ARM Instruction Set Architecture (III)

Lecture 7

Yeongpil Cho

Hanynag University

Topics

- ARM Memory Access
- ARM Flow Control

ARM Memory Access

Load-Modify-Store

C statement

```
x = x + 1;
```

```
; Assume the memory address of x is stored in r1


LDR r0, [r1]        ; load value of x from memory
ADD r0, r0, #1      ; x = x + 1
STR r0, [r1]        ; store x into memory
```

Load Instructions

- **LDR rt, [rs]**
  - fetch data from memory into register rt.
  - The memory address is specified in register rs.
  - For Example:

```
; Assume r0 = 0x08200004
; Load a word:
LDR r1, [r0]            ; r1 = Memory.word[0x08200004]
```

Store Instructions

- **STR rt, [rs]**
  - save data in register rt into memory
  - The memory address is specified in a base register rs.
  - For Example:

```
; Assume r0 = 0x08200004
; Store a word
STR r1, [r0]          ; Memory.word[0x08200004] = r1
```

Single register data transfer

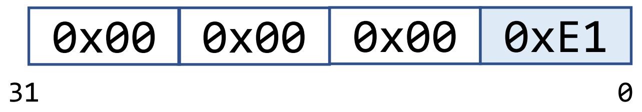
LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword

STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword

Load a Byte, Half-word, Word

