Chapter 3
ARM Assembly Language

3.1 ARM Assembly Language and programming environment

ARM assembly language

- ARM instructions are written in the form

```
Label  Op-code operand1, operand2, operand3  //comment
```

- Consider the following example implementing a loop.

```
Test_5 ADD  r0,r1,r2     //TotalTime = Time + NewTime
       SUBS r7,#1 //Decrement loop counter
       BEQ  Test_5       //IF zero THEN goto Test_5
```

- The Label field is a user-defined label that can be used by other instructions to refer to that line.
- Any text following a double slash is regarded as a comment field and is ignored by the assembler.

ARM assembly language and CPUlator

- We use CPUlator Simulator as the programming environment in this course. More details about this simulator will be provided in Lab 1.
- Web-based simulator and no installation needed: https://cpulator.01xz.net/?sys=arm-de1soc
- Consider this program that you will see in Lab 1:

```
.org    0x1000  // Start at memory location 1000
.text           // Code section
.global _start
_start:
mov r0, #9       // Store decimal 9 in register r0
mov r1, #0xE     // Store hex E (decimal 14) in register r1
add r2, r1, r0  // Add the contents of r0 and r1 and put result in r2

// End program
_stop:
b    _stop

.end
```

ARM assembly language and CPUlator

- Two types of statements:
  1. Assembler directives
     - Used to control the assembler
     - Do not generate machine code
     - Highlighted in red.
     - All directives in CPUlator starts with a period.

  2. Executable instructions
     - Do generate machine code to perform various operations
     - *Pseudoinstruction*: instruction available to the programmer but not part of the processor's instruction set. Shorthand expression that an assembler converts to appropriate machine code (e.g., ldr r1, =0xFF896788 – discuss in more detail after covering the fundamentals).

Directives used in the sample program

- .org
  - Format: .org memory_address
  - tells the compiler where in memory to place the program.
- .text
  - identifies the code section of the program.
- .global
  - Format: .global label
  - identifies globally visible labels.
- .end
  - identifies the end of the source code and data.

Instructions used in the sample program

`mov` (Move)

- Puts a value into a register.
- Two formats:
  - `mov r0, #9` puts the (immediate) decimal value 9 into register r0.
  - `mov r1, r0` copies the value in register r0 into register r1.

`add` (Add)

- adds two values and places the results into a register.
- Two formats:
  - `add r2, r1, r0` adds the contents of r0 and r1 and puts the result in register r2.
  - `add r3, r1, #7` adds the contents of r1 and the decimal value 7 and puts the result in register r3.

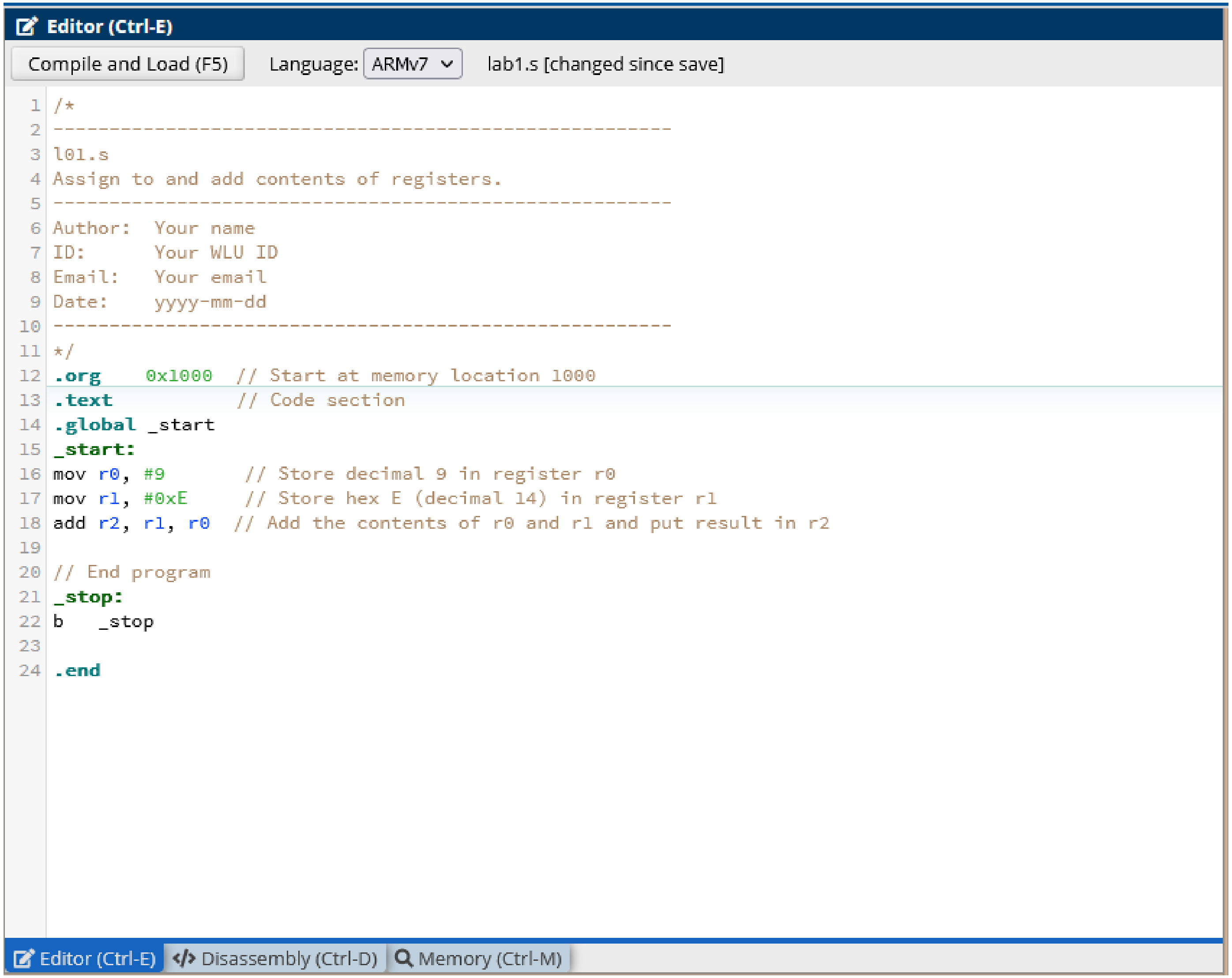
`b` (Branch)

- Branches execution to a memory location.
- The instruction:

```
_stop:
b _stop
```

terminates a program by branching to its own location.

CPUlator interface

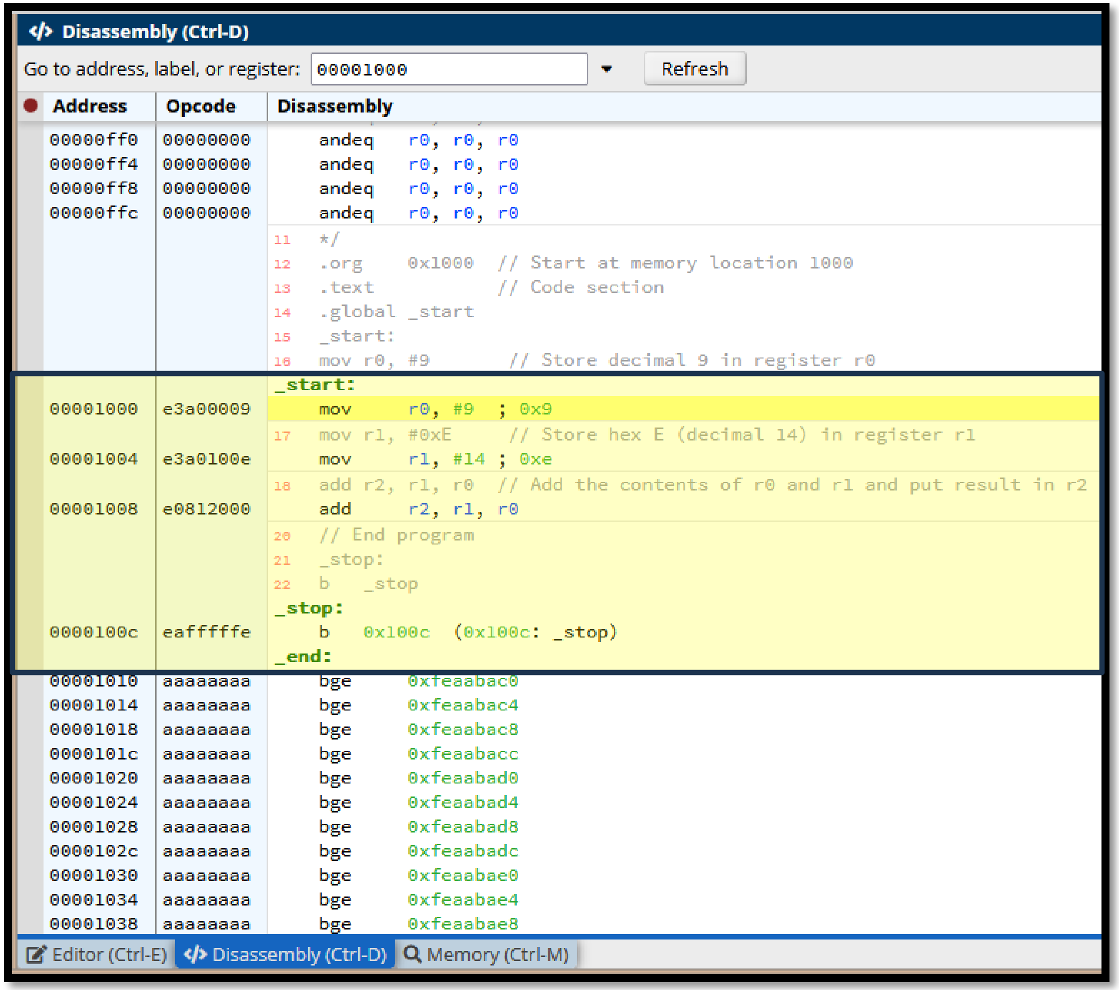


Language: ARMv7

lab1.s [changed since save]

```
/*
-------------------------------------------------------
l01.s
Assign to and add contents of registers.
-------------------------------------------------------
Author:  Your name
ID:      Your WLU ID
Email:   Your email
Date:    yyyy-mm-dd
-------------------------------------------------------
*/
.org    0x1000  // Start at memory location 1000
.text           // Code section
.global _start
_start:
mov r0, #9       // Store decimal 9 in register r0
mov r1, #0xE     // Store hex E (decimal 14) in register r1
add r2, r1, r0  // Add the contents of r0 and r1 and put result in r2

// End program
_stop:
b   _stop

.end
```

CPUlator interface



Go to address, label, or register: 00001000

```
Address   Opcode    Disassembly
00000ff0  00000000      andeq   r0, r0, r0
00000ff4  00000000      andeq   r0, r0, r0
00000ff8  00000000      andeq   r0, r0, r0
00000ffc  00000000      andeq   r0, r0, r0
                        */
                        .org    0x1000  // Start at memory location 1000
                        .text           // Code section
                        .global _start
                        _start:
                        mov r0, #9        // Store decimal 9 in register r0
                    _start:
00001000  e3a00009      mov     r0, #9  ; 0x9
                        mov r1, #0xE      // Store hex E (decimal 14) in register r1
00001004  e3a0100e      mov     r1, #14 ; 0xe
                        add r2, r1, r0  // Add the contents of r0 and r1 and put result in r2
00001008  e0812000      add     r2, r1, r0
                        // End program
                        _stop:
                        b    _stop
                    _stop:
0000100c  eafffffe      b    0x100c  (0x100c: _stop)
                    _end:
00001010  aaaaaaaa      bge     0xfeaabac0
00001014  aaaaaaaa      bge     0xfeaabac4
00001018  aaaaaaaa      bge     0xfeaabac8
0000101c  aaaaaaaa      bge     0xfeaabacc
00001020  aaaaaaaa      bge     0xfeaabad0
00001024  aaaaaaaa      bge     0xfeaabad4
00001028  aaaaaaaa      bge     0xfeaabad8
0000102c  aaaaaaaa      bge     0xfeaabadc
00001030  aaaaaaaa      bge     0xfeaabae0
00001034  aaaaaaaa      bge     0xfeaabae4
00001038  aaaaaaaa      bge     0xfeaabae8
```

Address

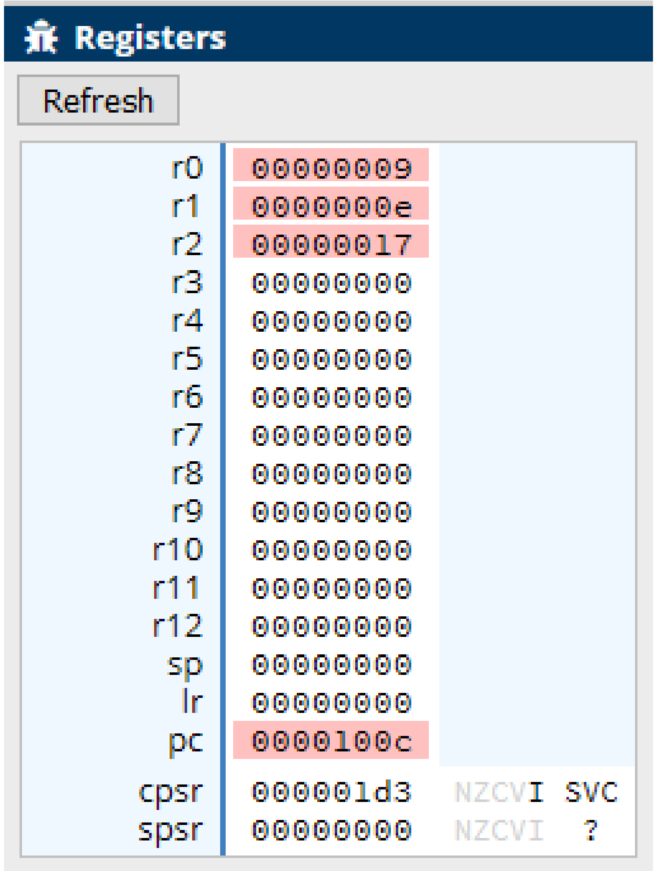
- The address of the instruction.
- The program starts at address 0x1000 as specified by the .org instruction.

Opcode

- 32-bit instruction encoded in binary format
- Specify the operation (e.g., ADD) and the operands.

