CET 241: Day 4

Dr. Noori KIM

Agenda

- Memory access, addressing mode
- ARM flow control: Branch instructions

Recap

- What is a C program?
  - Simply just a text file ≠ understandable
    - Contains only human readable characters
    - Can be opened with notepad
  - Computer stores characters via ASCII code mapping
    - Every character is represented by bits

- A heart of computers (the generalization of computing process)

CU

1. Fetch: instructions
   - From memory into CU
2. Decode: split instructions
   - Opcode, Operand..
   - Pass values to ALU

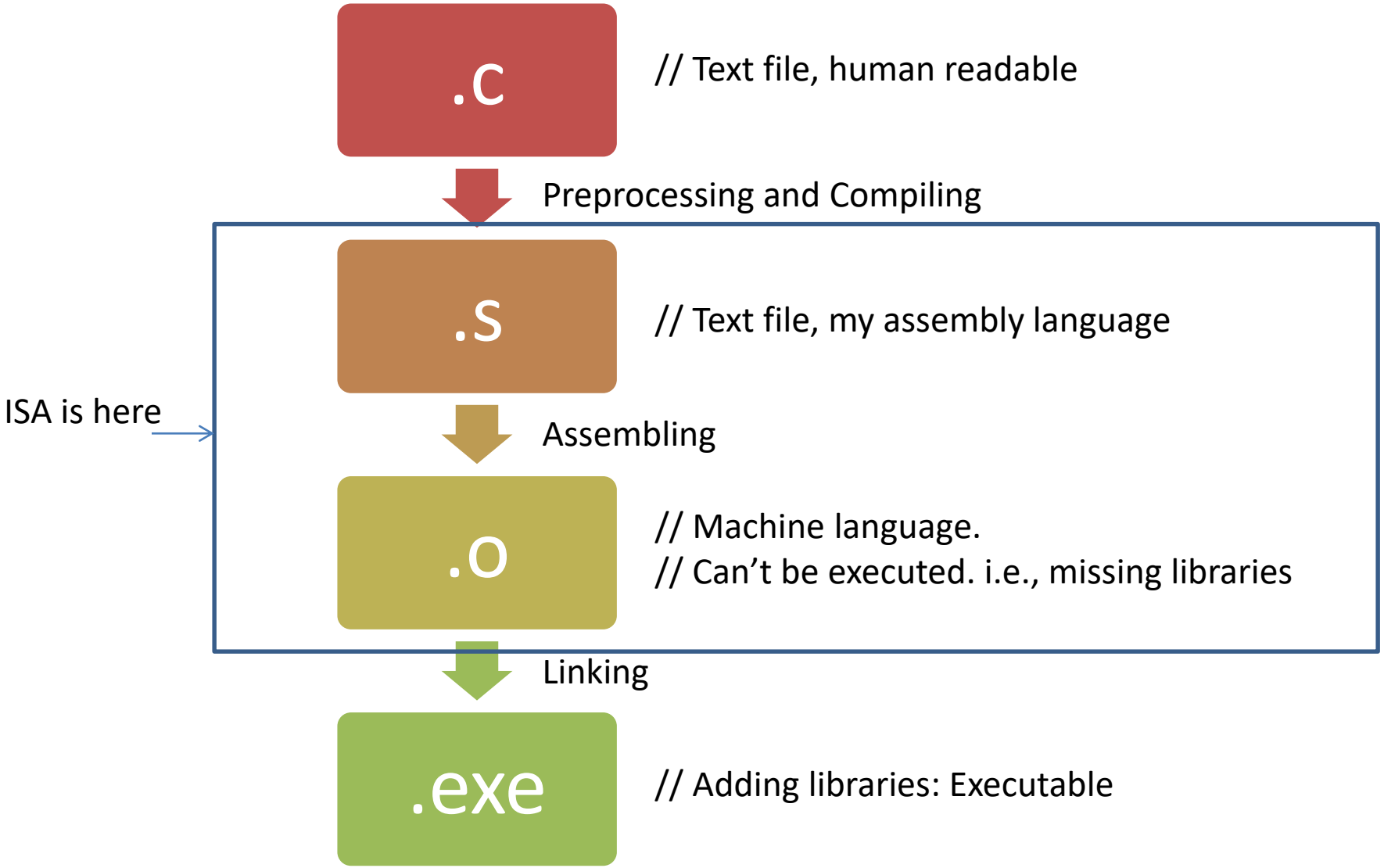
ALU

3. Execute: perform the operation

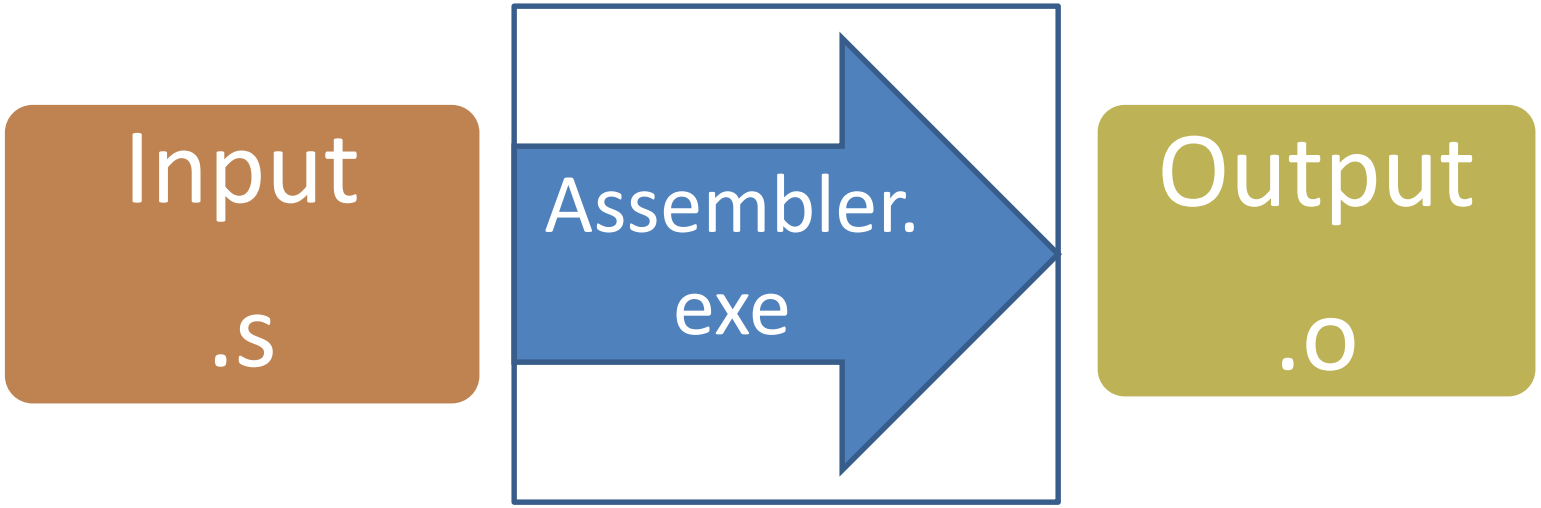
(4. Write back)

Cycle

- In short, the C program execution flow



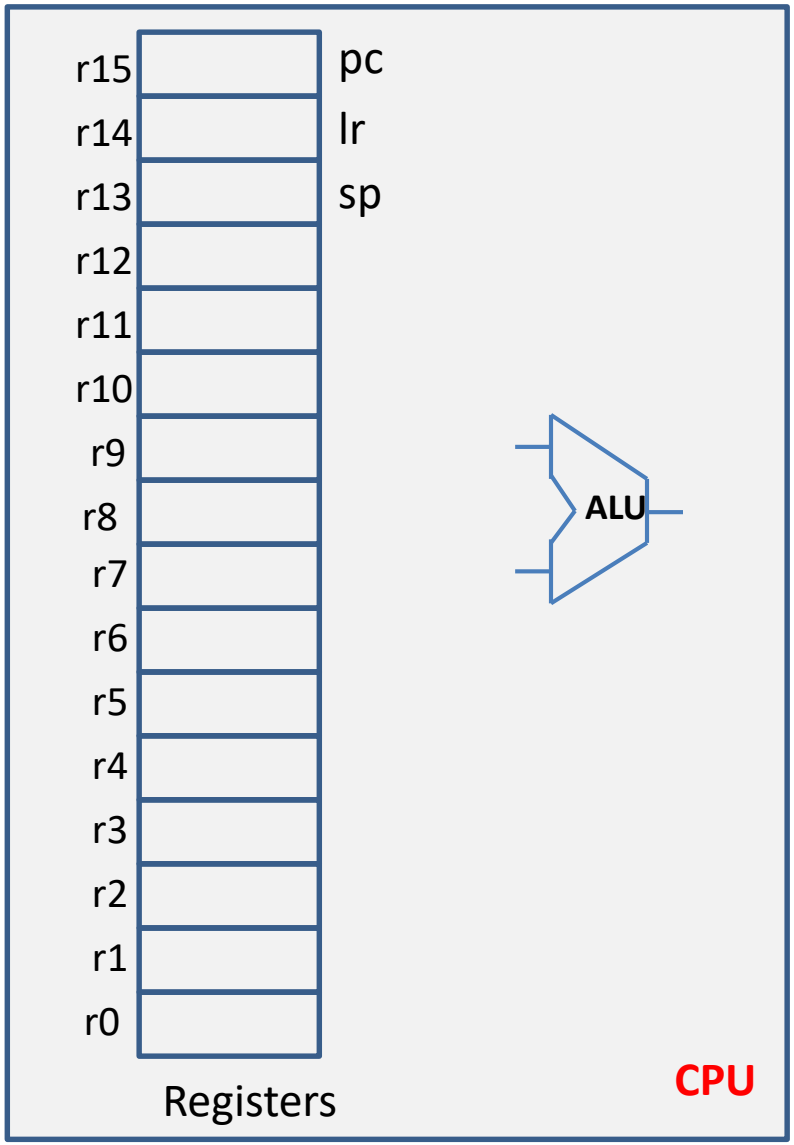
- Each process requires F-D-E process

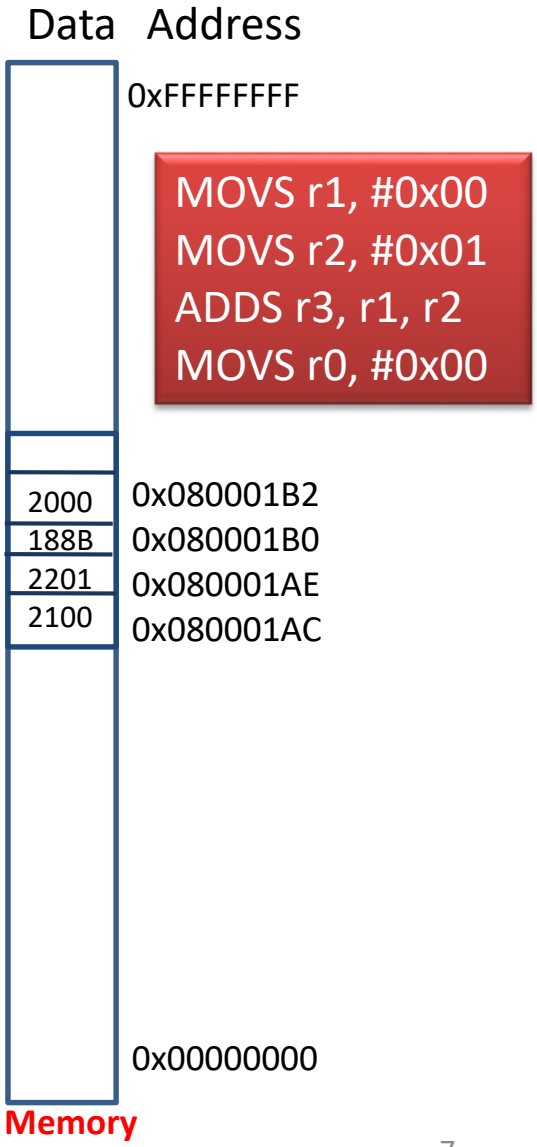


1. Fetch instructions
2. Decode via **ISA**
3. Execute

(4. Write back)

