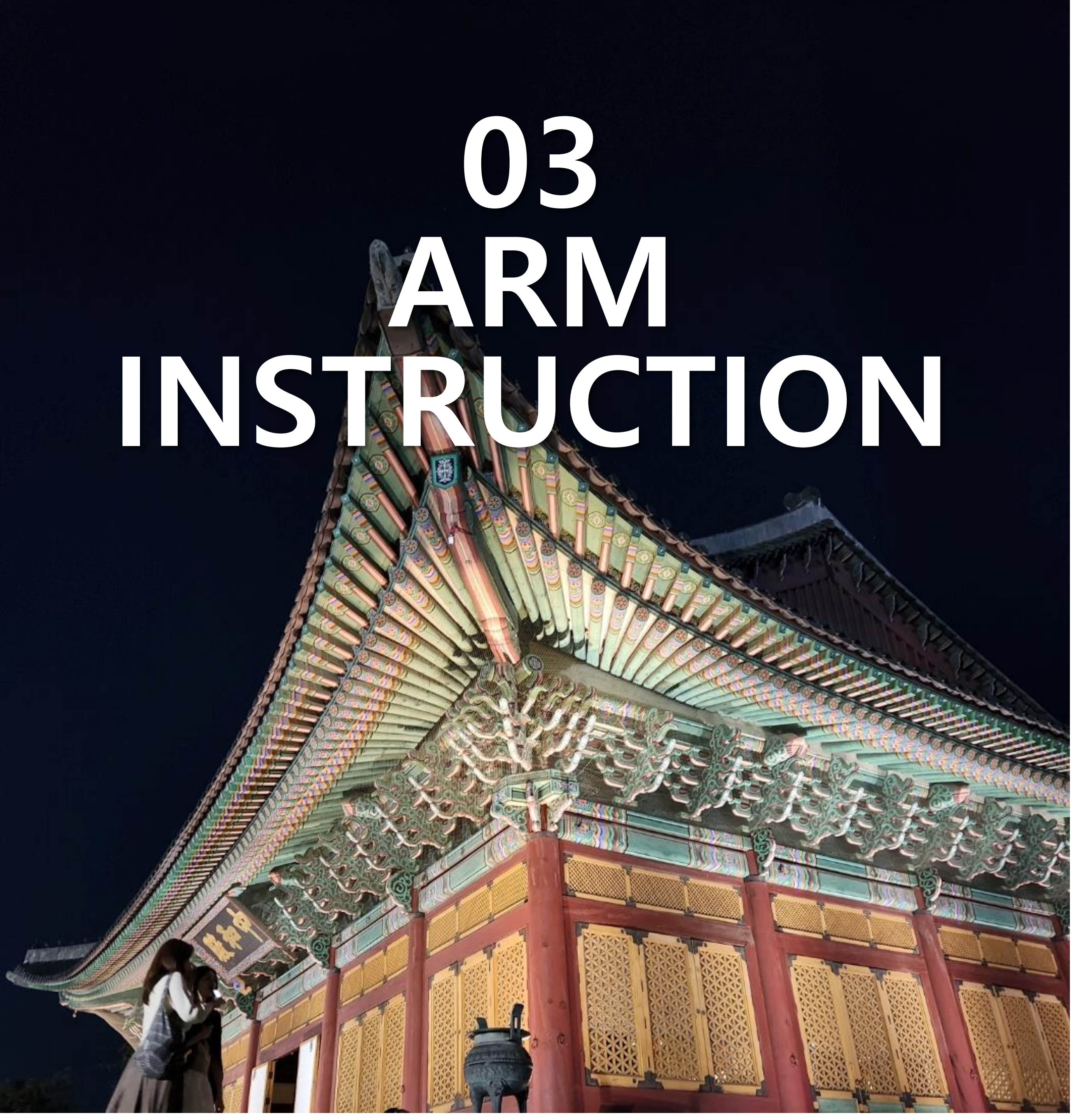


03
ARM INSTRUCTION

마이크로프로
세서응용

Handong
Global
University

Agenda

- <u>ARM Instruction Set Architecture Features</u>
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

ARM Instruction Set Feature

- 3-address data processing instructions
- Conditional Execution of each instruction
- Shift and ALU operation in single instruction
- Load-Store and Load-Store multiple instructions
- Single cycle execution of all instructions
- Instruction set extension through coprocessor instruction

Representation in this Slide

- Number representation
  - Hexadecimal numbers begin with 0x
  - binary numbers begin with 0b
- Literal
  - begins with #
- memory is denoted as M<data_size>[*address]*
  - <data_size> bits of memory starting at the byte *address*
  - <data_size> default is 32
- ARM instruction Notation
  - ADD r3, r2, r1
  - r3: destination register(Rd)
  - r2, r1: source resgisters (Rn, Rm)

ARM Addressing Mode (1)

- Immediate Addressing Mode
  - Data specified in Instruction
  - Example)
    - ✓MOV R2, #0x20 ; R2 ← 20H
    - ✓ADD R0, R1, #0x20 ; R0 ← R1 + 20H
- Register Addressing Mode
  - Data ← Register
  - All logical and Arithmetic instr. are based on this mode
  - Example)
    - ✓MOV R0, R1 ; R0 ← R1
    - ✓ADD R0, R1, R2 ; R0 ← R1 + R2

Addressing Mode (2)

- Direct Addressing Mode
  - Data ← mem[PC+Offset]
  - No direct Addressing mode but Assembler turns PC-relative addressing mode
  - Offset = Address – PC - 8
  - Example)
    - ✓LDR R5, Variable ; R5 ← mem[Variable]
    - ✓STR R5, Variable ; R5 → mem[Variable]
- Indirect Addressing Mode
  - Data ← mem[Register]
  - Example)
    - ✓LDR R5, [R1] ; R5 ← mem[R1]
    - ✓STR R5, [R1] ; R5 → mem[R1]

Addressing Mode (3)

- Register Relative Indirect Addressing Mode
  - Memory Address = Register + #imm
  - Example)
    - ✓LDR R0, [R1, #4] ; R0 ← mem[R1+4]
    - ✓LDR R0, [R1, #4]! ; (Pre-index) First, R1 ← R1+4 Then, R0 ← mem[R1]
    - ✓LDR R0, [R1], #4 ; (Post-index) First, R0 ← mem[R1] Then, R1 ← R1 + 4
- Base Indexed Indirect Addressing Mode
  - Memory Address = register 1(base) + register 2 (index)
  - Example)
    - ✓LDR R0, [R1,R2] ; R0 ← mem[R1+R2]
    - ✓LDR R0, [R1,R2]! ; (Pre-Index) First, R1 ← R1+R2 Then, R0 ← mem[R1]
    - ✓LDR R0, [R1], R2 ; (Post-Index)First, R0 ← mem[R1] Then, R1 ← R1 + R2

Addressing Mode (4)

- Base with Scaled Index Addressing Mode
  - memory address = address (base)+ scaled register2 (index)
  - [Rn, Rm, shifter]
  - [Rn, Rm, shifter]!
  - [Rn], Rm, shifter
  - Shifter = LSL/LSR #5-bit_unsigned_number
  - Example) LDR R0,[R1,R2, LSL #2] ; R0 ← mem[R1+R2 * 2]

Addressing mode

```
MOV R0, R1 ; register
MOV R0, #3 ; immediate
MOV R1, [R0] ; register indirect
MOV R2, [R0,#4] ; register relative indirect
MOV R3, [R0,#4]! ; register relative indirect Pre-index
MOV R4, [R0], #4 ; register relative indirect Post –index
MOV R5, [R0,R1] ; base indexed indirect register relative
MOV R6, [R0,R1]! ; base indexed  indirect register Pre-index
MOV R7, [R0], R1  ; base indexed indirect register Post-index
```

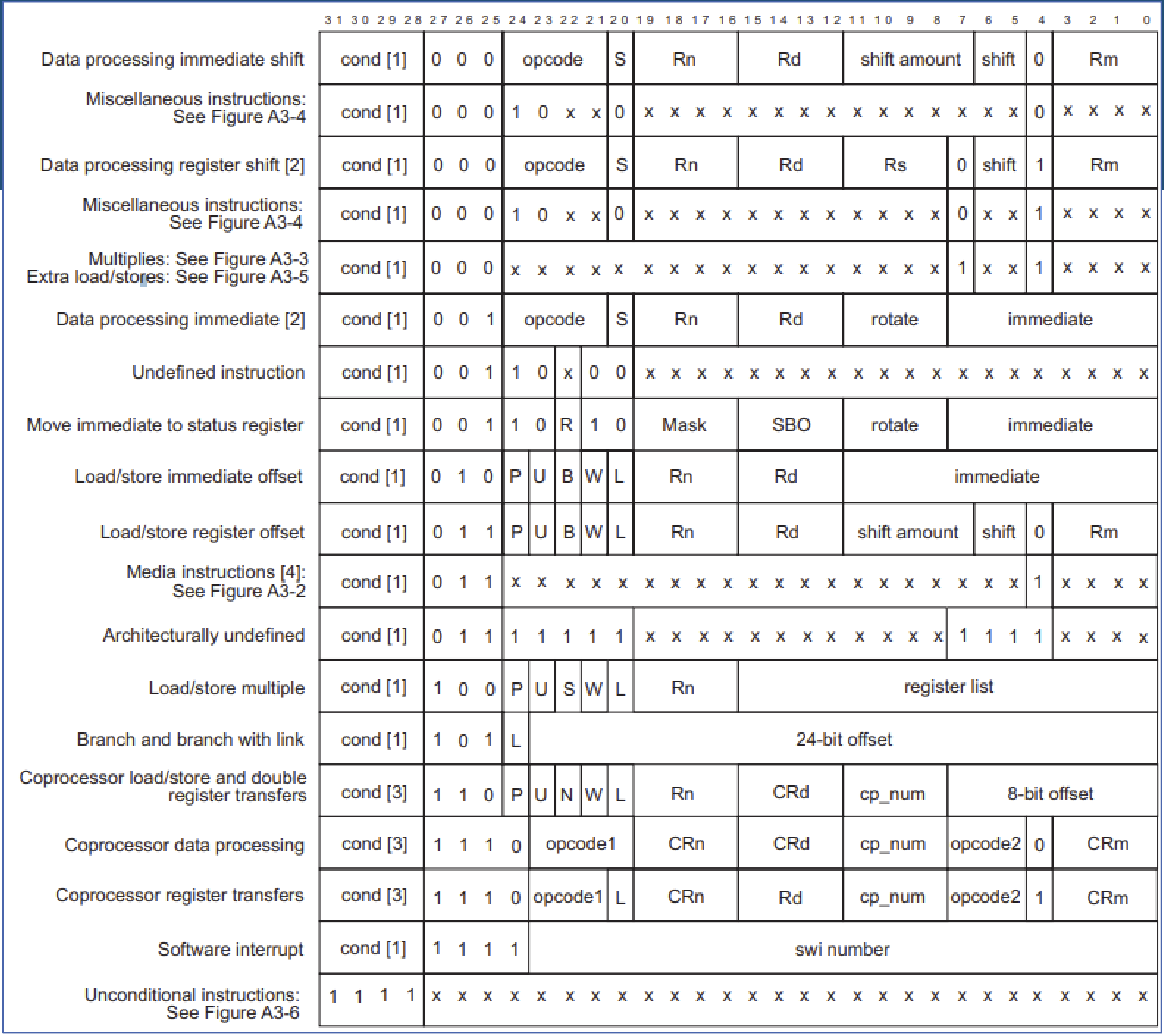
ARM Instruction Format

- For Data processing

S: Set condition(1)

- For load-store

P: pre-index (1)
post-index (0)
U: Up : add offset(1)
Down: subtract (0)
B: byte transfer (1)
word transfer (0)
W: write-back (1)
L: load from mem (1)
store to mem (0)



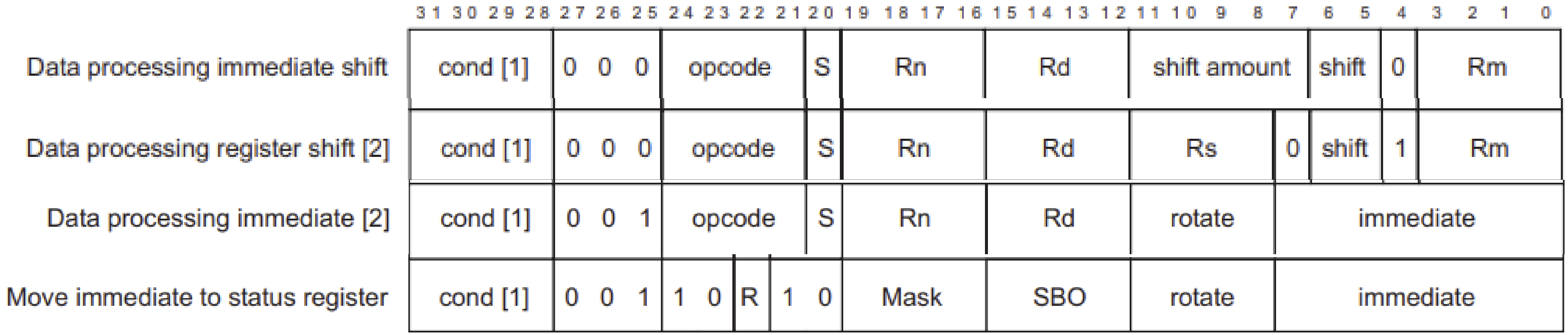
<table>
<tr><td></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td>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

Agenda

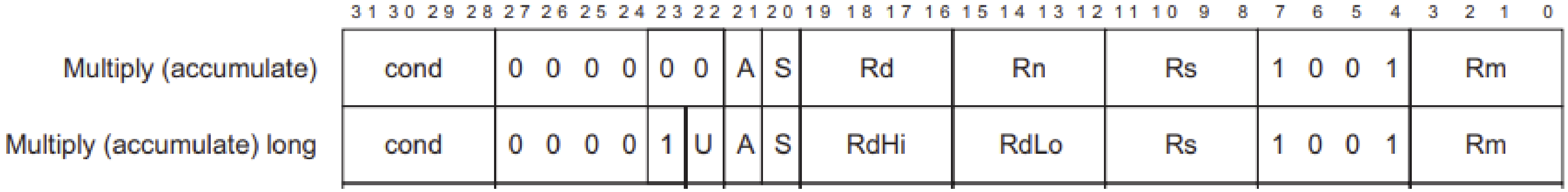
- ARM Instruction Set Architecture Features
- <u>Data Processing Instructions</u>
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

Data Processing Instruction Format

- Data processing instruction

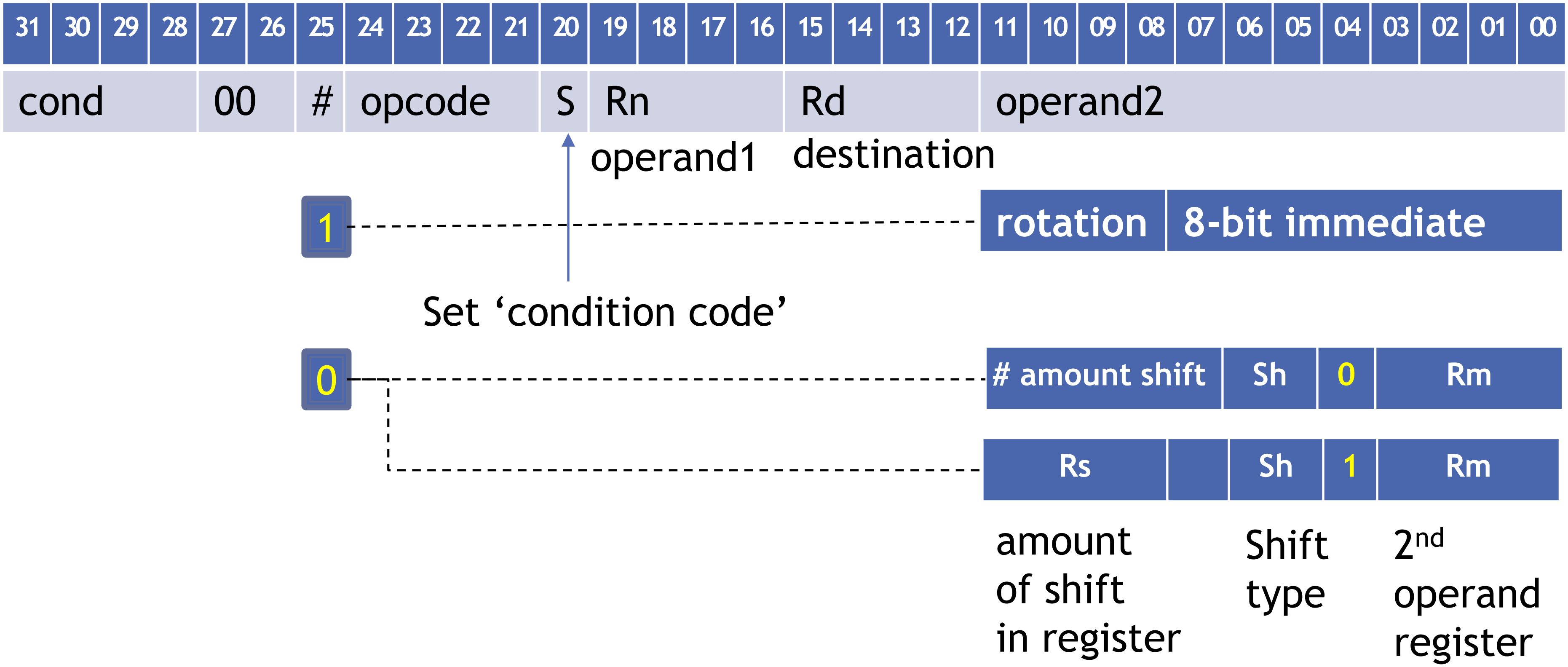


	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing immediate shift	cond [1]				0	0	0	opcode				S	Rn				Rd				shift amount					shift		0	Rm			
Data processing register shift [2]	cond [1]				0	0	0	opcode				S	Rn				Rd				Rs				0	shift		1	Rm			
Data processing immediate [2]	cond [1]				0	0	1	opcode				S	Rn				Rd				rotate				immediate							
Move immediate to status register	cond [1]				0	0	1	1	0	R	1	0	Mask				SBO				rotate				immediate							



	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
Multiply (accumulate)	cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
Multiply (accumulate) long	cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rs				1	0	0	1	Rm			

Binary encoding of data processing instructions



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

cond | 00 | # | opcode | S | Rn | Rd | operand2

operand1

destination

Set ‘condition code’