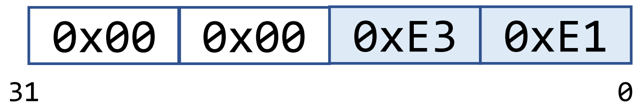
- Load a Byte

```
LDRB r1, [r0]
```



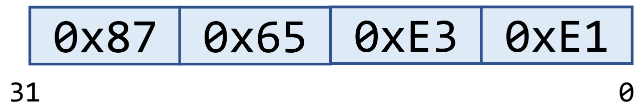
- Load a Halfword

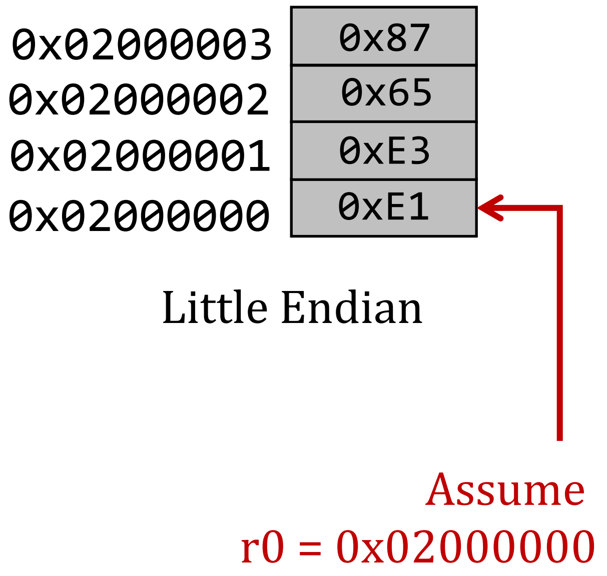
```
LDRH r1, [r0]
```



- Load a Word

```
LDR r1, [r0]
```

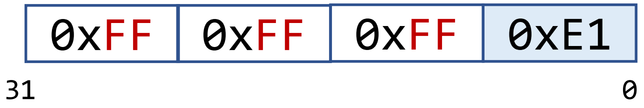




Sign Extension

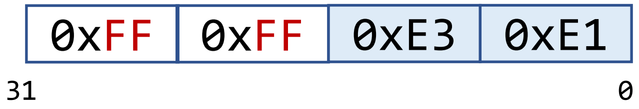
- Load a Signed Byte

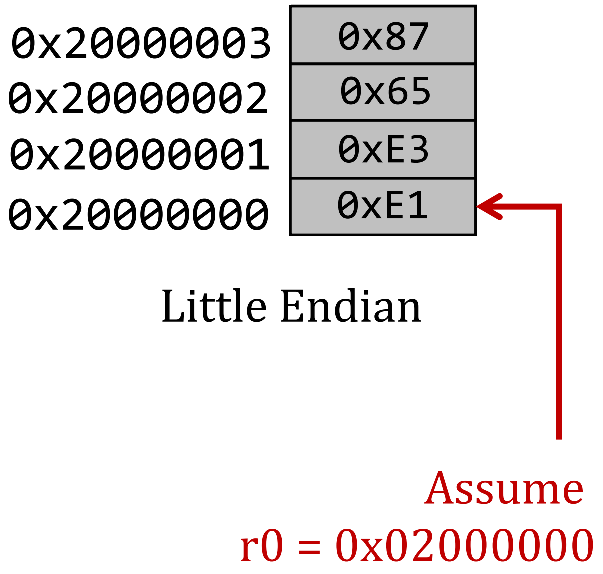
```
LDRSB r1, [r0]
```



- Load a Signed Halfword

```
LDRSH r1, [r0]
```





Address

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - A 12-bit immediate value

    ```
    LDR r0,[r1,#8]
    ```

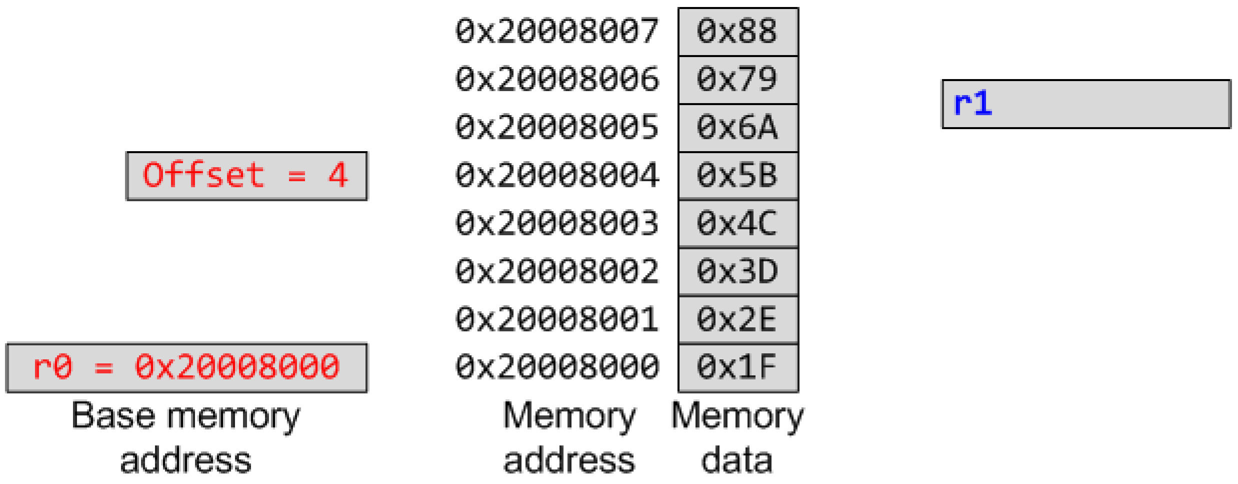
  - A register, optionally shifted by an immediate value

    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```

- For halfword and signed halfword / byte, offset can be:
  - A 8 bit immediate value
  - A register (unshifted).
- Choice of *pre-indexed* or *post-indexed* addressing

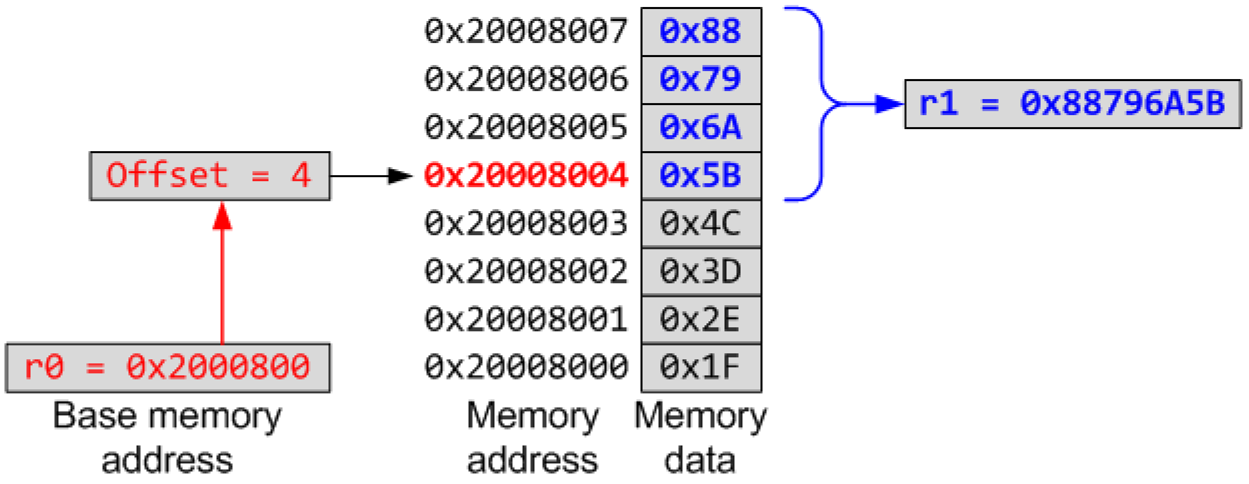
Pre-index

- Pre-Index: LDR r1, [r0, #4]



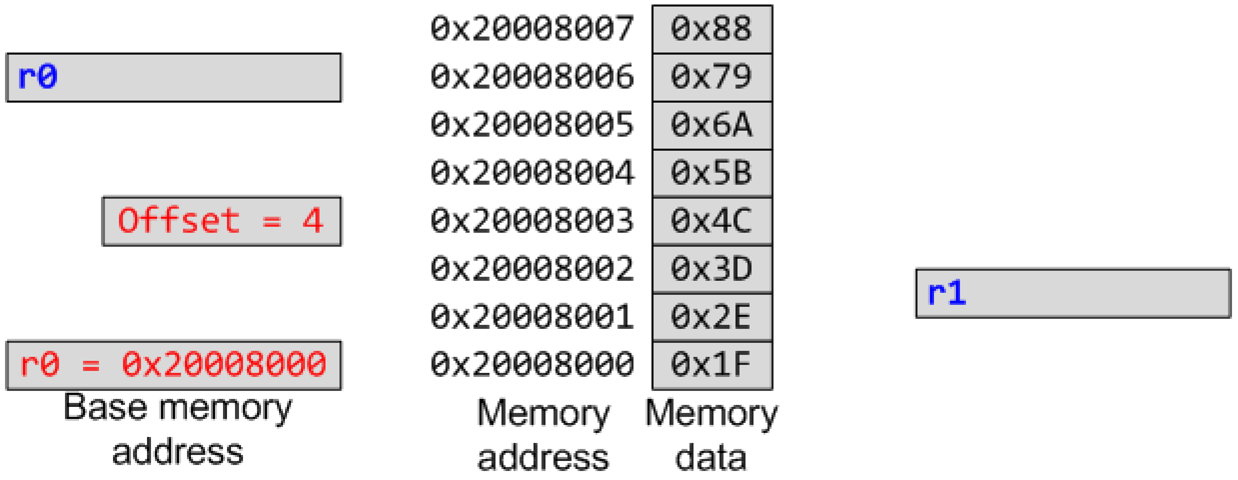
Pre-index

- Pre-Index: LDR r1, [r0, #4]