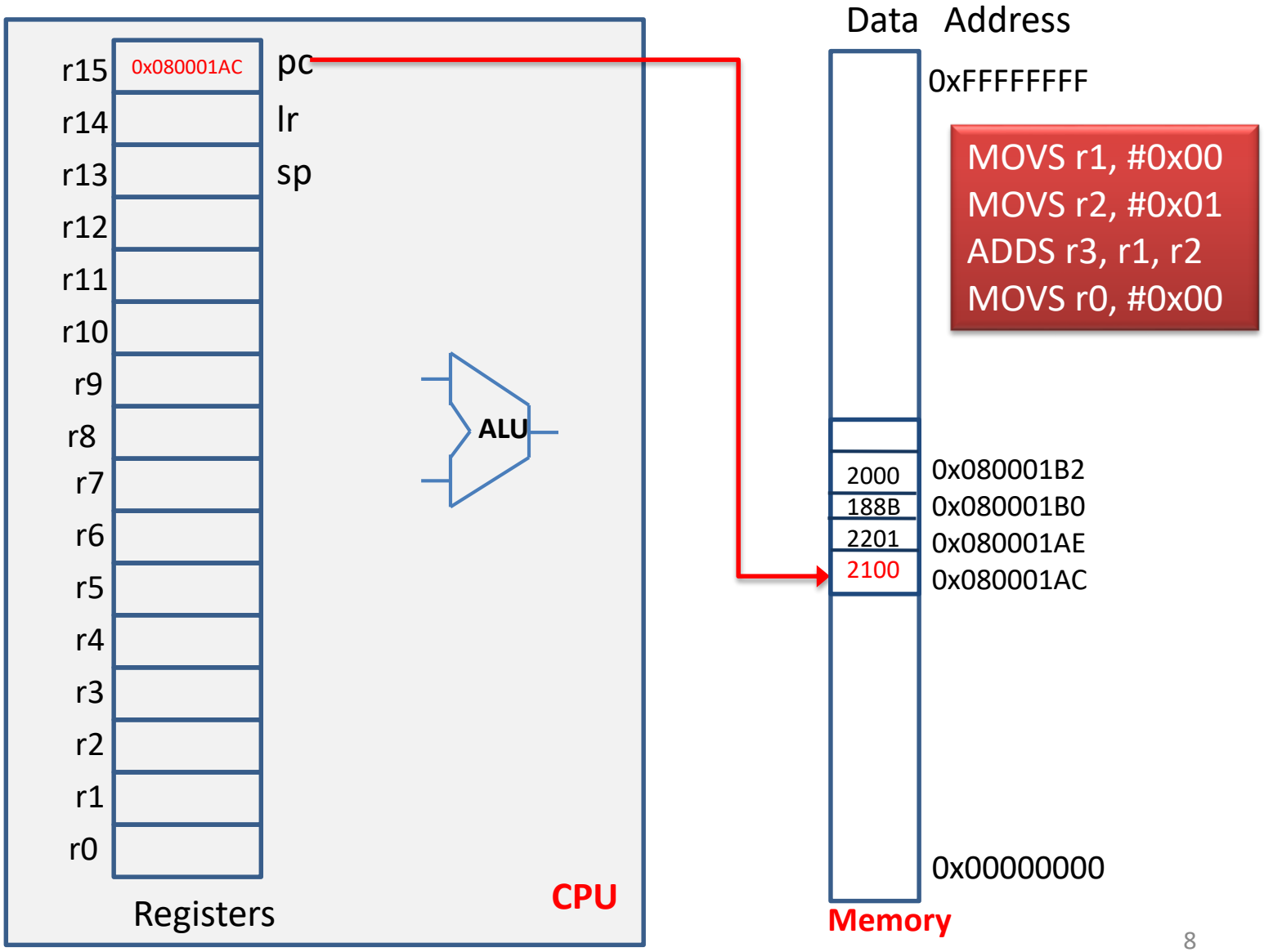
Machine codes are stored in memory





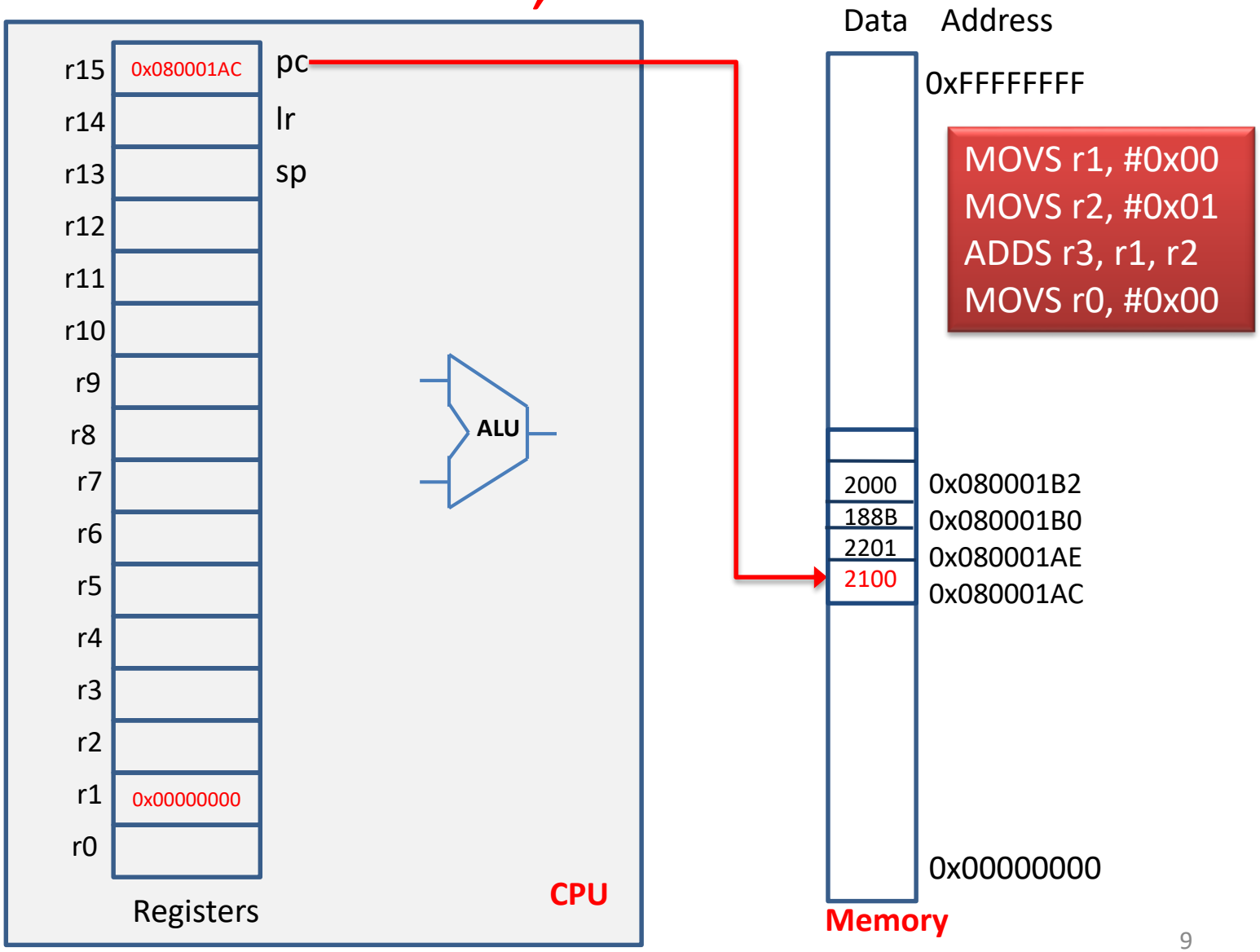
Fetch Instruction: pc = 0x08001AC

Decode Instruction: 2100 = MOVS r1, #0x00

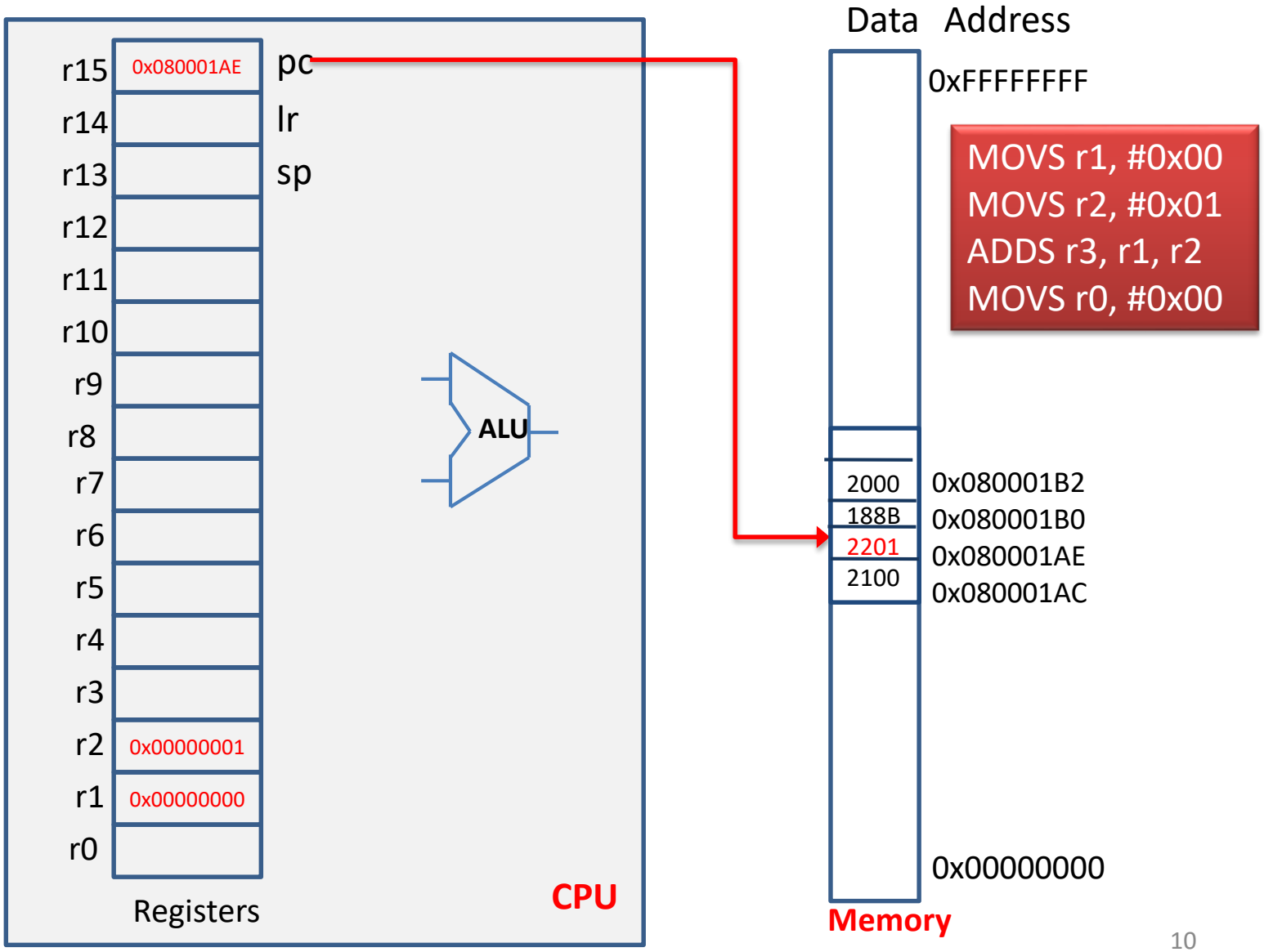


Execute Instruction:

MOVS r1, #0x00

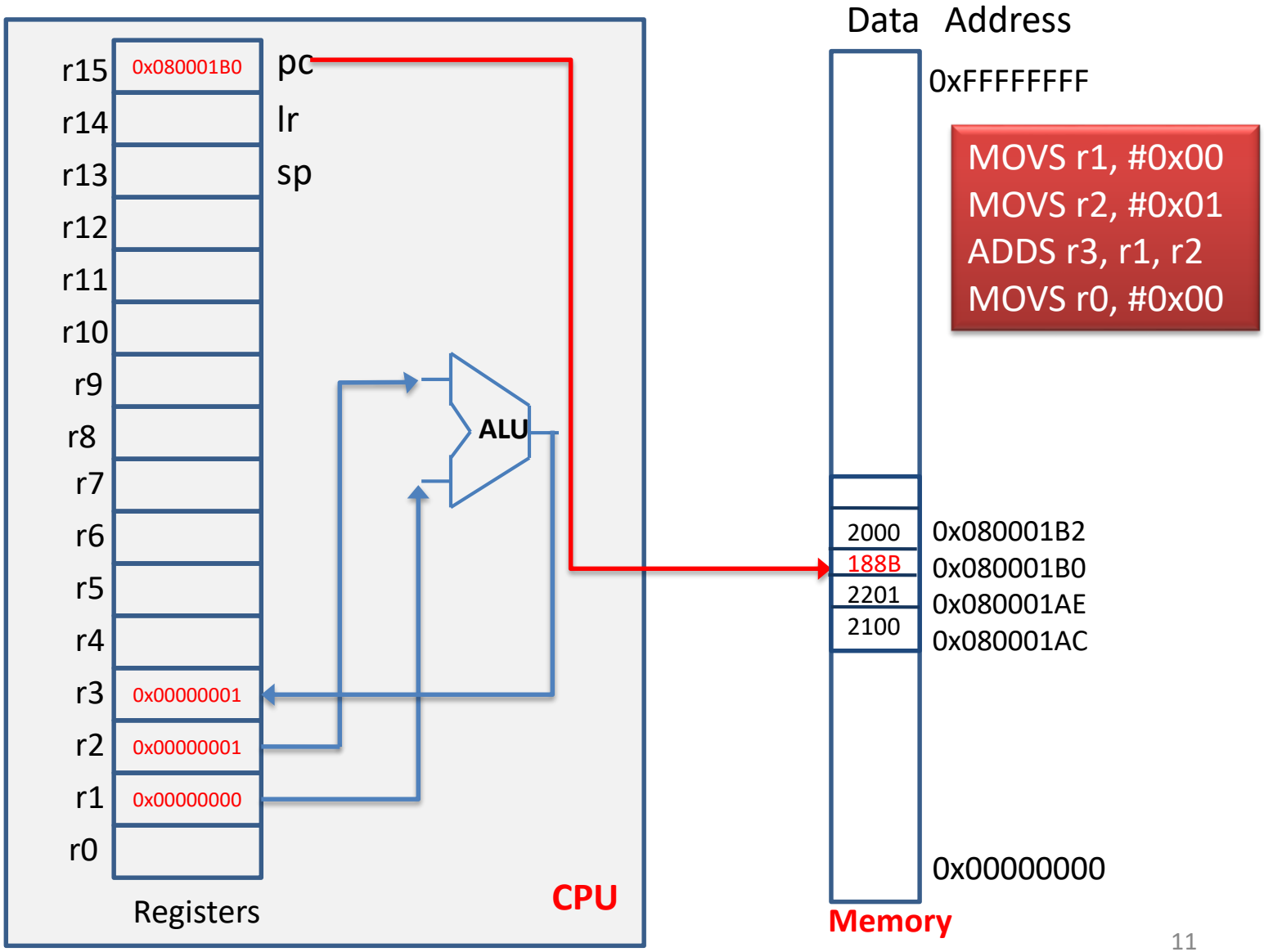


Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



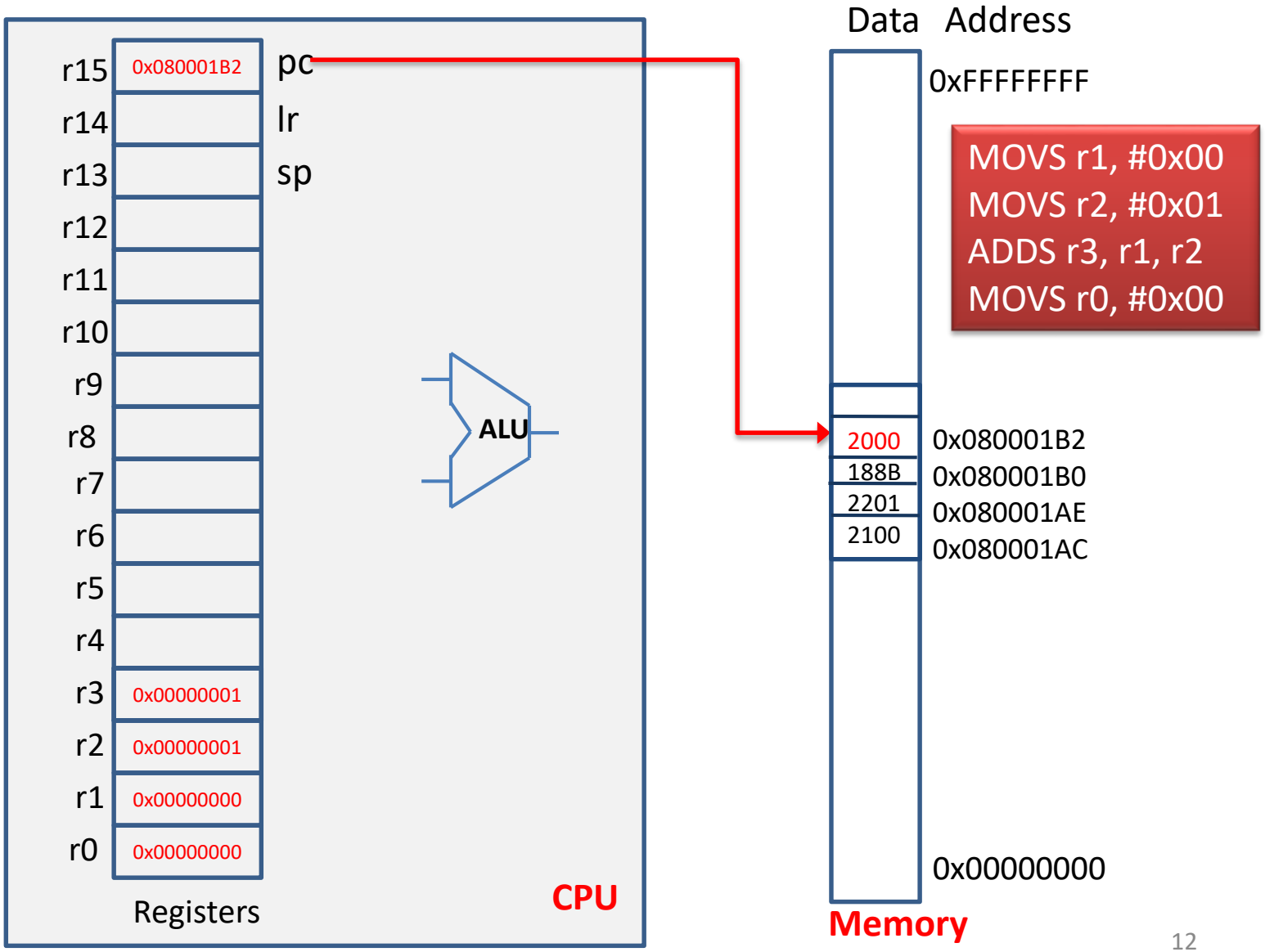