1

rotation | 8-bit immediate

0

# amount shift | Sh | 0 | Rm

Rs | | Sh | 1 | Rm

amount
of shift
in register

Shift
type

2nd
operand
register

Assembly Language Syntax

- Data Processing Assembly Instruction
  - instruction{<cond>}{S} Rd, Rn, N
  - ‘cond’ – indicates flag to test
    - ✓EQ, NE, GT, LT, CS
  - ‘S’ – set condition flag in cpsr
  - Rd – destination (register)
  - Rn – first operand (register)
  - ‘N’ – second operand register, immediate, shifter
    - ✓Rm,
    - ✓#const
    - ✓Rs, shifter

Basic Data Processing Instructions

MOV	**Move a 32-bit value**	**MOV Rd, n**	**Rd = n**
MVN	Move negated (logical Not) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rd + n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rd + n + C
SUB	Substract two 32-bit values	SUB Rd,Rn,n	Rd = Rn – n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn – n +C-1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n – Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n –Rn + C-1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear, every 1 in n clears corresponding bit of Rn	BIC Rd,Rn,n	Rd = Rn AND (~n)
CMP	Compare	CMP Rd, n	Rd-n & change flag
CMN	Compare Negative	CMP Rd, n	Rd+n & change flag
TST	Test for a bit in a 32-bit value	TST Rd, n	Rd AND n, change flag
TEQ	Test for equality	TEQ Rd, n	Rd XOR n, change flag
MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm * Rs
MLA	Multiply and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

Data Processing Instructions

- The data processing instructions manipulate data within registers.
  - move instructions
  - arithmetic instructions
  - logical instructions
  - comparison instructions
  - multiply instructions

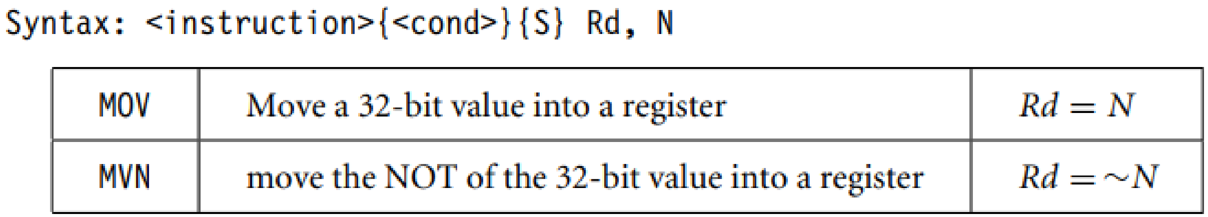
- Status Flag Update
  - If you use the S suffix on a data processing instruction, it updates the flags in the $cpsr$.
  - Move and logical operations update the carry flag C, negative flag N, and zero flag Z.
    - ✓ flag notation
      - lower case : 0
      - flag upper case : 1
    - ✓ nZcv : N=0, Z=1, C=0, V=0
    - ✓ nzCV : N=0, Z=0, C=1, V=1

Condition codes & flag states Tested

Mnemonic extension	Condition Tested	Cond. Code	Flag Tested	Mnemonic extension	Condition Tested	Cond. Code	Flag Tested
EQ	Equal	0000	Z=1	HI	Unsigned higher	1000	C=1, Z=0
NE	Not Equal	0001	Z=0	LS	Unsigned Lower or same	1001	C=0, Z=1
CS/HS	Carry Set/ Unsigned higher or same	0010	C=1	GE	Signed Greater than Equal	1010	N = V
CC/LO	Carry Clear/ Unsigned lower	0011	C=0	LT	Signed Less Than	1011	N ≠ V
MI	Minus/ Negative	0100	N=1	GT	Signed Greater Than	1100	Z=0 & N=V
PL	Plus/ Positive or Zero	0101	N=0	LE	Signed Less Than or Equal	1101	Z=1 or N ≠ V
VS	Overflow	0110	V=1	AL	Always	1110	---
VC	No overflow	0111	V=0	NV	Never (Don’t use)	1111	---

Data Processing Instructions

- Move Instructions
  - It copies N into a destination register Rd.

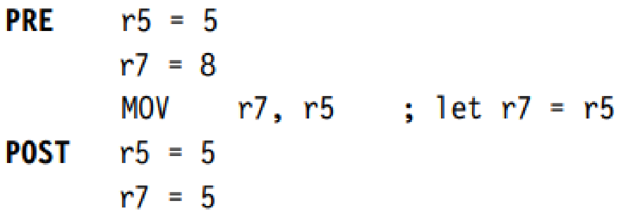


Syntax: `<instruction>{<cond>}{S} Rd, N`

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

✓Table 3.3 in the textbook gives a full description of the values allowed for the second operand N for all data processing instructions.

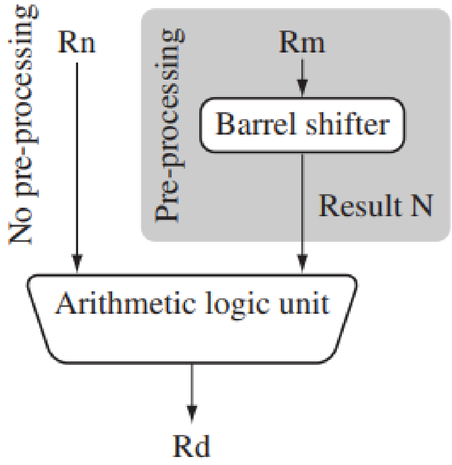
  - Ex)

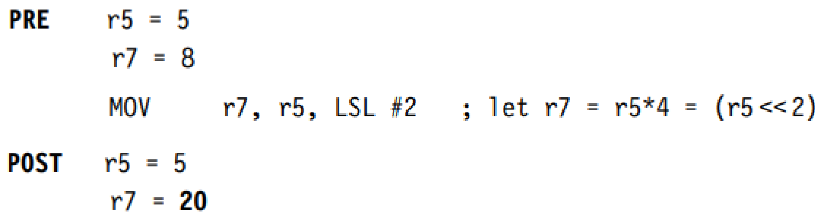


```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5     ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Data Processing Instructions

- Barrel Shifter
  - Instructions can also take a register Rm has been preprocessed by the barrel shifter prior to being used by a data processing instruction.
  - Pre-processing or shift occurs within the cycle time of the instruction.



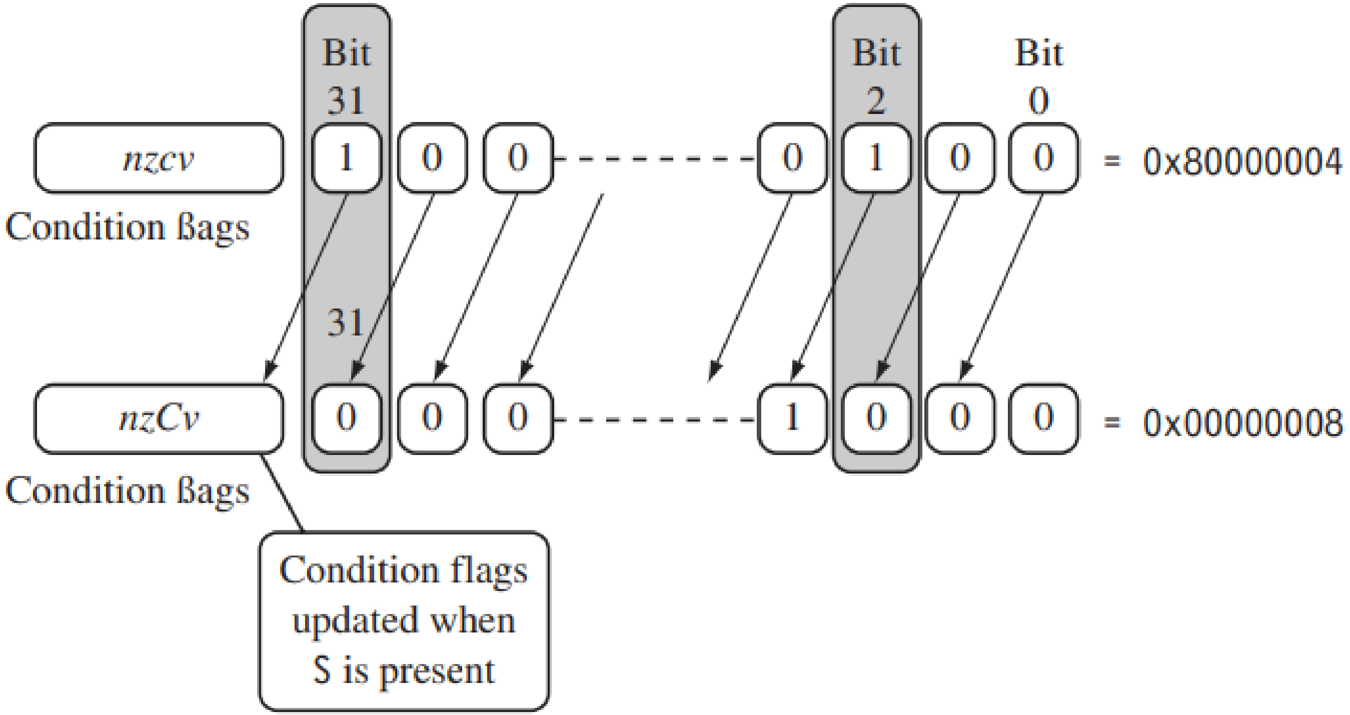


```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5, LSL #2    ; let r7 = r5*4 = (r5<<2)
POST    r5 = 5
        r7 = 20
```

  - Question: What does the shift processing mean for a binary number?

Data Processing Instructions

- Barrel Shifter
  - Logical shift left by one



Data Processing Instructions

- Barrel Shifter

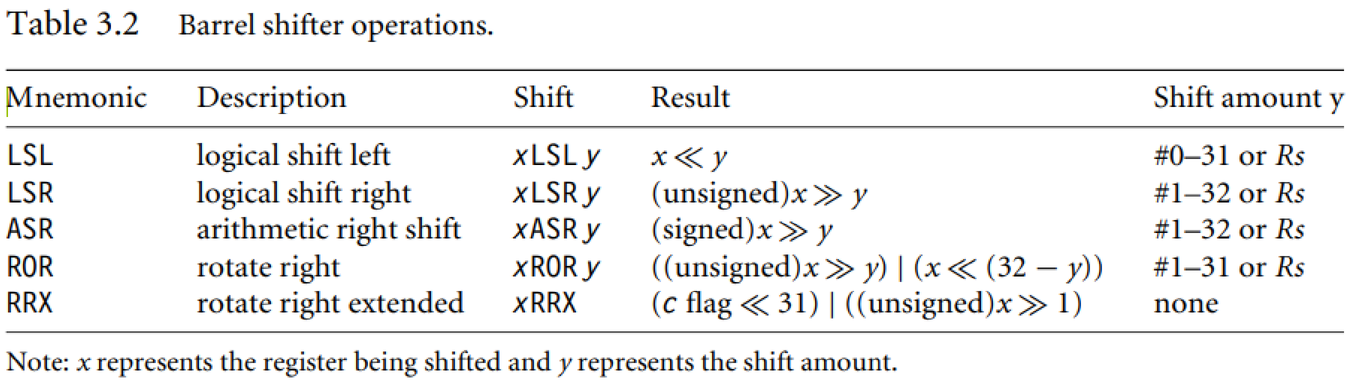


Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	x LSL y	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	x LSR y	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	x ASR y	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	x ROR y	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	x RRX	(c flag $\ll$ 31) | ((unsigned)$x \gg 1$)	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

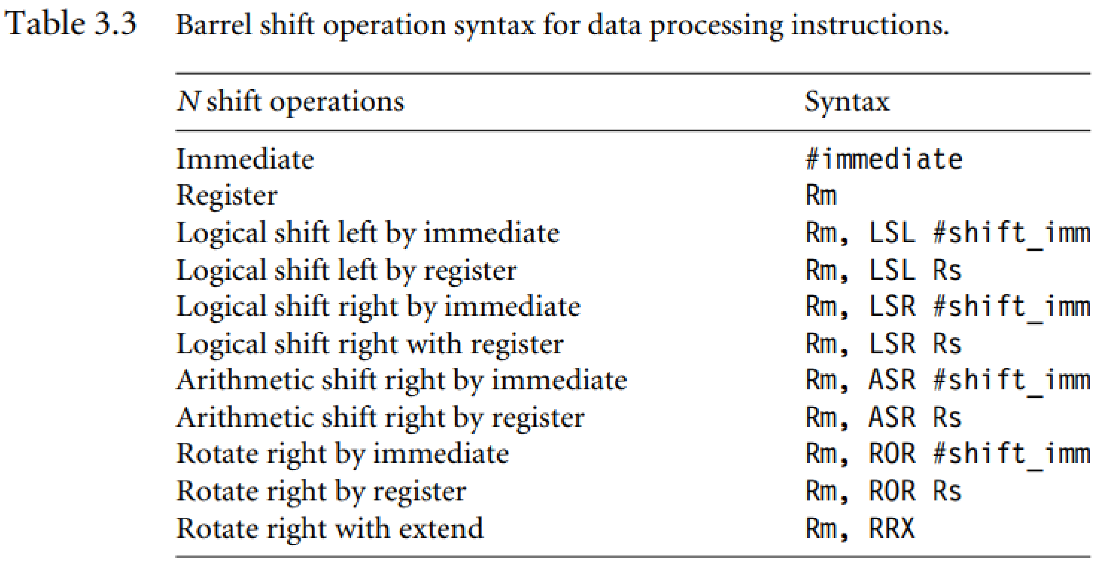
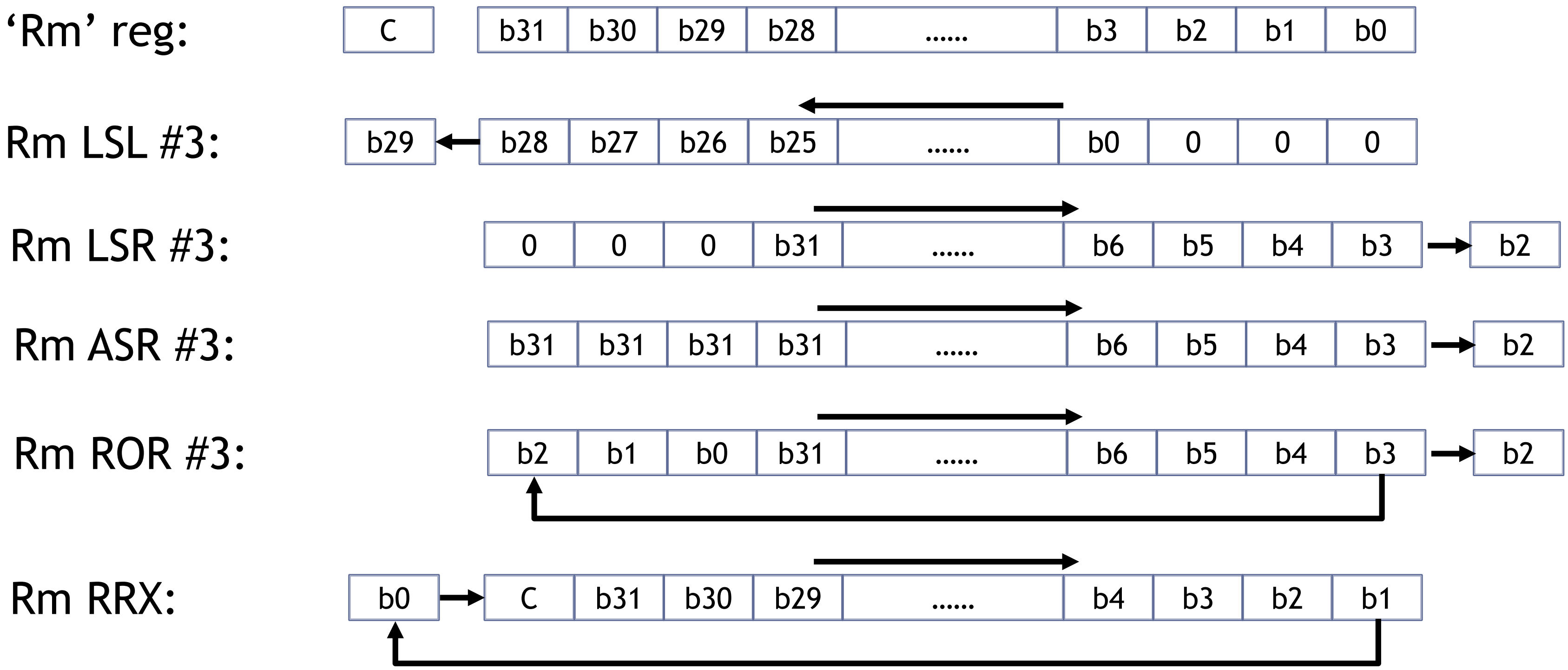


Table 3.3 Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Rs: register for shift

Examples of shift, rotate operations



'Rm' reg: C | b31 | b30 | b29 | b28 | …… | b3 | b2 | b1 | b0

Rm LSL #3: b29 ← b28 | b27 | b26 | b25 | …… | b0 | 0 | 0 | 0

Rm LSR #3: 0 | 0 | 0 | b31 | …… | b6 | b5 | b4 | b3 → b2

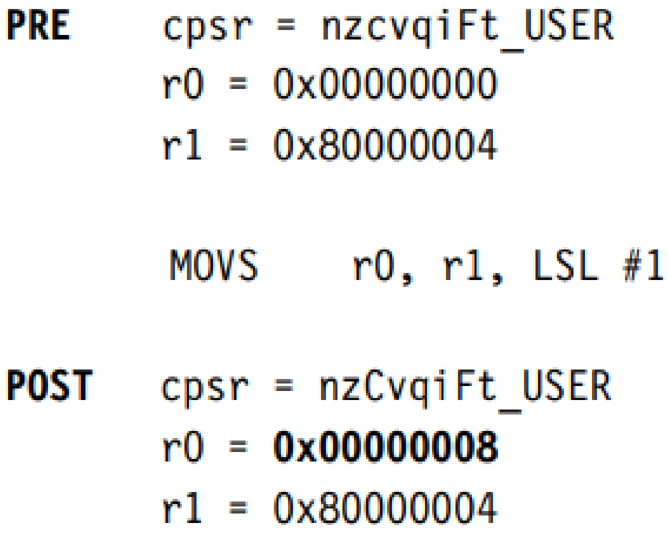
Rm ASR #3: b31 | b31 | b31 | b31 | …… | b6 | b5 | b4 | b3 → b2

Rm ROR #3: b2 | b1 | b0 | b31 | …… | b6 | b5 | b4 | b3 → b2

Rm RRX: b0 → C | b31 | b30 | b29 | …… | b4 | b3 | b2 | b1

Data Processing Instructions

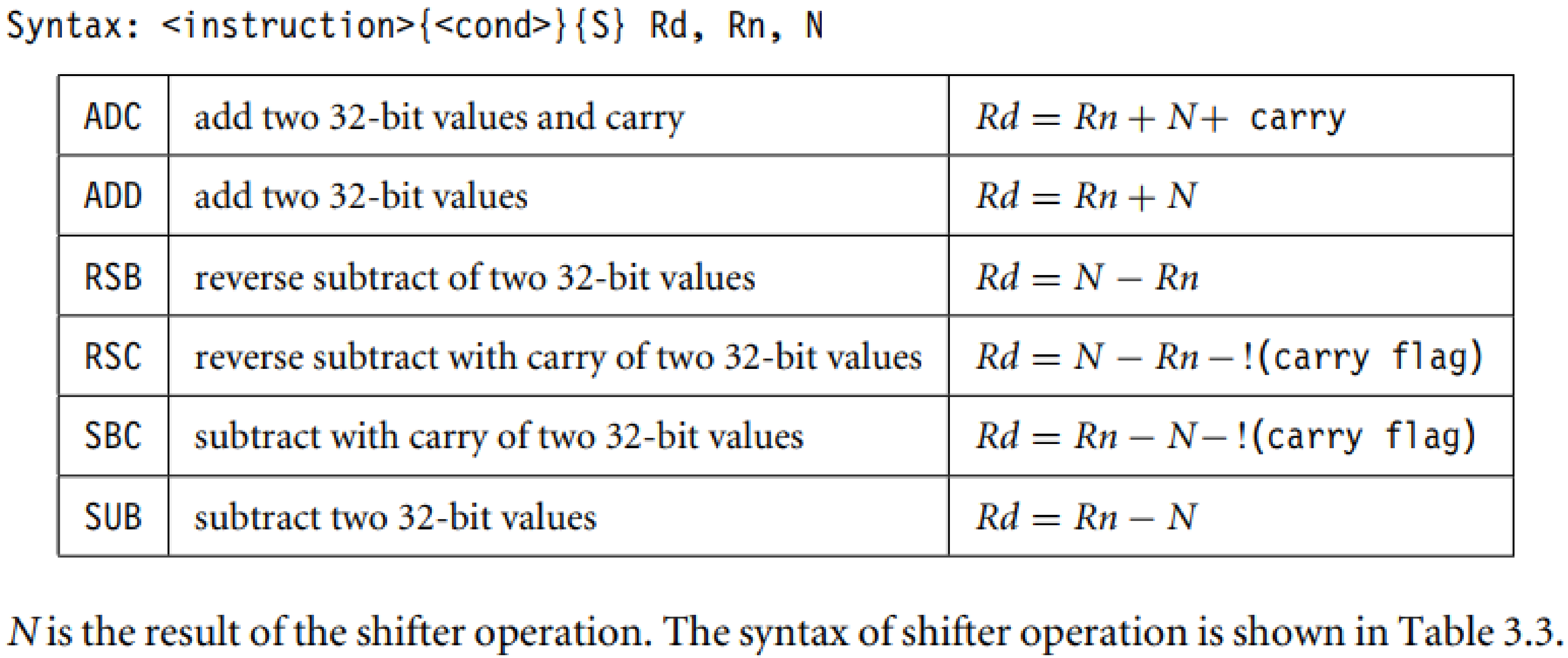
- Barrel Shifter
  - Example of a $MOVS$ instruction
    - ✓ It shifts register $r1$ left by one bit.
    - ✓ The C flag is updated in the $cpsr$
    - ✓ because the S suffix is present in the instruction mnemonic $MOVS$.