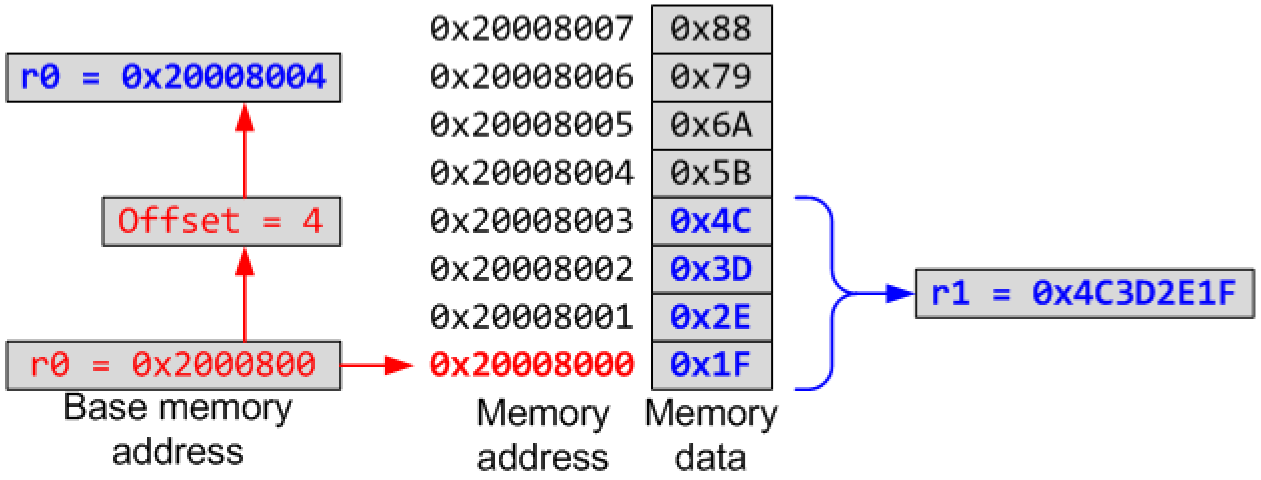
Post-index

- Post-Index: LDR r1, [r0], #4



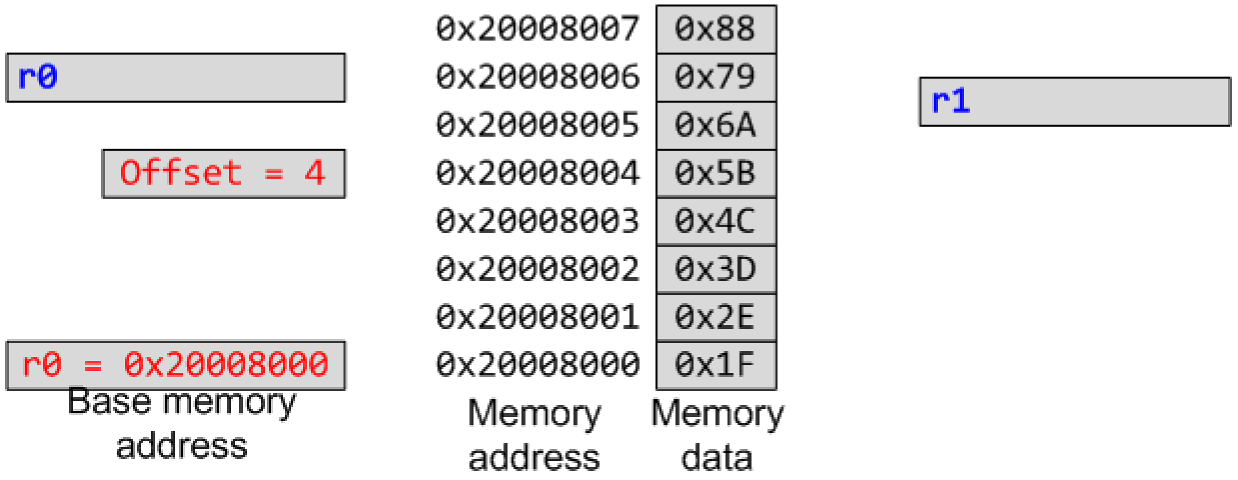
Post-index

- Post-Index: LDR r1, [r0], #4



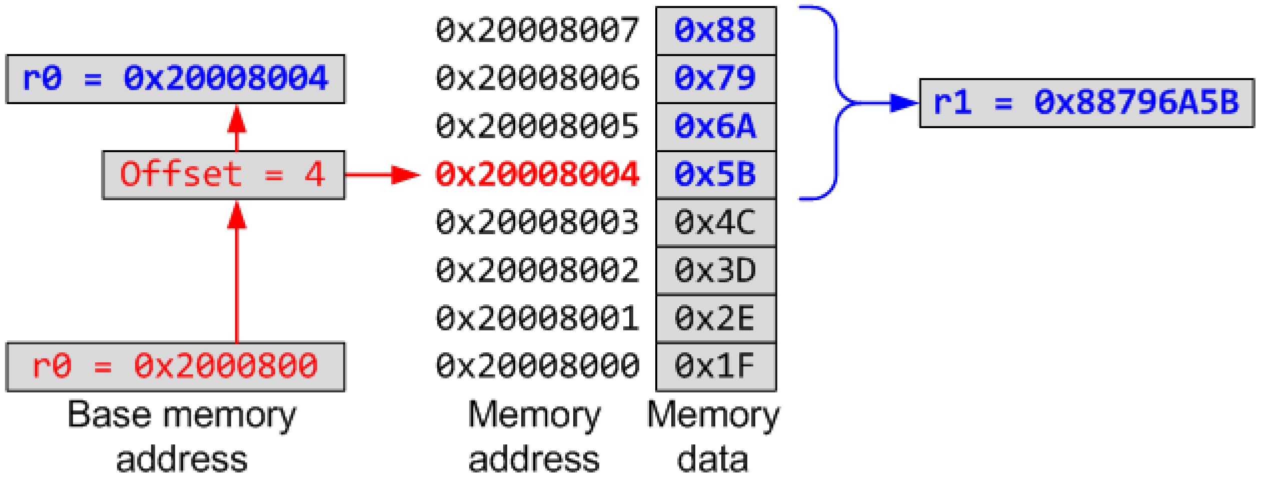
Pre-index with Updates

- Pre-Index with Update: LDR r1, [r0, #4]!