Fetch Next Instruction: pc = pc + 2

Decode & Execute: 188B = ADDS r3, r1, r2



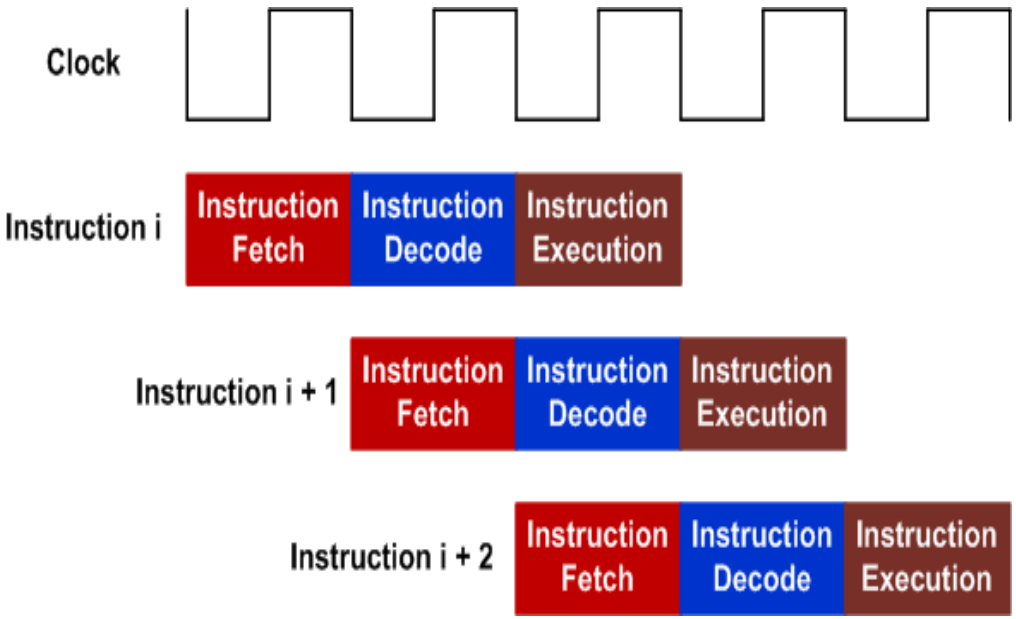
Fetch Next Instruction: pc = pc + 2

Decode & Execute: 2000 = MOVS r0, #0x00



Three-state pipeline: Fetch, Decode, Execution

- **Pipelining** allows hardware resources to be fully utilized
- One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of 32-bit instructions