```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

Data Processing Instructions

- Arithmetic Instructions
  - Addition and subtraction of 32-bit signed and unsigned values



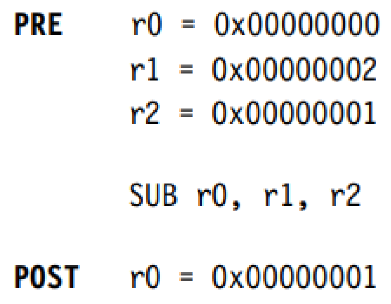
`Syntax: <instruction>{<cond>}{S} Rd, Rn, N`

`ADC`	add two 32-bit values and carry	$Rd = Rn + N + \texttt{carry}$
`ADD`	add two 32-bit values	$Rd = Rn + N$
`RSB`	reverse subtract of two 32-bit values	$Rd = N - Rn$
`RSC`	reverse subtract with carry of two 32-bit values	$Rd = N - Rn - !(\texttt{carry flag})$
`SBC`	subtract with carry of two 32-bit values	$Rd = Rn - N - !(\texttt{carry flag})$
`SUB`	subtract two 32-bit values	$Rd = Rn - N$

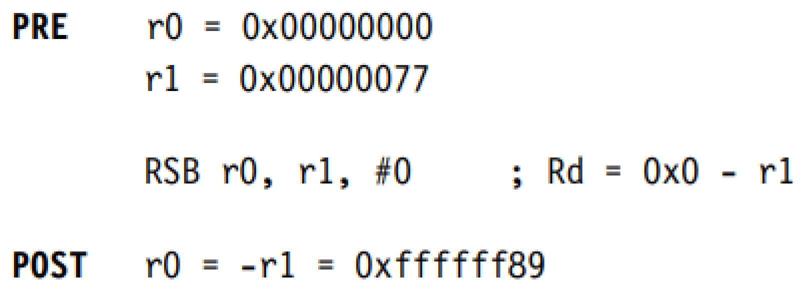
N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

Data Processing Instructions

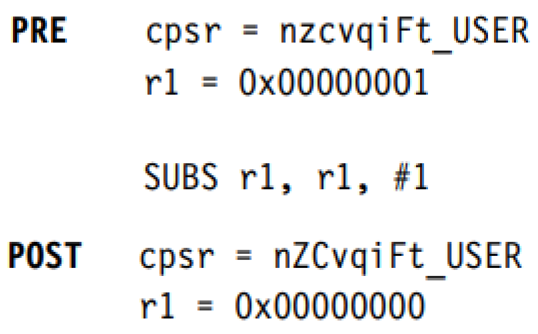
- Arithmetic Instructions
  - Addition and subtraction of 32-bit signed and unsigned values
  - Practice)



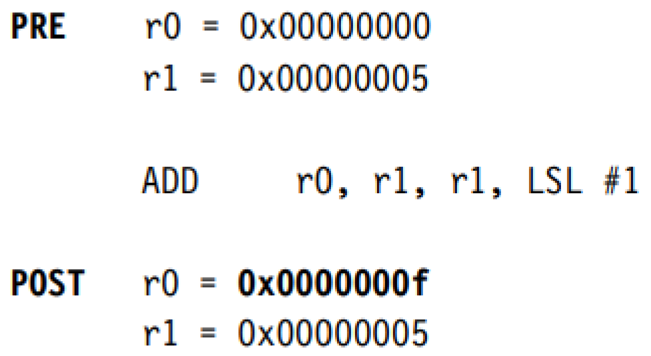
```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```



```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```



```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

        SUBS r1, r1, #1

POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```



```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Data Processing Instructions

- Logical Instructions
  - Bitwise* logical operations on the two source registers

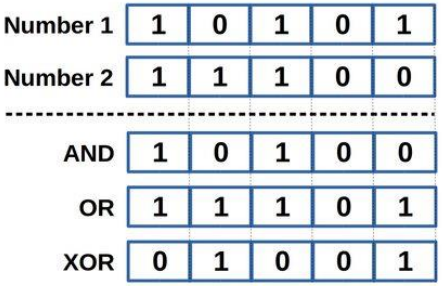
```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

  - Ex)

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

Rd = Rn AND NOT(N)

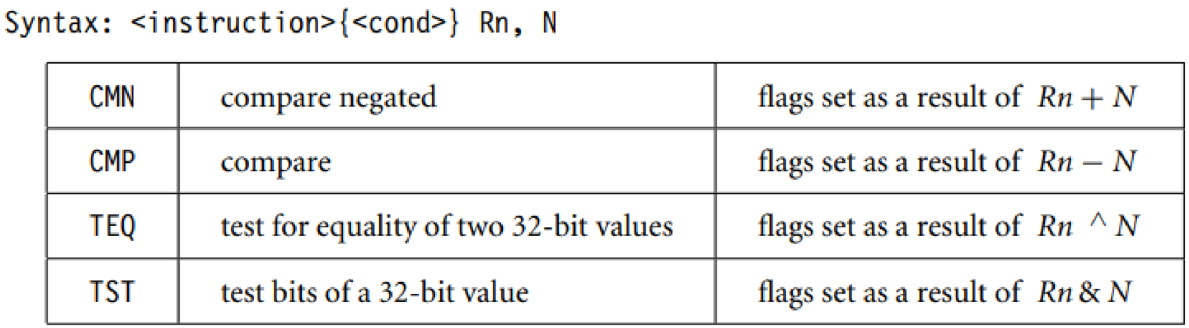


Number 1	1	0	1	0	1
Number 2	1	1	1	0	0
AND	1	0	1	0	0
OR	1	1	1	0	1
XOR	0	1	0	0	1

Bitwise: a level of operation that involves working with individual bits

Data Processing Instructions

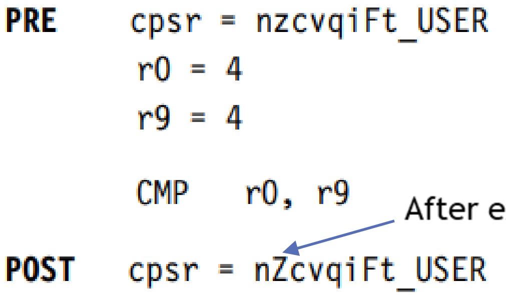
- Comparison Instructions
  - They compare or test a register with a 32-bit value.
  - They update the *cpsr* flag bits according to the result, but do not affect other registers.



Syntax: `<instruction>{<cond>} Rn, N`

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of Rn ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

  - Ex)

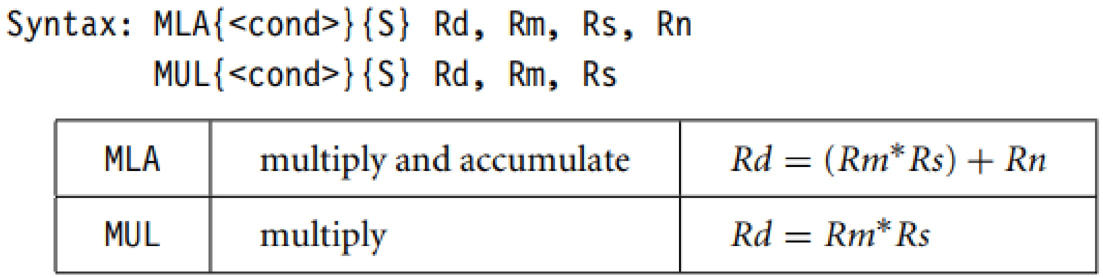


```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

After execution the z flag changes to 1 (uppercase Z)

Data Processing Instructions

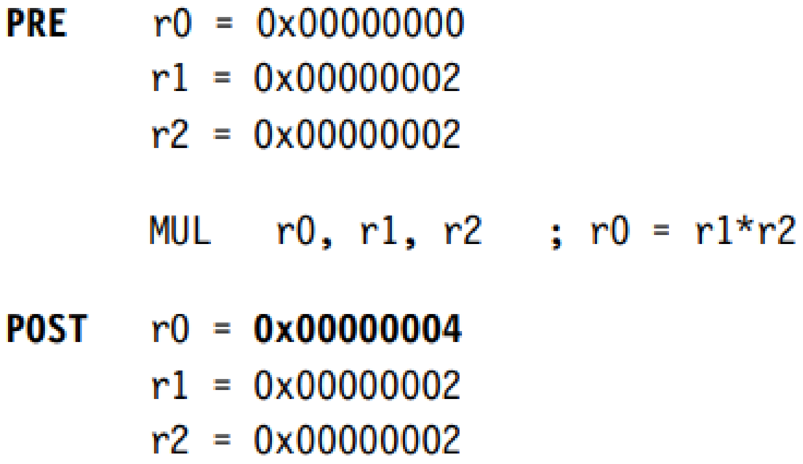
- Multiply Instructions
  - They multiply the contents of a pair of registers.



```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^{*}Rs) + Rn$
MUL	multiply	$Rd = Rm^{*}Rs$

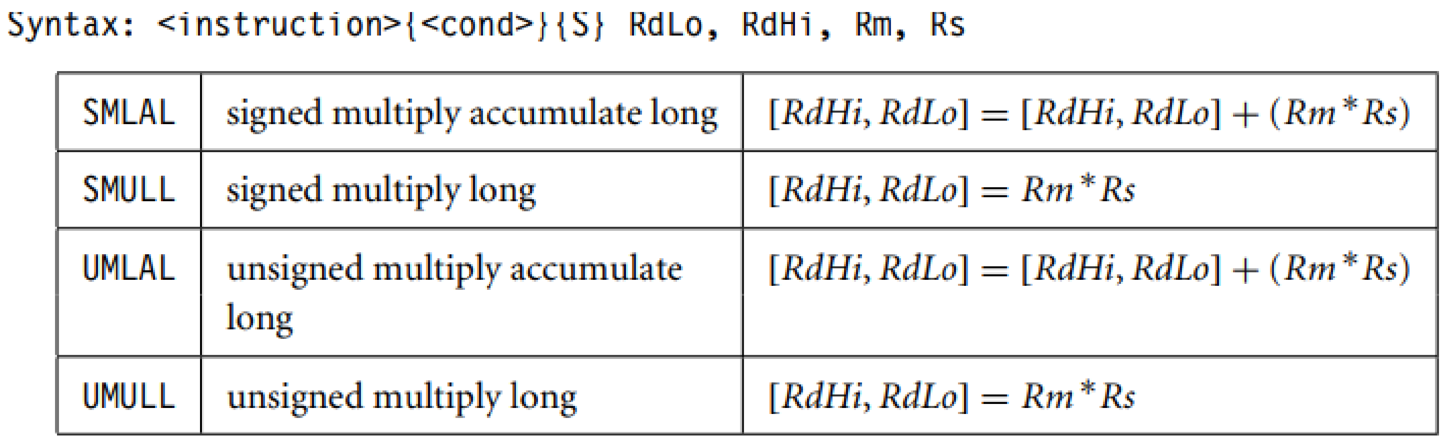
  - Ex)



```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

Data Processing Instructions

- Multiply Instructions
  - They multiply the contents of a pair of registers.
  - The long multiplies accumulate onto a pair of registers representing a 64-bit
    - ✓ long multiply instructions ($SMLAL$, $SMULL$, $UMLAL$, and $UMULL$) produce a 64-bit result.
    - ✓ The Result is placed in two registers labeled $RdLo$ and $RdHi$.
    - ✓ $RdLo$ holds the lower 32 bits of the 64-bit result, and $RdHi$ holds the higher 32 bits of the 64-bit result

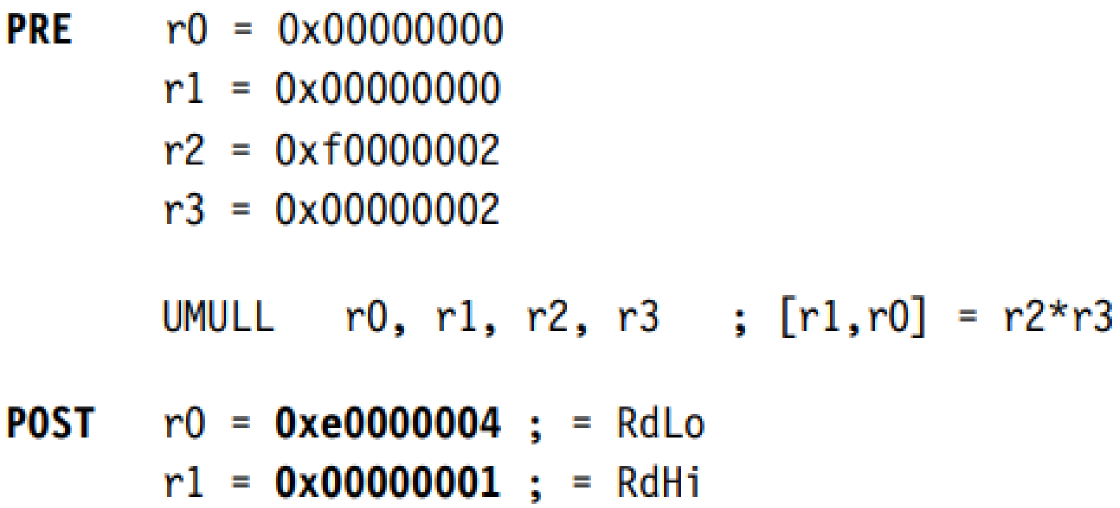


```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

Data Processing Instructions

- Multiply Long Instructions
  - Ex)



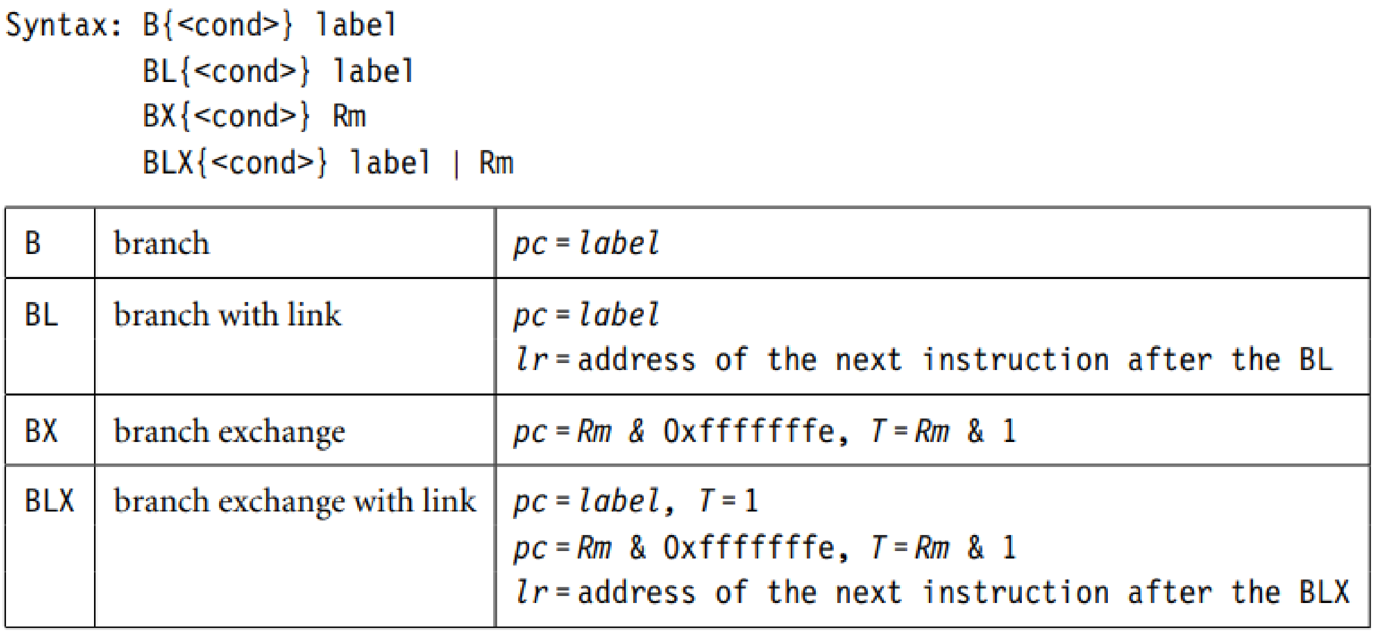
```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3    ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Agenda

- ARM Instruction Set Architecture Features
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

Branch Instructions

- A branch instruction changes the flow of execution or is used to call a routine.
- It allows programs to have subroutines, *if-then-else* structures, and loops.

