Instruction Encoding

Digital Computer Design

How to Encode Instructions?

- Assembly language is convenient for humans to read.
  - However, digital circuits understand only 1’ s and 0’ s.
- A program written in assembly language is to a representation using only 1’ s and 0’ s
  - called **Machine Language**.
- ARM uses a single instruction format
  - **32-bit instructions**
    - Variable-length instructions are avoided
      - would add complexity.

Design Principle 4

ARM defines three main instruction formats:

- **(1) Data-processing**
- **(2) Memory**
- **(3) Branch**

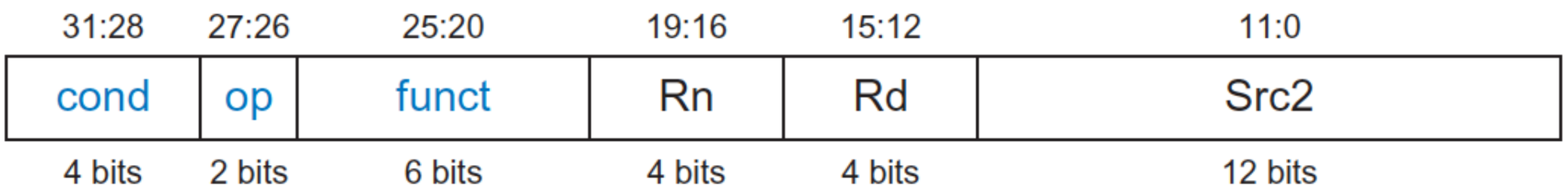
  - **Simpler decoder hardware**

- Number of instruction formats kept small
  - **smaller is faster**

Data-processing Instruction Format

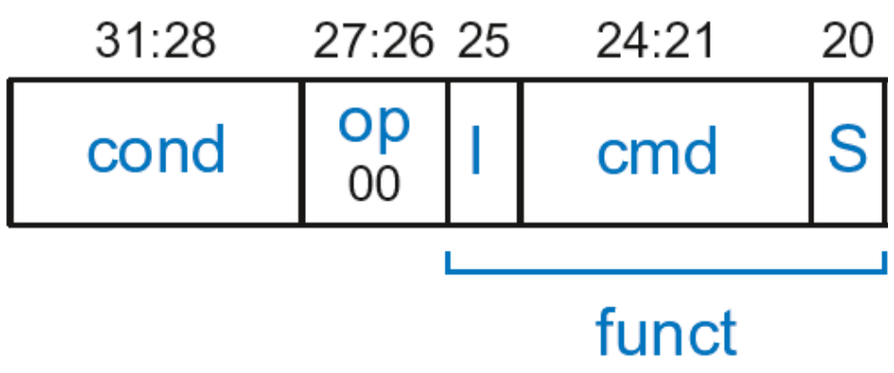
- The 32-bit instruction has six fields:
- **Operands:**
  - ***Rn*:** first ***source*** register
  - ***Src2*:** second ***source*** – register or immediate
  - ***Rd*:** ***destination*** register
- **Control fields:**
  - ***cond*:** specifies ***conditional*** execution
  - ***op*:** the *operation code* or ***opcode***
  - ***funct*:** the ***function*** to perform