Instruction Set Architecture: ISA

Machine language	Assembly language
Encoding 0's and 1's: binary	Writing in textual form
Copy the value from "Register 9" into "Register 3."	
1110 0001 1010 0000 0011 0000 0000 1001	MOV R3, R9
Assembler	

- ISA: The design of the machine language encoding

4 ARM Instruction Set

This chapter describes the ARM instruction set.

An important note:

- Although CORTEX-M4 uses Thumb2 ISA, we learn ARM ISA as it is a superset of Thumb2 ISA
- Therefore you may expect slight difference in terms of real execution of your assembly code.

Instruction	Binary format	HEX format
MOV R5, #0x12	$1110\_0011\_1010\_0000\_0101\_0000\_0001\_0010_2$	0xE3A0_5012

1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0

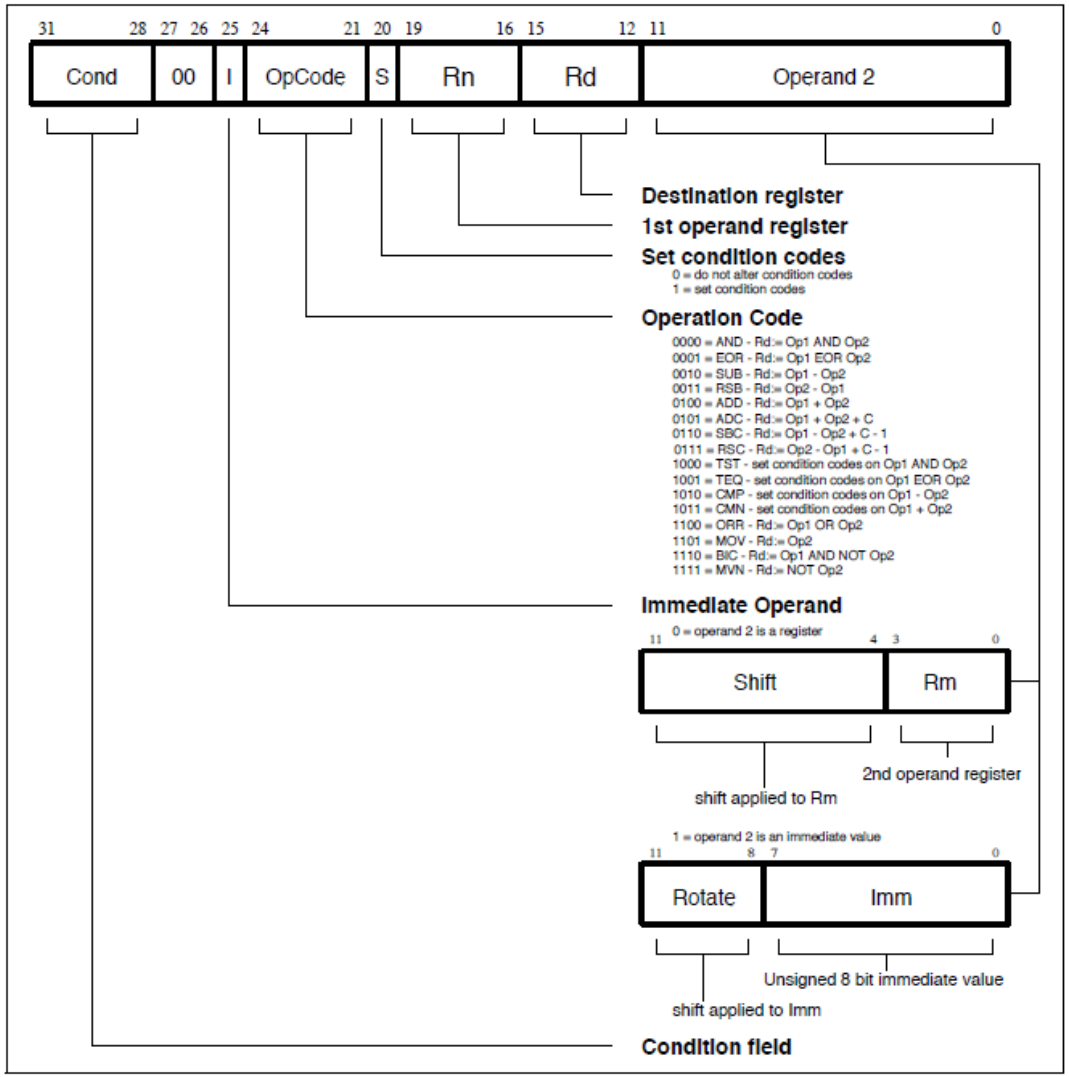


Figure 4-4: Data processing instructions

Instruction	Binary format	HEX format
MOV R5, #0x12	$1110\_0011\_1010\_0000\_0101\_0000\_0001\_0010_2$	0xE3A0_5012

MOV	Move register or constant	Rd : = Op2	4.5

4.2 The Condition Field

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.

There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set.

In practice, fifteen different conditions may be used: these are listed in ***Table 4-2: Condition code summary***. The sixteenth (1111) is reserved, and must not be used.

In the absence of a suffix, the condition field of most instructions is set to "Always" (sufix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Table 4-2: Condition code summary

An arbitrary example of arm asm code to examine .s to .o converting

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

mCoreClock

Registers

Register	Value
Core	
R0	0x10000000
R1	0x00000019
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000

Disassembly

```
0x00000104 4[illegible]      LDR     r0,[pc,[illegible]]   ; @0x00000128
0x00000106 F04F0119  MOV     r1,#0x19
0x0000010A 6001      STR     r1,[r0,#0x00]
0x0000010C F04F010A  MOV     r1,#0x0A
0x00000110 F04F0005  MOV     r0,#0x05
0x00000114 B500      PUSH    {lr}
0x00000116 F000F803  BL.W    0x00000120
0x0000011A F85DEB04  LDR     lr,[sp],#0x04
0x0000011E E7FE      B       0x0000011E
0x00000120 EB000201  ADD     r2,r0,r1
0x00000124 46F7      MOV     pc,lr
0x00000126 BF00      NOP
0x00000128 0000      DCW     0x0000
0x0000012A 1000      DCW     0x1000
0x0000012C 0000      DCW     0x0000
```

Addr	Contents
0x2000.0000	.
.	.
.	.
.	.
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

Check the following concepts from this picture

- ISA (assembly → machine code)
- ARM instruction (32 bits, 4 bytes)
- Thumb instruction (16 bits, 2 bytes)
- Byte addressable memory
- Etc …..

Addressing modes, starting from ISA memory access mechanism recap:

LDR and STR

- ONLY these two special instructions do this job
- Can access memory address

IMPORTANT CONCEPT to be checked

- To assign (represent) a number (constant) in a computer we need two parameters
  - Value (content)
  - Address

Addr	Contents
0x2000.0000	.
.	.
.	.
.	
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

```
Value1 EQU 0x2009.C040
LDR   R1, = Value1     (1)
LDR   R0,  [R1]        (2)
```

LDR R1, = Value1

R1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | . | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Value1
0x2009.C040

LDR R0, [R1]

R1: address on memory map

[R1] : values allocated in the address

Memory map

Addr	Contents
0x2000.0000	.
.	.
.	.
.	.
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	
.	
.	
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

R0 | ? | ? | ? | ? | ? | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | ? | ? | ? | ? | ? | ?

Don't know the contents in Value1, but this instruction will grab data in that address

LDR R0, [R1]

When we store it back

STR R0, [R1]

R0: Updated value

[R1] : values allocated in the address

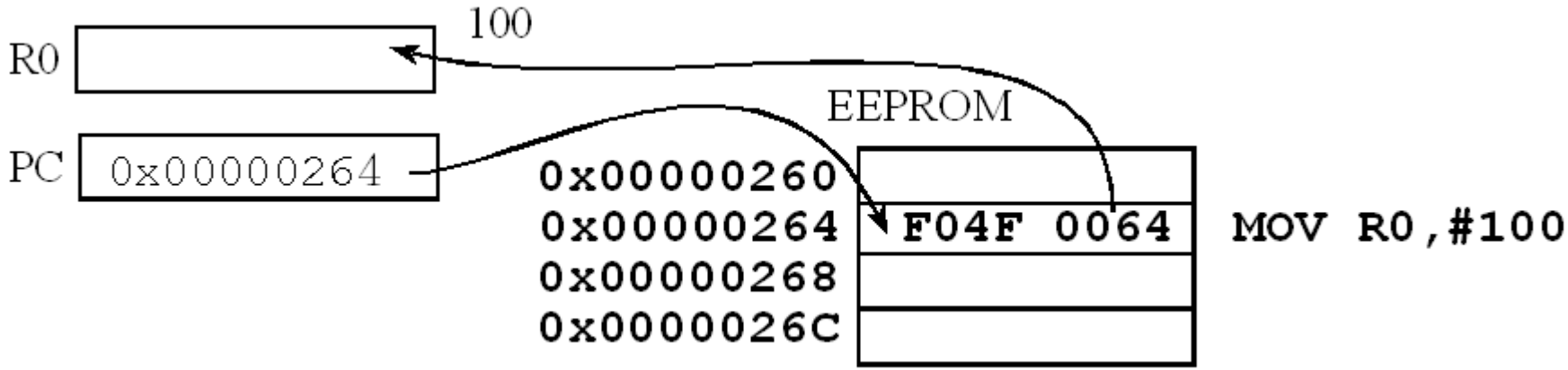
R0

Addr	Contents
0x2000.0000	.
.	.
.	.
.	.
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