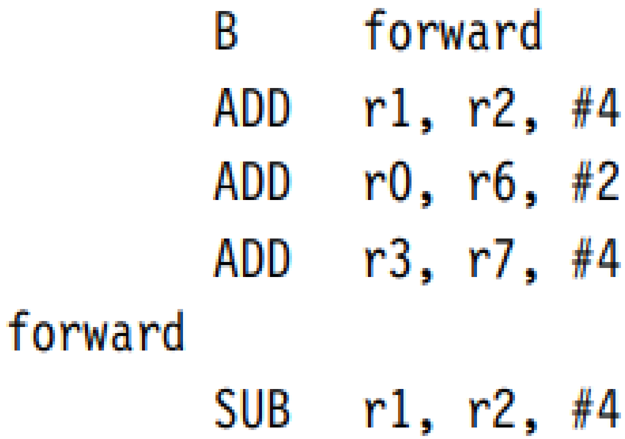


```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

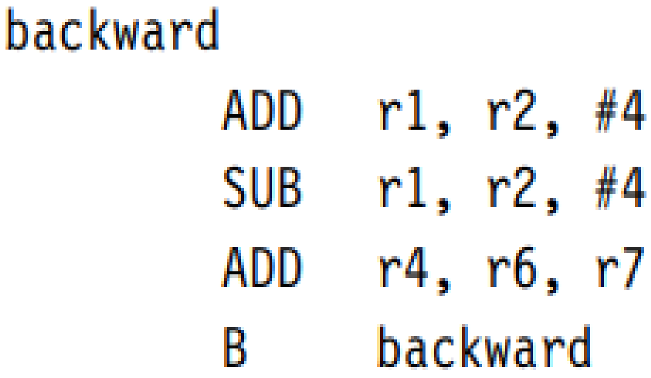
B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

Branch Instructions

- Ex)
  - In this example, *forward* and *backward* are the labels.
    - ✓ The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.
  - The forward branch skips three instructions.
  - The backward branch creates an infinite loop.



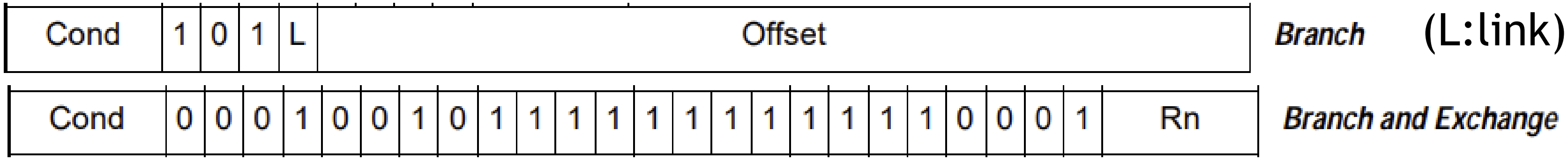
```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
```



```
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

Branch Instruction Format

- Branch Instruction Format
  - Opcode, offset that specifies the target address



Cond	1	0	1	L	Offset	***Branch***

(L:link)

Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	***Branch and Exchange***

  - Example
    - ✓B label (branch),
    - ✓BL label (branch with link) (pc = label, lr = address of next instr) L=1
    - ✓BX Rm (branch and exchange, pc = Rm, T = Rm & 1)
    - ✓BLX (branch and exchange with link, pc = label,T=1, lr =return addr.)

Branch Instructions

- Divert sequential execution / CALL a subroutine
- Range +/- 32MB from current position (PC-8), PC relative offset
- With 24-bit offset oppend ‘OO’ at LSB, extend sign bit, place into PC
- PC is set to point to next new address
- How is asm instruction ‘here B here’ encoded? – 0xEAFFFFFE
    - Hint: [31:28]=1110, [27:24]=1010, [23:0] = 24-bit offset

Branch Instructions - Examples

- Example of using ‘B’ instruction:

```
        CMP r0,#0    ; check if r0 == 0
        BNE  r2inc   ; if r0 !- 0 branch to r2inc
        ADD  r1,r1,#1  ; r1 += 1
        B   next
r2inc   ADD r2,r2,#1
next  ---
```

- Example of using ‘BL’ instruction

```
        BL  func1
 func1  ADD r0,r0,#1   ; subroutine
        ------              ; codes
        MOV pc,lr    ; return to program
```

Branch Instruction - Examples

- Examples using ‘BX’ instruction

```
; ARM state code
        CODE32     ; 32 instructions follow
        LDR   r0,=tcode+1   ; address of tcode to r0,
                            ; +1 enter Thumb state
        MOV lr, pc          ; save return address
        BX   r0             ; branch to thumb code
         ------------       ; ARM code continue
; Thumb state code
          CODE16            ; to begin Thumb code execution
```

- Example of using ‘BLX’ instruction

  In the above example replace ‘MOV lr, pc’ and ‘BX r0’ with ‘BLX r0’

Features of Conditional Execution Instructions

- Improves execution speed and offers high code density

‘C’ program	**ARM program using branch instructions**	**ARM program using conditional instructions**

If (r0 == 0)
r1 = r1 + 1;
else
r2 = r2 + 1;
	CMP r0,#0
	BNE else
	ADD r1,r1,#1
	B end
	else ADD r2,r2,#1
	end
		CMP r0, #0
		ADDEQ r1, r1, #1
		ADDNE r2, r2, #1

Comparison of the two approaches	Instructions : 5 Memory space : 20 bytes No. of Cycles: 5 or 6	Instructions : 3 Memory space: 12 bytes No. of cycles: 3

Agenda

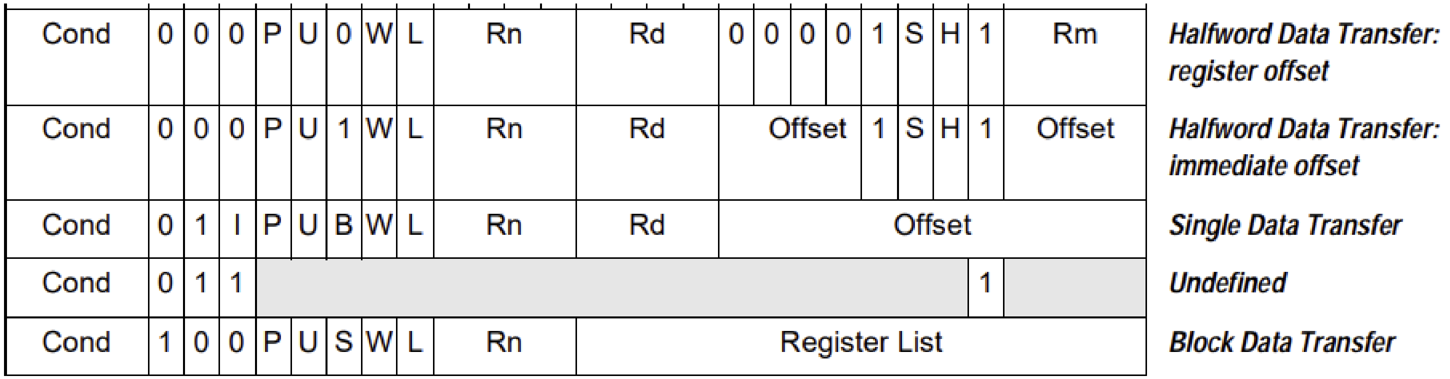
- ARM Instruction Set Architecture Features
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

Load-Store Instructions

- Load-store instructions transfer data between memory and processor registers.
- Three types of load-store instructions:
  - single-register transfer
  - multiple-register transfer
  - swap

Load/Store Instruction Format

- Load/Store Instruction Format

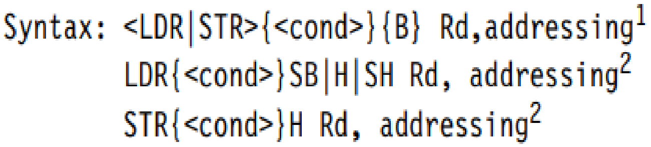


Cond	0	0	0	P	U	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm	***Halfword Data Transfer: register offset***
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset				1	S	H	1	Offset	***Halfword Data Transfer: immediate offset***
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset									***Single Data Transfer***
Cond	0	1	1															1		***Undefined***
Cond	1	0	0	P	U	S	W	L	Rn	Register List										***Block Data Transfer***

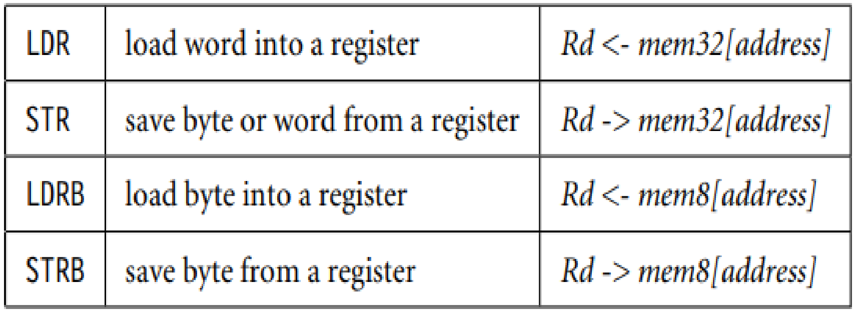
- Example
  - ✓LDR : load word into register (Rd ← mem32[addr]
  - ✓STR: store byte or word from a register
  - ✓LDRB: load byte into a register, STRMB : store byte from a register
  - ✓LDRH : load halfword , STRH : store halfword
  - ✓LDRSB : load signed byte, LDRSH : load signed short

Load-Store Instructions

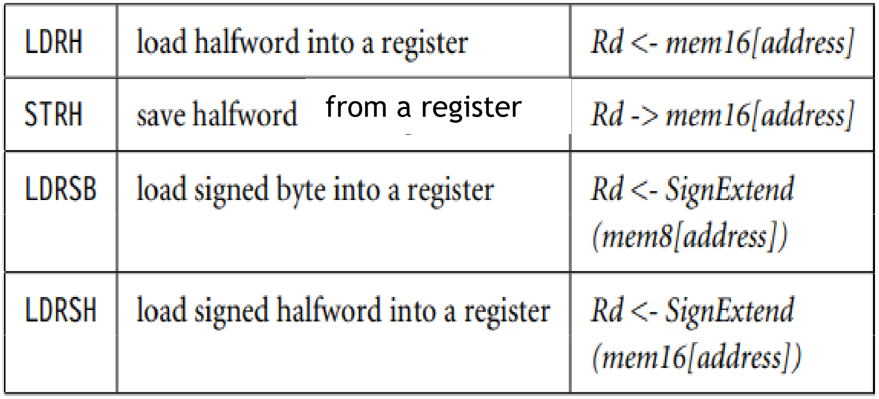
- Single-Register Transfer
  - The datatypes supported
    - ✓ signed and unsigned words (32-bit)
    - ✓ halfwords (16-bit)
    - ✓ bytes.



Syntax: <LDR|STR>{<cond>}{B} Rd,addressing^1
LDR{<cond>}SB|H|SH Rd, addressing^2
STR{<cond>}H Rd, addressing^2



LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword from a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Tables 3.5 and 3.7 in the textbook describe the $addressing^1$ and $addressing^2$ syntax

Load-Store Instructions

- Single-Register Transfer
  - Ex)
    - ✓ The first instruction loads a word from the address stored in register $r1$ and places it into register $r0$.
    - ✓ The second instruction goes the other way by storing the contents of register $r0$ to the address contained in register $r1$.
    - ✓ The offset from register $r1$ is zero in these examples.
    - ✓ Register $r1$ is called the ***base address register***.

```
LDR     r0, [r1]          ; = LDR r0, [r1, #0]

STR     r0, [r1]          ; = STR r0, [r1, #0]
```

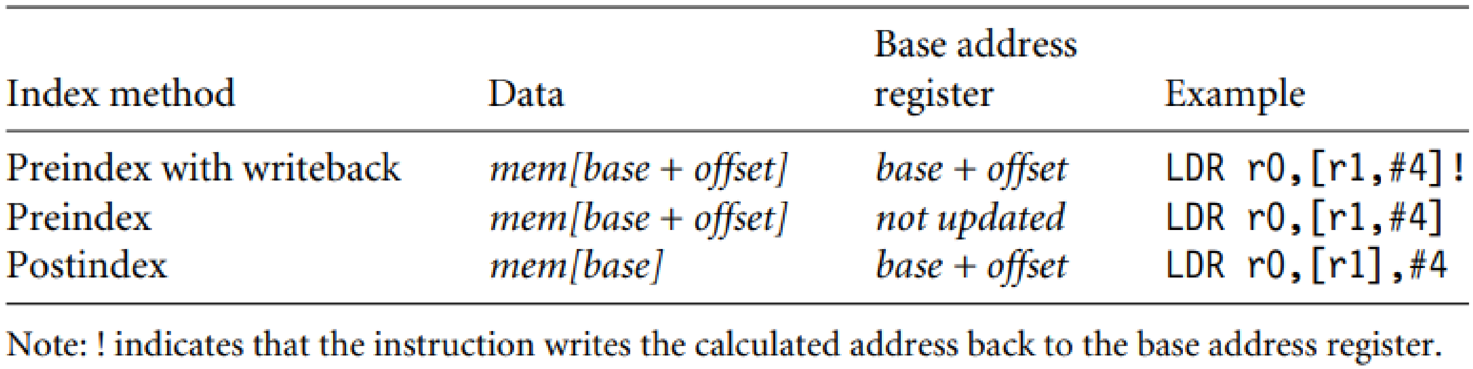
Indexing Method

- Preindexed
  - <opcode>{<cond>}{<type>} Rd, [Rn {, <offset>]
- Preindex with write back
  - <opcode>{<cond>}{<type>} Rd, [Rn { <offset>] !
- Postindexed
  - <opcode>{<cond>}Rd, [RN], <offset>

Indexing	Instr	Operation
Perindex	LDR Rd, [Rn,n]	Rd=M[Rn+n]
	STR Rd, [Rn,n]	M[Rn+n]=Rd
Peindex with write back	LDR Rd, [Rn,n]!	Rd=M[Rn+n], Rn=Rn+n
	STR Rd, [Rn,n]!	M[Rn+n]=Rd, Rn=Rn+n
Poxtindex	LDR Rd, [Rn],n	Rd=M[Rn], Rn=Rn+n
	STR Rd, [Rn],n	M[Rn]=Rd, Rn=Rn+n

Load-Store Instructions

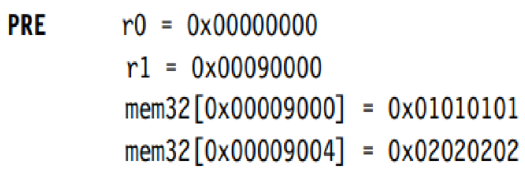
- Single-Register Load-Store Addressing Modes



Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

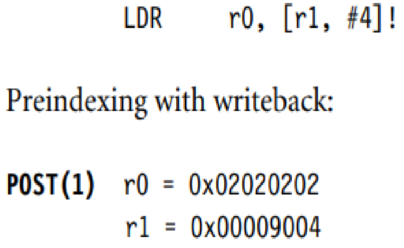
Note: ! indicates that the instruction writes the calculated address back to the base address register.