Data-processing



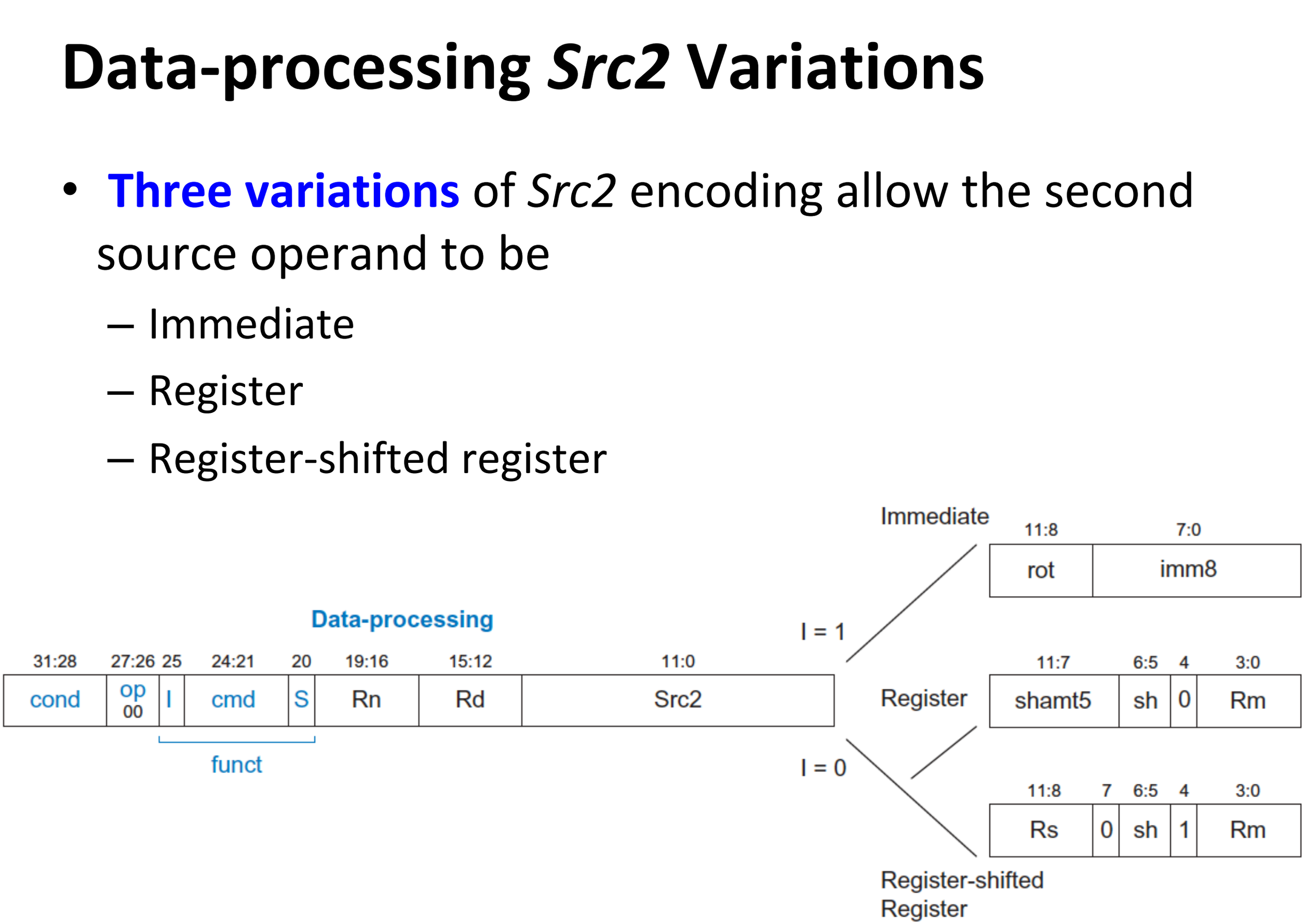
Data-processing Control Fields

- ***cond*** **=1110$_2$** for unconditional execution
- ***op*** **= 00$_2$** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit
  - ***cmd:*** specifies the specific data-processing instruction. For example,
    - ***cmd*** = 0100$_2$ for `ADD`
    - ***cmd*** = 0010$_2$ for `SUB`
  - ***I***-bit
    - ***I*** = 0: *Src2* is a register
    - ***I*** = 1: *Src2* is an immediate
  - ***S***-bit: 1 if sets condition flags
    - ***S*** = 0: `SUB  R0, R5, R7`
    - ***S*** = 1: `ADDS R8, R2, R4` **or** `CMP R3, #10`



Data-processing *Src2* Variations

- **Three variations** of *Src2* encoding allow the second source operand to be
  - Immediate
  - Register
  - Register-shifted register



Data-processing instructions with three register operands

Assembly Code

```
ADD R5, R6, R7
    (0xE0865007)
SUB R8, R9, R10
    (0xE049800A)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
1110_2	00_2	0	0010_2	0	9	8	0	0	0	10
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

cond = 1110_2 (14) for unconditional execution
op = 00_2 (0) for data-processing instructions
cmd = 0100_2 (4) for ADD and 0010_2 (4) for SUB
Src2 is a register so ***I*** = 0

Data-processing instructions with an immediate and two register operands

Assembly Code

```
ADD R0, R1, #42
    (0xE281002A)
SUB R2, R3, #0xFF0
    (0xE2432EFF)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8

The immediate of the ADD instruction (42) can be encoded in 8 bits, so no rotation is needed (**imm8=42, rot=0**)

However, the immediate of SUB R2, R3, 0xFF0 cannot be encoded directly using the 8 bits of imm8 . Instead, imm8 is 255 (0xFF), and it is rotated right by 28 bits (**rot = 14**).