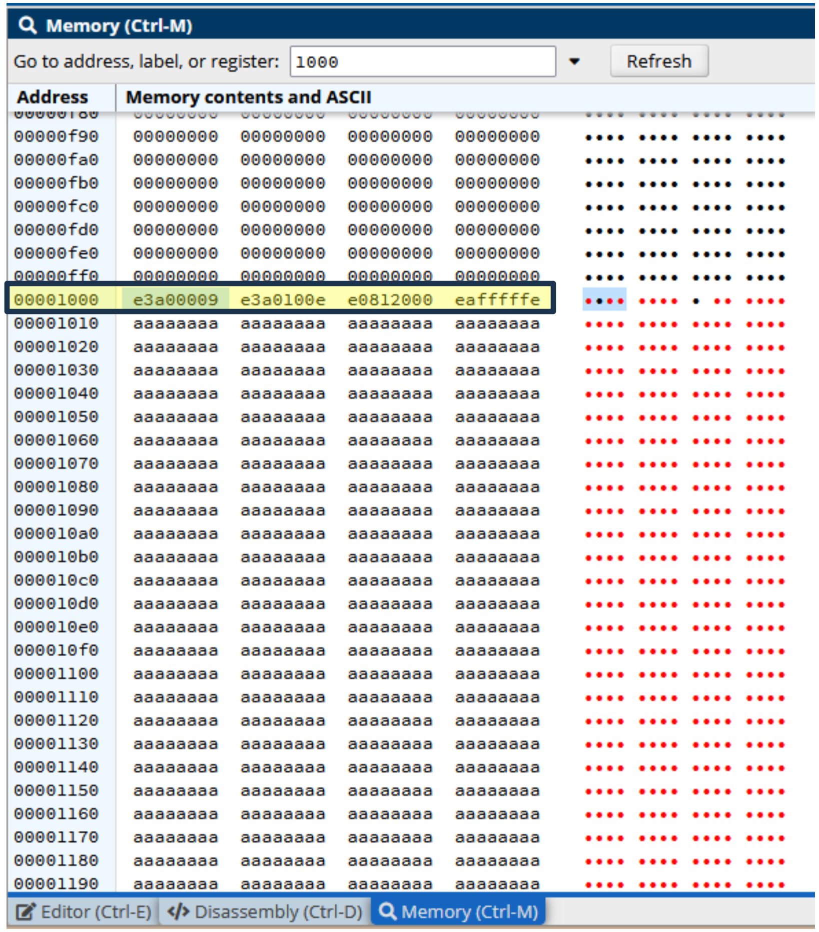
CPUlator interface

Registers



```
   r0   00000009
   r1   0000000e
   r2   00000017
   r3   00000000
   r4   00000000
   r5   00000000
   r6   00000000
   r7   00000000
   r8   00000000
   r9   00000000
  r10   00000000
  r11   00000000
  r12   00000000
   sp   00000000
   lr   00000000
   pc   0000100c

 cpsr   000001d3   NZCVI  SVC
 spsr   00000000   NZCVI   ?
```

Memory



Go to address, label, or register: 1000

```
Address    Memory contents and ASCII
00000f80   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000f90   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000fa0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000fb0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000fc0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000fd0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000fe0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00000ff0   00000000  00000000  00000000  00000000    •••• •••• •••• ••••
00001000   e3a00009  e3a0100e  e0812000  eafffffe    •••• •••• •  •• ••••
00001010   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001020   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001030   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001040   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001050   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001060   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001070   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001080   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001090   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010a0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010b0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010c0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010d0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010e0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
000010f0   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001100   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001110   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001120   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001130   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001140   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001150   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001160   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001170   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001180   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
00001190   aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa    •••• •••• •••• ••••
```

CPUlator interface

- You will learn the use of debugging tools CPUlator in Lab 2 and 3.
- The following statement you encounter in Lab 2 is a pseudoinstruction:
  - `ldr r0, =0x12345678`
  - This pseudoinstruction puts a number into a register
  - You will learn more why we need this statement and how this pseudoinstruction works later.

3.2 ARM Data-processing instructions

ARM Data Processing Instructions

- Consist of :
  - Arithmetic: **`ADD`** **`ADC`** **`SUB`** **`SBC`** **`RSB`** **`RSC`**
  - Logical: **`AND`** **`ORR`** **`EOR`** **`BIC`**
  - Comparisons: **`CMP`** **`CMN`** **`TST`** **`TEQ`**
  - Data movement: **`MOV`** **`MVN`**
- These instructions only work on registers, NOT memory.
- Syntax:

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

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via *barrel shifter*.

Updating ARM conditional codes

- ARM does not automatically update its status flags after an operation
- *Update-on-demand*: status flags are updated only the programmer puts an “S” suffix in the mnemonic, i.e.,
  - ADD r1, r2, r3 does not update status flags
  - ADDS r1, r2, r3 updates status flags
- This allows the programmer performs a test, and then carry out other instructions without changing the flags:

```
SUBS r1, r1, #1      ; subtract 1 from r1 and set status bits
ADD r2, r2, #4       ; increment r2 (don’t update status bits)
BEQ Error            ; if r1 is zero then deal with the problem
```

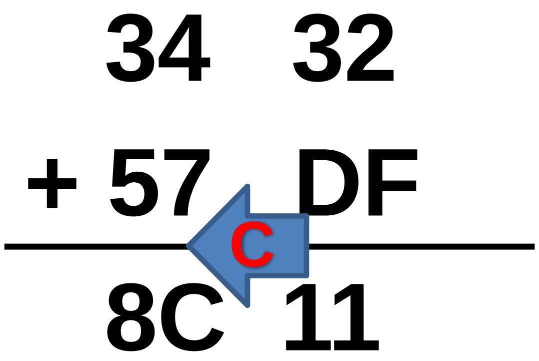
Addition

- Suppose you have a computer capable of adding 8-bit values but you want to add two 16-bit values.
- You can divide the 16-bit addition to two 8-bit additions, but the answer is not correct if there is a carry in the addition of the least significant byte:

Addition of two 8-bit number without carry

```
   34  32
+  57  DF
----------
   8B  11
```

Addition of two 8-bit number with carry

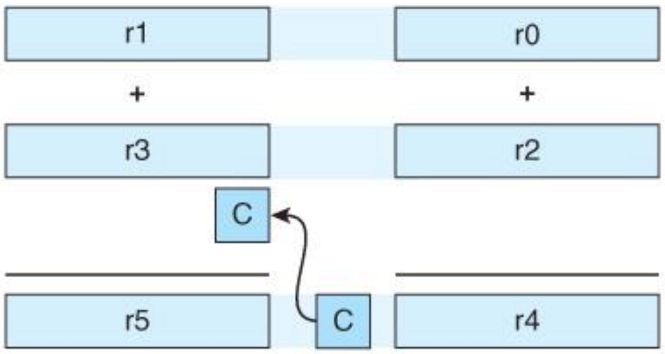


```
   34  32
+  57  DF
----------
   8C  11
```

Multi-Word Addition

- The following shows how a 64-bit addition is performed:

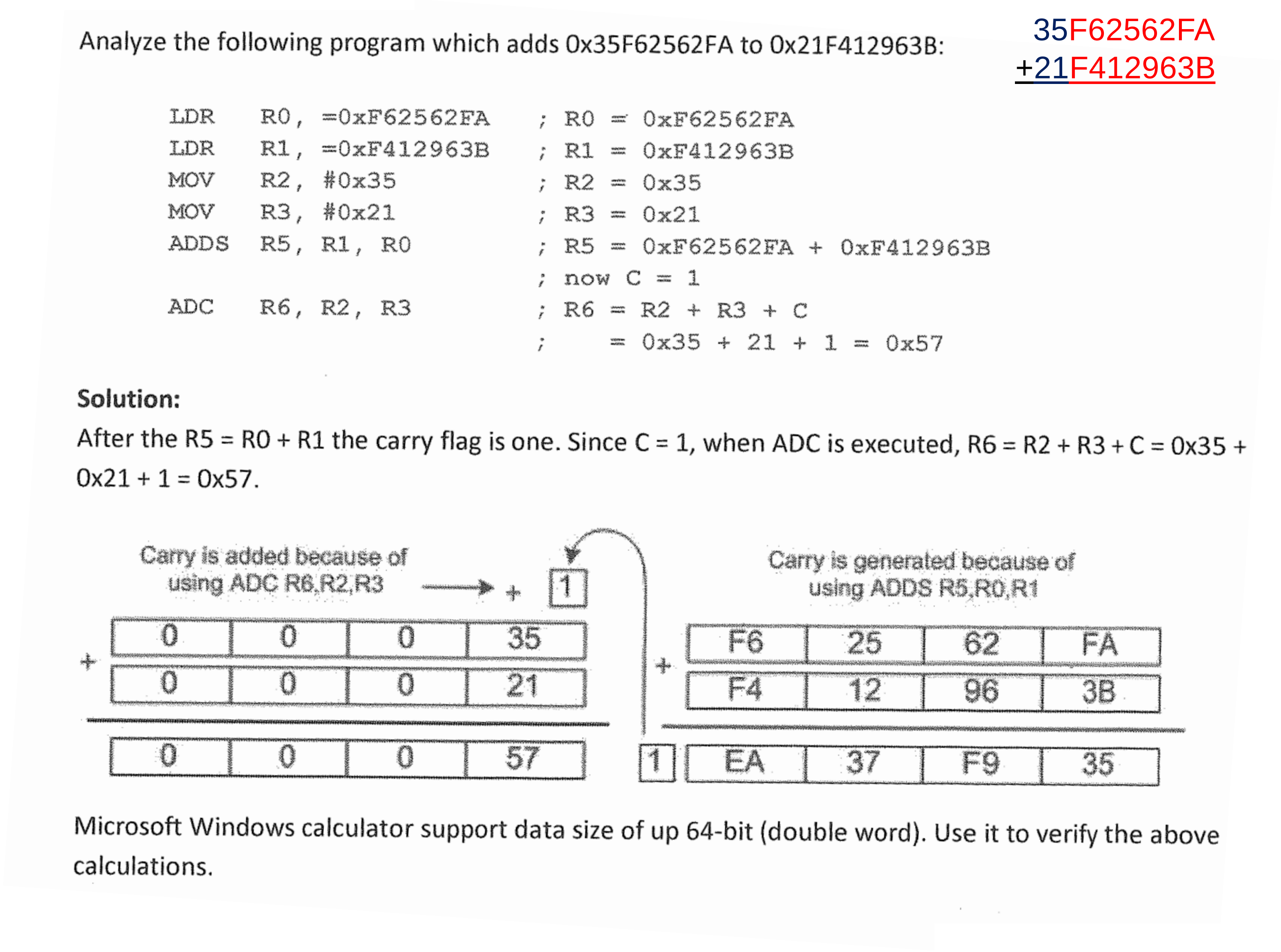
```
ADDS r4, r0, r2
ADC  r5, r1, r3
```



- Must use S after ADD, as the C flag in the status register (CPSR) must be updated.
- `ADC`: add with carry

Example

35F62562FA
+21F412963B



Analyze the following program which adds 0x35F62562FA to 0x21F412963B:

```
LDR    R0, =0xF62562FA     ; R0 = 0xF62562FA
LDR    R1, =0xF412963B     ; R1 = 0xF412963B
MOV    R2, #0x35           ; R2 = 0x35
MOV    R3, #0x21           ; R3 = 0x21
ADDS   R5, R1, R0          ; R5 = 0xF62562FA + 0xF412963B
                           ; now C = 1
ADC    R6, R2, R3          ; R6 = R2 + R3 + C
                           ;    = 0x35 + 21 + 1 = 0x57
```

Solution:

After the R5 = R0 + R1 the carry flag is one. Since C = 1, when ADC is executed, R6 = R2 + R3 + C = 0x35 + 0x21 + 1 = 0x57.

Carry is added because of using ADC R6,R2,R3 → + 1

	0	0	0	35
+	0	0	0	21
	0	0	0	57

Carry is generated because of using ADDS R5,R0,R1

	F6	25	62	FA
+	F4	12	96	3B
1	EA	37	F9	35

Microsoft Windows calculator support data size of up 64-bit (double word). Use it to verify the above calculations.

Subtraction

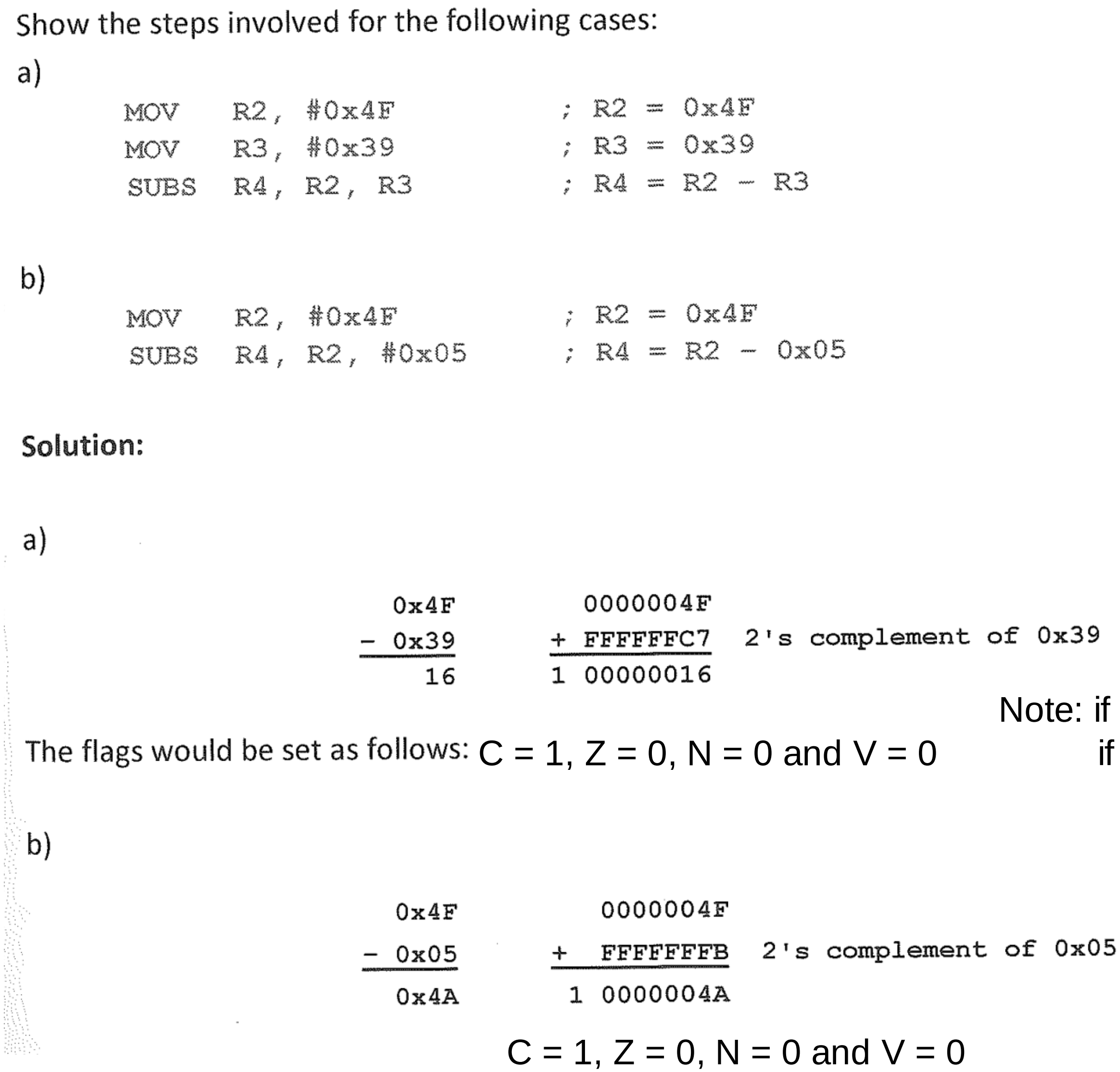
Subtraction:

- `SUB r1, r2, r3`: `[r1]←[r2]-[r3]`
- `SBC`: subtract with borrow

Reverse subtraction:

- `RSB r1, r2, r3: [r1]←[r3]-[r2]`
- As the literal must be in the second operand, RSB allows you to subtract the register content from a literal, e.g.,
  - `RSB r1, r2, #10`: `[r1]←10-[r2]`
- No negation operation in ARM. RSB also allows you to implement negation:
  - `RSB r1, r1, #0: [r1]← 0-[r1]`

Examples: SUBS



Show the steps involved for the following cases:

a)

```
MOV    R2, #0x4F          ; R2 = 0x4F
MOV    R3, #0x39          ; R3 = 0x39
SUBS   R4, R2, R3         ; R4 = R2 - R3
```

b)

```
MOV    R2, #0x4F          ; R2 = 0x4F
SUBS   R4, R2, #0x05      ; R4 = R2 - 0x05
```

Solution:

a)

```
  0x4F        0000004F
- 0x39      + FFFFFFC7   2's complement of 0x39
------      ----------
    16      1 00000016
```

The flags would be set as follows: C = 1, Z = 0, N = 0 and V = 0

Note: if C = 1, No Borrow
if C = 0, Borrow

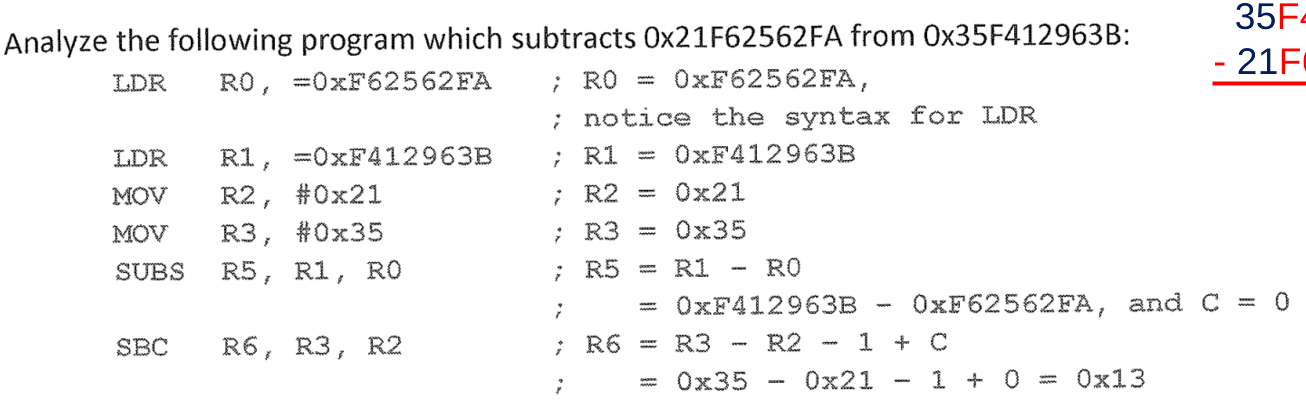
b)

```
  0x4F          0000004F
- 0x05      +   FFFFFFFB   2's complement of 0x05
------      ------------
  0x4A        1 0000004A
```

C = 1, Z = 0, N = 0 and V = 0

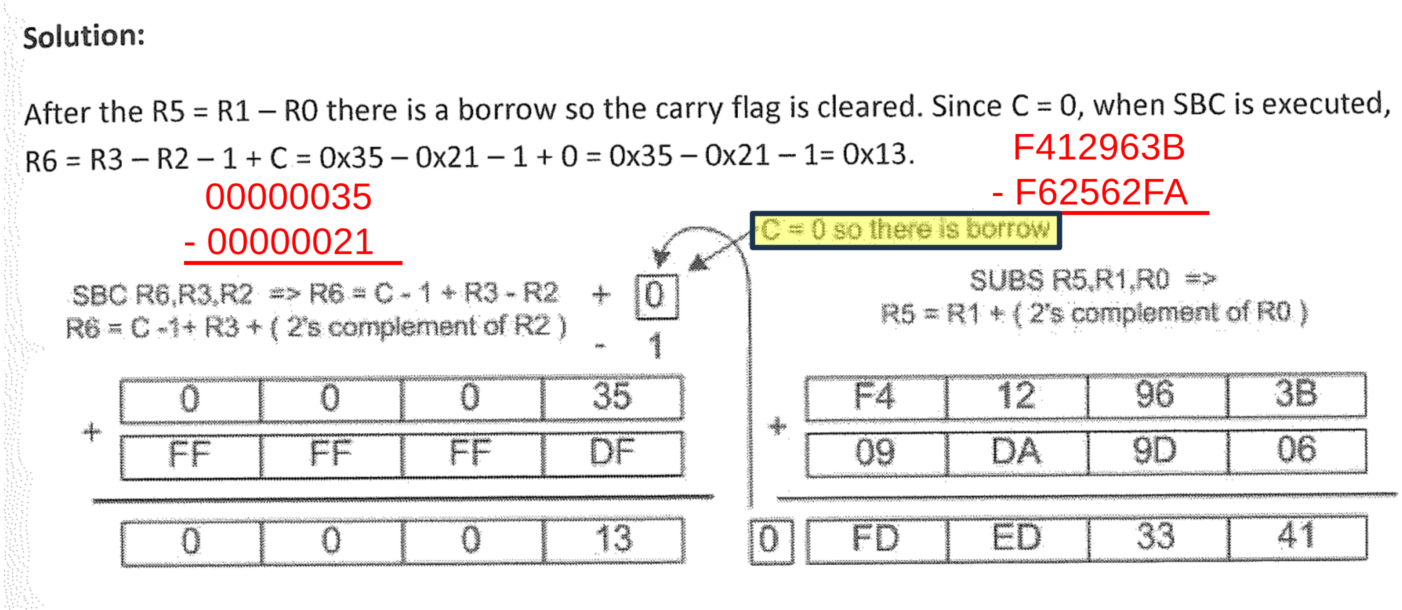
Example: Multi-Word Subtraction

35F412963B
- 21F62562FA



Analyze the following program which subtracts 0x21F62562FA from 0x35F412963B:

```
        LDR     R0, =0xF62562FA     ; R0 = 0xF62562FA,
                                    ; notice the syntax for LDR
        LDR     R1, =0xF412963B     ; R1 = 0xF412963B
        MOV     R2, #0x21           ; R2 = 0x21
        MOV     R3, #0x35           ; R3 = 0x35
        SUBS    R5, R1, R0          ; R5 = R1 - R0
                                    ;    = 0xF412963B - 0xF62562FA, and C = 0
        SBC     R6, R3, R2          ; R6 = R3 - R2 - 1 + C
                                    ;    = 0x35 - 0x21 - 1 + 0 = 0x13
```



Solution:

After the R5 = R1 – R0 there is a borrow so the carry flag is cleared. Since C = 0, when SBC is executed, R6 = R3 – R2 – 1 + C = 0x35 – 0x21 – 1 + 0 = 0x35 – 0x21 – 1= 0x13.

F412963B
- F62562FA

00000035
- 00000021

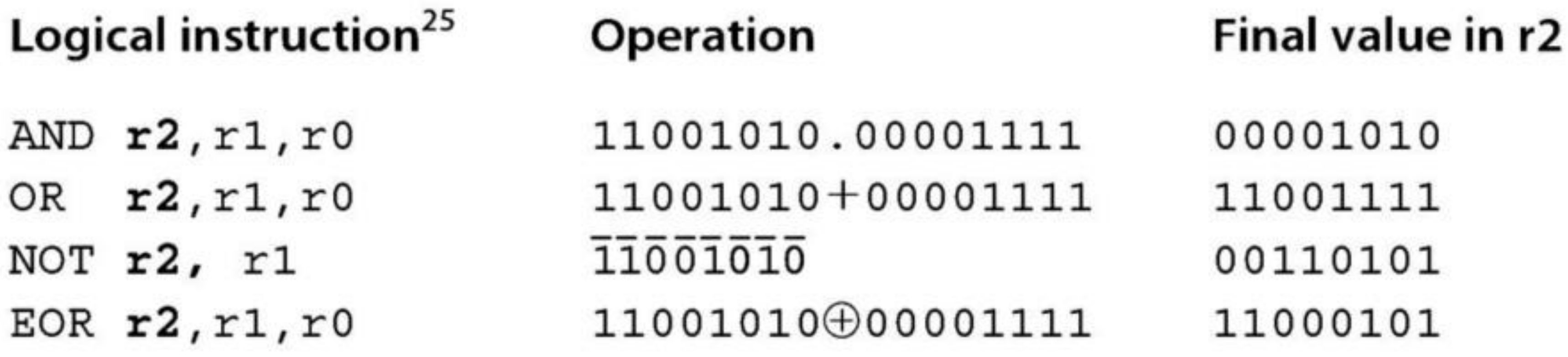
C = 0 so there is borrow

SBC R6,R3,R2 => R6 = C - 1 + R3 - R2
R6 = C -1+ R3 + (2's complement of R2)

SUBS R5,R1,R0 =>
R5 = R1 + (2's complement of R0)

```
                                + 0
                                - 1
    0    0    0    35                 F4   12   96   3B
+   FF   FF   FF   DF           +     09   DA   9D   06
    --------------------             ---------------------
    0    0    0    13           0     FD   ED   33   41
```

Bitwise logical operations

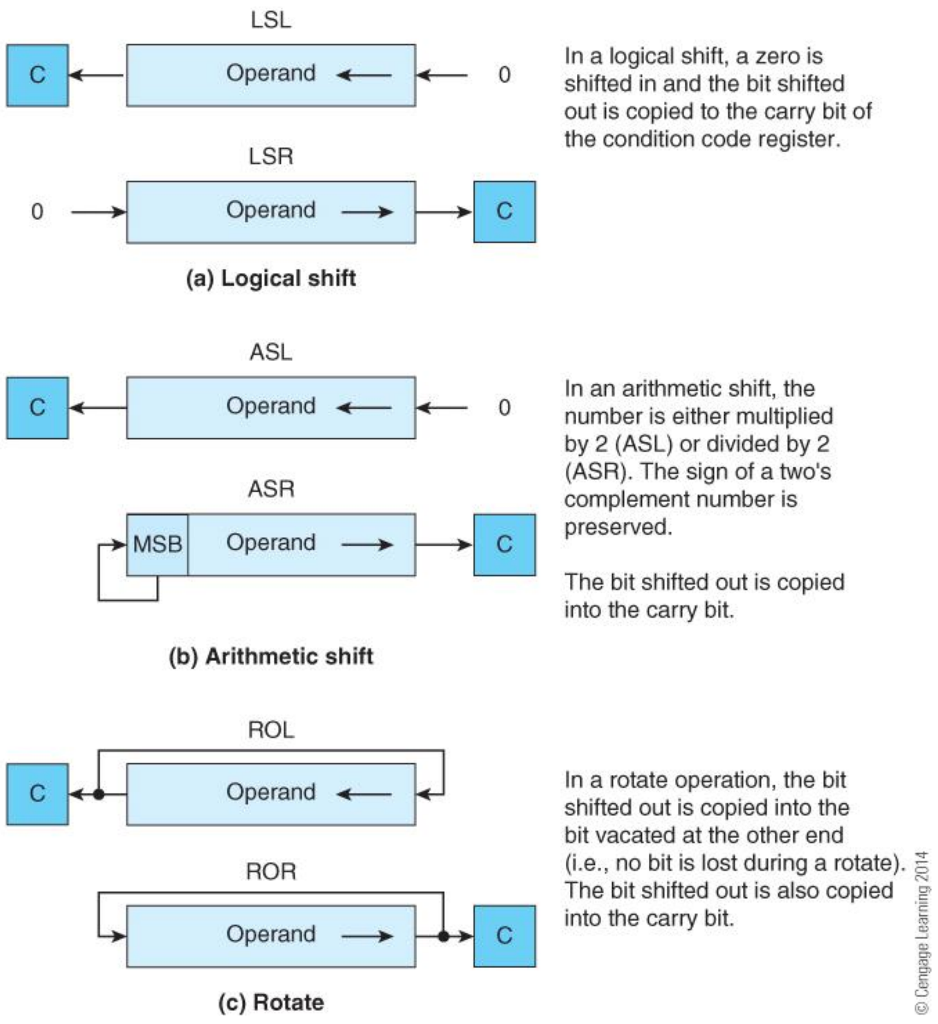


Logical instruction[25]	Operation	Final value in r2
`AND` **`r2`**`,r1,r0`	11001010.00001111	00001010
`OR` **`r2`**`,r1,r0`	11001010+00001111	11001111
`NOT` **`r2,`** `r1`	$\overline{11001010}$	00110101
`EOR` **`r2`**`,r1,r0`	11001010⊕00001111	11000101

- ARM does not have a NOT instruction of this form, but the NOT instruction can be implemented by one of the following two approaches:
  1. `mvn r2, r1`: mvn copies the logical complement of r1 into r2
  2. Apply EOR with the second operand equal to 0xFFFFFFFF. Note for each bit x, $x\ EOR\ 1\ = \bar{x}$

(i.e., 0 EOR 1 = 1
1 EOR 1 = 0)

Shift operations



(a) Logical shift

In a logical shift, a zero is shifted in and the bit shifted out is copied to the carry bit of the condition code register.

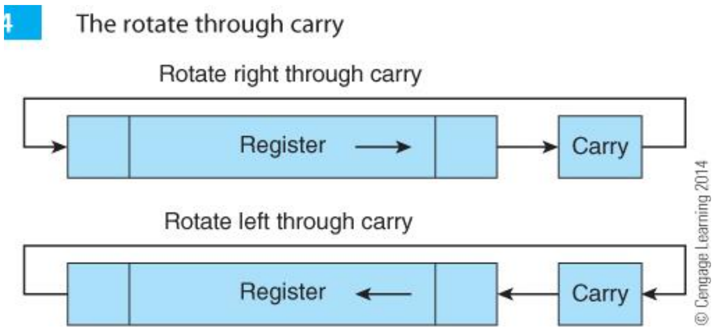
(b) Arithmetic shift

In an arithmetic shift, the number is either multiplied by 2 (ASL) or divided by 2 (ASR). The sign of a two's complement number is preserved.