- Ex)



```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202
```

Pre-indexing with writeback

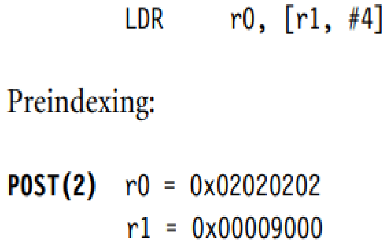


```
        LDR     r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)   r0 = 0x02020202
          r1 = 0x00009004
```

Pre-indexing

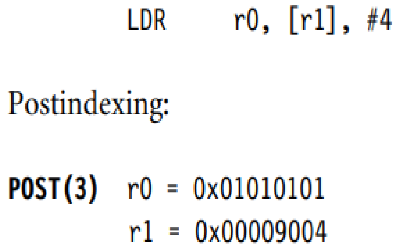


```
        LDR     r0, [r1, #4]
```

Preindexing:

```
POST(2)   r0 = 0x02020202
          r1 = 0x00009000
```

Post-indexing



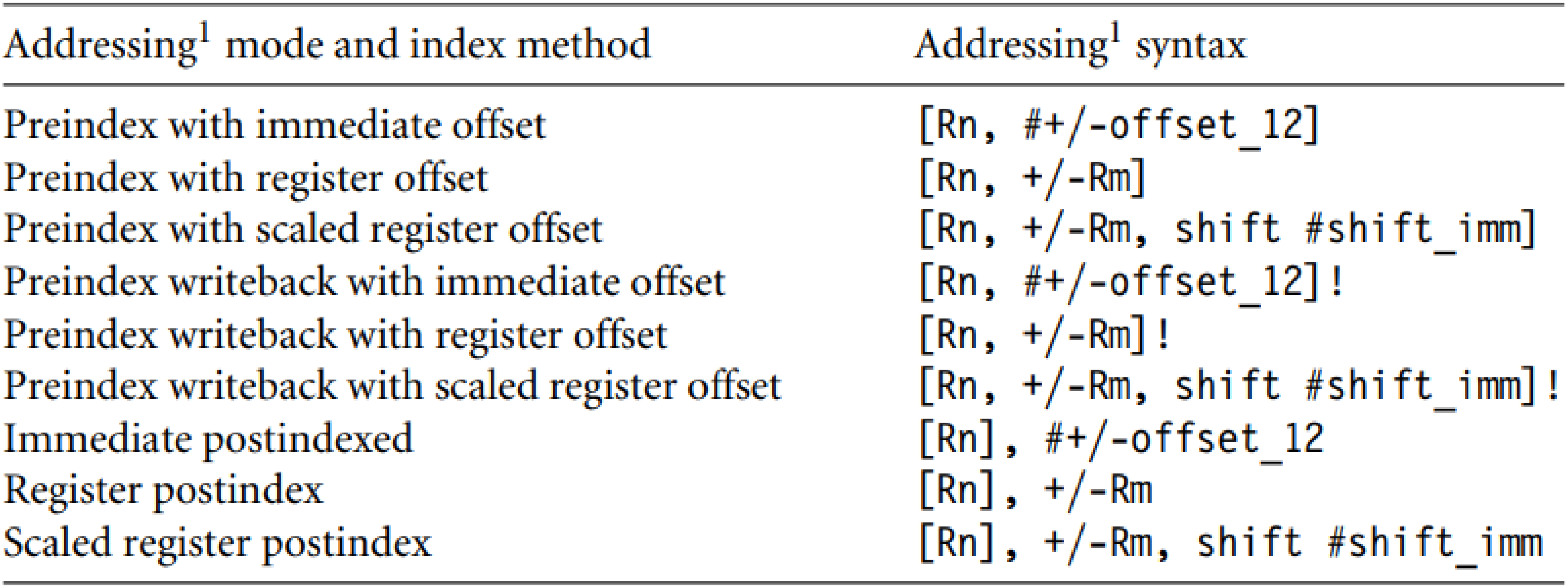
```
        LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)   r0 = 0x01010101
          r1 = 0x00009004
```

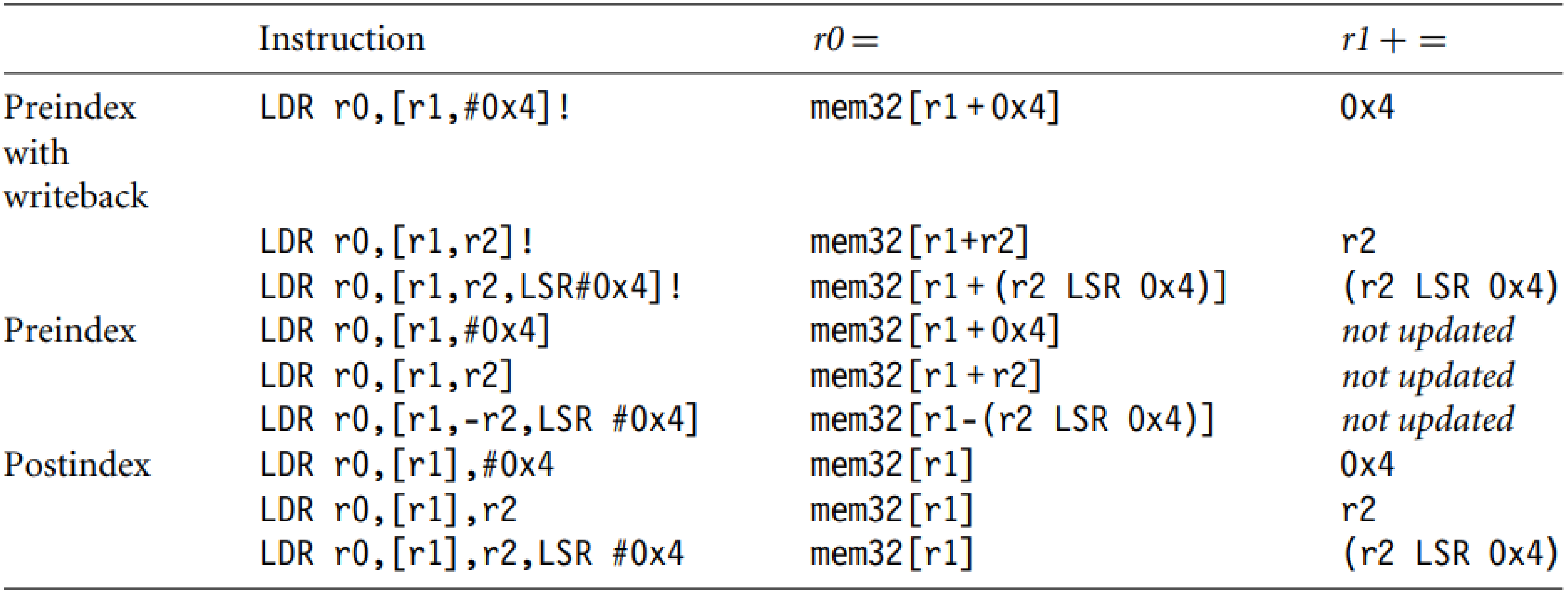
Load/Store Instructions

- Single-register Load-Store **Addressing**, word or unsigned byte
  - Preindex: access Mem[base+offset]
  - preindex with writeback : access Mem[base+offset] and the update base
  - Postindex: access Mem[base] and then update base
  - offset: number of bytes from byte address pointed by base



Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

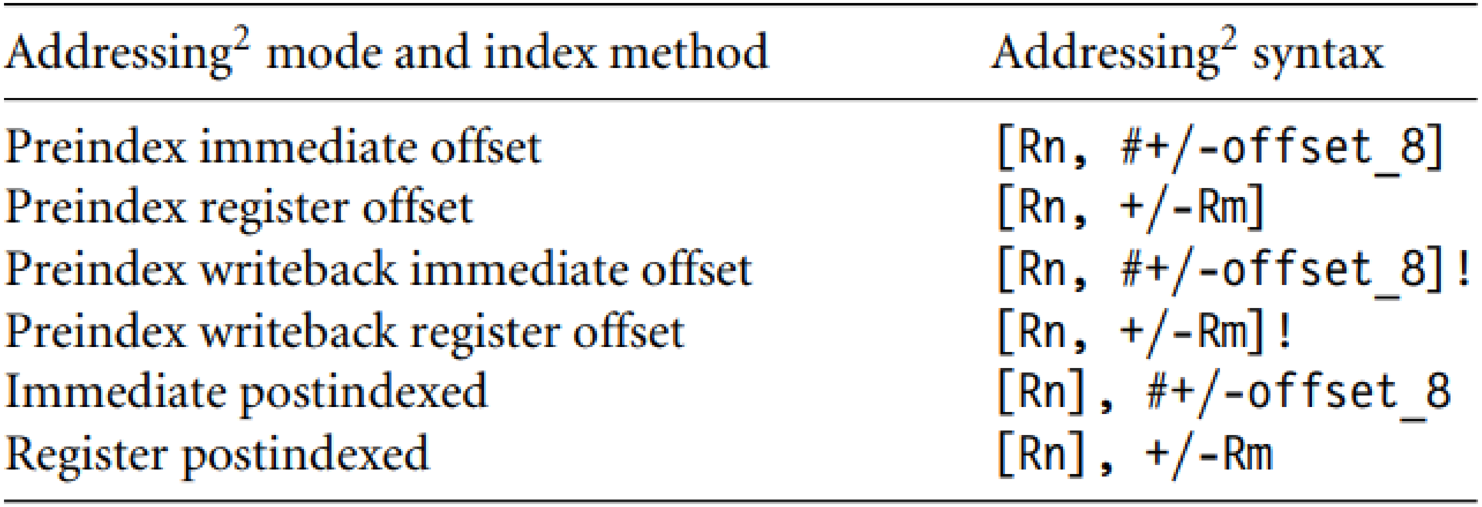
Examples of LDR instructions with different addressing modes



	Instruction	*r0* =	*r1* + =
Preindex with writeback	`LDR r0,[r1,#0x4]!`	mem32[r1+0x4]	0x4
	`LDR r0,[r1,r2]!`	mem32[r1+r2]	r2
	`LDR r0,[r1,r2,LSR#0x4]!`	mem32[r1+(r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	`LDR r0,[r1,#0x4]`	mem32[r1+0x4]	*not updated*
	`LDR r0,[r1,r2]`	mem32[r1+r2]	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	mem32[r1-(r2 LSR 0x4)]	*not updated*
Postindex	`LDR r0,[r1],#0x4`	mem32[r1]	0x4
	`LDR r0,[r1],r2`	mem32[r1]	r2
	`LDR r0,[r1],r2,LSR #0x4`	mem32[r1]	(r2 LSR 0x4)

Load-Store Instructions (Halfword, Byte)

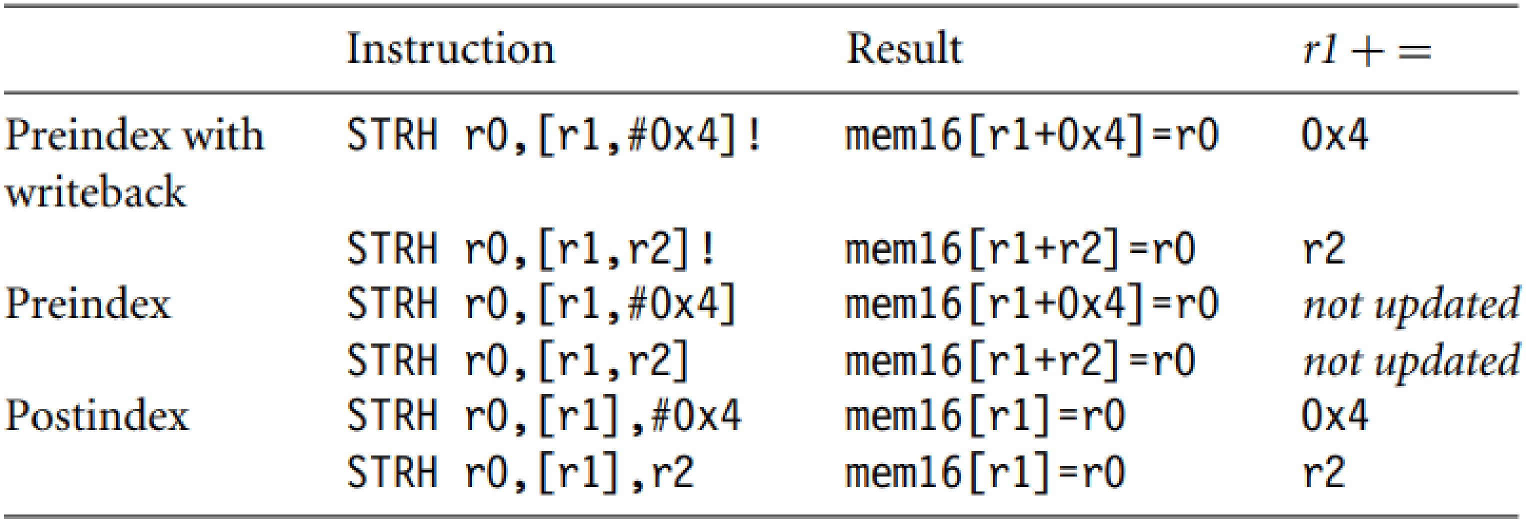
- single register load-store addressing, 16-bit halfword (STRH), signed halfword, signed byte (STRB)
    - STRH support both signed and unsigned halfword
    - STRB support both signed and unsigned byte
- These operations cannot use barrel shifter



Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

Load-Store Instructions

- Various examples of STRH (Store Halfword) instructions



	Instruction	Result	$r1 +=$
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Load-Store Instructions (Block)

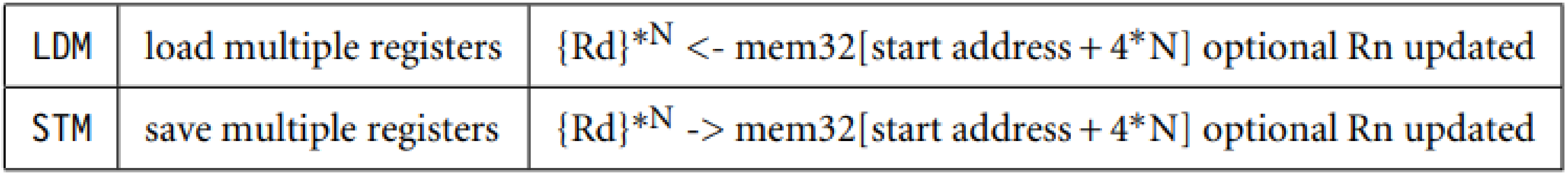
- Transfer data between multiple regs & mem in single instruction
- Use : Stack, block move, temporary store&restore
- Advantage: small code size, single instr. Fetch
- Disadvantage: can’t be interrupted, increase interrupt latency
- Syntax: <opcode>{<cond>}<mode>Rn{!}, <registers>
    - Opcode: STM, LDM
    - Rn –base register, ‘!’ updates base register after data transfer

<mode>	description	Start address	End address	Rn!
IA	Increment After	Rn	Rn+N*4-4	Rn+N*4
IB	Increment Before	Rn+4		Rn+N*4
DA	Decrement After	Rn-N*4+4	Rn	Rn-N*4
DB	Decrement Before	Rn-N*4	Rn-4	Rn-N*4

Load-Store Instructions (Block)

- Multiple-Register Transfer
  - Multiple-register transfer instructions are more efficient from single-register transfers <u>for moving blocks of data</u> around memory

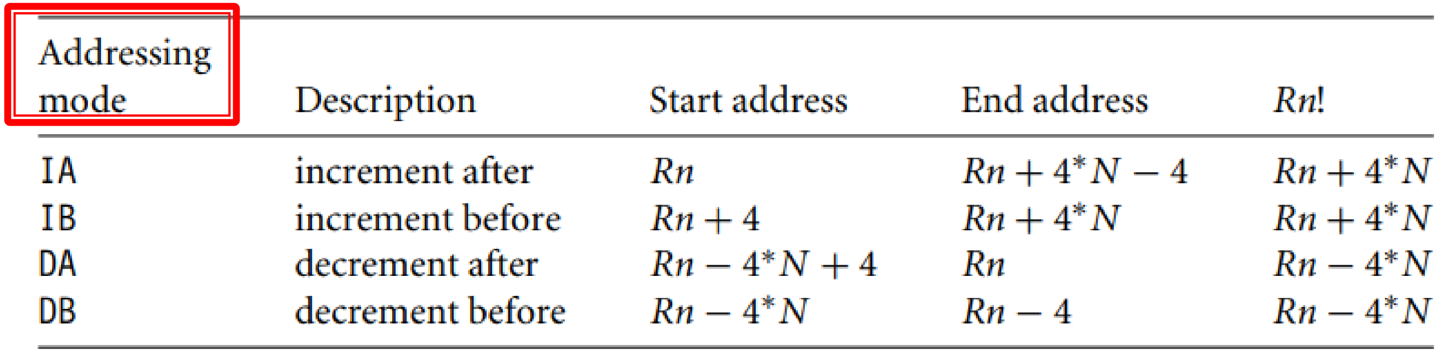
```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```



`LDM`	load multiple registers	$\{Rd\}^{*N}$ <- mem32[start address + 4*N] optional Rn updated
`STM`	save multiple registers	$\{Rd\}^{*N}$ -> mem32[start address + 4*N] optional Rn updated

N: number of registers in the list

Addressing mode for load-store multiple instructions.



Addressing mode	Description	Start address	End address	$Rn!$
`IA`	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
`IB`	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
`DA`	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
`DB`	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

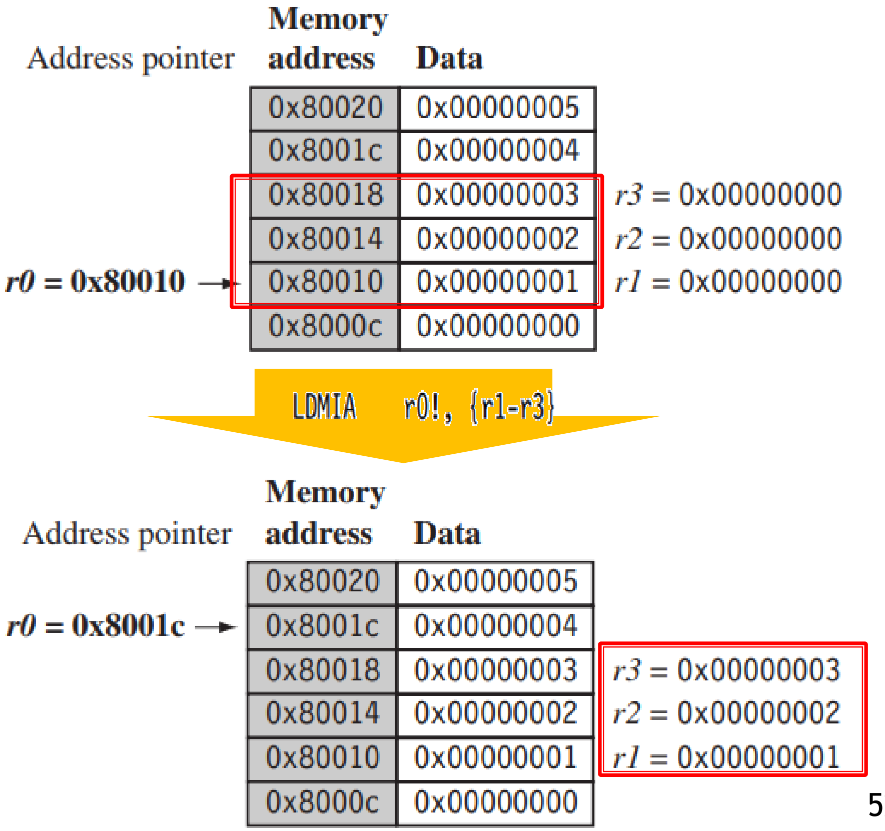
Load-Store Instructions (Block)

- Multiple-Register Transfer
  - ✓In this example, register $r0$ is the base register.
  - ✓“-” character is used to identify a range of register.

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000
```

```
LDMIA     r0!, {r1-r3}
```

```
POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```



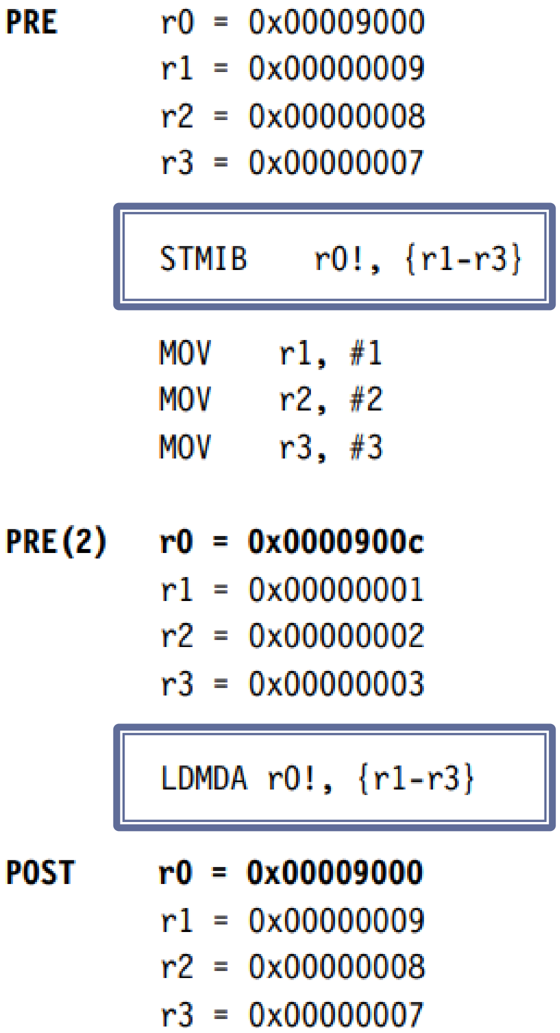
Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	r3 = 0x00000000
	0x80014	0x00000002	r2 = 0x00000000
r0 = 0x80010 →	0x80010	0x00000001	r1 = 0x00000000
	0x8000c	0x00000000	

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	r3 = 0x00000003
	0x80014	0x00000002	r2 = 0x00000002
	0x80010	0x00000001	r1 = 0x00000001
	0x8000c	0x00000000	

Load-Store Instructions (Block)

- Multiple-Register Store
  - Ex)
    - ✓ The **STMIB** instruction stores the values 7, 8, 9 to memory. We then corrupt register $r1$ to $r3$.
    - ✓ The **LDMDA** <u>reloads the original values </u>and restores the base pointer $r0$.

Store Multiple	Load Multiple	Stack Operation
STMIA	LDMDB	Empty Ascending
STMIB	LDMDA	Full Ascending
STMDA	LDMIB	Empty Descending
STMDB	LDMIA	Full Descending

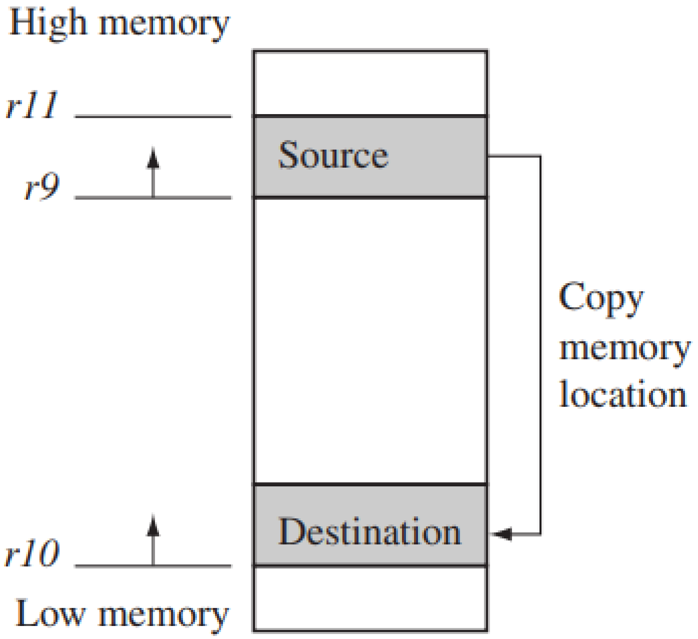