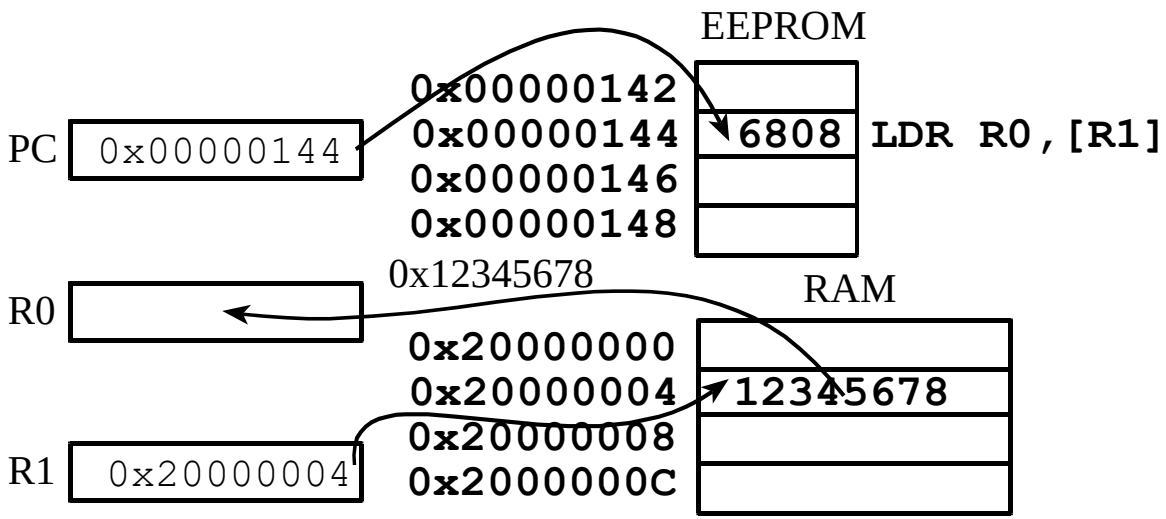
```
LDR   R0,  [ R1 ]

STR   R0,  [ R1 ]
```

Test differences

```
MOV   R0,   #100        ; R0=100, immediate addressing
```




```
LDR   R0,   [R1]        ; R0= value pointed to by R1
```



Load/Store Instructions variation

- General load/store instruction format (Homework for exam)

LDR Rd,[Rn] ;load memory at [Rn] to Rd

STR Rt,[Rn] ;store Rt to memory at [Rn]

LDR Rd,[Rn, #n] ;load memory at [Rn+n] to Rd, no update on Rn

LDR Rd,[Rn, #n]! ; **Rn**=Rn+n then load memory at the new **Rn** to Rd

LDR Rd,[Rn], #n ; load memory at Rn to Rd, then **Rn**=Rn+n

Study this for your exam.

Offset modes

1. an immediate as offset
   - ldr r3, [r1, #4]
2. a register as offset
   - ldr r3, [r1, r2]
3. a scaled register as offset
   - ldr r3, [r1, r2, LSL#2]

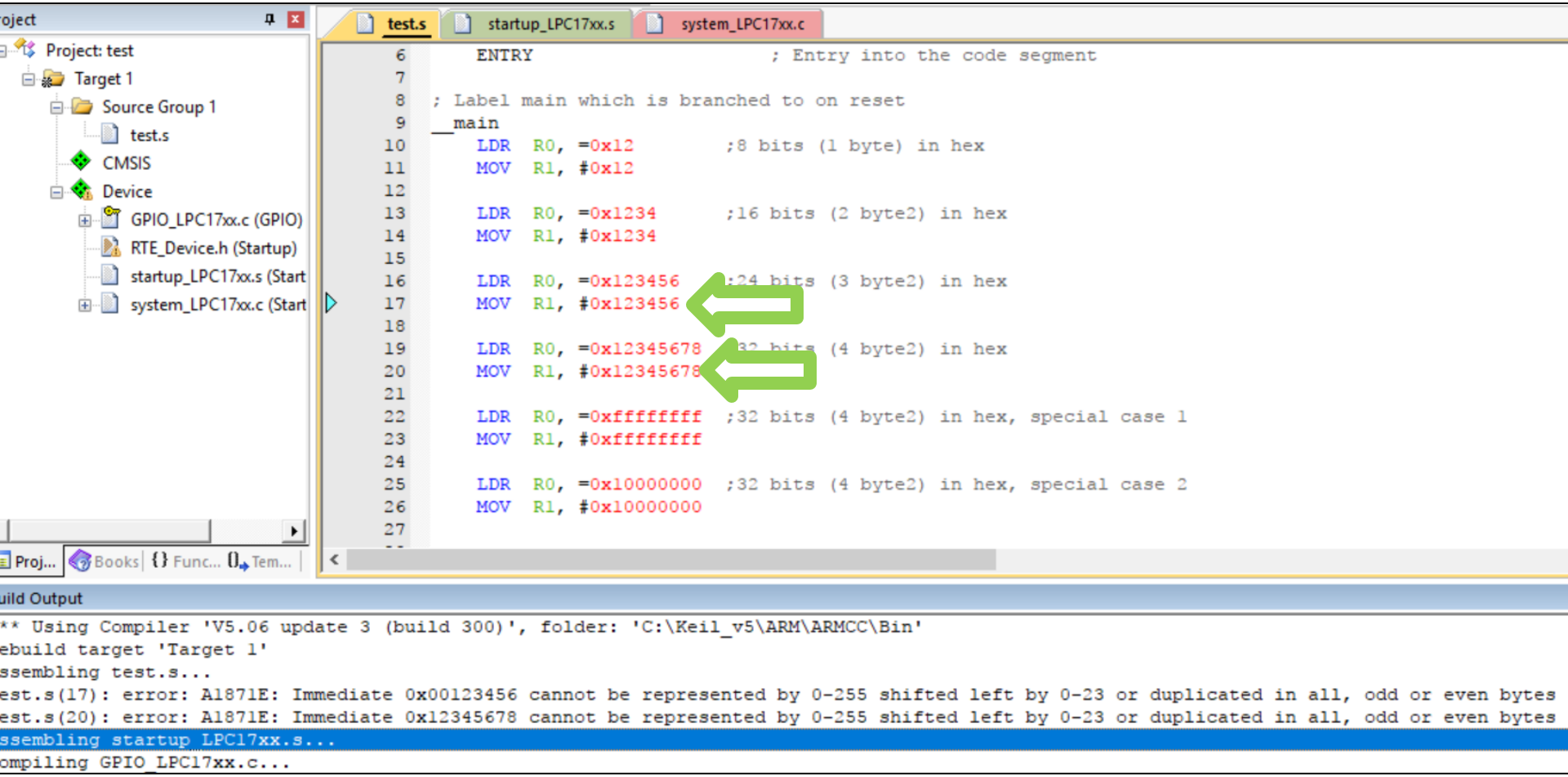
Addressing modes

The different ways of determining the address of the operands

1. A prefix address mode (!)
   - ldr r3, [r1, #4]!
   - ldr r3, [r1, r2]!
   - ldr r3, [r1, r2, LSL#2]!
2. Postfix address mode (brackets)
   - ldr r3, [r1], #4
   - ldr r3, [r1], r2
   - ldr r3, [r1], r2, LSL#2
3. Offset address mode (etc)
   - ldr r3, [r1, #4]
   - ldr r3, [r1, r2]
   - ldr r3, [r1, r2, LSL#2]
4. PC relative address mode
   - ldr r0, [PC, #offset]
   - ldr r0, =0x12345676
5. Immediate address mode
   - mov r0, #15

Register indirect address mode

Practice: LDR, MOV

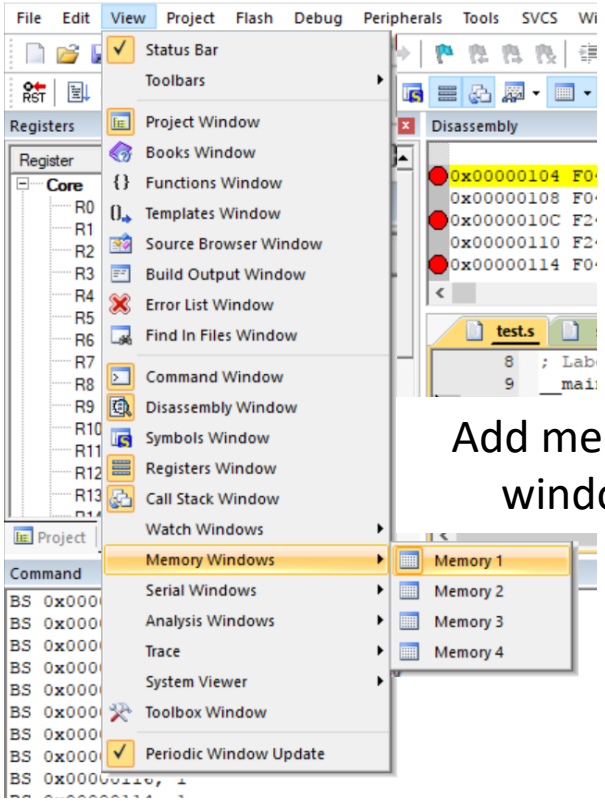


test.s | startup_LPC17xx.s | system_LPC17xx.c

```
    ENTRY                          ; Entry into the code segment

; Label main which is branched to on reset
__main
    LDR  R0, =0x12          ;8 bits (1 byte) in hex
    MOV  R1, #0x12

    LDR  R0, =0x1234        ;16 bits (2 byte2) in hex
    MOV  R1, #0x1234

    LDR  R0, =0xffffffff    ;32 bits (4 byte2) in hex, special case 1
    MOV  R1, #0xffffffff

    LDR  R0, =0x10000000    ;32 bits (4 byte2) in hex, special case 2
    MOV  R1, #0x10000000

; Infinitely loop when we're done
__loop
    B    __loop             ; Branch to the label __loop

    END ; End of file
```