The bit shifted out is copied into the carry bit.

(c) Rotate

In a rotate operation, the bit shifted out is copied into the bit vacated at the other end (i.e., no bit is lost during a rotate). The bit shifted out is also copied into the carry bit.



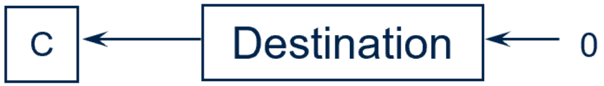
The rotate through carry

- In eight bits, if the carry C = 1 and the word to be shifted is 01101110, a rotate left through carry would give

- 11011101 and carry = 0

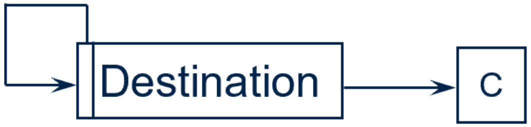
The Barrel Shifter in ARM

LSL : Logical Left Shift



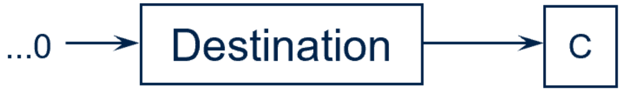
Multiplication by a power of 2

ASR: Arithmetic Right Shift



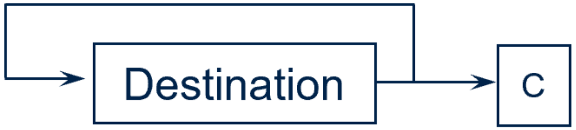
Division by a power of 2,
preserving the sign bit
e.g.,
10000100 = -7C$_{16}$= -124$_{10}$
11000010 = -3E$_{16}$= -62$_{10}$

LSR : Logical Shift Right



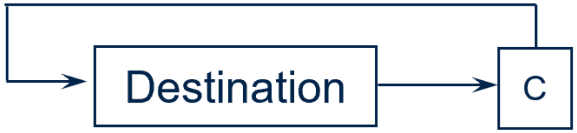
Division by a power of 2

ROR: Rotate Right



Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended



Single bit rotate with wrap around
from C to MSB

Using the Barrel shifter: The second operand

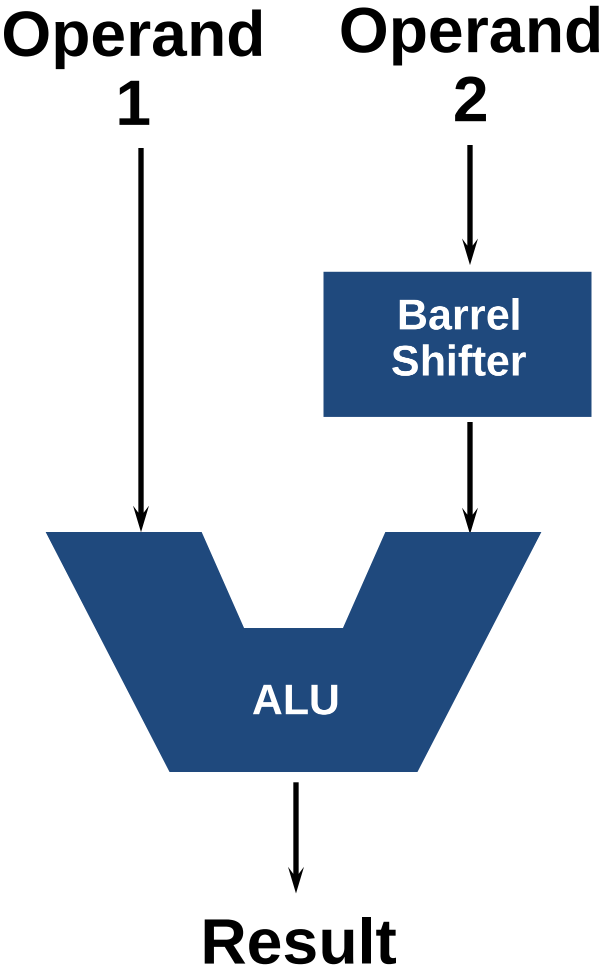
- ARM combines shifting with other data processing operations, because the second operand can be shifted before it is used. Consider:

```
ADD r0,r1,r2, LSL #1
```

- A logical shift left is applied to the contents of r2 before they are added to the contents of r1. This operation is equivalent to [r0] ← [r1] + [r2] x 2.

- Logical shift without other operations can be achieved by, e.g.:

```
MOV r0, r0, LSL #1
```



Static vs. Dynamic Shifts

Static shift

- The number of shifts is fixed in the code.
- e.g., `ADD r0,r1,r2, LSL #1` implements

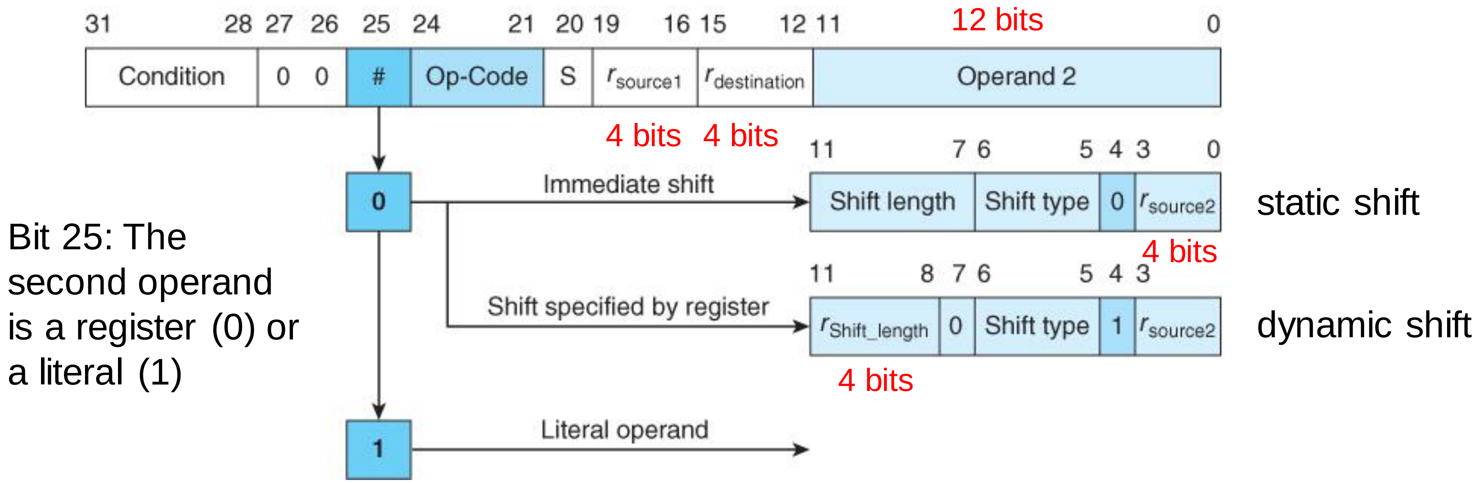
`[r0]←[r1]+[r2]x2`

Dynamic shift

- Consider `ADD r0,r1,r2, LSL r3`, which implements `[r0]←[r1]+[r2]x2`$^{[r3]}$
- This allows you to change the number of shifts at runtime.

Instruction Encoding

FIGURE 3.26 Encoding the ARM's data processing instructions



Bit 25: The second operand is a register (0) or a literal (1)

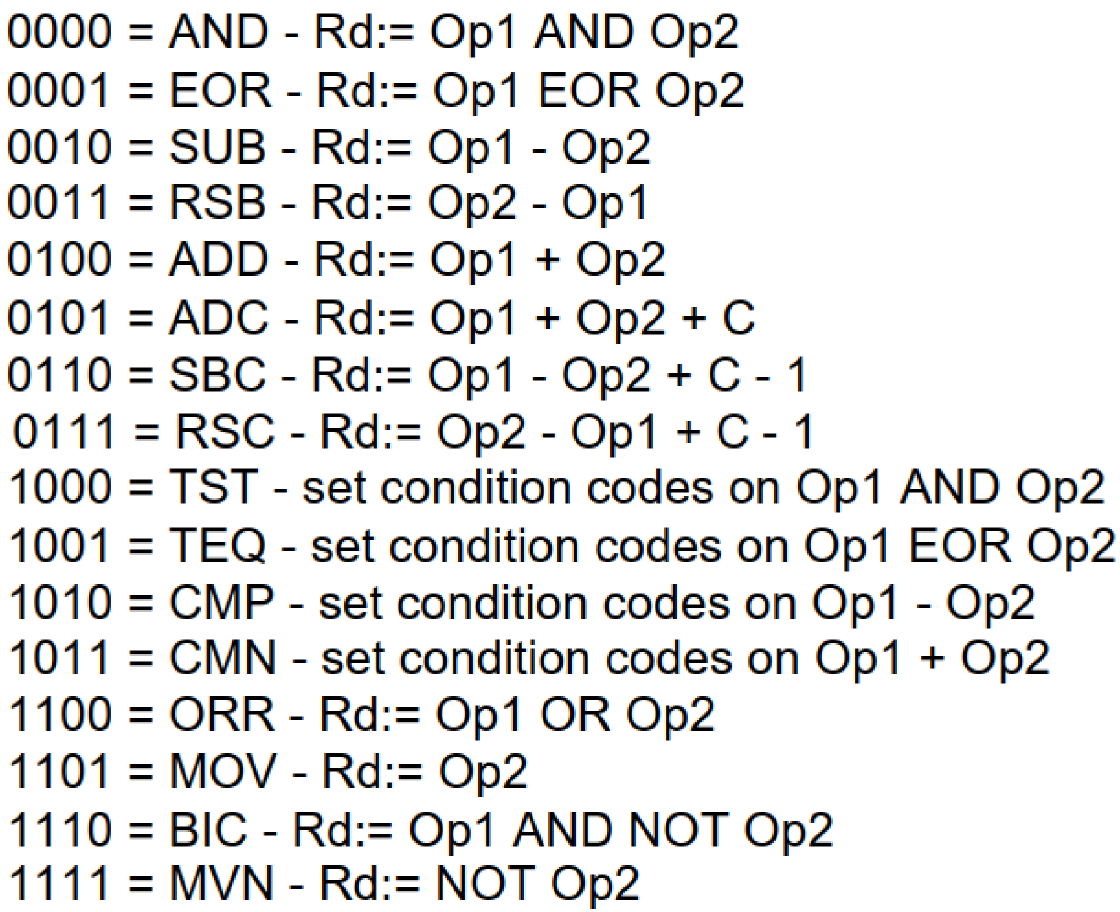
static shift

dynamic shift

- 4 bits for each of the destination and first source operand.
- 12 bits for the second operand consisting of:
  - shift type
  - static shift: 5-bit unsigned integer in Bit 7 to 11 specifies the number of shifts. Range is 0 to 31.
  - dynamic shift: 4-bit address of the register in Bit 8 to 11 specifies the register containing the number of shifts.

Instruction Encoding

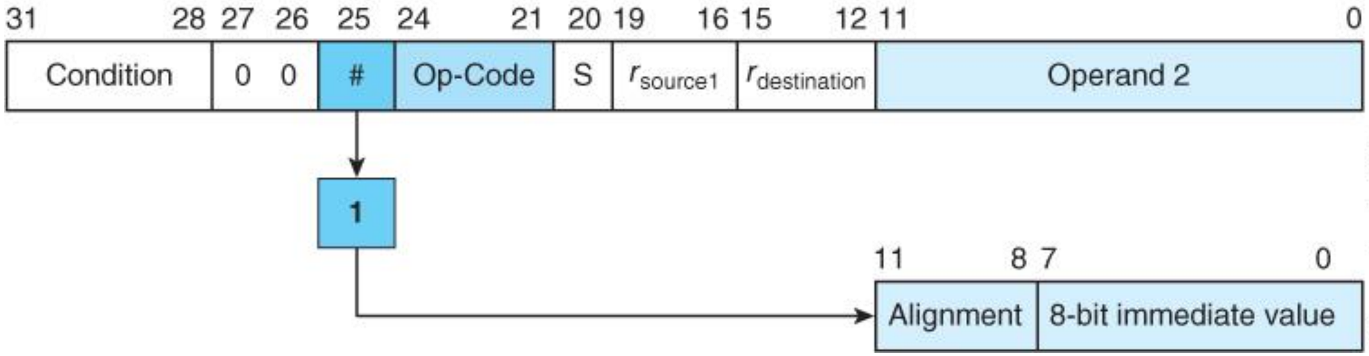
- Bit 21-24: Opcode

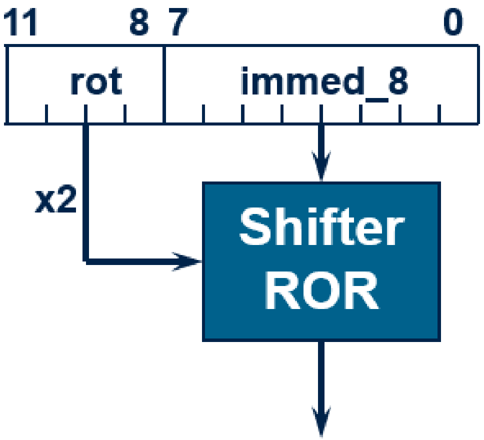


0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Literal Encoding

FIGURE 3.28 Diagram of ARM's literal operand encoding





- No ARM instruction can contain a 32-bit immediate constant
- The data processing instruction format has 12 bits available for operand2
- The literal processed by the instruction is the 8-bit value stored in Bits 0-7, shifted by ROR by 2 times the value stored in Bits 8-11.