```
PRE       r0 = 0x00009000
          r1 = 0x00000009
          r2 = 0x00000008
          r3 = 0x00000007

          STMIB    r0!, {r1-r3}

          MOV      r1, #1
          MOV      r2, #2
          MOV      r3, #3

PRE(2)    r0 = 0x0000900c
          r1 = 0x00000001
          r2 = 0x00000002
          r3 = 0x00000003

          LDMDA r0!, {r1-r3}

POST      r0 = 0x00009000
          r1 = 0x00000009
          r2 = 0x00000008
          r3 = 0x00000007
```

Load-Store Instructions (Block)

- Block Copy Example

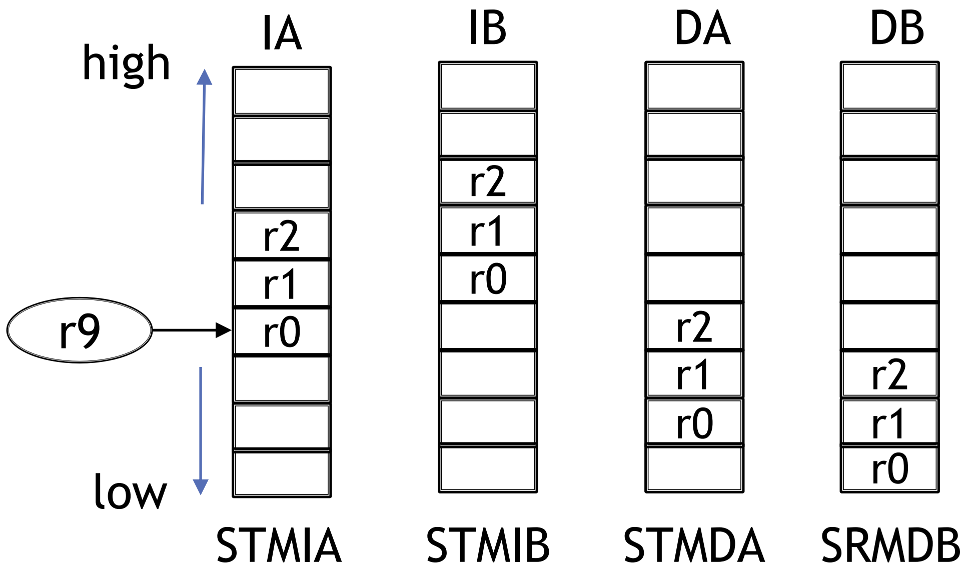
```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to the end of source data

loop
    ; load 32 bytes from source and update r9
    LDMIA r9!, {r0-r7}

    ; store 32 bytes to destination and updates r10
    STMIA r10! {r0-r7}

    ; have we reached the end?
    CMP r9, r11
    BNE loop
```



Load-Store Examples

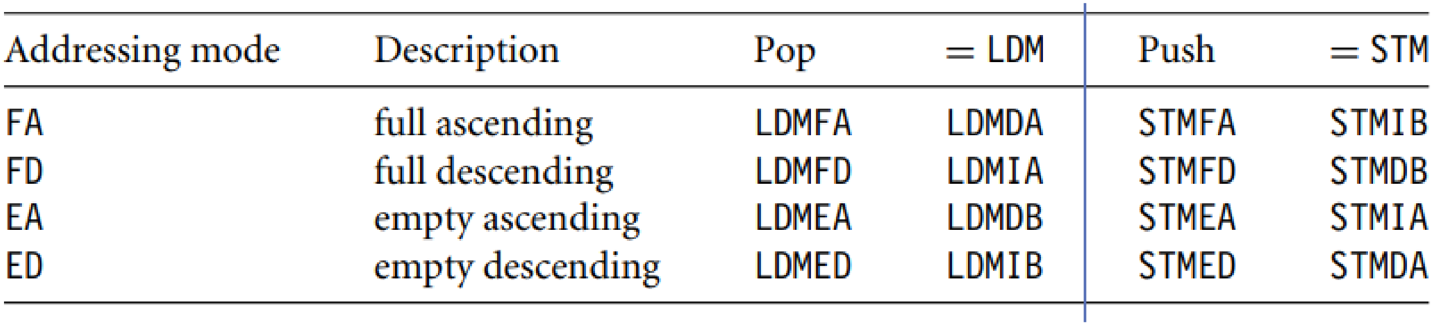
- STMxx r9, {r0-r2}
- LDMxx r9, {r0-r2}

xx = IA, IB, DB, or DA



Stack Operation

- Stack Operations
  - Push (placing data onto stack): STM (store multiple instruction)
  - Pop (removing data from stack): LDM (load multiple instruction)
- 4 types of Stack
  - Stack growing direction: Ascending(A) vs. Descending (D)
  - Stack Pointer(sp) points to last used location (Full) or first unused location (Empty)
- Stack Addressing Mode Aliases (4 types)



Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

Stack Operation (Push Example)

PRE

```
r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080014
```

```
STMFD   sp!, {r1,r4}
```

PRE	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

PRE

```
r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080010
```

```
STMED   sp!, {r1,r4}
```

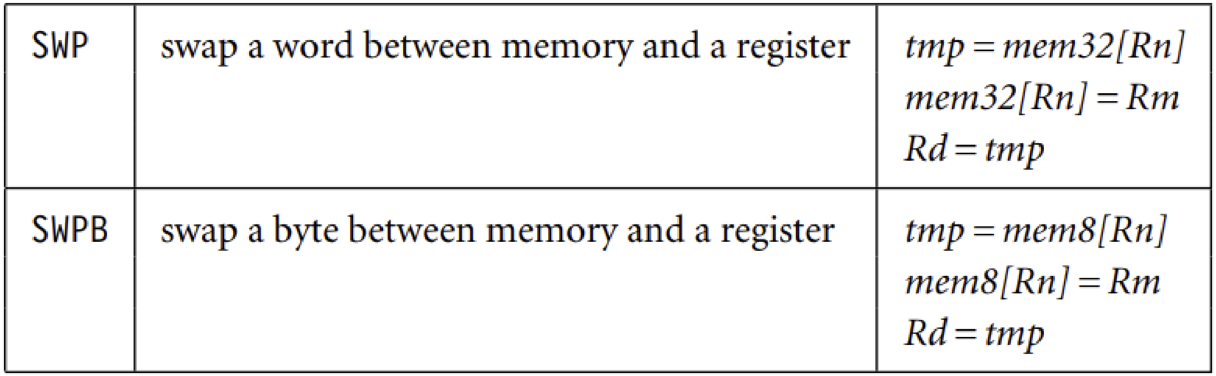
PRE	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

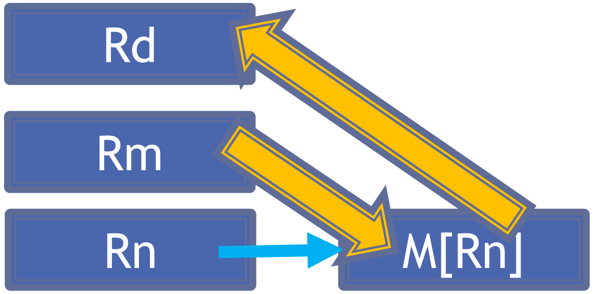
SWAP Instruction

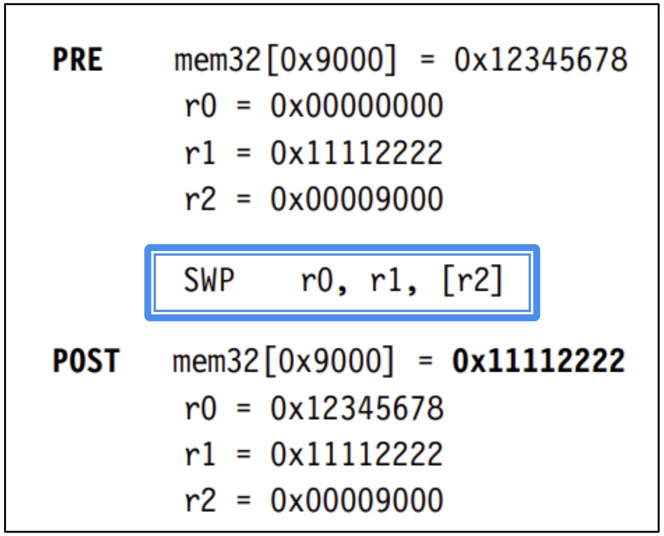
`Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]`

- Swaps contents of register and contents of memory
  - SWP : Word
  - SWPB : Byte



SWP	swap a word between memory and a register	*tmp* = *mem32[Rn]* *mem32[Rn]* = *Rm* *Rd* = *tmp*
SWPB	swap a byte between memory and a register	*tmp* = *mem8[Rn]* *mem8[Rn]* = *Rm* *Rd* = *tmp*

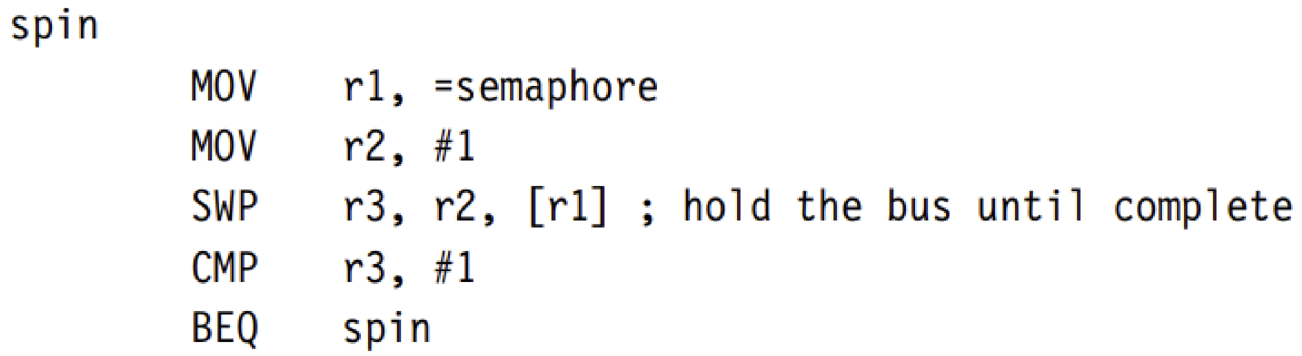




```
PRE     mem32[0x9000] = 0x12345678
         r0 = 0x00000000
         r1 = 0x11112222
         r2 = 0x00009000

        SWP     r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
         r0 = 0x12345678
         r1 = 0x11112222
         r2 = 0x00009000
```

SWAP Instruction

- SWP(B) is an atomic operation
  - Read and Writes a location in the same bus operation, preventing other instruction from accessing to that location until completion
  - useful to implement semaphore and mutual exclusion in OS
  - example) a simple data guard that can be used to protect data from being written by any other task



```
spin
        MOV     r1, =semaphore
        MOV     r2, #1
        SWP     r3, r2, [r1] ; hold the bus until complete
        CMP     r3, #1
        BEQ     spin
```

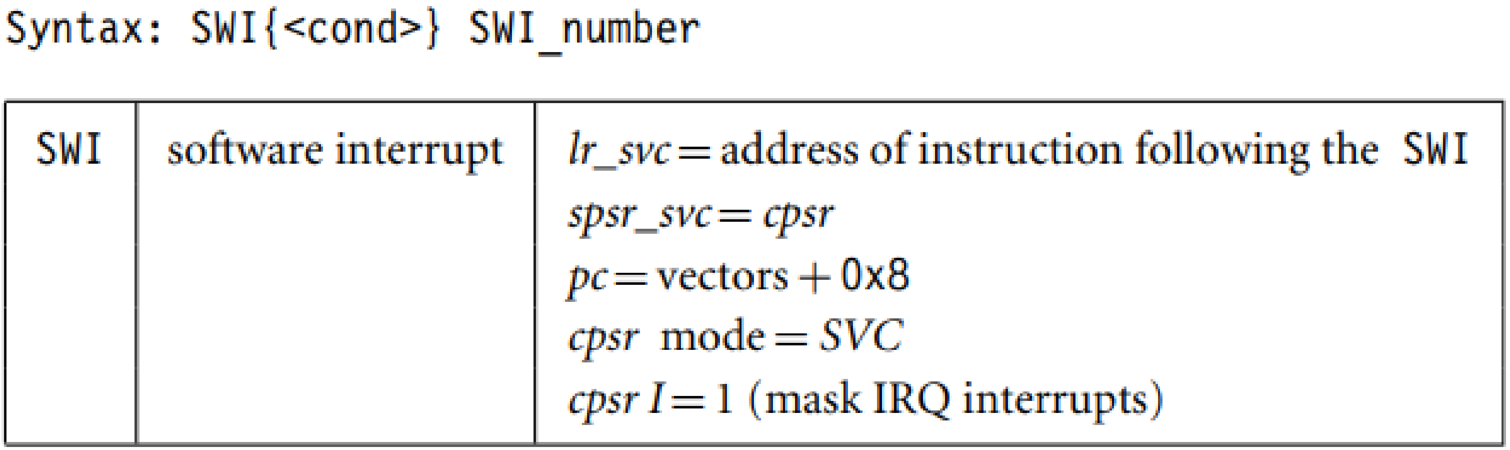
  - The routine continues to loop until the service is released by other process (when the semaphore location contains value 0)

Agenda

- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- **Software Interrupt Instruction**
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

Software Interrupt

- It causes Software interrupt exception proving user mode applications to execute OS routine
- When executed, mode changes to supervisor mode
- Syntax: SWI{<cond>} SWI_number



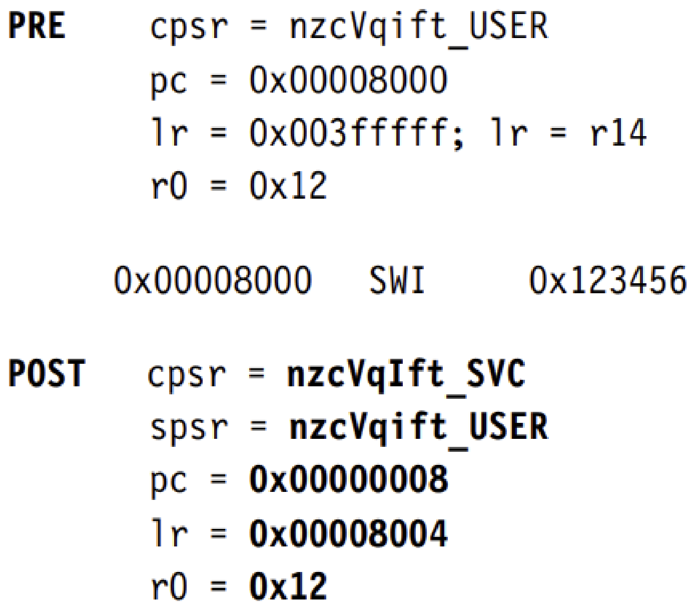
`Syntax: SWI{<cond>} SWI_number`

SWI	software interrupt	$lr\_svc =$ address of instruction following the SWI $spsr\_svc = cpsr$ $pc =$ vectors $+$ 0x8 $cpsr$ mode $= SVC$ $cpsr\ I = 1$ (mask IRQ interrupts)

- SWI handler location is fetched @vector table address + 8

SWI Instruction Example

- SWI 0x123456