The right rotation by 28 bits is equivalent to a left rotation by 32− 28 = 4 bits.

Shift instructions with immediate shift amounts

cmd: field is 13 **(1101_2)** for all shift instruction
Rm**:** the second source operand
shamt5**:** the amount Rm is shifted
sh**:** the type of shift

Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Assembly Code

```
LSL   R0, R9, #7
      (0xE1A00389)
ROR   R3, R5, #21
      (0xE1A03AE5)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	1101_2	0	0	0	7	00_2	0	9
1110_2	00_2	0	1101_2	0	0	3	21	11_2	0	5
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Shift instructions with register shift amounts

Assembly Code

```
LSR R4, R8, R6
   (0xE1A04638)
ASR R5, R1, R12
   (0xE1A05C51)
```

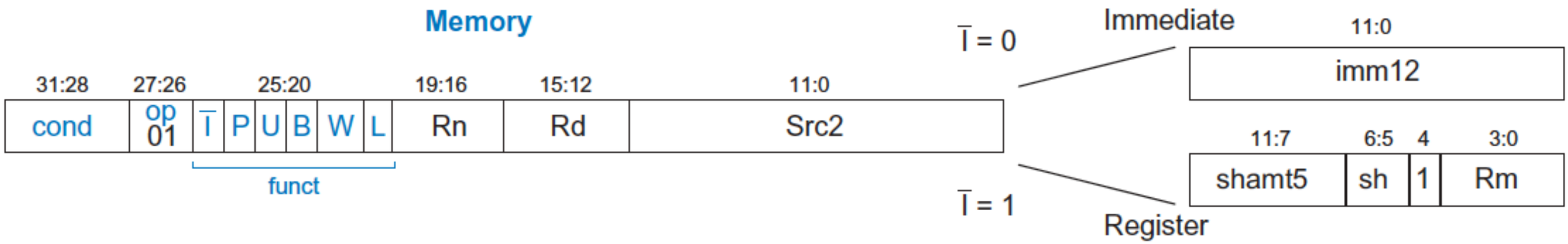
Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
1110_2	00_2	0	1101_2	0	0	4	6	0	01_2	1	8
1110_2	00_2	0	1101_2	0	0	5	12	0	10_2	1	1
cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

This instruction uses the register-shifted register addressing mode, where one register (Rm) is shifted by the amount held in a second register (Rs).

Memory Instruction Format

- **Encodes:** `LDR`, `STR`, `LDRB`, `STRB`
- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = offset: immediate or register (optionally shifted)
- ***funct*** = $\overline{I}$ (immediate bar), *P* (preindex), *U* (add), *B* (byte), *W* (writeback), *L* (load)



Memory Format *funct* Encodings

Type of Operation

L	*B*	Instruction
0	0	`STR`
0	1	`STRB`
1	0	`LDR`
1	1	`LDRB`

Indexing Mode

P	*W*	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

Add/Subtract Immediate/Register Offset

Value	*I*	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

Indexing Modes

Mode	Address	Base Reg. Update
Offset	Base register ± Offset	No change
Preindex	Base register ± Offset	Base register ± Offset
Postindex	Base register	Base register ± Offset

Examples

- **Offset:**

```
LDR R1, [R2, #4]    ; R1 = mem[R2+4]
```

- **Preindex:**

```
LDR R3, [R5, #16]! ; R3 = mem[R5+16]
                   ; R5 = R5 + 16
```

- **Postindex:**

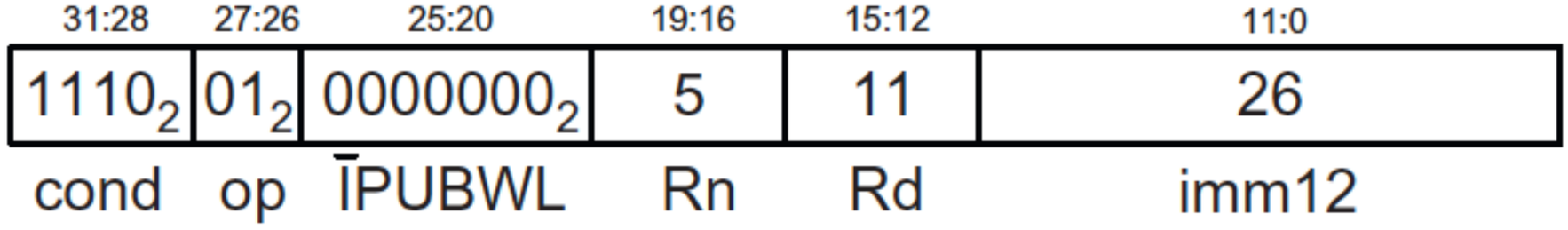
```
LDR R8, [R1], #8    ; R8 = mem[R1]
                    ; R1 = R1 + 8
```

Memory Instruction Example

Assembly Code