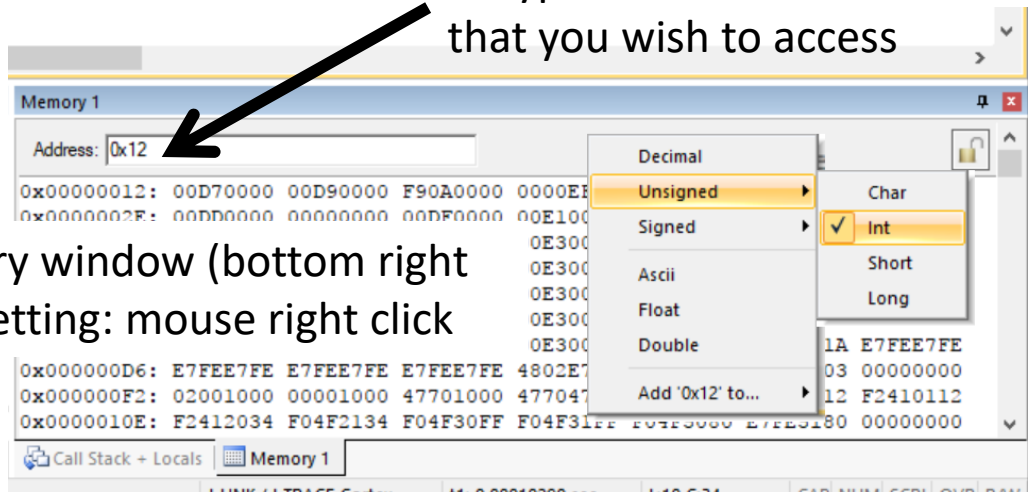
```
p_LPC17xx.s...
C17xx.c...
LPC17xx.c...

e=564 RO-data=204 RW-data=0 ZI-data=512
xf" - 0 Error(s), 0 Warning(s).
d:  00:00:00
```

Go to debug mode



Add memory window

2. Type-in the address that you wish to access



1. Memory window (bottom right corner) setting: mouse right click

Address that you wish to access

4 bytes of Contents at address 0x12-0x15 → 4 bytes →int

4 bytes of Contents at address 0x16-0x19 → 4 bytes →int

at address 0x1A-0x1D → 4 bytes →int

0x1E-0x21

0x22-0x25

0x26-0x29

0x2A-0x2D

Address: 0x12

```
0x00000012: 00D70000 00D90000 F90A0000 0000EFFF 00000000 00000000 00DB0000
0x0000002E: 00DD0000 00000000 00DF0000 00E10000 00E30000 00E30000 00E30000
0x0000004A: 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000
0x00000066: 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000
0x00000082: 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000
0x0000009E: 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000 00E30000
0x000000BA: 00E30000 00E30000 00E30000 00E30000 F0000000 E7FEB81A E7FEE7FE
0x000000D6: E7FEE7FE E7FEE7FE E7FEE7FE 4802E7FE 4A014903 47704B03 00000000
0x000000F2: 02001000 00001000 47701000 47704770 F04F0000 F04F0012 F2410112
0x0000010E: F2412034 F04F2134 F04F30FF F04F31FF F04F5080 E7FE5180 00000000
```

C:\Users\Noori\Desktop\CS100\CS100C_Fall2017\0_Lab_testing\General_testing\test.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
R0	0x00000012
R1	0x00000012
R7	0x00000000
R8	0x00000000
R9	0x100001DC
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x10000200
R14 (LR)	0x1FFF0D5F
R15 (PC)	0x0000010C
xPSR	0x61000000

R0 and R1 have the same values

Disassembly

```
__main:
0x00000104 F04F0012  MOV     r0,#0x12
0x00000108 F04F0112  MOV     r1,#0x12
0x0000010C F2412034  MOVW    r0,#0x1234
0x00000110 F2412134  MOVW    r1,#0x1234
0x00000114 F04F30FF  MOV     r0,#0xFFFFFFFF
```

Real execution
Just MOV

test.s | startup_LPC17xx.s | system_LPC17xx.c

```
__main
    LDR  R0, =0x12          ;8 bits (1 byte) in hex
    MOV  R1, #0x12

    LDR  R0, =0x1234        ;16 bits (2 byte2) in hex
    MOV  R1, #0x1234

    LDR  R0, =0xffffffff    ;32 bits (4 byte2) in hex, special ca
    MOV  R1, #0xffffffff
```

Command

```
BS 0x00000114
BS 0x0000010C
BS 0x00000104
BS 0x00000106, 1
BS 0x00000104, 1
BS 0x00000108, 1
BS 0x00000118, 1
```

Memory 1

Address: 0x12

```
0x00000012: 00D70000 00D90000
0x                   00000000
0x          000      00E30000
0x          000      00E30000
```

No change

C:\Users\Noori\Desktop\CS100\CS100C_Fall2017\0_Lab_testing\General_testing\test.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
R0	0x00001234
R1	0x00001234
R7	0x00000000
R8	0x00000000
R9	0x100001DC
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x10000200
R14 (LR)	0x1FFF0D5F
R15 (PC)	0x00000114
xPSR	0x61000000

R0 and R1 have the same values

Disassembly

```
__main:
0x00000104 F04F0012  MOV     r0,#0x12
0x0000010C F2412034  MOVW    r0,#0x1234
0x00000110 F2412134  MOVW    r1,#0x1234
0x00000118 F04F31FF  MOV     r1,#0xFFFFFFFF
```

Real execution
Just **MOVW**

test.s | startup_LPC17xx.s | system_LPC17xx.c

```
12
13      LDR  R0, =0x1234       ;16 bits (2 byte2) in hex
14      MOV  R1, #0x1234

16      LDR  R0, =0xffffffff   ;32 bits (4 byte2) in hex, special case 1
17      MOV  R1, #0xffffffff
18
19      LDR  R0, =0x10000000   ;32 bits (4 byte2) in hex, special case 2
20      MOV  R1, #0x10000000
```

Command

```
BS 0x00000114
BS 0x0000010C
BS 0x00000104
BS 0x00000106, 1
BS 0x00000104, 1
BS 0x00000108, 1
BS 0x00000118, 1
```

ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess

Memory 1

Address: 0x1234

```
0x00001234: FFFFFFFF FFFFFFFF FFFFFFFF
0x000012A4: FFFFFFFF FFFFFFFF FFFFFFFF
0x000012C0: FFFFFFFF FFFFFFFF FFFFFFFF
0x000012DC: FFFFFFFF FFFFFFFF FFFFFFFF
0x000012F8: FFFFFFFF FFFFFFFF FFFFFFFF
```

No change

Call Stack + Locals | Memory 1

C:\Users\Noori\Desktop\CS100\CS100C_Fall2017\0_Lab_testing\General_testing\test.uvprojx - µVision

ile Edit View Project Flash Debug Peripherals Tools SVCS Window Help

gisters

Register	Value
R0	0xFFFFFFFF
R1	0xFFFFFFFF
R7	0x00000000
R8	0x00000000
R9	0x100001DC
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x10000200
R14 (LR)	0x1FFF0D5F
R15 (PC)	0x0000011C
xPSR	0x61000000
Banked	
System	

R0 and R1 have the same values

Project | Registers

Disassembly

```
0x00000114 F04F30FF  MOV    r0,#0xFFFFFFFF
0x00000118 F04F31FF  MOV    r1,#0xFFFFFFFF
0x0000011C F04F5080  MOV    r0,#0x10000000
0x00000120 F04F5180  MOV    r1,#0x10000000
0x00000124 E7FE      B      0x00000124
0x00000126 0000      MOVS   r0,r0
0x00000128 0000      MOVS   r0,r0
0x0000012A 0000      MOVS   r0,r0
```

Real execution
Just **MOV**

test.s | startup_LPC17xx.s | system_LPC17xx.c

```
15
16      LDR  R0, =0xffffffff  ;32 bits (4 byte) in hex, special case 1
17      MOV  R1, #0xffffffff
18
19      LDR  R0, =0x10000000  ;32 bits (4 byte) in hex, special case 2
20      MOV  R1, #0x10000000
21
22
23  ; Infinitely loop when we're done
24  __loop
```

mmand

```
0x00000124
0x0000011A
\\test\RTE/Device/LPC1769/startup_LPC17xx.s\138
0x0000011E
0x0000010C
```

Memory 1

```
0xF  A 0000CD10 0000D100
0x0  0 000000EF 00000000
0x00000037: 0000DF00 0000E100 0000E300
```

No change

C:\Users\Noori\Desktop\CS100\CS100C_Fall2017\0_Lab_testing\General_testing\test.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
R0	0x10000000
R1	0x10000000
R7	0x00000000
R8	0x00000000
R9	0x100001DC
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x10000200
R14 (LR)	0x1FFF0D5F
R15 (PC)	0x00000124
xPSR	0x61000000
Banked	
System	

R0 and R1 have the same values

Project | Registers

Disassembly

```
0x00000114 F04F30FF  MOV   r0,#0xFFFFFFFF
0x0000011C F04F5080  MOV   r0,#0x10000000
0x00000120 F04F5180  MOV   r1,#0x10000000
0x00000126 0000      MOVS  r0,r0
0x00000128 0000      MOVS  r0,r0
0x0000012A 0000      MOVS  r0,r0
```

Real execution
Just **MOV**

test.s | startup_LPC17xx.s | system_LPC17xx.c

```
15
16      LDR  R0, =0xffffffff  ;32 bits (4 byte) in hex, special case 1
18
19      LDR  R0, =0x10000000  ;32 bits (4 byte) in hex, special case 2
20      MOV  R1, #0x10000000
22
23  ; Infinitely loop when we're done
24  __loop
```

Command

```
BS 0x00000124
```

Memory 1

Address: 0x10000000

```
0x10000000: E00AE7FE 62D780D 24084068
0x:  00 08C0D301 4770300E
0x:  5D 47904620 28006960
0x10000070: 28274449 DC02D01C D1062820
```

No change

RECALL:
test.s(17): error: A1871E: Immediate 0x00123456 cannot be represented by 0-255 shifted left by 0-23 or duplicated in all, odd or even bytes

C:\Users\Noori\Desktop\CS100\CS100C_Fall2017\0_Lab_testing\General_testing\test.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
ore	
R0	0x12345678
R4	0x3456ABCD
R5	0x3456ABCD
R6	0x12345678
R7	0x00000000
R8	0x00000000
R9	0x100001DC
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x10000200
R14 (LR)	0x1FFF0D5F
R15 (PC)	0x00000126
xPSR	0x61000000
Banked	
System	

R0 is updated

Project | Registers

Disassembly

```
0x00000124 4800    LDR    r0,[pc,#0]   ; @0x0 00012
0x00000126 E7FE    B      0x00000126
0x00000128 5678    DCW    0x5678
0x0000012A 1234    DCW    0x1234
0x0000012E 0000    DCW    0x0000
0x00000130 0000    DCW    0x0000
```

test.s | startup_LPC17xx.s | system_LPC17xx.c

```
16      LDR  R0, =0xffffffff  ;32 bits (4 byte) in
17      MOV  R1, #0xffffffff
18
19      LDR  R0, =0x10000000  ;32 bits (4 byte) in hex, special case 2
20      MOV  R1, #0x10000000
22      LDR  R0, =0x12345678
24
25  ; Infinitely loop when we're done
```

Real execution

LDR with PC related addressing

[a square bracket]

Although we didn't specify memory address, PC related address is assigned to load the value to R0 register

Command

```
BS 0x00000106
BS 0x00000116
BS 0x0000011A
BS \\test\RTE/Device/LPC1769/startup_LPC17xx.s\138
```