```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff; lr = r14
        r0 = 0x12

      0x00008000    SWI      0x123456

POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

- Return instruction from SWI routine: MOVS PC, r14

SWI Handler

```
SWI_handler
     STMFD  sp!, {r0-r12, lr} ; Store registers r0-r12 and link register
     LDR     r10, [lr, #-4] ; read SWI instruction itself
     BIC     r10, r10, #0xff000000  ; Mask off top 8 bits
     BL     service_routine ; r10 – contains the SWI number

    ; return from SWI handler
    LDMFD    sp!, {r0-r12,pc}
```

- (Example)
  - SWI_Number = <SWI instruction code> AND NOT(0xff000000)
  - The number in r10 is then used by the SWI handler to call the appropriate SWI service routine

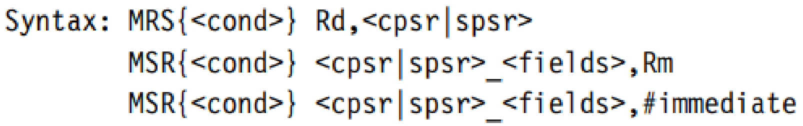
Agenda

- ARM Instruction Set Architecture Overview
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- **Program Status Register Instructions**
- Loading Constants
- ARMv5E Extensions

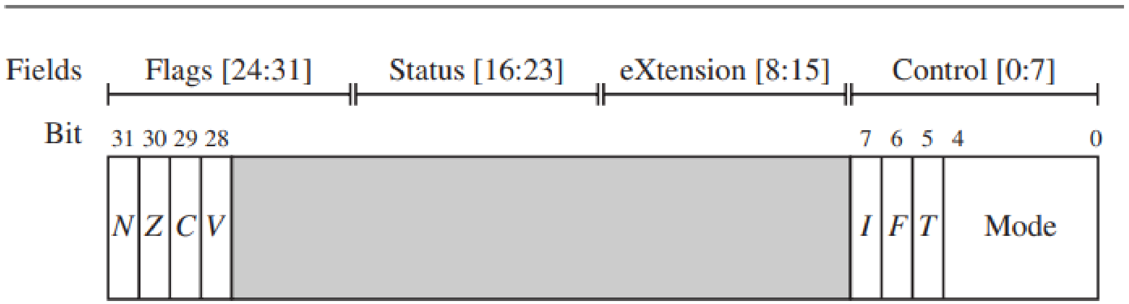
Program Status Register Instructions

- The ARM instruction set provides two instructions to directly control a program status register (psr) to read and write the $cpsr$ and $spsr$. These are MRS instructions.

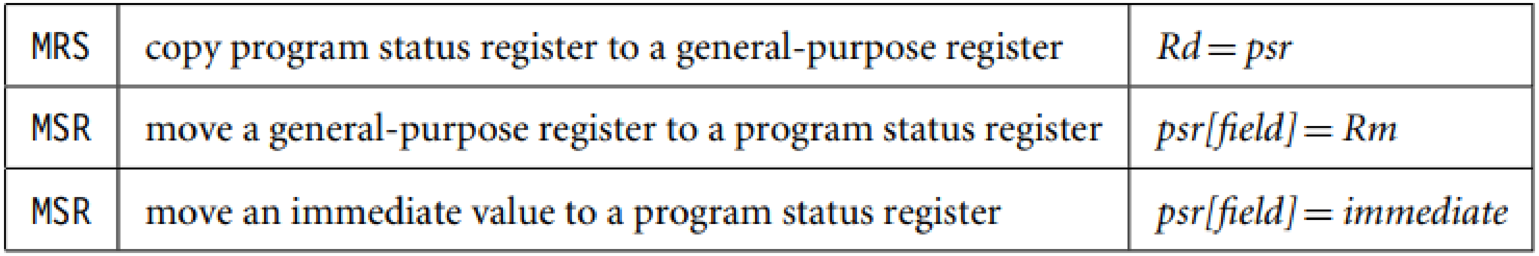
- Fields
  - Combination of control (c), extension (x), status (s), and flags (f)
  - These fields relate to particular byte regions in a psr.



```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

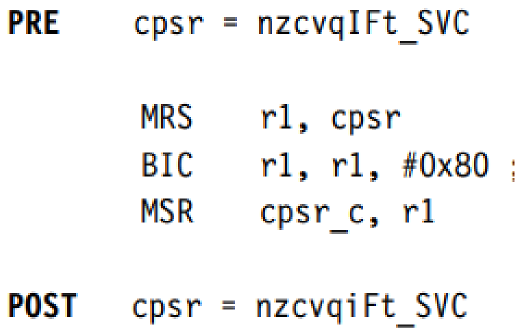


Program Status Register Instructions



MRS	copy program status register to a general-purpose register	$Rd = psr$
MSR	move a general-purpose register to a program status register	$psr[field] = Rm$
MSR	move an immediate value to a program status register	$psr[field] = immediate$

- Ex)
  - The MSR first copies the $cpsr$ into register $r1$.
  - The BIC instruction clears bit 7 or $r1$.
  - Register $r1$ is the copied back into the $cpsr$, which enables IRQ interrupts.



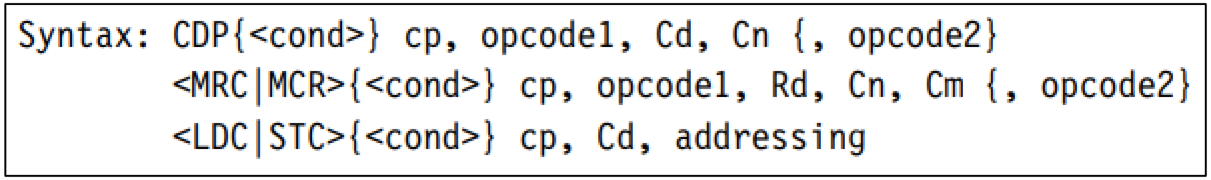
```
PRE     cpsr = nzcvqIFt_SVC

         MRS    r1, cpsr
         BIC    r1, r1, #0x80 ;
         MSR    cpsr_c, r1

POST    cpsr = nzcvqiFt_SVC
```

Program status register instructions

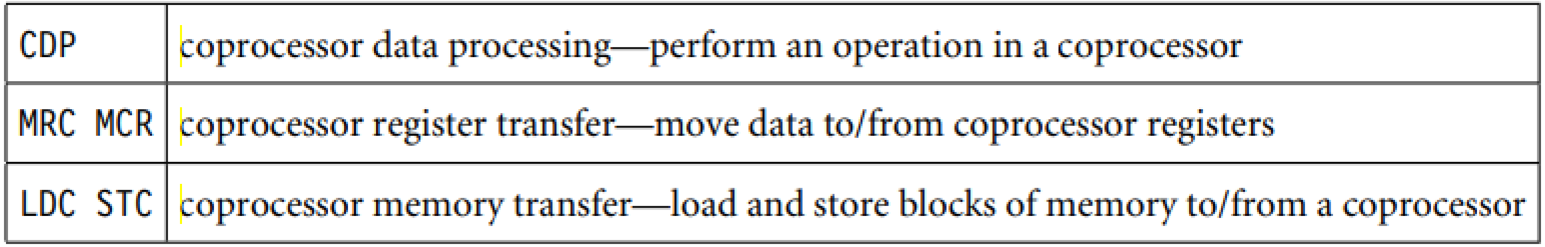
- Instructions to read/write from/to cpsr or spsr
- Instructions: MRS, MSR
- Syntax:
  - MRS{<cond>} Rd, <cpsr|spsr>
  - MSR{<cond>} <cpsr|spsr>, Rm
  - MSR{<cond>}, <cpsr|spsr>_<fields>, Rm
  - MSR{<cond>|}, <cpsr|spsr>_<fields>, #imm
- Modifying cpsr, spsr: read, modify, and write back technique
  - Read cpsr/spsr using MRS
  - Modify relevant bits
  - Transfer to cpsr/spsr using MSR

Coprocessor Instructions

- Coprocessor instructions are used to extend ARM instruction set
  - to provide additional computation capability
  - control memory subsystem including cache and MMU
- 3 types of coprocessor instructions: data processor, register transfer, memory transfer
- Syntax:



```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```



CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

- cp: coprocessor number between p0 to p15
- Cn , Cm, Cd : coprocessor register

Coprocessor Instructions

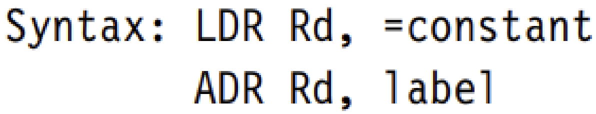
instruction	description	example	interpretation
CDP	Coprocessor data processing	CDP p10, #0, c2, c4, c6, #1	coprocessor p10 opcode: 0 coprocessor register : c2, c4, c6 opcode 2: 1
MCR	Move to cop register from ARM general register	MCR p6, #0, r4, c5, c6	request coprocessor p6 to perform operation 0 in R4 and place the results in c6
MRC	Move to ARM general register from coprocessor register	MRC p15, #0, r10, c0, c0, #0	request CP15 to perform operation 0 (copy) on CP15 register 0 and place the results on r10. This CP15 register0 is copied to general purpose rgister r10

Agenda

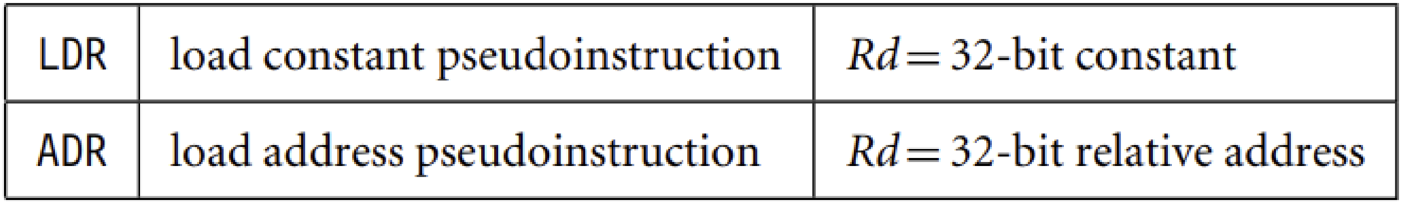
- ARM Instruction Set Architecture Overview
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

Loading Constants

- Since ARM instructions are 32-bit size, they cannot specify a general 32-bit constant
- Two pseudo instructions to move a 32-bit constant value into a register



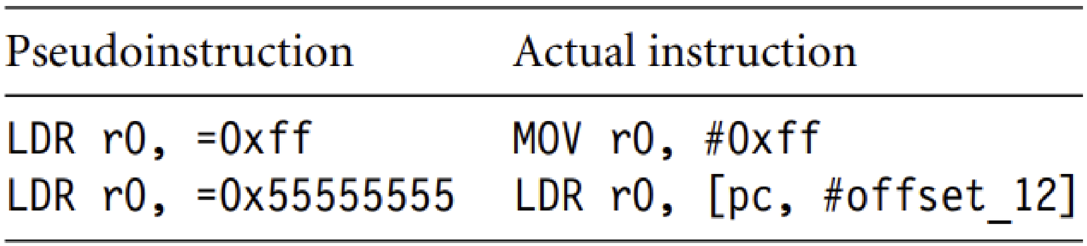
```
Syntax: LDR Rd, =constant
        ADR Rd, label
```



LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

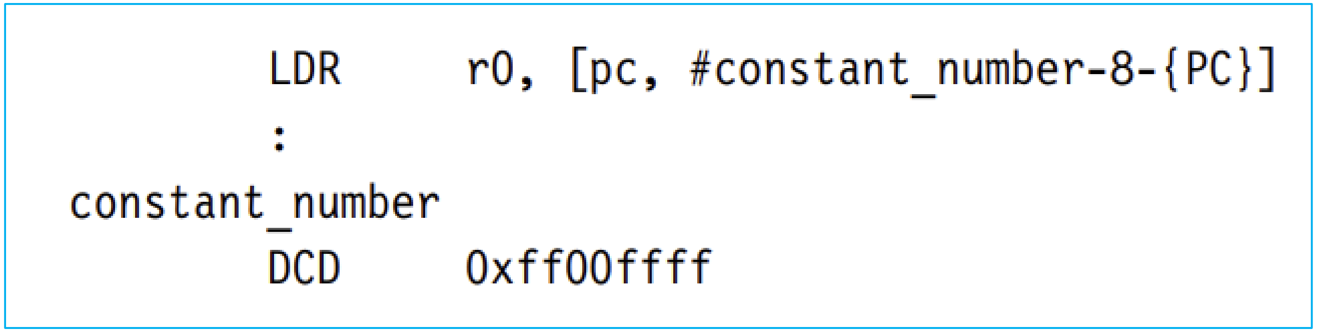
Loading Constants

- LDR pseudo instruction conversion



Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

- Loading a 32-bit constant value method without pseudo instruction (or interpreting pseudo instruction LDR into real LDR)



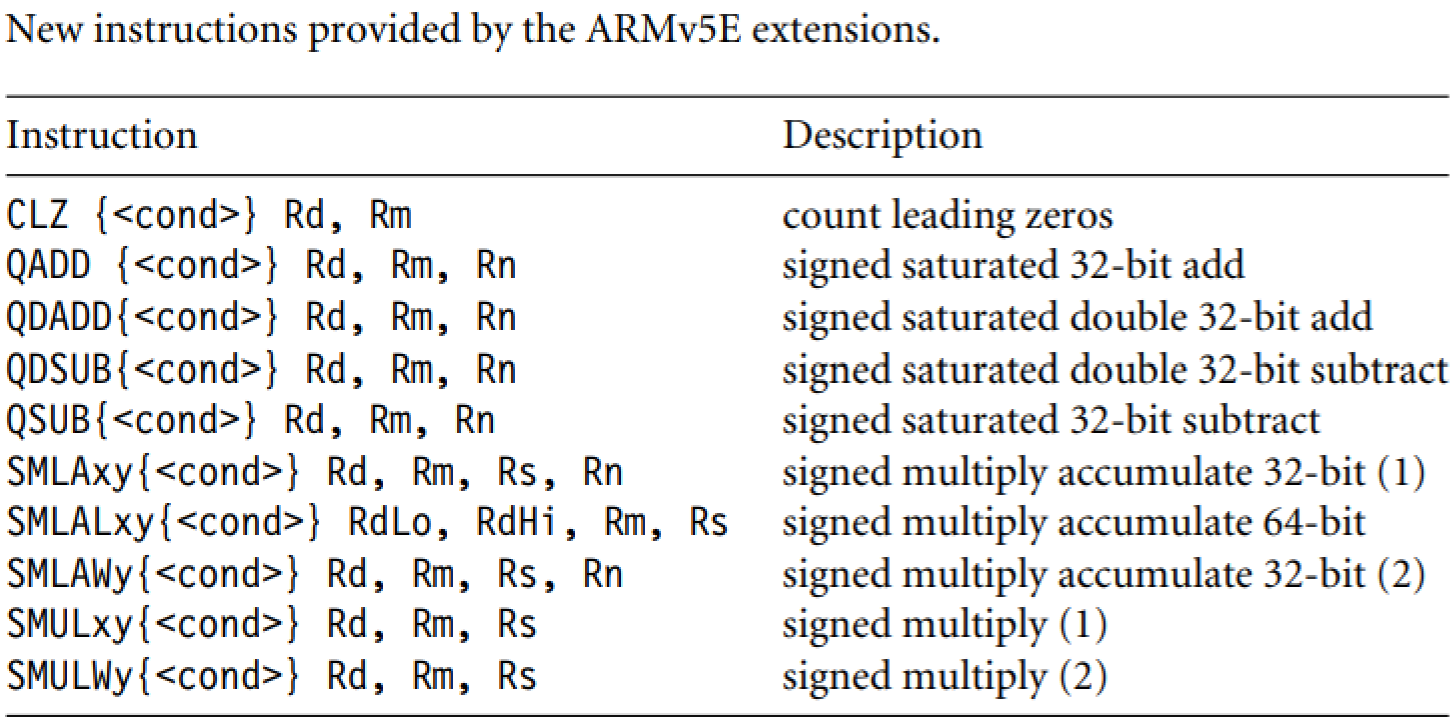
```
        LDR     r0, [pc, #constant_number-8-{PC}]
        :
constant_number
        DCD     0xff00ffff
```

Agenda

- ARM Instruction Set Architecture Overview
- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions

ARMv5E Extensions

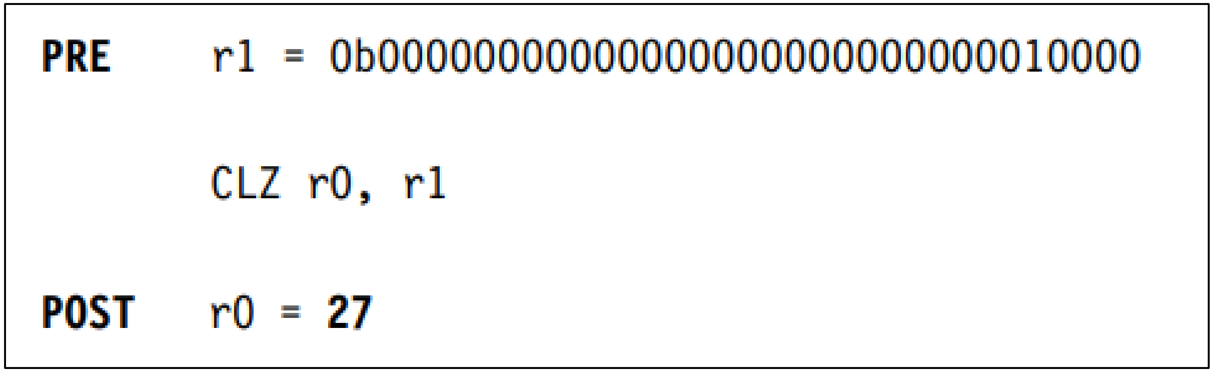
- The ARMv5E extensions provide many new instructions.
  - Please refer to the textbook for more information on this part.



New instructions provided by the ARMv5E extensions.