```
STR R11, [R5], #-26
```

Field Values



Operation: mem[R5] <= R11; R5 = R5 - 26

cond = 1110_2 (14) for unconditional execution

op = 01_2 (1) for memory instruction

funct = 0000000_2 (0)

I = 0 (immediate offset), ***P*** = 0 (postindex),

U = 0 (subtract), ***B*** = 0 (store word), ***W*** = 0 (postindex),

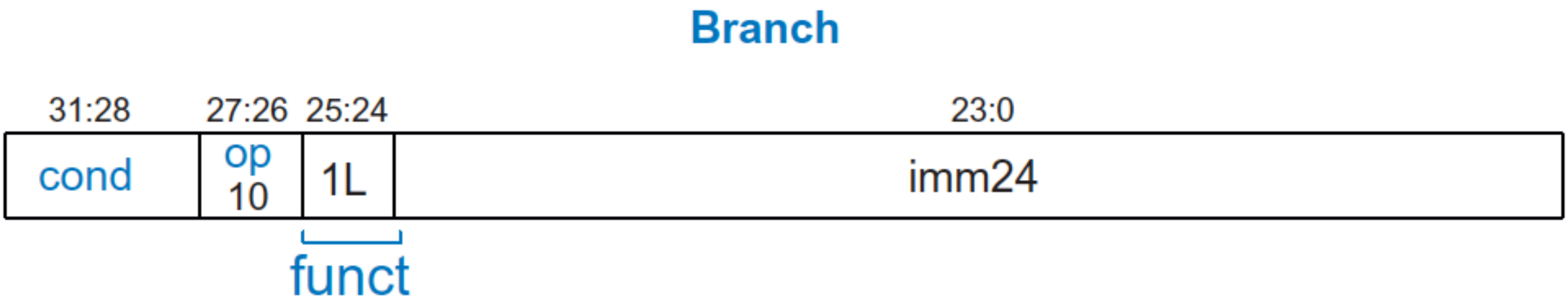
L = 0 (store)

Rd = 11, ***Rn*** = 5, ***imm12*** = 26

Branch Instruction Format

Encodes `B` and `BL`

- ***op*** = 10_2
- ***imm24*:** 24-bit immediate
- ***funct*** = $1L_2$: ***L*** = 1 for BL, ***L*** = 0 for B



Encoding Branch Target Address

- ***Branch Target Address* (BTA)*:*** The branch target address (BTA) is the address of the next instruction to execute if the branch is taken.
- *imm24* encodes BTA
- ***imm24*** = # of words BTA is away from PC+8

Branch Instruction: Example 1

ARM assembly code

```
0xA0            BLT THERE          ← PC
0xA4            ADD R0, R1, R2
0xA8            SUB R0, R0, R9     ← PC+8
0xAC            ADD SP, SP, #8
0xB0            MOV PC, LR
0xB4  THERE     SUB R0, R0, #1     BTA
0xB8            BL  TEST
```

PC = 0xA0

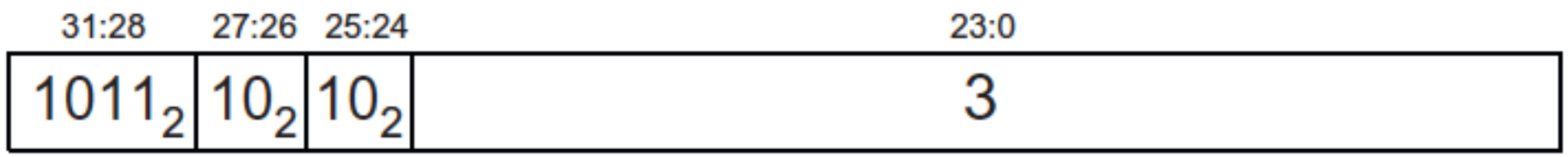
PC + 8 = 0xA8

THERE label is 3 instructions past PC+8

So, *imm24* = 3

Assembly Code **Field Values**

BLT THERE
(0xBA000003)



Branch Instruction: Example 2

ARM assembly code