Example of literal encoding

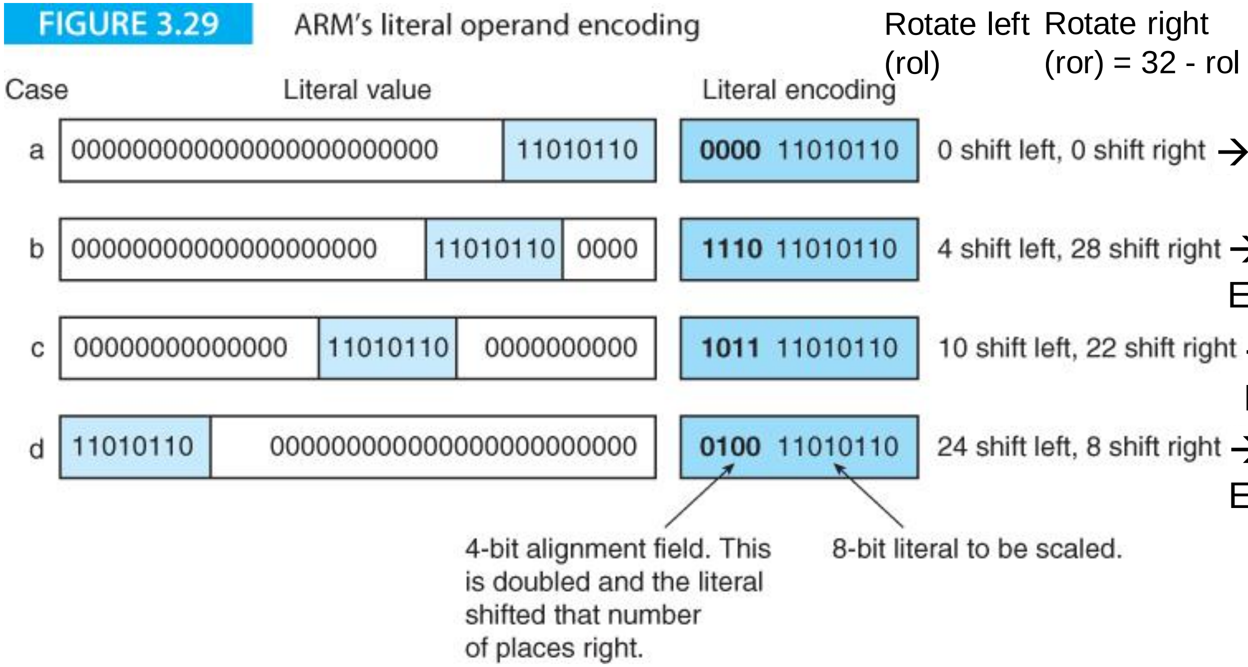


FIGURE 3.29 ARM's literal operand encoding

Case	Literal value	Literal encoding	Rotate left (rol) Rotate right (ror) = 32 - rol	Encode ror shift divided by 2
a	000000000000000000000000 11010110	0000 11010110	0 shift left, 0 shift right →	Encode 0
b	00000000000000000000 11010110 0000	1110 11010110	4 shift left, 28 shift right →	28 = 2 x14 Encode 14 decimal or E in hex
c	00000000000000 11010110 0000000000	1011 11010110	10 shift left, 22 shift right →	22 = 2 x11 Encode 11 decimal or B in hex
d	11010110 000000000000000000000000	0100 11010110	24 shift left, 8 shift right →	8 = 2 x 4 Encode 4 decimal or 4 in hex

4-bit alignment field. This is doubled and the literal shifted that number of places right.

8-bit literal to be scaled.

Example of literal encoding

- The assembler converts immediate values to the rotate form:
  - `MOV r0,#4096; uses 0x01, left shift 12, ror 20, encode 10 or A`

    `machine code: E3A00A01`
  - `ADD r1,r2,#0xFF0000; uses 0xFF, left shift 16, ror 16, encode 8`

    `machine code: E28218FF`
- Should be able to convert least significant 12 bit of machine code to the literal being encoded
- The MVN (move negative) instruction MVN r0, #XX can specify unshifted constant in the range 0xFFFFFF00 to 0XFFFFFFFF. e.g.,
  - `MOV r0, #0xFFFFFFFF  ;assembles to MVN r0,#0`
- Values that cannot be generated in this way will cause an error.

More examples on literal encoding

Determine the literal operand encoded by the following 32-bit machine code. Express your answer in hexadecimal representation.

(a) E3A00C67

(b) E2810E89

(c) E283260F

(d) E3A04807

More examples on literal encoding

Determine the literal operand encoded by the following 32-bit machine code. Express your answer in hexadecimal representation.

(a) E3A00C67

Encoded C → ror $12 \times 2 = 24$ → left shift = 32-24 = 8.

Literal = 67 shifted by 8 bits to the left = 6700

(b) E2810E89

Encoded E → ror $14 \times 2 = 28$ → left shift = 32-28 = 4.

Literal = 89 shifted by 4 bits to the left = 890

(c) E283260F

Encoded 6 → ror $6 \times 2 = 12$ → left shift = 32-12 = 20.

Literal = 0F shifted by 20 bits to the left = F00000

(d) E3A04807

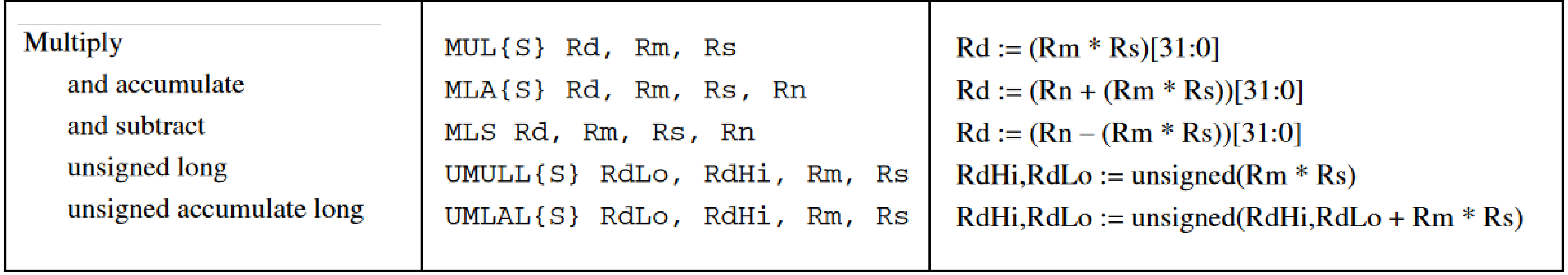
Encoded 8 → ror $8 \times 2 = 16$ → left shift = 32-16 = 16.

Literal = 07 shifted by 16 bits to the left = 70000

Multiplication

- Take two m-bit operands → output a 2m-bit result.
- Multiplication would not be correct using 2's complement values.
- Cannot use the same multiplication operation in signed and unsigned values. Focus on unsigned here.
- For more information: https://bohr.wlu.ca/cp216/docs/QRC0001_UAL.pdf

Unsigned multiplications



Operation	Assembler	Action
Multiply	`MUL{S} Rd, Rm, Rs`	Rd := (Rm * Rs)[31:0]
and accumulate	`MLA{S} Rd, Rm, Rs, Rn`	Rd := (Rn + (Rm * Rs))[31:0]
and subtract	`MLS Rd, Rm, Rs, Rn`	Rd := (Rn – (Rm * Rs))[31:0]
unsigned long	`UMULL{S} RdLo, RdHi, Rm, Rs`	RdHi,RdLo := unsigned(Rm * Rs)
unsigned accumulate long	`UMLAL{S} RdLo, RdHi, Rm, Rs`	RdHi,RdLo := unsigned(RdHi,RdLo + Rm * Rs)

- Cannot use the same register to specify both `Rd` and `Rm`.
- MUL, MLA and MLS have the multiplication results truncated to the lower-order 32 bits.
- UMULL and UMLAL store the full 64-bit result into two registers `RdLo` and `RdHi`.

3.3 ARM Flow Control Instructions

Unconditional branch

- Format: `B target`, where `target` denotes the branch target address (the address of the next instruction to be executed).
- Example:

```
..   do this    Some code
..   then that  Some other code
     B    Next  Now skip past next instructions
..         …the code being skipped past
..         …the code being skipped past
Next ..     Target address for the branch
```

Conditional branch

```
IF (X == Y)
    THEN Y = Y + 1;
    ELSE Y = Y + 2
```

- A test is performed and one of two courses of action is carried out depending on the outcome. We can translate this as:

```
      CMP r1,r2     ;r1 contains y and r2 contains x: compare them
      BNE Plus2     ;if not equal, then branch to the else part
      ADD r1,r1,#1  ;if equal, fall through to here and + 1 to y
      B   leave     ;now skip past the else part
Plus2 ADD r1,r1,#2  ;ELSE part add 2 to y
leave …             ;continue from here
```

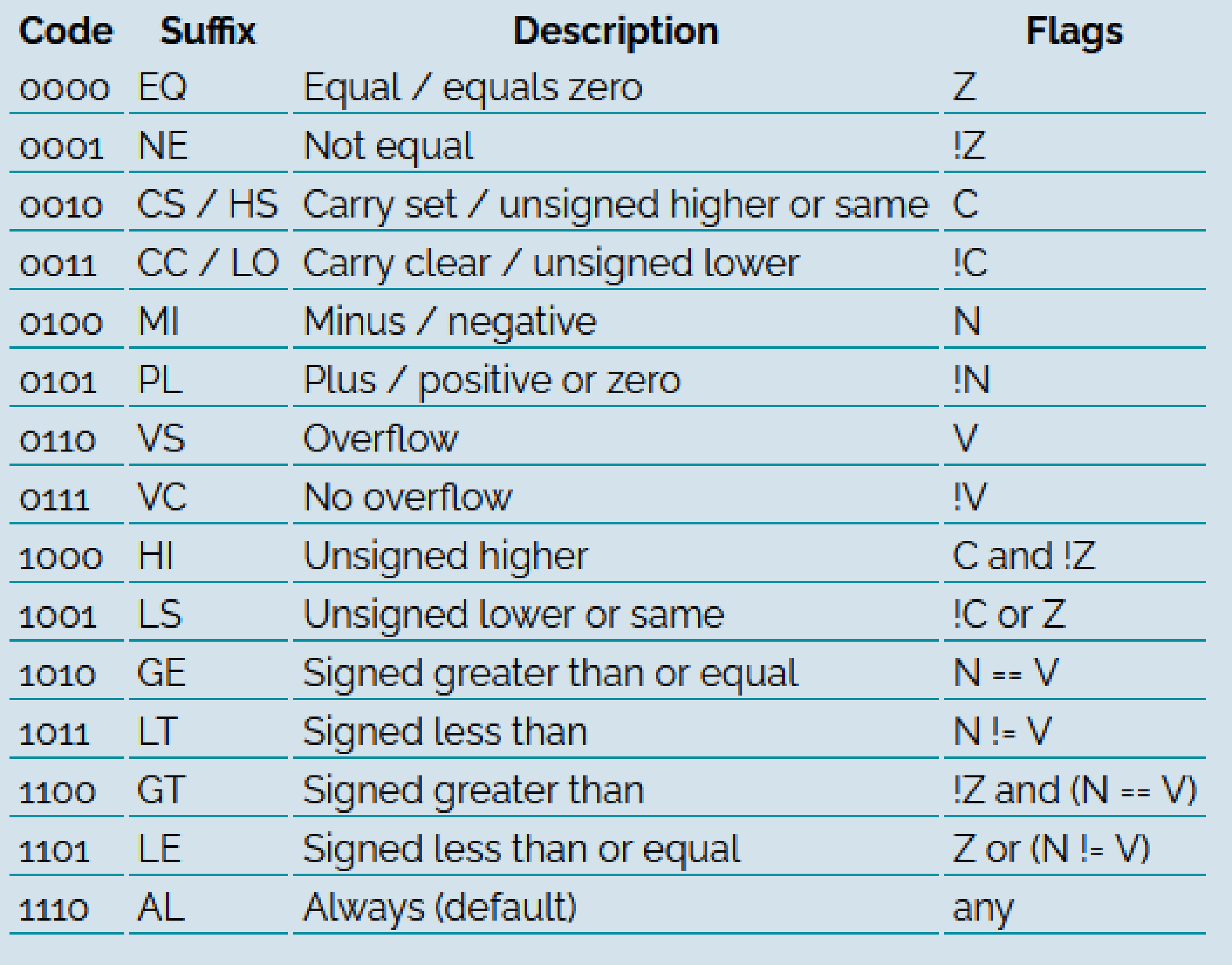
- The *conditional branch* instruction tests flag bits in the processor's CPSR, then takes the branch if the tested condition is true.

- Here `CMP` computes `[r1]-[r2]` and update the CPSR, the conditional branch tests for the condition NE (not equal), corresponding to !Z, i.e., condition is true when Z = 0.

- `CMP` always updates the CPSR flags (i.e., no S suffix is needed)

Condition Codes

The code of possible conditions is tabulated here:



Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

Branching for signed and unsigned data

- Branch instruction can refer to signed and unsigned data
- Consider the example:
    - [r1] = 0x90000075
    - [r2] = 0x50000075
    - Which value is larger? The answer depends on whether **you interpret** the number as signed or unsigned numbers.
    - if interpreted as unsigned numbers, [r1] > [r2].
    - if interpreted as signed numbers, [r1] is a negative number and [r2] is a positive number. Thus [r2] > [r1].

Branching for signed and unsigned data

- The computer does not know whether you want to interpret the comparison as signed and unsigned.
- When executing `CMP r1, r2`, the computer would just perform [r1] – [r2] and update CPSR.

```
  90000075                    90000075
- 50000075                  +AFFFFF8B (2’s complement)
                            1 40000000
```

CPSR: C = 1, N = 0, V = 1 (-ve + -ve = +ve), Z = 0

- If programmer decides to test [r1]>[r2] and consider the operands as *unsigned numbers* → Use BHI (Unsigned higher)
  - BHI branches as the condition C and !Z are satisfied
- If programmer decides to test [r1]>[r2] and consider the operands as *signed numbers* → Use BGT (Signed greater than)
  - BGT does not take the branch as the condition N != V

Test instruction: TST

Two test instructions that explicitly update CPSR:

```
TST Rn, Operand2
```

- updates CPSR on Rn AND Operand2
- Useful in testing whether an individual bit of a word is 1.
- E.g., Bit 5 of a lower-case ASCII character is set to 1. To test whether an ASCII character in r0 is lower-case:

```
TST r0, #2_0010000
BNE LowerCase // jump to code handling lower case
```

  - If lower-case, then the result is non-zero (i.e., Z = 0) → Branch taken
  - If capital, result is zero (i.e., Z = 1) → Branch not taken

Test instruction: TEQ

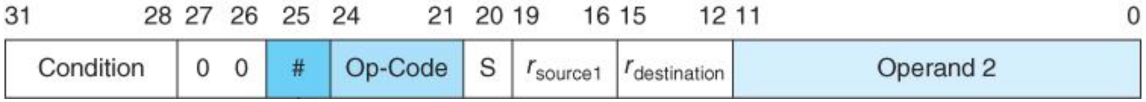
```
TEQ Rn, <Operand2>
```

- updates CPSR on Rn EOR Operand2
- Z-bit is high if [Rn] == [Operand2]
  - [Rn] == [Operand2]: All bits in the result are 0 (i.e., Z = 1)
  - [Rn] != [Operand2]: At least one bit in the result are 1 (i.e., Z = 0)
- Similar to CMP but TEQ does not update the overflow flag (no concept of V in logical operations).
- Only used to test equivalence, not less/larger than conditions.

Conditional Execution

- In ARM, each instruction is conditionally executed (i.e., an instruction is executed only if a condition after the conditional code list is satisfied)
- So far, we have only dealt with the default case *always*:
  - – ADD r0, r1, r2 means ADDAL r0, r1, r2
- Bits 28-31 in Fig. 3.26 are used to encode the condition.