Instruction	Description
`CLZ {<cond>} Rd, Rm`	count leading zeros
`QADD {<cond>} Rd, Rm, Rn`	signed saturated 32-bit add
`QDADD{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit add
`QDSUB{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit subtract
`QSUB{<cond>} Rd, Rm, Rn`	signed saturated 32-bit subtract
`SMLAxy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (1)
`SMLALxy{<cond>} RdLo, RdHi, Rm, Rs`	signed multiply accumulate 64-bit
`SMLAWy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (2)
`SMULxy{<cond>} Rd, Rm, Rs`	signed multiply (1)
`SMULWy{<cond>} Rd, Rm, Rs`	signed multiply (2)

Count Leading Zeros Instructions

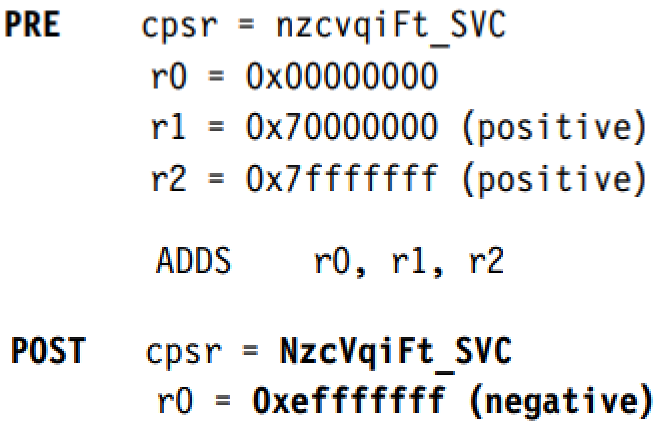


```
PRE     r1 = 0b00000000000000000000000000010000

        CLZ r0, r1

POST    r0 = 27
```

- Count the first bit set to 1 has 27 zeros preceding it

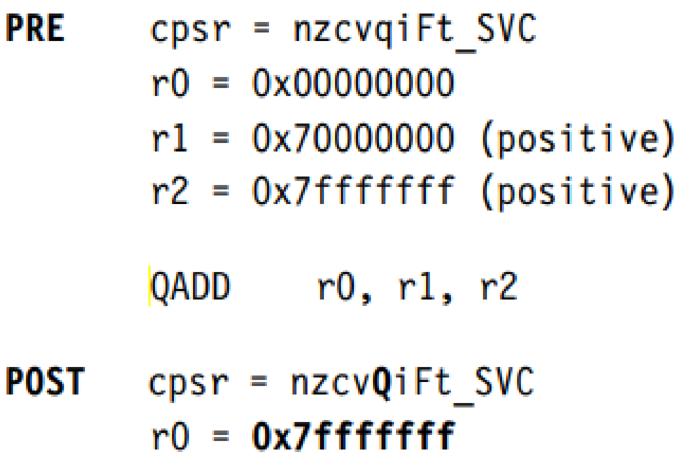
Saturated Arithmetic

- Normal ARM arithmetic instructions wrap around when you overflow an integer value
  - For example, 0x7fffffff + 1 = -0x80000000
- In ARMv5E Saturated instruction, overflow results remain maximum value of 0x7fffffff



```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        ADDS    r0, r1, r2

POST    cpsr = NzcVqiFt_SVC
        r0 = 0xefffffff (negative)
```

Overflow with ADD

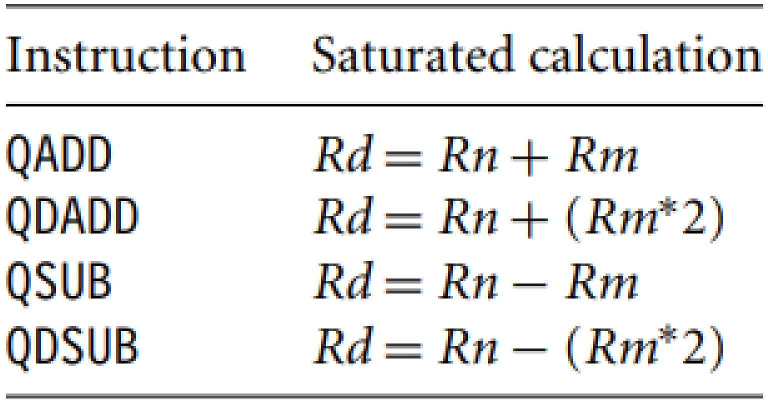


```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        QADD    r0, r1, r2

POST    cpsr = nzcvQiFt_SVC
        r0 = 0x7fffffff
```

Saturated (No Overflow) with QADD

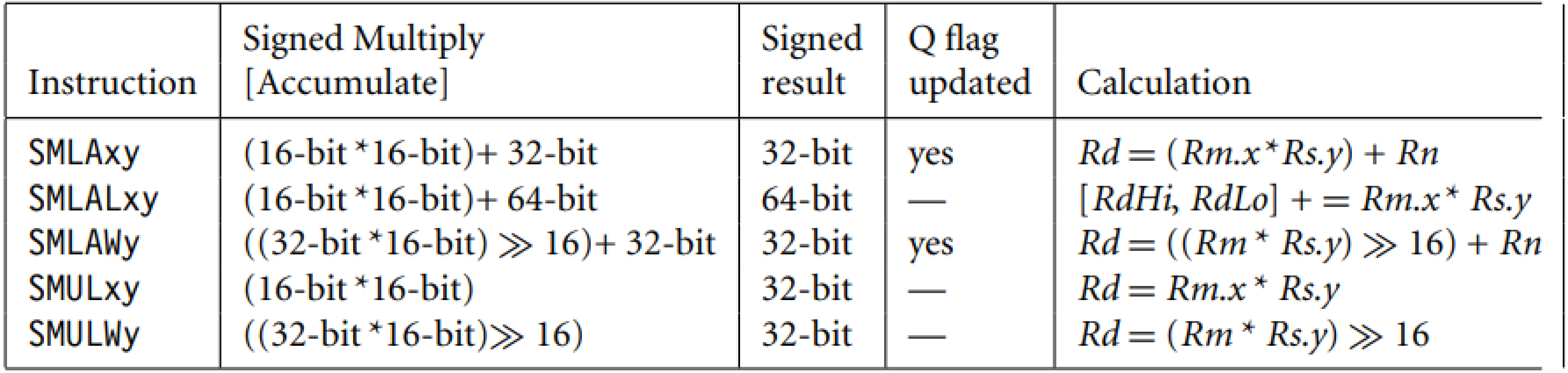
More Saturation Instructions

- All saturation instructions of ARMv4E



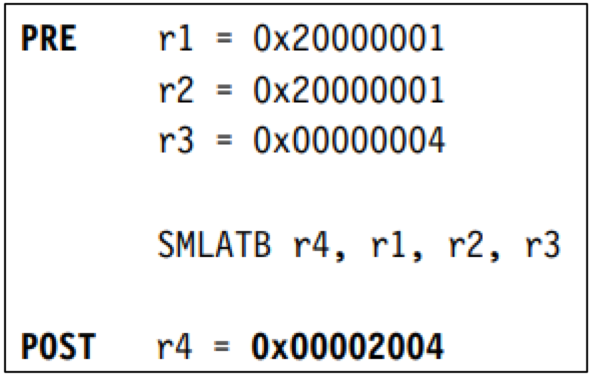
Instruction	Saturated calculation
QADD	$Rd = Rn + Rm$
QDADD	$Rd = Rn + (Rm{*}2)$
QSUB	$Rd = Rn - Rm$
QDSUB	$Rd = Rn - (Rm{*}2)$

Signed Multiply Instructions



Instruction	Signed Multiply [Accumulate]	Signed result	Q flag updated	Calculation
SMLAxy	(16-bit * 16-bit)+ 32-bit	32-bit	yes	$Rd = (Rm.x * Rs.y) + Rn$
SMLALxy	(16-bit * 16-bit)+ 64-bit	64-bit	—	$[RdHi, RdLo] += Rm.x * Rs.y$
SMLAWy	((32-bit * 16-bit) ≫ 16)+ 32-bit	32-bit	yes	$Rd = ((Rm * Rs.y) \gg 16) + Rn$
SMULxy	(16-bit * 16-bit)	32-bit	—	$Rd = Rm.x * Rs.y$
SMULWy	((32-bit * 16-bit)≫ 16)	32-bit	—	$Rd = (Rm * Rs.y) \gg 16$

- x, y select 16-bits of a 32-bit register: T for top 16 bit, B for bottom 16-bit



```
PRE     r1 = 0x20000001
        r2 = 0x20000001
        r3 = 0x00000004

        SMLATB r4, r1, r2, r3

POST    r4 = 0x00002004
```

ARM MEMORY MODEL

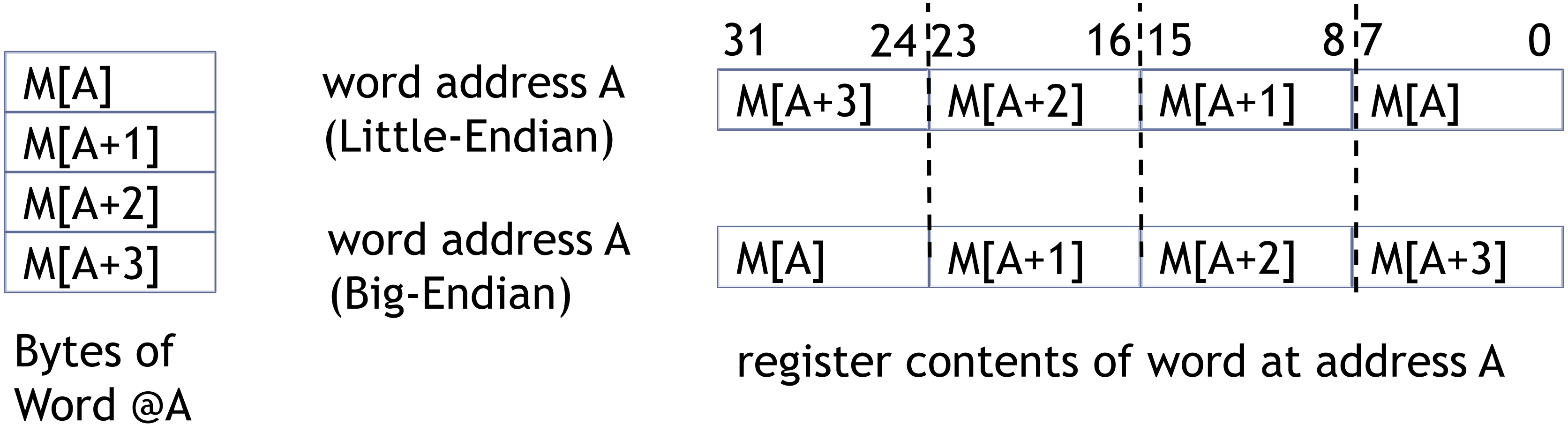
ARM Memory Model

- Alignment
- Endianness
- Memory types and attributes

Memory Alignment Schemes

- Word Align Memory access (if not, alignment fault)
  - LDREX, STREX
  - LDRD, LDMIA, LDMDB, POP, LDC, VLDR, VLDM, VPOP
  - STRD, STMIA, STMDB, PUSH, DTC, VSTR, VSTM, VPUSH
- Halfword Align Memory access (if not, alignment fault)
  - LDREXH, STREXH
- Unaligned Memory addressing supported (only if Configuration and Control Register’s TRP bit=0)
  - LDR{S}H{T}, STRH{T}, TBH
    - ✓but if CCR.TRP=1, should be halfword aligned
  - LDR{T}, STR{T}
    - ✓but if CCR.TRP=1, should be Word aligned

Endianness of Memory System



M[A]
M[A+1]
M[A+2]
M[A+3]

Bytes of Word @A

	31 24	23 16	15 8	7 0
word address A (Little-Endian)	M[A+3]	M[A+2]	M[A+1]	M[A]
word address A (Big-Endian)	M[A]	M[A+1]	M[A+2]	M[A+3]

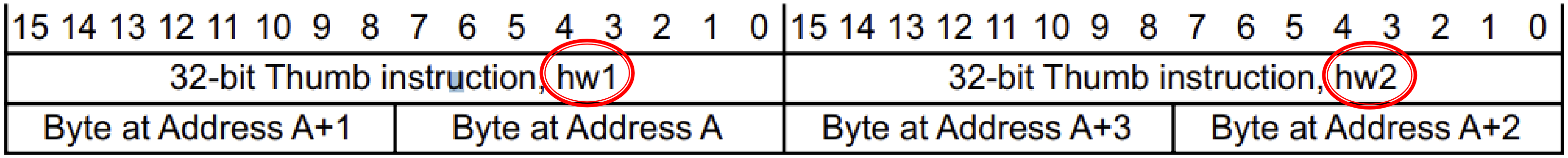
register contents of word at address A

- Address space rule
  - Word at address A consists of bytes at A, A+1, A+2, A+3
  - Half word at address A consists of bytes at A, A+1
- The mapping between bytes and interpreted value
  - little-endian: byte at A is LBS
  - Big-endian: byte at A is MSB

Control of Endianness in ARMv7-M

- Endianness for data access can be selected by the control input, on reset
  - AIRCR.ENDIANNESS bit (Application Interrupt & Reset Conftol Register)
- Instruction Fetches are Always Little-endian
- All Access to System Control Space are Little-endian
- Thumb instruction fetches enforces 16-bit alignment (32-bit instructions treated as two halfwords, hw1, hw2)

  hw1 at lower address



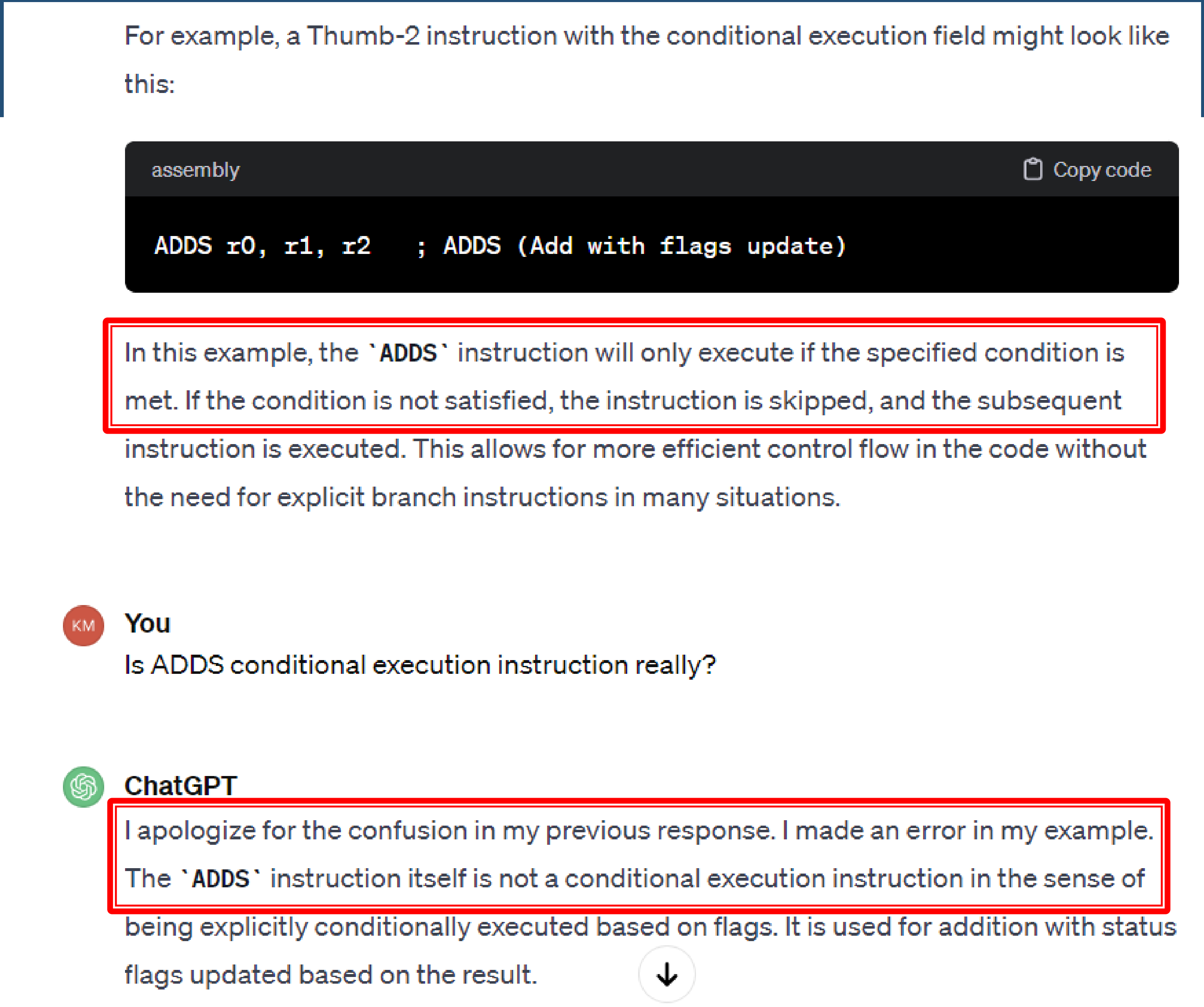
15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
32-bit Thumb instruction, hw1		32-bit Thumb instruction, hw2	
Byte at Address A+1	Byte at Address A	Byte at Address A+3	Byte at Address A+2

bye order of a 32-bit Thumb instruction

Memory type

- 3 types of Memory Regions
    - Normal Memory: Flash, ROM, SRAM, DRAM
    - Device
    - Strongly-Ordered
- I/O Mapped Peripherals : FIFO, Interrupt Controller, Memory Controller
    - Device and Strongly-Ordered Memory
    - Rules
        - ✓Read and Write can have side-effects
        - ✓Accesses must not be repeated
        - ✓Number of memory accesses, Order, Size must be maintained

Chat-GPT를 믿을 수 없는 이유:



For example, a Thumb-2 instruction with the conditional execution field might look like this:

```assembly
ADDS r0, r1, r2    ; ADDS (Add with flags update)
```

In this example, the `ADDS` instruction will only execute if the specified condition is met. If the condition is not satisfied, the instruction is skipped, and the subsequent instruction is executed. This allows for more efficient control flow in the code without the need for explicit branch instructions in many situations.

You

Is ADDS conditional execution instruction really?

ChatGPT

I apologize for the confusion in my previous response. I made an error in my example. The `ADDS` instruction itself is not a conditional execution instruction in the sense of being explicitly conditionally executed based on flags. It is used for addition with status flags updated based on the result.