```
0x8040 TEST   LDRB R5, [R0, R3]   ← BTA
0x8044        STRB R5, [R1, R3]
0x8048        ADD  R3, R3, #1
0x8044        MOV  PC, LR
0x8050        BL   TEST           ← PC
0x8054        LDR  R3, [R1], #4
0x8058        SUB  R4, R3, #9     ← PC+8
```

PC = 0x8050

PC + 8 = 0x8058

`TEST` label is 6 instructions before PC+8

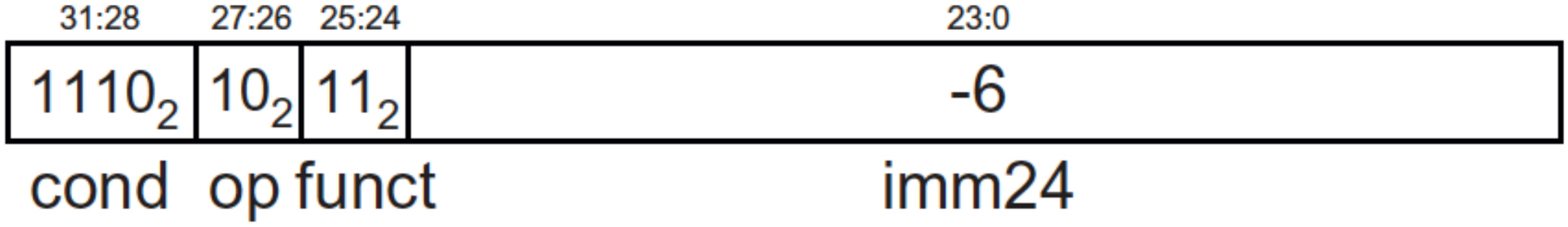
So, *imm24* = -6

Assembly Code

`BL TEST`

(0xEBFFFFFA)

Field Values



Review: Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	
0001	NE	Not equal	
0010	CS / HS	Carry set / Unsigned higher or same	
0011	CC / LO	Carry clear / Unsigned lower	
0100	MI	Minus / Negative	
0101	PL	Plus / Positive of zero	
0110	VS	Overflow / Overflow set	
0111	VC	No overflow / Overflow clear	
1000	HI	Unsigned higher	
1001	LS	Unsigned lower or same	
1010	GE	Signed greater than or equal	
1011	LT	Signed less than	
1100	GT	Signed greater than	
1101	LE	Signed less than or equal	
1110	AL (or none)	Always / unconditional	ignored

Conditional Execution: Machine Code

Assembly Code

Field Values