FIGURE 3.26 Encoding the ARM's data processing instructions



31 28	27 26	25	24 21	20	19 16	15 12	11 0
Condition	0 0	#	Op-Code	S	$r_{source1}$	$r_{destination}$	Operand 2

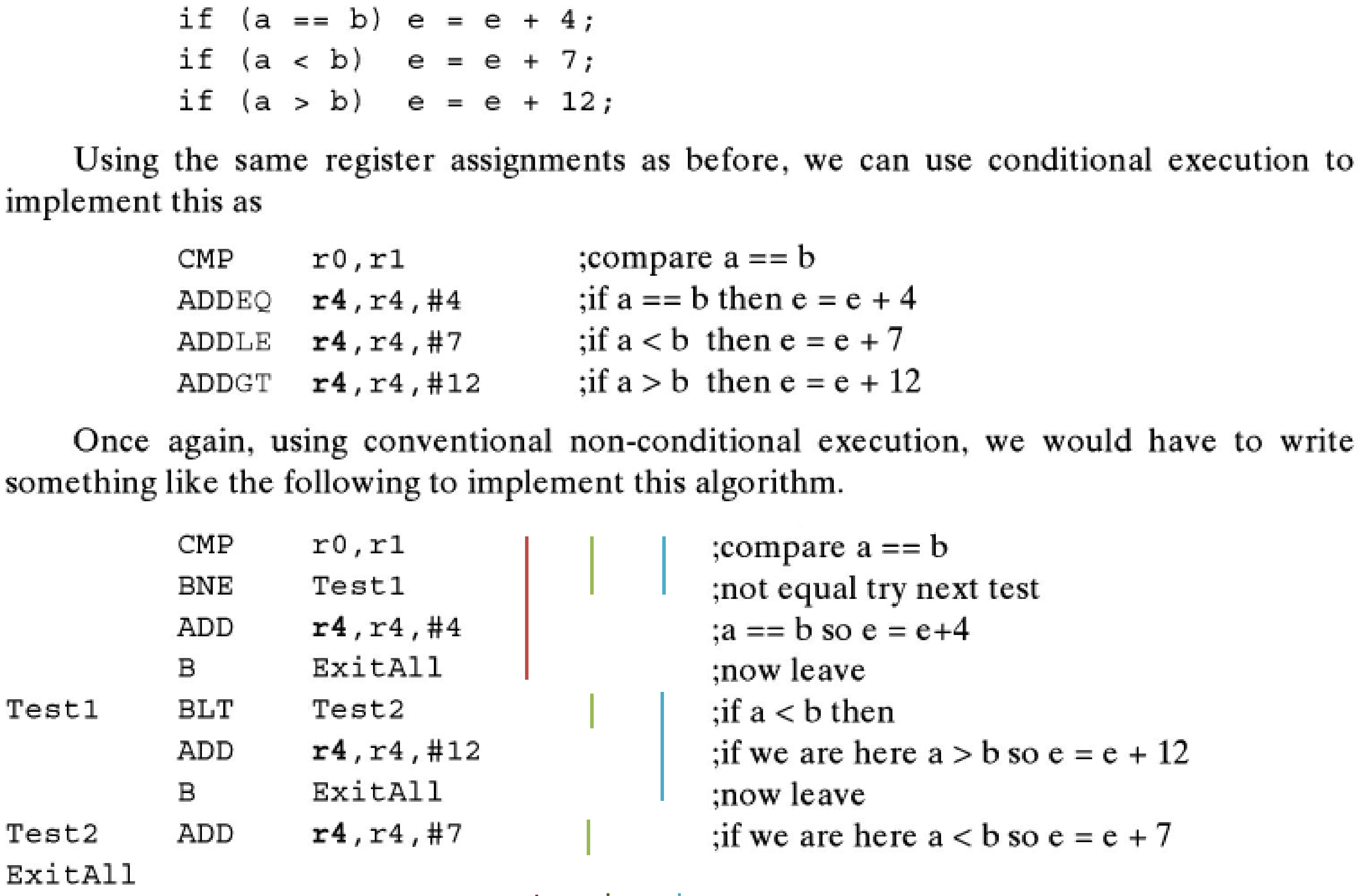
- e.g., if `Z = 1 then [r0] ←[r1]+[r2]` can be implemented as: `ADDEQ r0, r1, r2`

Example 1

- This improves code density and performance by reducing the number of forward branch instructions.
- E.g., if (a == 0); b = c + d

```
CMP   r3,#0                 CMP   r3,#0
BNE   skip                  ADDEQ r0,r1,r2
ADD   r0,r1,r2
skip
```

Example 2



```
if (a == b) e = e + 4;
if (a < b)  e = e + 7;
if (a > b)  e = e + 12;
```

Using the same register assignments as before, we can use conditional execution to implement this as

```
CMP     r0,r1          ;compare a == b
ADDEQ   r4,r4,#4       ;if a == b then e = e + 4
ADDLE   r4,r4,#7       ;if a < b  then e = e + 7
ADDGT   r4,r4,#12      ;if a > b  then e = e + 12
```

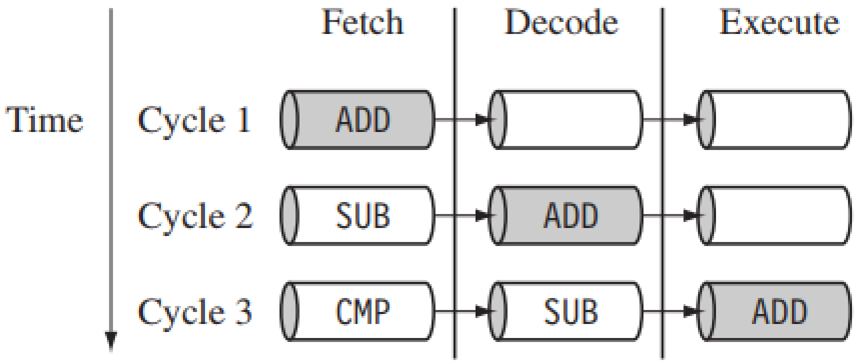
Once again, using conventional non-conditional execution, we would have to write something like the following to implement this algorithm.

```
        CMP     r0,r1          ;compare a == b
        BNE     Test1          ;not equal try next test
        ADD     r4,r4,#4       ;a == b so e = e+4
        B       ExitAll        ;now leave
Test1   BLT     Test2          ;if a < b then
        ADD     r4,r4,#12      ;if we are here a > b so e = e + 12
        B       ExitAll        ;now leave
Test2   ADD     r4,r4,#7       ;if we are here a < b so e = e + 7
ExitAll
```

a==b a<b a>b

Pipelining

- ARM7 has a three-stage pipeline:
  - fetch
  - decode
  - execute

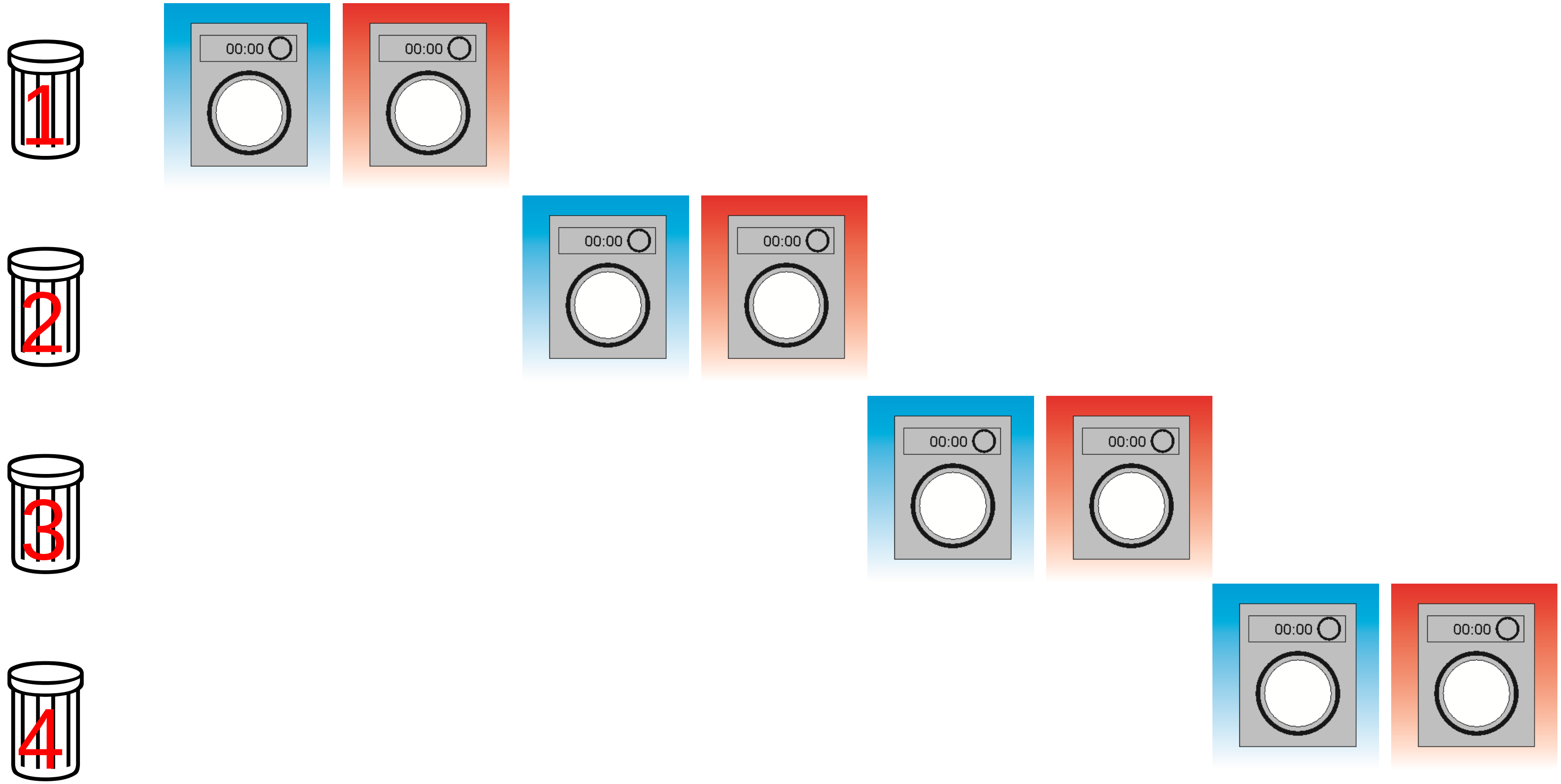


- When ADD is being executed (Cycle 3), the pc is pointing to CMP, which is two instructions (8 bytes) below ADD.
- pc = executed address + 8
- For our example, when ADD is executed:
  - pc = 0x100C + 8 = 0x1014

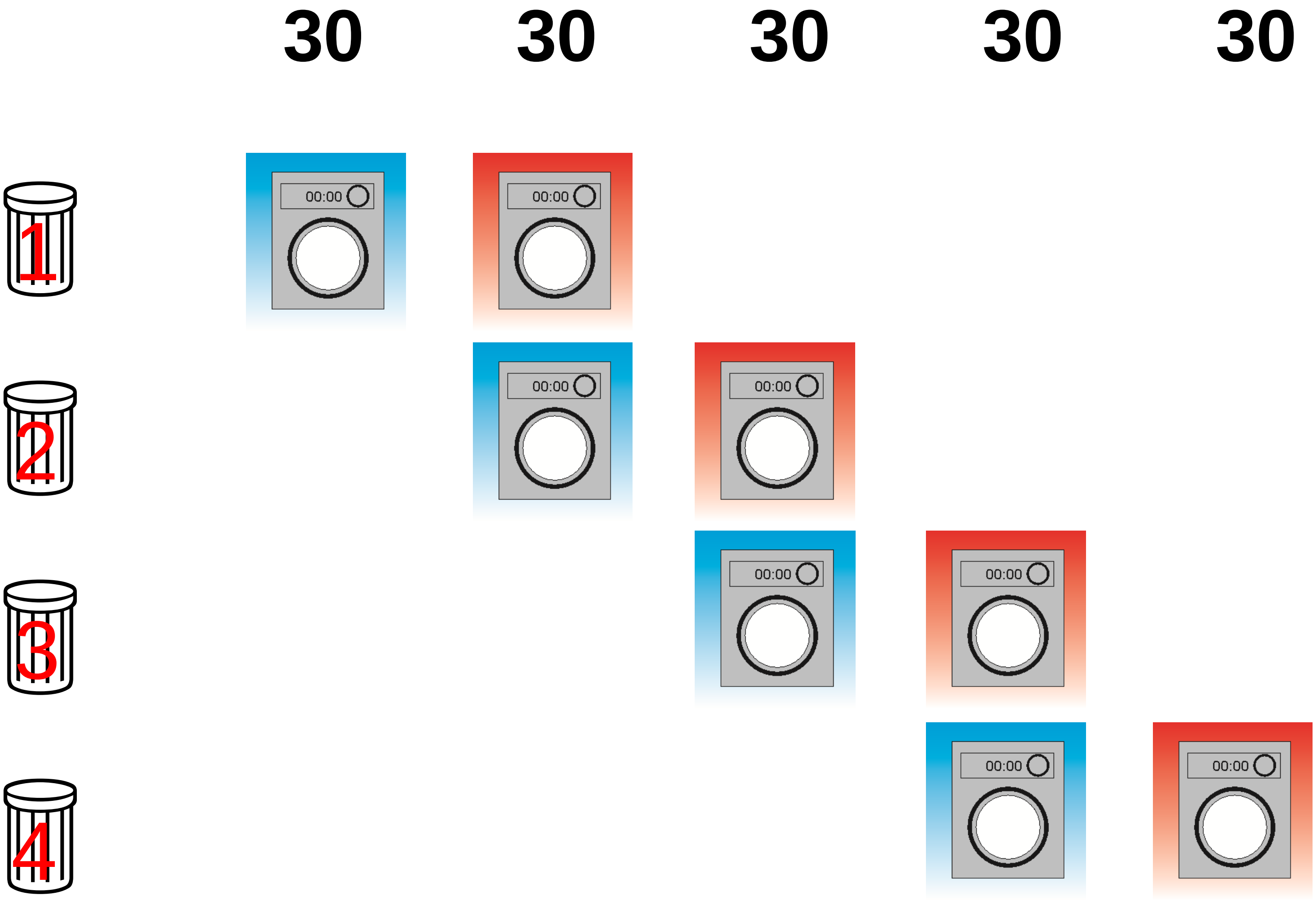
	Address	Instruction
	100C	ADD
	1010	SUB
pc →	1014	CMP

No Pipelining

30 30 30 30 30 30 30 30



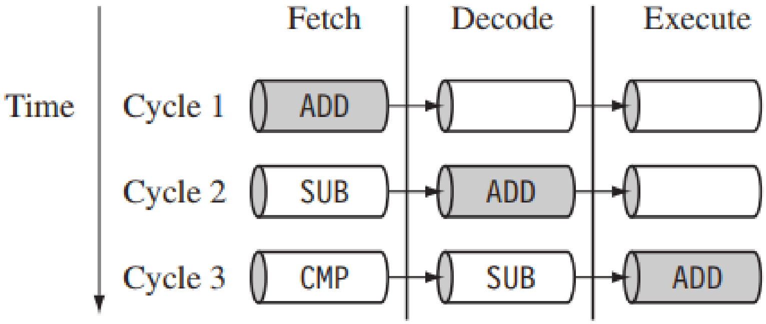
With Pipelining



30 30 30 30 30

Pipelining

- When ADD is being executed (Cycle 3), the PC is pointing to CMP, which is two instructions (8 bytes) below ADD.