Pre-index with Updates

- Pre-Index with Update: LDR r1, [r0, #4]!



Summary of Pre-index and Post-index

Index Format	**Example**	**Equivalent**
Pre-index	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged

Pre-index wit
h update
	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4]
		r0 ← r0 + 4

Post-index	LDR r1, [r0], #4	r1 ← memory[r0]
		r0 ← r0 + 4

Example

```
LDRH r1, [r0]
; r0 = 0x20008000
```

r1 before load

```
0x12345678
```

r1 after load

```
0x0000CDEF
```

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example

```
LDSB r1, [r0]
; r0 = 0x20008000
```

r1 before load

`0x12345678`

r1 after load

`0xFFFFFFEF`

Memory Address	Memory Data
0x20008003	0x89
0x20008002	0xAB
0x20008001	0xCD
0x20008000	0xEF

Example

```
STR r1, [r0], #4
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

`0x20008000`

r0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Example

```
STR r1, [r0], #4
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

$\boxed{\texttt{0x20008000}}$

r0 after store

$\boxed{\texttt{0x20008004}}$

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

Example

```
STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210
```

r0 before the store

```
0x20008000
```

r0 after the store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Example

```
STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

`0x20008000`

r0 after store

`0x20008000`

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Example

```
STR r1, [r0, #4]!
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Example

```
STR r1, [r0, #4]!
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

`0x20008000`

r0 after store

`0x20008004`

Memory Address	Memory Data
0x20008007	**0x76**
0x20008006	**0x54**
0x20008005	**0x32**
0x20008004	**0x10**
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Load/Store Multiple Instructions

- `STMxx rn{!}, {register_list}`