	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
`SUBS R1, R2, R3`	14	0	0	2	1	2	1	0	0	0	3
`ADDEQ R4, R5, R6`	0	0	0	4	0	5	4	0	0	0	6
`ANDHS R7, R5, R6`	2	0	0	0	0	5	7	0	0	0	6
`ORRMI R8, R5, R6`	4	0	0	12	0	5	8	0	0	0	6
`EORLT R9, R5, R6`	11	0	0	1	0	5	9	0	0	0	6
	cond	op	I	cmd	S	rn	rd	shamt5	sh		rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0	
1110	00	0	0010	0	0010	0001	00000	00	0	0011	(0xE0421003)
0000	00	0	0100	0	0101	0100	00000	00	0	0110	(0x00854006)
0010	00	0	0000	0	0101	0111	00000	00	0	0110	(0x20057006)
0100	00	0	1100	0	0101	1000	00000	00	0	0110	(0x41858006)
1011	00	0	0001	0	0101	1001	00000	00	0	0110	(0xB0259006)
cond	op	I	cmd	S	rn	rd	shamt5	sh		rm	

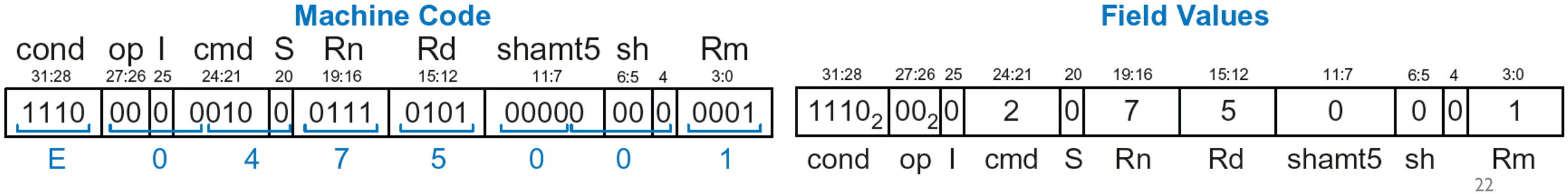
Interpreting Machine Code

- **Start with *op*:** tells how to parse rest
  - ***op*** = 00 (Data-processing)
  - ***op*** = 01 (Memory)
  - ***op*** = 10 (Branch)
- ***I*-bit:** tells how to parse ***Src2***
- **Data-processing instructions:**
  - If ***I***-bit is 0, bit 4 determines if ***Src2*** is a register (bit 4 = 0) or a register-shifted register (bit 4 = 1)