Memory 1

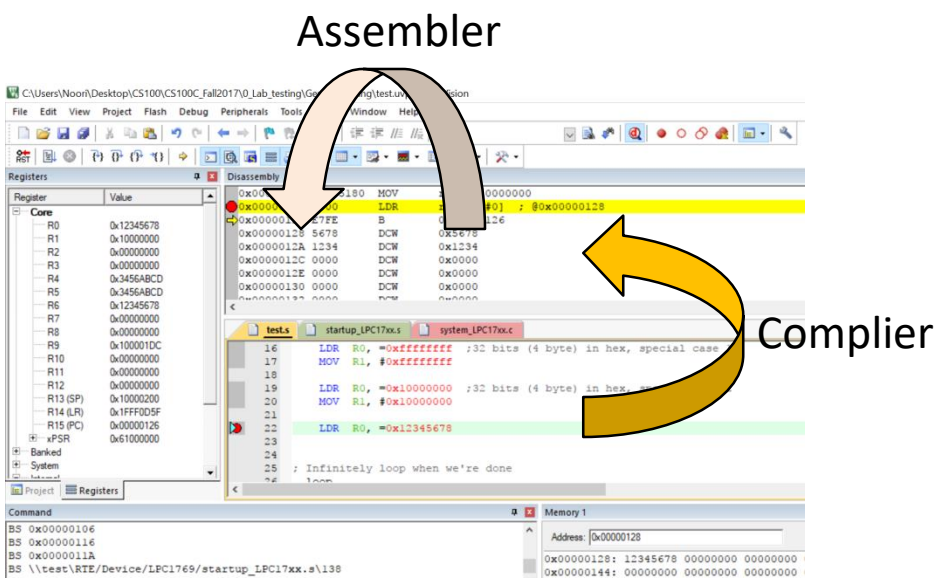
Address: 0x00000128

```
0x00000128: 12345678 00000000 00000000
0x00000144: 00000000 00000000 00000000
```

The memory contents are changed

A message from previous examples

- The complier is smart enough to analyze codes case by case written by users
- The complier is always finding an efficient way to optimize code and program



A summary of MOV and LDR with addressing modes

- MOV Rd, #number ;can handle limited size of number less than 2 bytes (Immediate addressing)
- LDR Rd, [Rn] ; Contents at address Rn will be loaded to Rd (Indexed addressing)
- LDR Rd, =Number_more_than_2 byte_such_as_address ; PC relative addressing LDR Rd, [pc]
- LDR Rd, =Number_less_than_2 byte ; immediate addressing such as MOV

MOV instructions

MOV Rd , operand2	Rd ← operand2
MVN Rd , operand2	Rd ← NOT operand2

```
MOV r4, r5              ; Copy r5 to r4
MVN r4, r5              ; r4 = bitwise logical NOT of r5
MOV r1, r2, LSL #3      ; r1 = r2 << 3
MOV r0, PC              ; Copy PC (r15) to r0
MOV r1, SP              ; Copy SP (r13) to r1
```

MOVW Rd, #imm16	**Move Wide**, Rd ← #imm16
MOVT Rd, #imm16	**Move Top**, Rd ← #imm16 << 16
MOV Rd, #const	**Move**, Rd ← const

Example: Load a 32-bit number into a register

```
MOVW r0, #0x4321    ; r0 = 0x00004321
MOVT r0, #0x8765    ; r0 = 0x87654321
```

Order **does matter**!

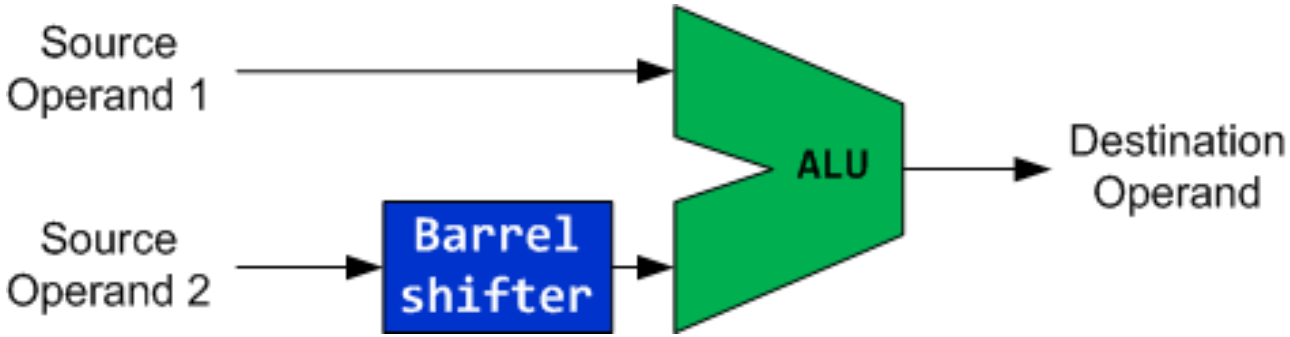
- MOVW will zero the upper halfword
- MOVT won't zero the lower halfword

```
MOVT r0, #0x8765    ; r0 = 0x8765xxxx
MOVW r0, #0x4321    ; r0 = 0x00004321
```

Playing with bits (binary #)

1. Identify even or odd numbers by checking LSB
   - 0:
   - 1:
2. Division by 2^n: shift to by n bits
3. Multiply by 2^n: shift to by n bits

Shift instructions



- The second operand of ALU has a special hardware called Barrel shifter
- Example:

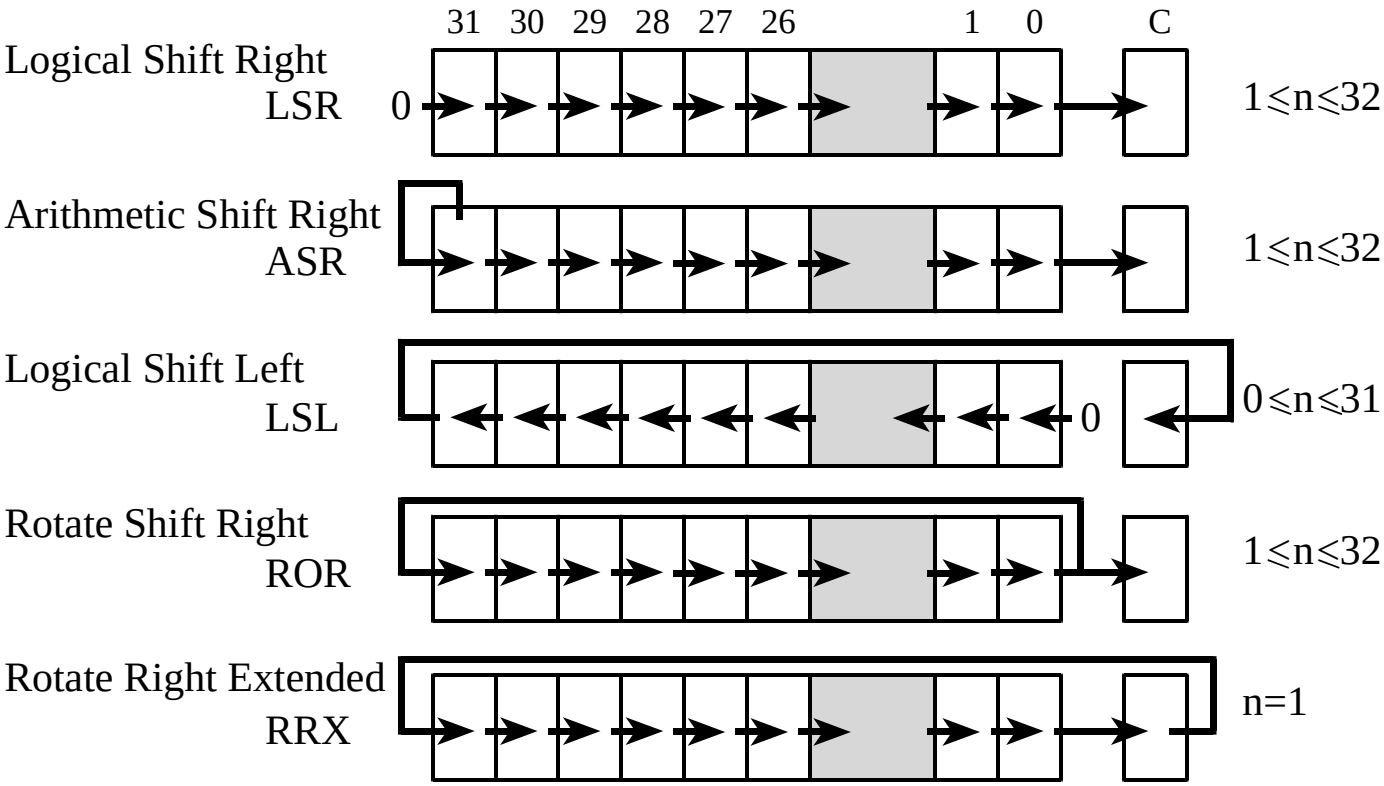
```
ADD r1, r0, r0, LSL #3  ; r1 = r0 + r0 << 3 = 9 × r0
```

- Examples:
  - **ADD r1, r0, r0, LSL #3**
    ; r1 = r0 + r0 << 3 = r0 + 8 × r0
  - **ADD r1, r0, r0, LSR #3**
    ; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)
  - **ADD r1, r0, r0, ASR #3**
    ; r1 = r0 + r0 >> 3 = r0 + r0/8 (signed)

- Use Barrel shifter to speed up the application

```
ADD r1, r0, r0, LSL #3  <=>  MOV r2, #9      ; r2 = 9
                             MUL r1, r0, r2 ; r1 = r0 * 9
```

Shift Operations

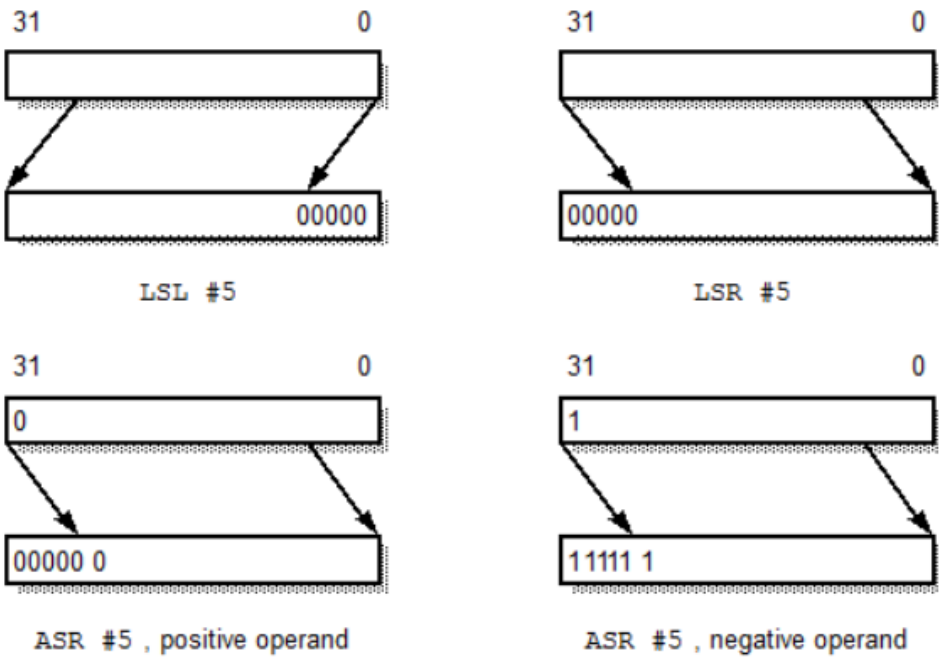


*Use the **ASR** instruction when manipulating signed numbers,*

*and use the **LSR** instruction when shifting unsigned numbers*

3

I had hard time to understand the real difference between LSR and ASR but hope this image helps you to understand the same. In LSR(`Logical Shift Right`) the MSB(`Most Significant Bit`) is replaced by 0 where as In ASR(`Arithematic Shift Right`) MSB is same as the earlier MSB before being shifted .(Similar for Left Shift) ASR is useful in computing with signed values in two-complement representation.