- `LDMxx rn{!}, {register_list}`

  e.g., stmia r5, {r0, r1, r2} → mem[r5]=r0, mem[r5+4]=r1, mem[r5+8]=r2

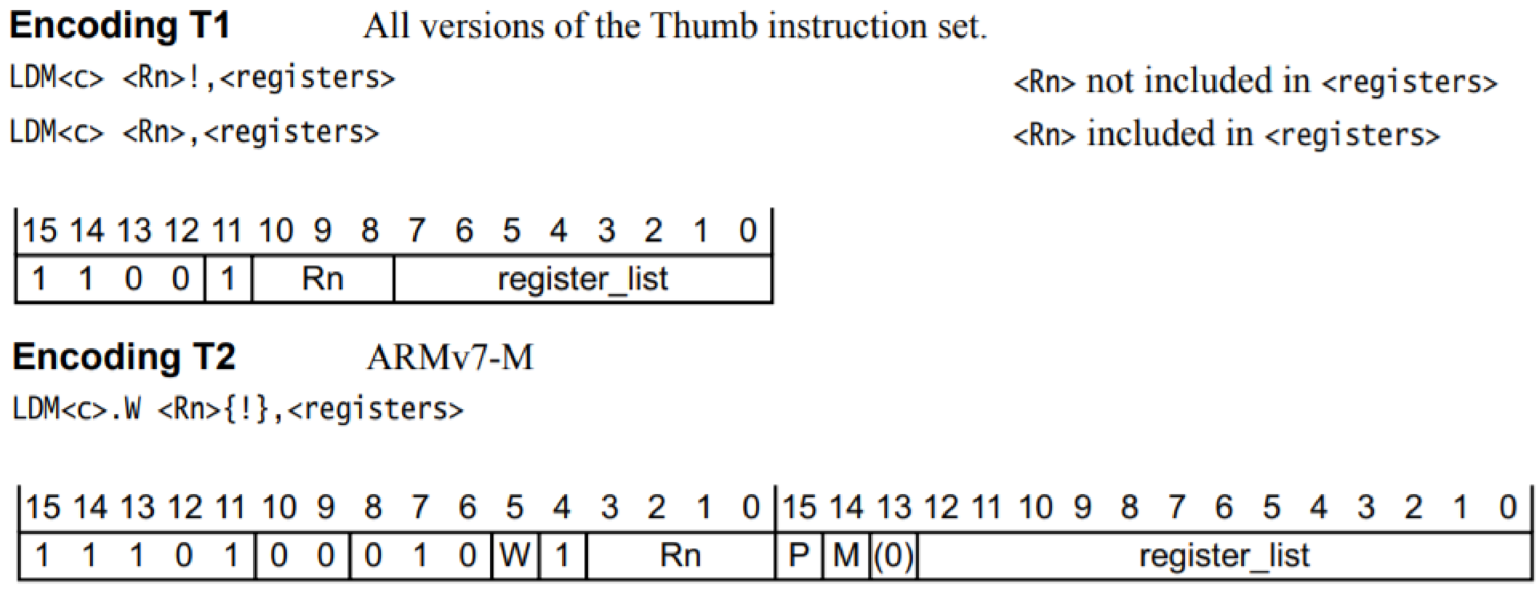
- xx = IA, IB, DA, or DB

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

- IA: address is incremented by 4 after a word is loaded or stored.
- IB: address is incremented by 4 before a word is loaded or stored.
- DA: address is decremented by 4 after a word is loaded or stored.
- DB: address is decremented by 4 before a word is loaded or stored.

Load/Store Multiple Instructions

- The following are synonyms.
  - STM = STMIA (Increment After)
  - LDM = LDMIA (Increment After)
- The order in which registers are listed does not matter
  - For STM/LDM, the lowest-numbered register is stored/loaded at the lowest memory address.
    - stm r5, {r0, r1, r2} == stm r5, {r2, r1, r0}



Encoding T1 All versions of the Thumb instruction set.

`LDM<c> <Rn>!,<registers>` <Rn> not included in <registers>

`LDM<c> <Rn>,<registers>` <Rn> included in <registers>

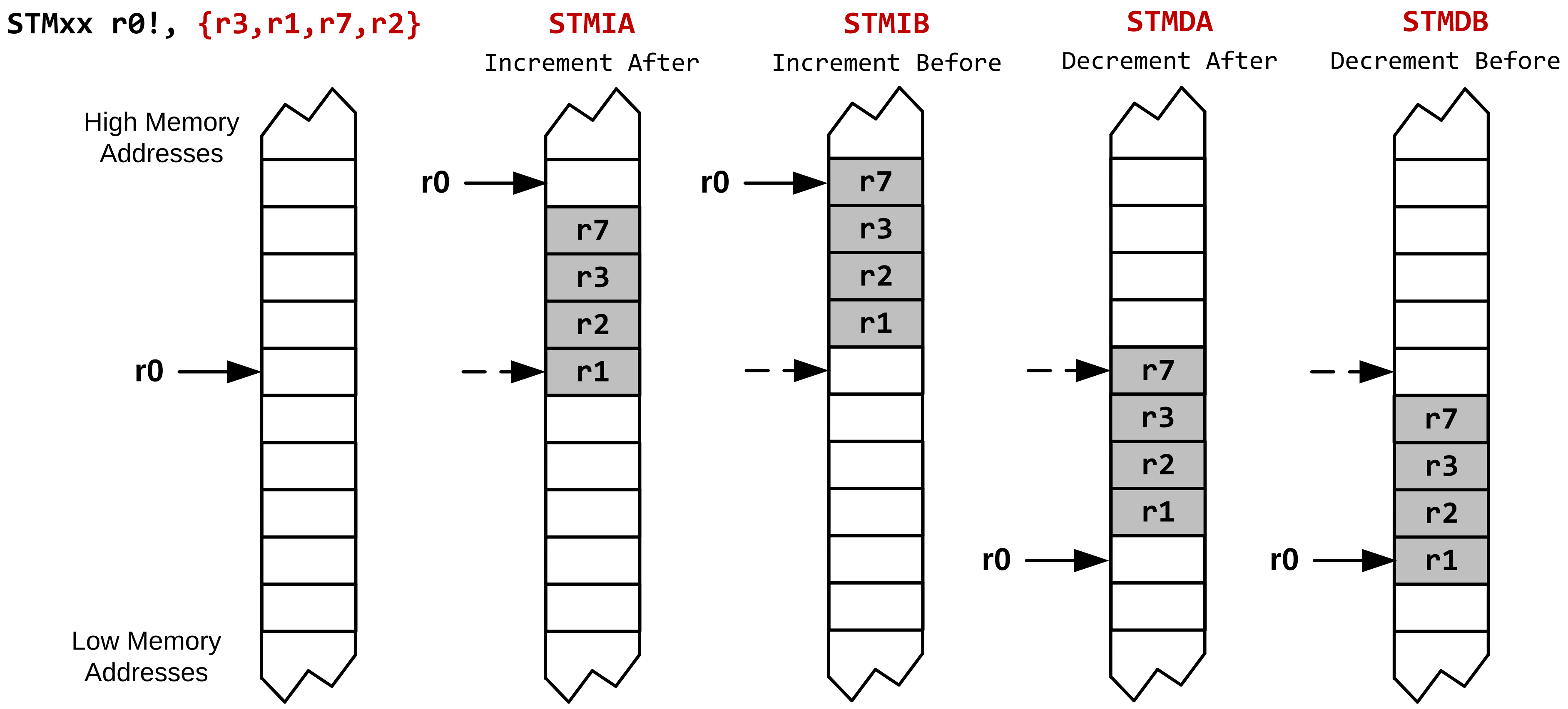
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	Rn			register_list							

Encoding T2 ARMv7-M

`LDM<c>.W <Rn>{!},<registers>`

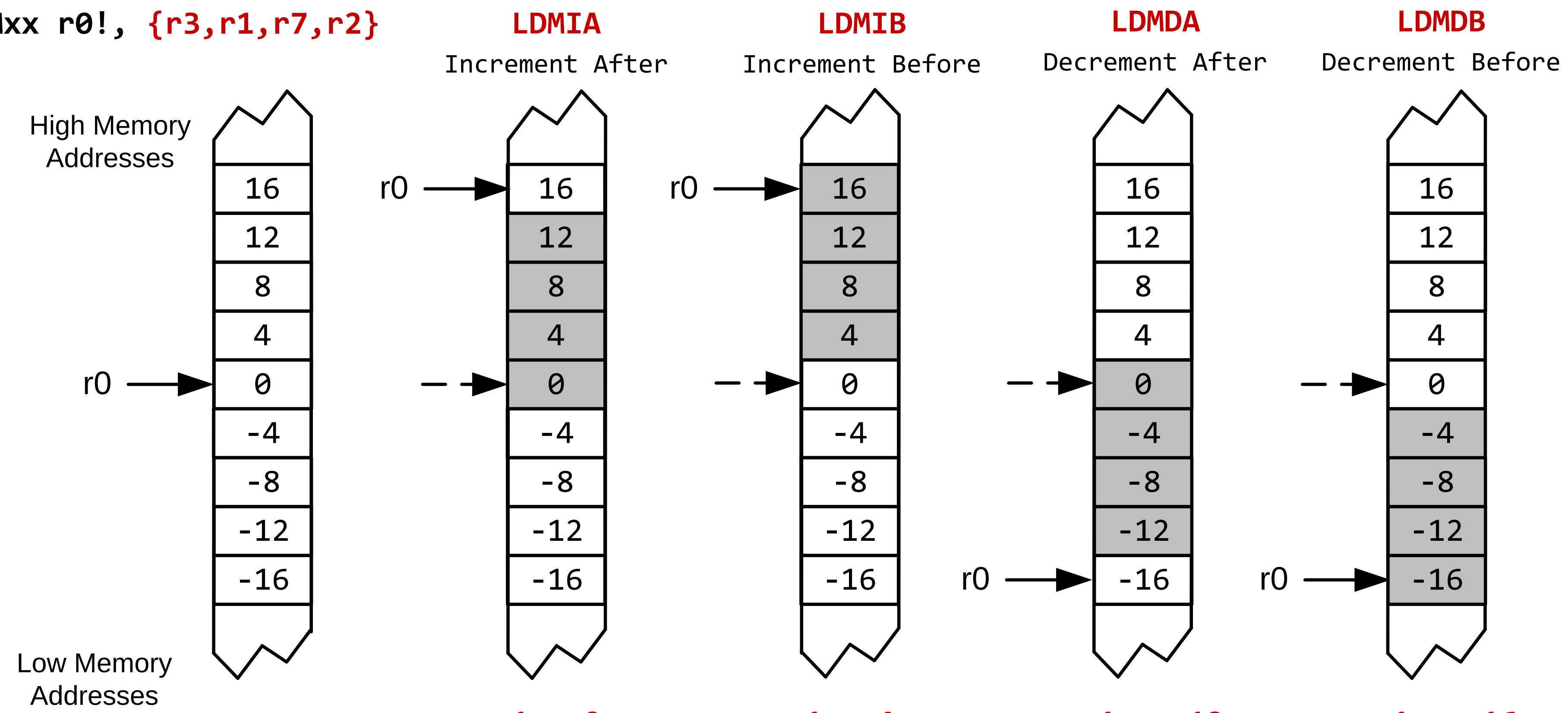
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	1	Rn				P	M	(0)	register_list												

Store Multiple Instructions



Load Multiple Instructions

`LDMxx r0!, {r3,r1,r7,r2}`



LDMIA	LDMIB	LDMDA	LDMDB
r1 = 0	r1 = 4	r1 = -12	r1 = -16
r2 = 4	r2 = 8	r2 = -8	r2 = -12
r3 = 8	r3 = 12	r3 = -4	r3 = -8
r7 = 12	r7 = 16	r7 = -0	r7 = -4

Arm 32-bit load pseudo-op*

- In LDR, Operand cannot be memory address or large constant
  - LDR r3, =0x55555555   ; place 0x55555555 in r3

* Not an actual Arm instruction – translated to Arm ops by the assembler

32-bit constant or symbol
e.g., LDR r3, =foo

  - Produces MOV if immediate constant can be found
  - Otherwise put constant in a "**literal pool**" and use:

```
LDR r3, [PC, #offset]
...
...
...
.word   0x55555555           ;in literal pool following code
```

PC-relative address

ARM Flow Control

Branch Instructions

Instruction	Operands	Brief description	Flags
B	label	Branch	-
BL	label	Branch with Link	-
BLX	Rm	Branch indirect with Link	-
BX	Rm	Branch indirect	-

- *B label*: causes a branch to label.
- *BL label*: instruction copies the address of the next instruction into r14 (lr, the link register), and causes a branch to label.
- *BX Rm*: branch to the address held in Rm
- *BLX Rm*: copies the address of the next instruction into r14 (lr, the link register) and branch to the address held in Rm

Branch With Link

- The "Branch with link (BL)" instruction implements a subroutine call by writing the next instruction's address (return address) into the LR.
- To return from subroutine, simply need to restore the PC from the LR:
  - **`MOV pc, lr`   or   `BX lr`**
  - Again, pipeline has to refill before execution continues.

```
   bl  foo
    ....
foo:
    ...
    bx  lr
```

- The "Branch" instruction does not affect LR.

Condition Codes

- The possible condition codes are listed below:
  - **N**egative, **Z**ero, **C**arry/borrow, o**V**erflow,

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 (Negative)	N=1
PL	PLus (Positive or Zero)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Clear	V=0
HI	Unsigned HIgher	C=1 & Z=0
LS	Unsigned Lower or Same	C=0 or Z=1
GE	Signed Greater or Equal	N=V
LT	Signed Less Than	N!=V
GT	Signed Greater Than	Z=0 & N=V
LE	Signed Less than or Equal	Z=1 or N!=V
AL	ALways	

Note AL is the default and does not need to be specified

Conditional Branch Instructions

	Instruction	Description	Flags tested
Unconditional Branch	B	Branch to label	
Conditional Branch	BEQ	Branch if EQual	Z = 1
	BNE	Branch if Not Equal	Z = 0
	BCS/BHS	Branch if unsigned Higher or Same	C = 1
	BCC/BLO	Branch if unsigned LOwer	C = 0
	BMI	Branch if MInus (Negative)	N = 1
	BPL	Branch if PLus (Positive or Zero)	N = 0
	BVS	Branch if oVerflow Set	V = 1
	BVC	Branch if oVerflow Clear	V = 0
	BHI	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE	Branch if signed Greater or Equal	N = V
	BLT	Branch if signed Less Than	N != V
	BGT	Branch if signed Greater Than	Z = 0 & N = V
	BLE	Branch if signed Less than or Equal	Z = 1 or N = !V

Comparison Instructions

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

- The only effect of the comparisons is to **update the condition flags**.
  - No need to set S bit.
  - No need to specify Rd.
- Operations are:

CMP	operand1 - operand2, but result not written
CMN	operand1 + operand2, but result not written
TST	operand1 & operand2, but result not written
TEQ	operand1 ^ operand2, but result not written

- Examples:

```
CMP     r0, r1
SUBS    r0, r1
```

CMP and CMN

CMP{cond} Rn, Operand2

CMN{cond} Rn, Operand2

- The CMP instruction subtracts the value of Operand2 from the value in Rn.
  - This is the same as a SUBS instruction, except that the result is discarded.
- The CMN instruction adds the value of Operand2 to the value in Rn.
  - This is the same as an ADDS instruction, except that the result is discarded.
- These instructions update the N, Z, C and V flags according to the result.

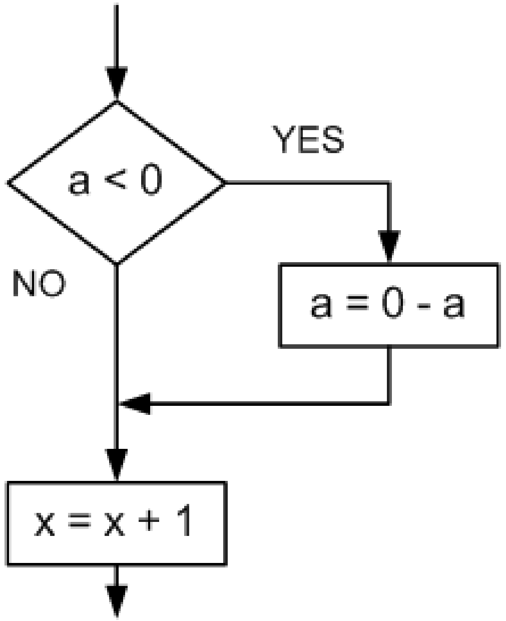
TST and TEQ