- **Memory instructions:**
  - Examine ***funct*** bits for indexing mode, instruction, and add or subtract offset

Interpreting Machine Code: Example 1

0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction
- ***I*-bit:** 0, so *Src2* is a register
- **bit 4:** 0, so *Src2* is a register (optionally shifted by *shamt5*)
- ***cmd*:** 0010_2 (2), so SUB
- **Rn**=7, **Rd**=5, **Rm**=1, ***shamt5*** = 0, ***sh*** = 0
- So, instruction is: **`SUB R5,R7,R1`**



Interpreting Machine Code: Example 2

0xE5949010

- **Start with *op*:** 01_2, so memory instruction
- ***funct*:** *B*=0, *L*=1, so `LDR`; *P*=1, *W*=0, so offset indexing; *I*=0, so immediate offset, *U*=1, so add offset
- **Rn**=4, **Rd**=9, ***imm12*** = 16
- So, instruction is: **`LDR R9,[R4,#16]`**

Machine Code

cond	op	IPUBWL	Rn	Rd	imm12
31:28	27:26	25:20	19:16	15:12	11:0
1110	01	011001	0100	1001	0000 0001 0000
E	5	9	4	9	0 1 0

Field Values

31:28	27:26	25:20	19:16	15:12	11:0
1110_2	01_2	25	4	9	16
cond	op	IPUBWL	Rn	Rd	imm12