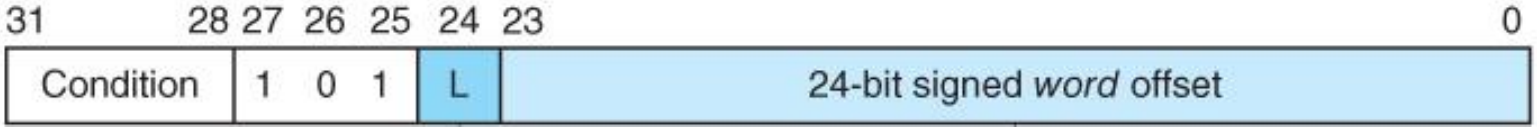
- PC = executed address + 8
- This has an effect on address encoding, as described below.

	Address	Instruction
	100C	ADD
	1010	SUB
pc →	1014	CMP

Branch instruction encoding

- How does the machine code encode the destination address of the branching operation?

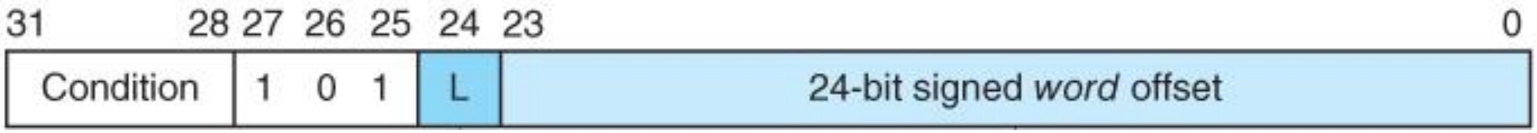
FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions



31 28	27	26	25	24	23 0
Condition	1	0	1	L	24-bit signed *word* offset

Branch instruction encoding

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions

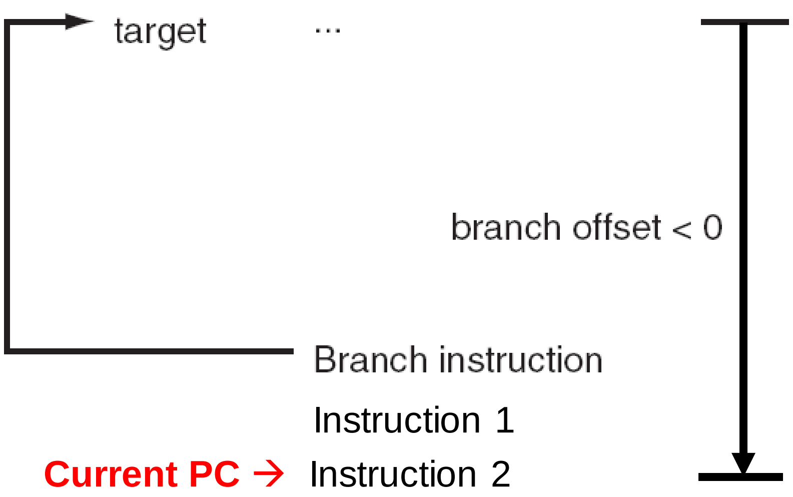


31 28	27	26	25	24	23 0
Condition	1	0	1	L	24-bit signed *word* offset

- The 24-bit signed offset stores the number of instructions of the destination instruction with respect to the current PC.
- Note each instruction has 4 bytes.
- Thus, the 24-bit offset is shifted left twice to convert the instruction offset to byte offset → 26-bit signed offset
- Range of offset: -32 Mbytes to +32 Mbytes or -8 M instructions to +8 M instructions

26-bit signed: -2^{25} to $2^{25}-1$ bytes 24-bit signed: -2^{23} to $2^{23}-1$ instructions 1M = 2^{20}

Branch offset



target ...

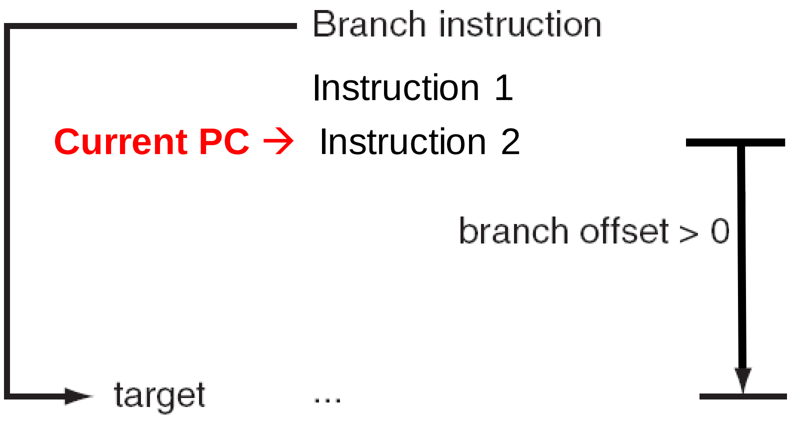
branch offset < 0

Branch instruction

Instruction 1

Current PC → Instruction 2

(a) Backward branch (-ve offset)



Branch instruction

Instruction 1

Current PC → Instruction 2

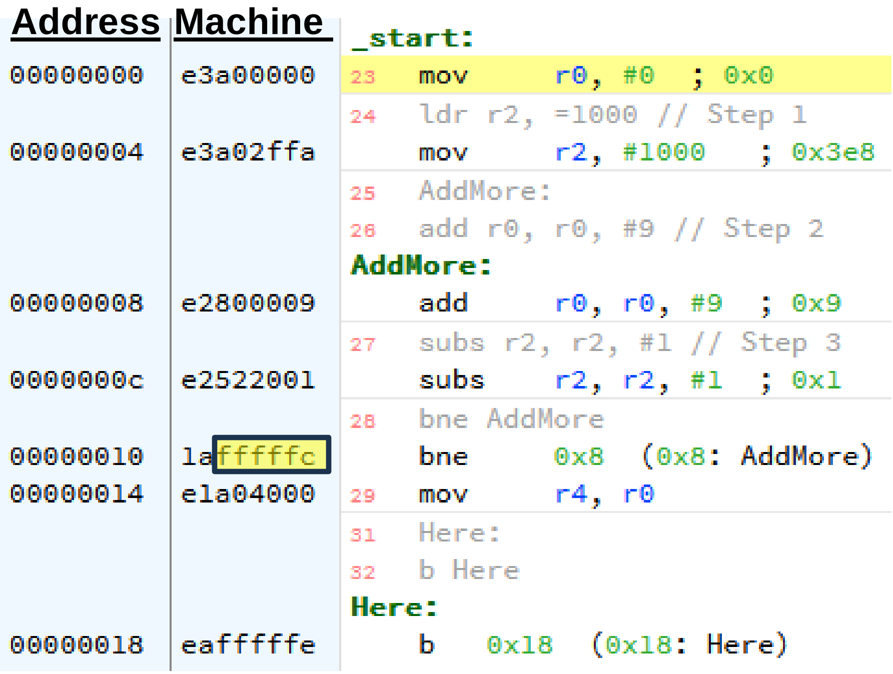
branch offset > 0

target ...

(a) Forward branch (+ve offset)

- Note that due to pipelining, when a branch instruction is executed, the PC points to two instructions below the branching instruction.

Example 1



```
Address   Machine
                      _start:
00000000  e3a00000       mov       r0, #0   ; 0x0
                         ldr r2, =1000 // Step 1
00000004  e3a02ffa       mov       r2, #1000    ; 0x3e8
                         AddMore:
                         add r0, r0, #9 // Step 2
                      AddMore:
00000008  e2800009       add       r0, r0, #9   ; 0x9
                         subs r2, r2, #1 // Step 3
0000000c  e2522001       subs      r2, r2, #1   ; 0x1
                         bne AddMore
00000010  1afffffc       bne       0x8   (0x8: AddMore)
00000014  e1a04000       mov       r4, r0
                         Here:
                         b Here
                      Here:
00000018  eafffffe       b    0x18   (0x18: Here)
```

2's complement

```
  000004
  FFFFFB
+      1
  FFFFFC
```

Encoded address

$$= \frac{Dest - Current\ PC}{4}$$

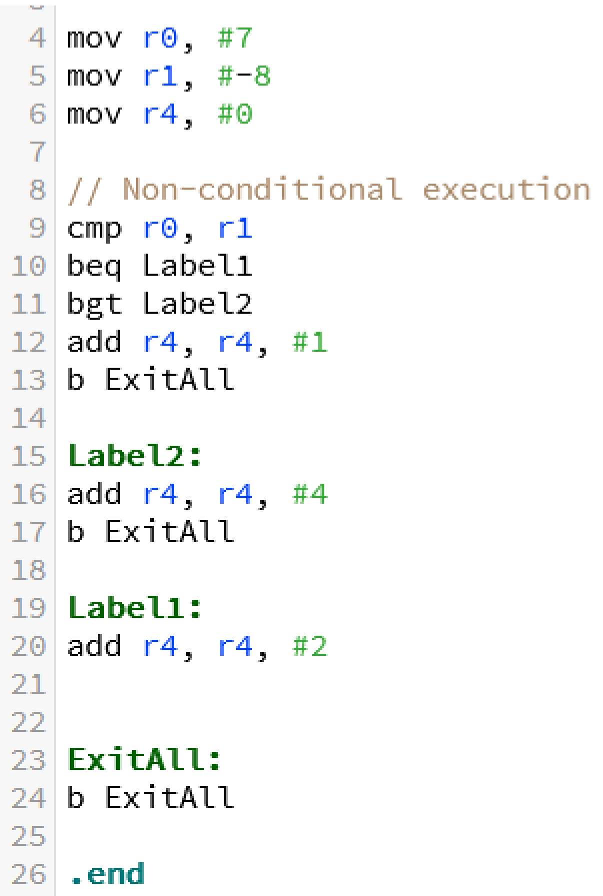
- Dest = 0x08
- Current PC = 0x18 (two instructions beyond bne)

Encoded address = $\frac{0x08-0x18}{4} = -4$

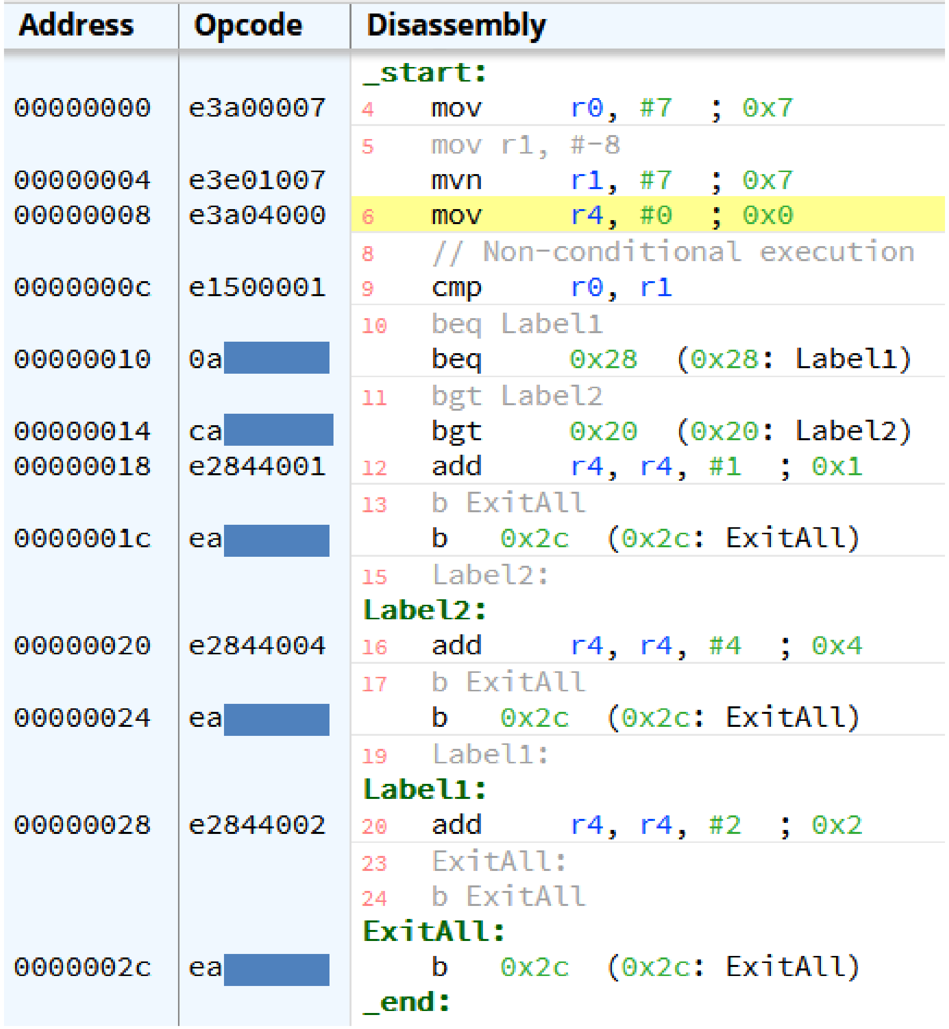
= 0xFFFFFC

Example 2

Determine the encoded destination addresses in the covered machine code:

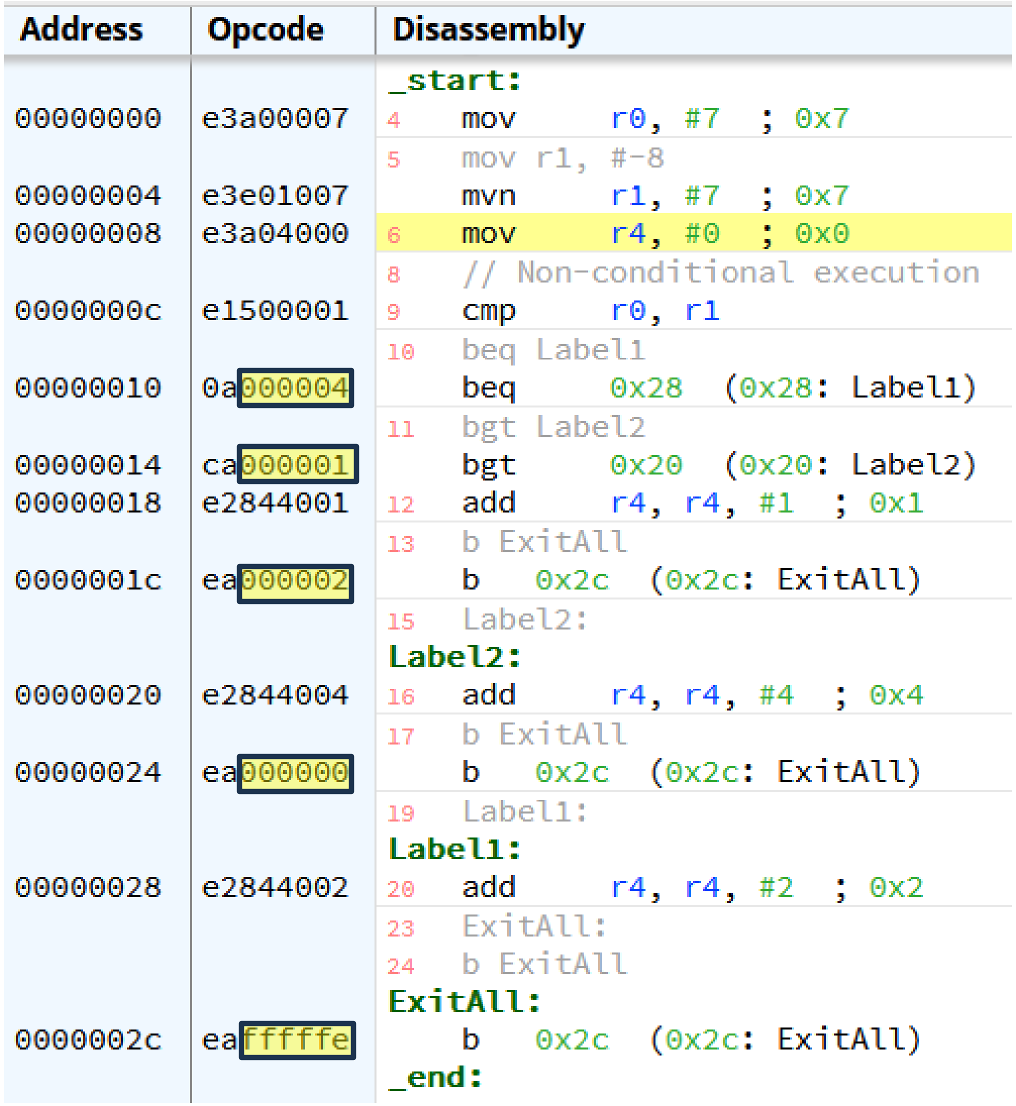


```
mov r0, #7
mov r1, #-8
mov r4, #0

// Non-conditional execution
cmp r0, r1
beq Label1
bgt Label2
add r4, r4, #1
b ExitAll

Label2:
add r4, r4, #4
b ExitAll

Label1:
add r4, r4, #2


ExitAll:
b ExitAll

.end
```



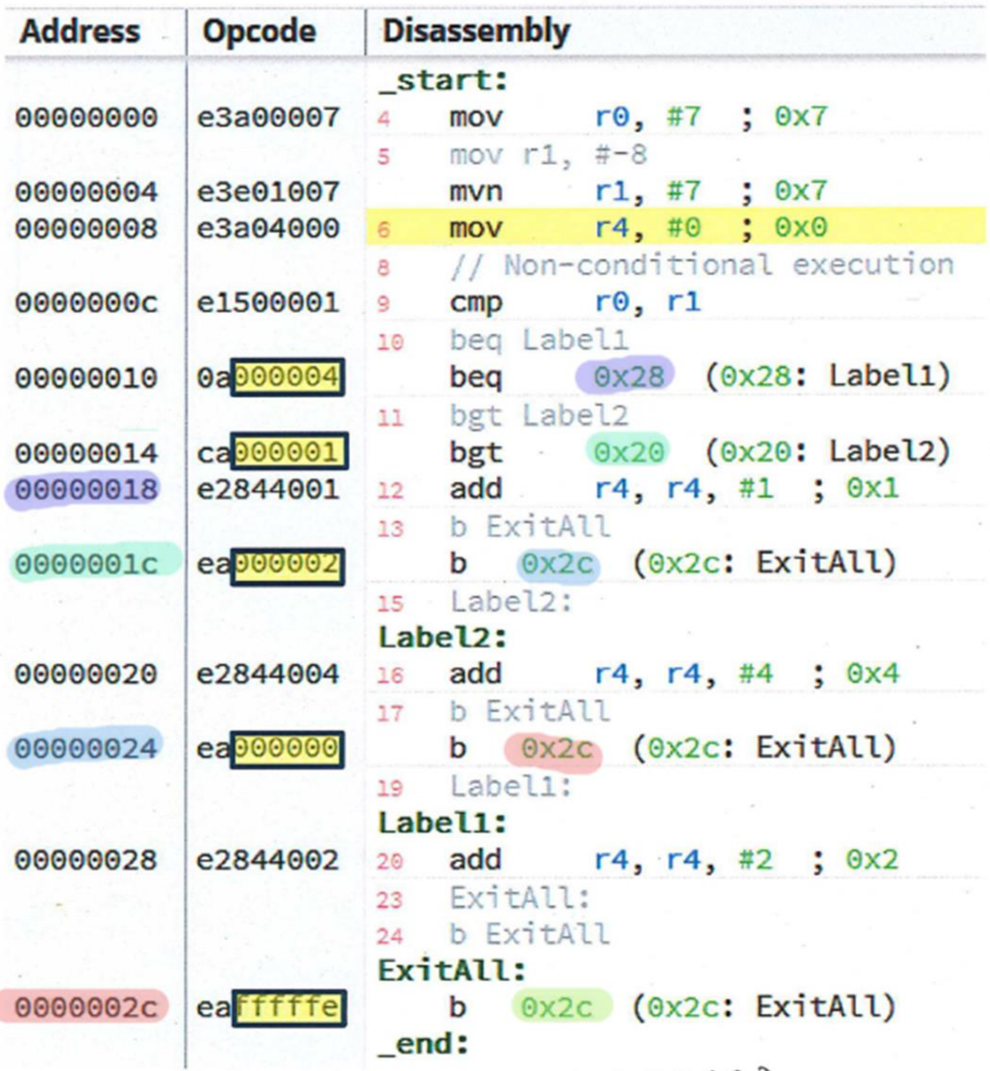
```
Address    Opcode       Disassembly
                        _start:
00000000   e3a00007         mov      r0, #7    ; 0x7
                            mov r1, #-8
00000004   e3e01007         mvn      r1, #7    ; 0x7
00000008   e3a04000         mov      r4, #0    ; 0x0
                            // Non-conditional execution
0000000c   e1500001         cmp      r0, r1
                            beq Label1
00000010   0a[redacted]     beq      0x28   (0x28: Label1)
                            bgt Label2
00000014   ca[redacted]     bgt      0x20   (0x20: Label2)
00000018   e2844001         add      r4, r4, #1   ; 0x1
                            b ExitAll
0000001c   ea[redacted]     b    0x2c   (0x2c: ExitAll)
                            Label2:
                        Label2:
00000020   e2844004         add      r4, r4, #4   ; 0x4
                            b ExitAll
00000024   ea[redacted]     b    0x2c   (0x2c: ExitAll)
                            Label1:
                        Label1:
00000028   e2844002         add      r4, r4, #2   ; 0x2
                            ExitAll:
                            b ExitAll
                        ExitAll:
0000002c   ea[redacted]     b    0x2c   (0x2c: ExitAll)
                        _end:
```

Example 2



```
Address   Opcode     Disassembly
                     _start:
00000000  e3a00007       mov      r0, #7   ; 0x7
                         mov r1, #-8
00000004  e3e01007       mvn      r1, #7   ; 0x7
00000008  e3a04000       mov      r4, #0   ; 0x0
                         // Non-conditional execution
0000000c  e1500001       cmp      r0, r1
                         beq Label1
00000010  0a000004       beq      0x28   (0x28: Label1)
                         bgt Label2
00000014  ca000001       bgt      0x20   (0x20: Label2)
00000018  e2844001       add      r4, r4, #1   ; 0x1
                         b ExitAll
0000001c  ea000002       b    0x2c   (0x2c: ExitAll)
                         Label2:
                     Label2:
00000020  e2844004       add      r4, r4, #4   ; 0x4
                         b ExitAll
00000024  ea000000       b    0x2c   (0x2c: ExitAll)
                         Label1:
                     Label1:
00000028  e2844002       add      r4, r4, #2   ; 0x2
                         ExitAll:
                         b ExitAll
                     ExitAll:
0000002c  eafffffe       b    0x2c   (0x2c: ExitAll)
                     _end:
```

Example 2



```
Address    Opcode     Disassembly
                      _start:
00000000   e3a00007       mov      r0, #7   ; 0x7
                          mov r1, #-8
00000004   e3e01007       mvn      r1, #7   ; 0x7
00000008   e3a04000       mov      r4, #0   ; 0x0
                          // Non-conditional execution
0000000c   e1500001       cmp      r0, r1
                          beq Label1
00000010   0a000004       beq      0x28   (0x28: Label1)
                          bgt Label2
00000014   ca000001       bgt      0x20   (0x20: Label2)
00000018   e2844001       add      r4, r4, #1   ; 0x1
                          b ExitAll
0000001c   ea000002       b     0x2c   (0x2c: ExitAll)
                          Label2:
                      Label2:
00000020   e2844004       add      r4, r4, #4   ; 0x4
                          b ExitAll
00000024   ea000000       b     0x2c   (0x2c: ExitAll)
                          Label1:
                      Label1:
00000028   e2844002       add      r4, r4, #2   ; 0x2
                          ExitAll:
                          b ExitAll
                      ExitAll:
0000002c   eafffffe       b     0x2c   (0x2c: ExitAll)
                      _end:
```

00000034 // add 8 from address to get PC

000002
(-2) FFFFFD
+ 1
Encode: FFFFFE

Address 0x10

Dest = 0x28, Current PC = 0x18

$R = \frac{0x28 - 0x18}{4} = 4$

Encode: $\boxed{000004}$

Address 0x14

Dest = 0x20, Current PC = 0x1C

$R = \frac{0x20 - 0x1C}{4} = 1$

Encode: $\boxed{000001}$

Address 0x1C

Dest = 0x2c, Current PC = 0x24

$R = \frac{0x2C - 0x24}{4} = 2$

Encode: $\boxed{000002}$

Address 0x24

Dest = 0x2C, Current PC =0x2C

$R = \frac{0x2C - 0x2C}{4} = 0$

Encode: $\boxed{000000}$

Address 0x2C

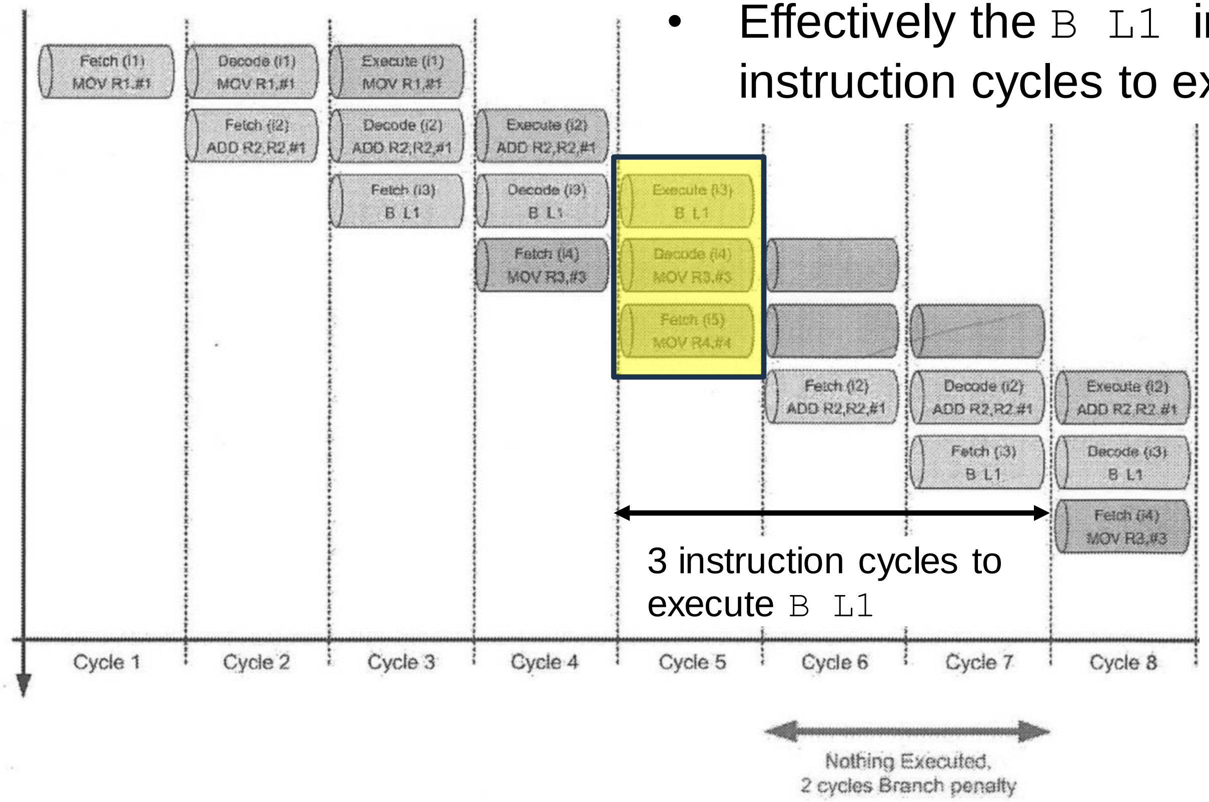
Dest = 0x2C, Current PC = 0x34

$R = \frac{0x2C - 0x34}{4} = -2$

Branch penalty

```
MOV R1, #1
L1:
ADD R2, R2, #1
B L1
MOV R3, #3
MOV R4, #4
```

- First instruction takes 3 cycles to pass through the stages of the pipeline
- Thereafter, instructions take one cycle to execute
- The `B L1` instruction changes the sequential order of program execution.
- The instruction in the fetch and decode stages must be dumped (or flushed).
- Effectively the `B L1` instruction takes 3 instruction cycles to execute.



3 instruction cycles to execute `B L1`

Example

```
       mov r0, #255
Again: subs r0, r0, #1 //1 cycle
       bne Again       //1 cycle if not
                          branch, 3 if branch
```

How many instruction cycles are generated by the Again loop?

	[r0]	**Instruction Cycles**	
1st repetition	255 → 254	4	254 x 4
2nd repetition	254 → 253	4	
⋮	⋮	⋮	
255th repetition	1 → 0	2 (bne branch not taken)	
	Total	1018	