```
TST{cond} Rn, Operand2    ; Bitwise AND
TEQ{cond} Rn, Operand2    ; Bitwise Exclusive OR
```

- The TST instruction performs a bitwise AND operation on the value in Rn and the value of Operand2.
  - This is the same as a ANDS instruction, except that the result is discarded.
- The TEQ instruction performs a bitwise Exclusive OR operation on the value in Rn and the value of Operand2.
  - This is the same as a EORS instruction, except that the result is discarded.
- Update the N and Z flags according to the result
- Can update the C flag during the calculation of Operand2
- Do not affect the V flag.

If-then Statement

```
C Program
if (a < 0 ) {
  a = 0 - a;
}
x = x + 1;
```



```
        ; r1 = a, r2 = x
        CMP r1, #0          ; Compare a with 0
        BGE endif           ; Go to endif if a ≥ 0
then:   RSB r1, r1, #0      ; a = 0 - a
endif:  ADD r2, r2, #1      ; x = x + 1
```

Note: RSB = Reverse SuBtract

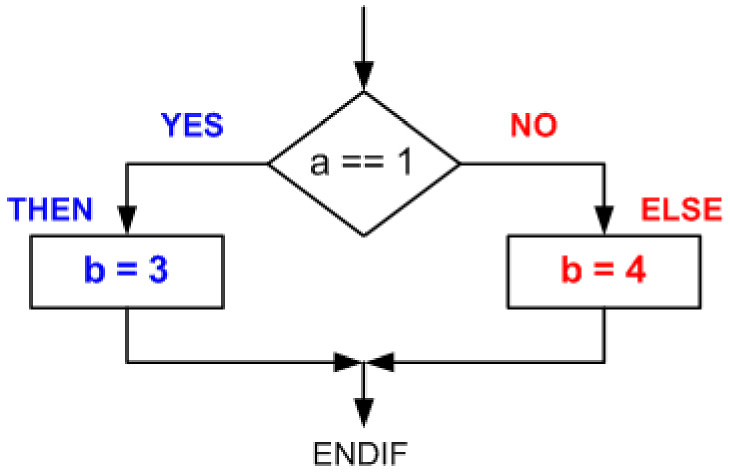
Compound Boolean Expression

C Program	Assembly Program
`// x is a signed integer`	`       ; r0 = x`
`if(x <= 20 || x >= 25){`	`       CMP  r0, #20   ; compare x and 20`
`   a = 1;`	`       BLE  then      ; go to then if x ≤ 20`
`}`	`       CMP  r0, #25   ; compare x and 25`
	`       BLT  endif     ; go to endif if x < 25`
	`then:  MOV  r1, #1    ; a = 1`
	`endif:`

If-then-else

C Program

```
if (a == 1)
    b = 3;
else
    b = 4;
```

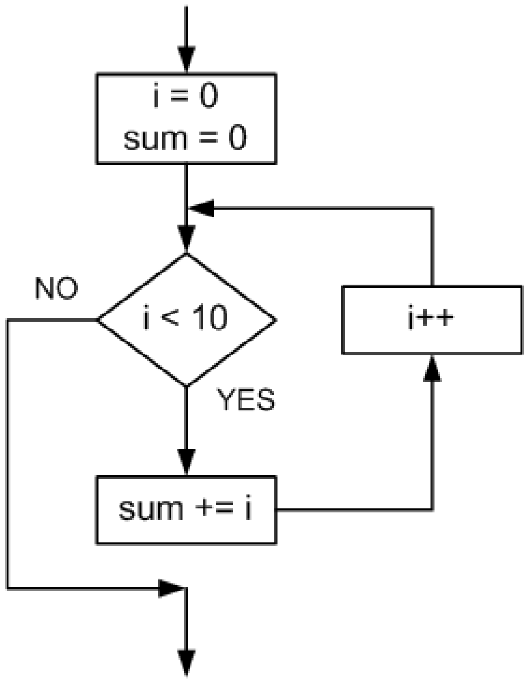


```
         ; r1 = a, r2 = b
         CMP r1, #1    ; compare a and 1
         BNE else      ; go to else if a ≠ 1
then:    MOV r2, #3    ; b = 3
         B   endif     ; go to endif
else:    MOV r2, #4    ; b = 4
endif:
```

For Loop

C Program

```
int i;
int sum = 0;
for(i = 0; i < 10; i++){
  sum += i;
}
```



Implementation:

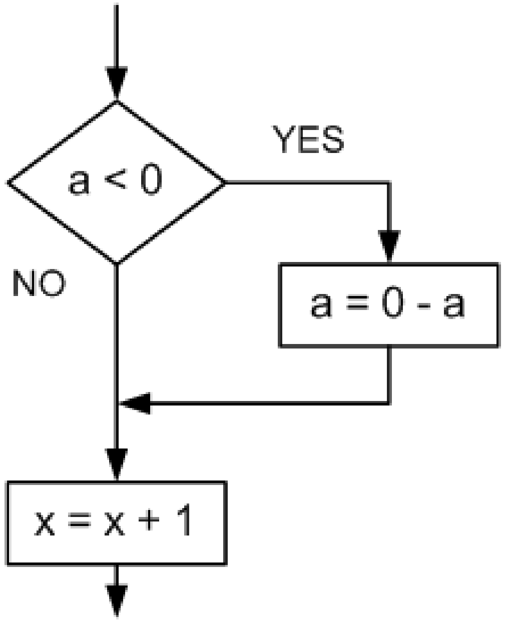
```
           MOV r0, #0   ; i
           MOV r1, #0   ; sum

loop:      CMP r0, #10
           BGE endloop
           ADD r1, r1, r0
           ADD r0, r0, #1
           B   loop
endloop:
```

Conditional Execution

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Example of Conditional Execution