*Use the **ASR** instruction when manipulating signed numbers,*

*and use the **LSR** instruction when shifting unsigned numbers*

Table 1: Initial memory address and content

Address data	Content data
0x1000.000C	0xAAAA.AAAA
0x1000.0008	0x5555.5555
0x1000.0004	0x89AB.CDEF
0x1000.0000	0x0123.4567

(d) Which memory address data will be updated after executing the following code?

```
MOV R0, #0x10000000
MOV R1, #0x0C
MOV R2, #0x08
STR R1, [R0, R2, LSR#1]
```

(1)0x04 (2)0x1000.0000
(3)0x08 (4)0x1000.0004
(5)0x0C (6)0x89AB.CDEF

(e) Before executing the code from the previous question, what is the original content data in the memory address of part (d)?
(1)0x04 (2)0x5555.5555
(3)0x08 (4)0x0123.4567
(5)0x0C (6)0x89AB.CDEF

(f) After executing the code from the previous question, what is the new content data in the memory address?
(1)0x04 (2)0x1000.0000
(3)0x08 (4)0x1000.0004
(5)0x0C (6)0x89AB.CDEF

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

- The "Branch with link (BL)" instruction implements a subroutine call by writing PC-4 (returning address) into the LR of the current.
- To return from subroutine, simply need to restore the PC from the LR:
  - **MOV pc, lr**
- By default, the "Branch" instruction does not affect LR, unless otherwise “BL”

Number Interpretation

Which is greater?

`0xFFFFFFFF` or `0x00000001`

- If they represent signed numbers, the latter is greater
  (`-1 < 1`).
- If they represent unsigned numbers, the former is greater
  (`4294967295 > 1`).

Which is Greater: 0xFFFFFFFF or 0x00000001?

It's **software's responsibility** to tell computer how to interpret data:

- If written in C, declare the signed vs unsigned variable
- If written in Assembly, use signed vs unsigned branch instructions

```
signed int x, y ;
x = -1;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF
    MOVS r5, #0x00000001
    CMP  r5, r6
    BLE  Then_Clause
    ...
```

BLE: Branch if less than or equal, signed ≤

```
unsigned int x, y ;
x = 4294967295;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF
    MOVS r5, #0x00000001
    CMP  r5, r6
    BLS  Then_Clause
    ...
```

BLS: Branch if lower or same, unsigned ≤

Condition Codes

Suffix	Description	Flags tested
EQ	EQual	
NE	Not Equal	

MI	MInus (Negative)	
PL	PLus (Positive or Zero)	
VS	oVerflow Set	
VC	oVerflow Clear	
HI	Unsigned HIgher	
LS	Unsigned Lower or Same	
GE	Signed Greater or Equal	
LT	Signed Less Than	
GT	Signed Greater Than	
LE	Signed Less than or Equal	
AL	ALways	

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

Suffix	Description	Flags tested
EQ	EQual	Z=1
NE	Not Equal	Z=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	

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 0
V = 0	• No overflow, implying the result is correct. • The result is non-negative, • Thus r0 – r1 ≥ 0, i.e., r0 ≥ r1

R0=0x1111.1111
R1=0x1111.1110
R0-R1=0x0000.0001
N=0 (pos) , V=0 (P-P=P, can)
R0-R1>0 ➔ R0>R1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 1
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as negative and in fact it should be non-negative. • Thus r0 – r1 ≥ 0 in reality., i.e. r0 ≥ r1

R0=0x0000.0000 =0

R1=0x8000.0000 =-2^{31}

R0-R1=0x8000.0000

N=1 (neg) , V=1 (P-N=N, can't)

R0-R1=0- -2^{31} > 0 ➔ R0>R1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 1
V = 0	• No overflow, implying the result is correct. • The result is negative. • Thus r0 – r1 < 0, i.e., r0 < r1

R0=0xF000.0000
R1=0x7000.0000
R0-R1=0x8000.0000=-2^{31}
N=1 (neg) , V=0 (N-P=N, can)
R0-R1< 0 ➔ R0<R1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 0
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as non-negative and in fact it should be negative. • Thus r0 – r1 < 0 in reality, i.e., r0 < r1

R0=0x8000.0000 $=-2^{31}$

R1=0x1000.0000 $=2^{28}$

R0-R1=0x7000.0000

N=0 (pos) , V=1 (N-P=P, can't)

R0-R1$=-2^{31}-2^{28}<0$ ➔ R0<R1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 0	N = 1
V = 0	• No overflow, implying the result is correct. • The result is non-negative, • Thus r0 – r1 ≥ 0, i.e., r0 ≥ r1	• No overflow, implying the result is correct. • The result is negative. • Thus r0 – r1 < 0, i.e., r0 < r1
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as non-negative and in fact it should be negative. • Thus r0 – r1 < 0 in reality, i.e., r0 < r1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as negative and in fact it should be non-negative. • Thus r0 – r1 ≥ 0 in reality., i.e. r0 ≥ r1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Signed vs. Unsigned

Conditional codes applied to branch instructions

Compare	Signed	Unsigned
==	EQ	EQ
≠	NE	NE
>	GT	HI
≥	GE	HS
<	LT	LO
≤	LE	LS

Compare	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS

Branch Instructions

- Branches are a necessary evil
  - Software can't avoid them
  - Hardware engineers' anathema (hate them)
  - Messing up pipeline

Cycle				1	2	3	4	5
Address	Operation							
0x8000	BL	Fetch	Decode	Execute	Linkret	Adjust		
0x8004	X		Fetch	Decode				
0x8008	XX			Fetch				
0x8FEC	ADD				Fetch	Decode	Execute	
0x8FF0	SUB					Fetch	Decode	Execute
0x8FF4	MOV						Fetch	Decode
								Fetch

Branch Instructions + condition codes

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

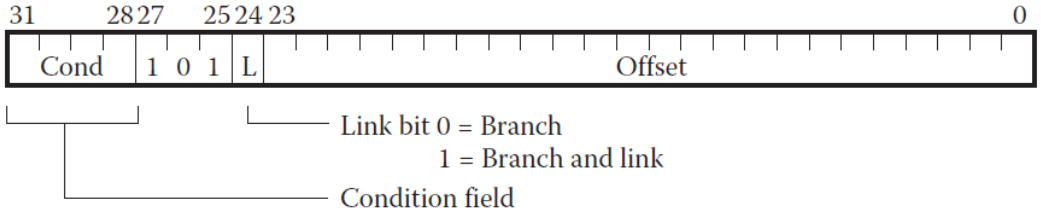


FIGURE 8.2 The B and BL instruction.

Example 1: CMP

$$f(x) = |x|$$

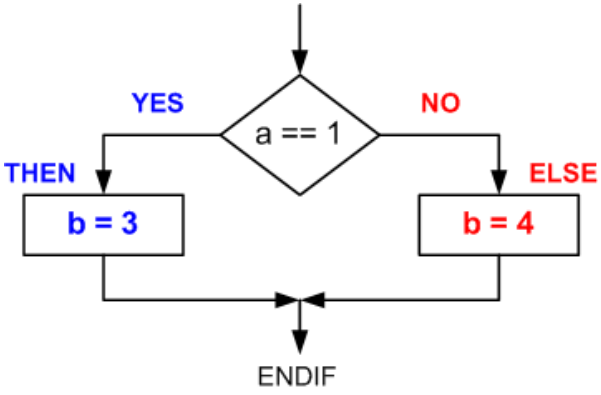
```
        Area absolute, CODE, READONLY
        EXPORT __main
        ENTRY

__main PROC
                          ?


done    B done      ; deadloop

        ENDP
        END
```

Note: RSB = Reverse SuBtract

Example 2: If-then-else

C Program

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

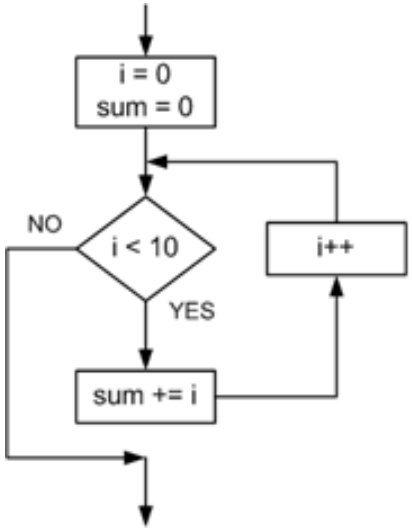



```
        ; r1 = a, r2 = b
                        ; compare a and 1
                        ; go to else if a ≠ 1
then          ?         ; b = 3
                        ; go to endif
else                    ; b = 4
endif
```

Example 3-1: For Loop

C Program

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



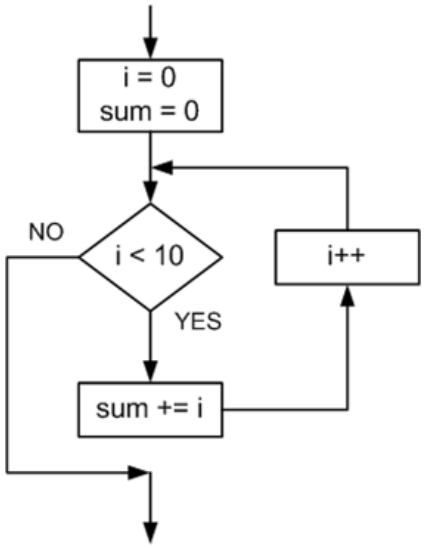
Implementation 1:

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

        B   check
loop

                ?
check

endloop
```

Example 3-2: For Loop

C Program

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



Implementation 2:

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

loop

                              ?


            B    loop
endloop
```