```
C Program
if (a < 0 ) {
  a = 0 - a;
}
x = x + 1;
```



```
; r1 = a, r2 = x
CMP   r1, #0        ; Compare a with 0
RSBLT r1, r1, #0    ; a = 0 - a if a < 0
ADD   r2, r2, #1    ; x = x + 1
```

Example of Conditional Execution

```
a ⟶ r0
y ⟶ r1
```

```
if (a <= 0)
  y = -1;
else
  y = 1;
```

$\Rightarrow$

```
CMP   r0, #0
MOVLE r1, #-1
MOVGT r1, #1
```

LE: Signed Less than or Equal
GT: Signed Greater Than

Compound Boolean Expression

C Program	Assembly Program

`// x is a signed integer`
`if(x <= 20 || x >= 25){`
`   a = 1;`
`}`
	`       ; r0 = x, r1 = a`
	`       CMP    r0, #20  ; compare x and 20`
	`       MOVLE  r1, #1   ; a=1 if less or equal`
	`       CMP    r0, #25  ; CMP if greater than`
	`       MOVGE  r1, #1   ; a=1 if greater or equal`
	`endif:`

Compound Boolean Expression

$a \longrightarrow r0$
$y \longrightarrow r1$

```
if (a==1 || a==7 || a==11)
    y = 1;
else
    y = -1;
```

$\Longrightarrow$

```
CMP   r0, #1
CMPNE r0, #7
CMPNE r0, #11
MOVEQ r1, #1
MOVNE r1, #-1
```

NE: Not Equal
EQ: Equal

Combination

Instruction	Operands	Brief description	Flags
CBZ	Rn, label	Compare and Branch if Zero	-
CBNZ	Rn, label	Compare and Branch if Non Zero	-

- Except that it does not change the condition code flags, CBZ Rn, label is equivalent to:

```
CMP Rn, #0
BEQ label
```

- Except that it does not change the condition code flags, CBNZ Rn, label is equivalent to:

```
CMP  Rn, #0
BNE  label
```

Example

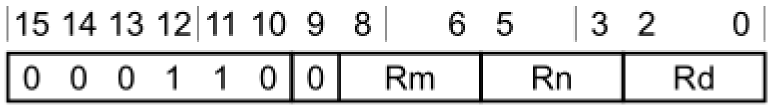
C Program	Assembly Program
`// Find string length`	`; r0 = string memory address`
`char str[] = "hello";`	`; r1 = string length`
`int len = 0;`	`MOV  r1, #0         ; len = 0`
`for( ; ; ) {`	`loop:     LDRB r2, [r0]`
`    if (*str == '\0')`	**CBZ** `r2, endloop`
`        break;`	`ADD  r0, r0, #1     ; str++`
`    str++;`	`ADD  r1, r1, #1     ; len++`
`    len++;`	`B    loop`
`}`	`endloop:`

IT (If-Then) instruction

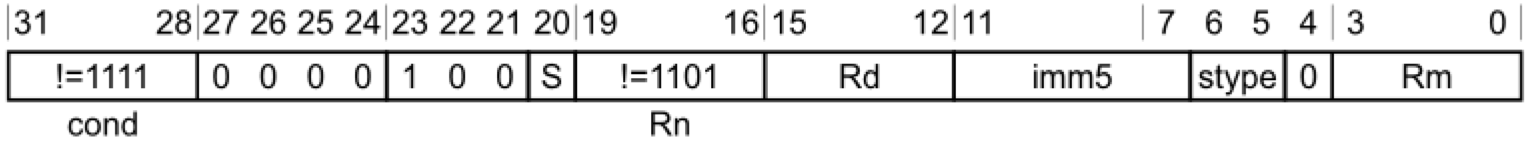
IT{x{y{z}}} {cond}

- Both ARM and Thumb instruction sets support conditional execution.
  - On ARM instruction set, the conditions are embedded in the instructions themselves.
  - But, Thumb instructions cannot embed the conditions due to the short of unused bits in instructions.
- For UAL, conditional execution is implemented using **IT** instruction in Thumb mode.

Thumb's ADD



ARM's ADD



IT (If-Then) instruction

IT{x{y{z}}} {cond}

- where the x, y, and z specify the existence of the optional second, third, and fourth conditional instruction respectively.
  - e.g., ITTTE → If-Then-Then-Then-Else
- x, y, and z are either T (Then) or E (Else)

Examples:

```
IT   EQ
AND  r0,r0,r1   ; 16-bit AND, not ANDS
```

=

```
ANDEQ  r0,r0,r1   ; 16-bit AND, not ANDS
```

```
ITET   NE
AND  r0,r0,r1   ; 16-bit AND, not ANDS
ADD  r2,r2,#1  ; 32-bit ADDS
MOV  r2,r3       ; 16-bit MOV
```

=

```
ANDNE  r0,r0,r1   ; 16-bit AND, not ANDS
ADDEQ  r2,r2,#1  ; 32-bit ADDS
MOVNE  r2,r3       ; 16-bit MOV
```

IT (If-Then) instruction

IT{x{y{z}}} {cond}

- where the x, y, and z specify the existence of the optional second, third, and fourth conditional instruction respectively.
- x, y, and z are either T (Then) or E (Else)
- You do not need to write IT instructions in your code.
- The assembler generates them for you automatically according to the conditions specified.
  - Only, in Thumb mode.
  - ARM mode allows instructions to have conditions on their own.