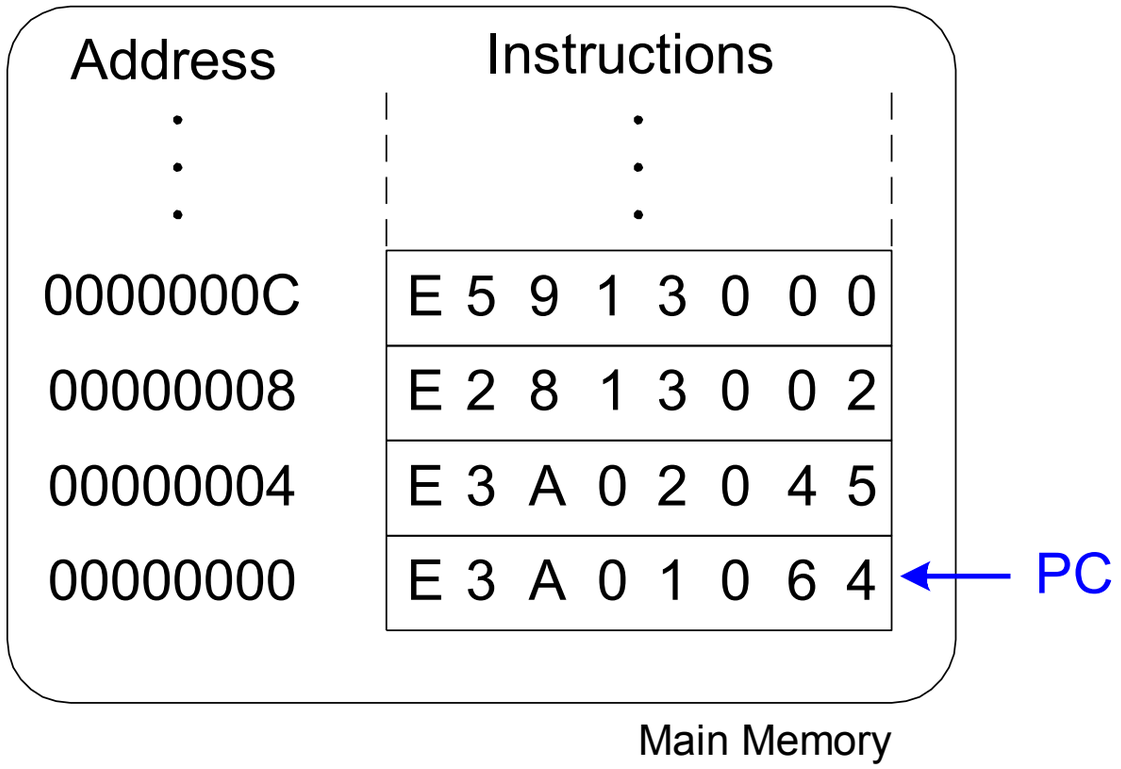
Power of the Stored Program

- **32-bit instructions & data** stored in memory
- **Sequence of instructions:** only difference between two applications
- **To run a new program:**
  - No rewiring required
  - Simply store new program in memory
- **Program Execution:**
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
`MOV R1, #100`	`0xE3A01064`
`MOV R2, #69`	`0xE3A02045`
`ADD R3, R1, R2`	`0xE2813002`
`STR R3, [R1]`	`0xE5913000`

Stored Program



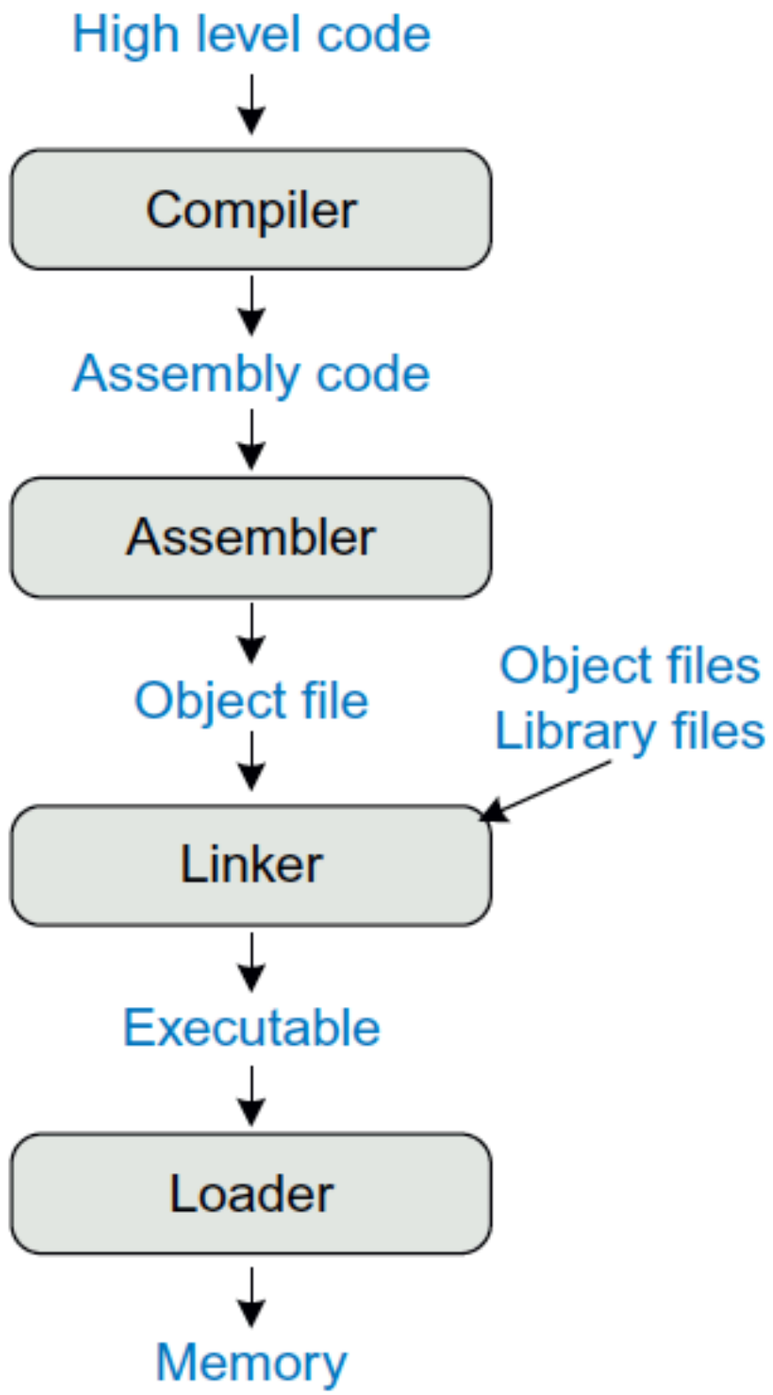
Program Counter (PC): keeps track of current instruction

Up Next

- How to implement the ARM Instruction Set Architecture in Hardware?

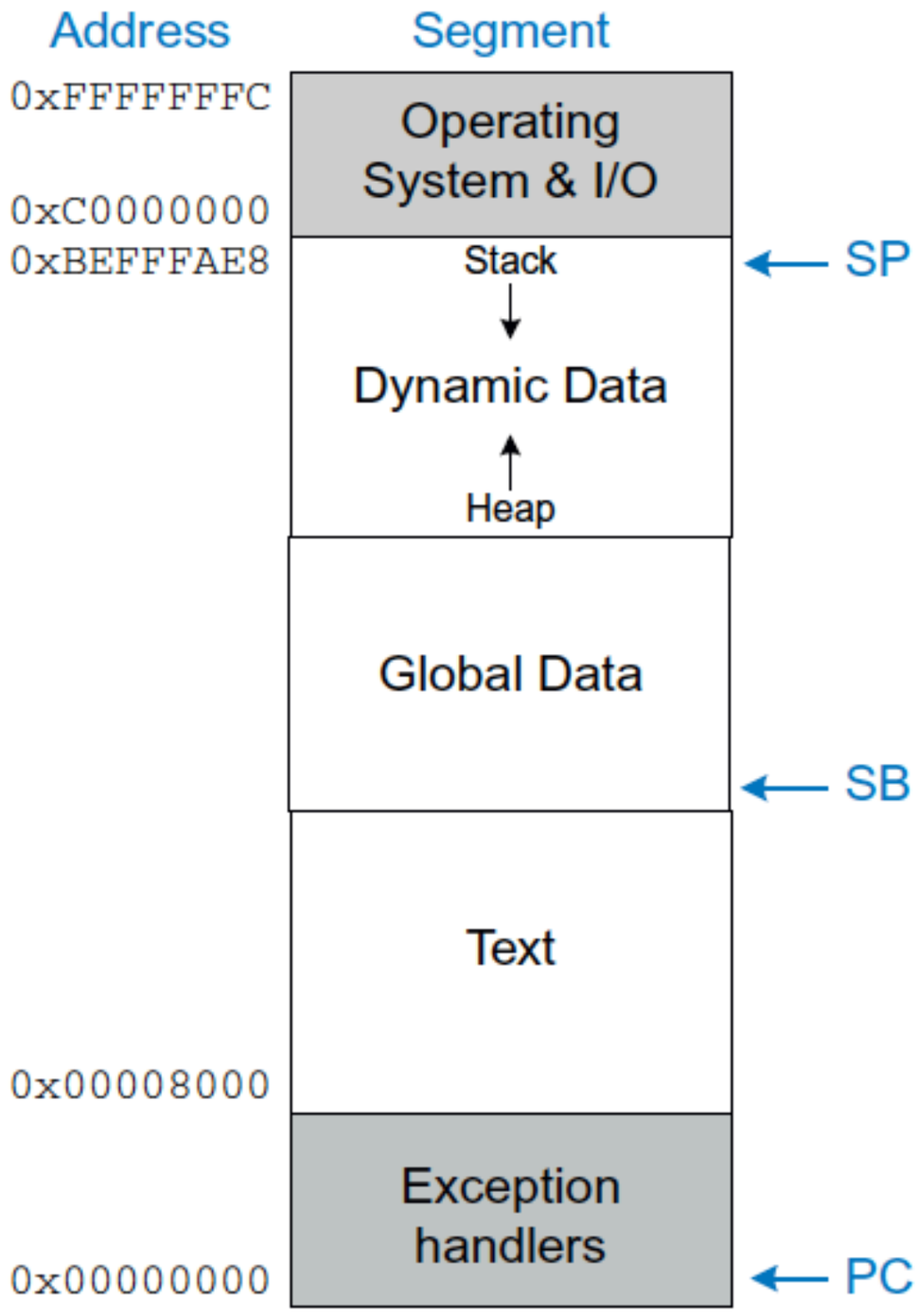
Microarchitecture

Steps for translating and starting a program



Memory MAP

- The ARM architecture divides the address space into five parts or segments:
  - the text segment
    - stores the machine language program
  - global data segment
    - global variables that
  - dynamic data segment
    - the stack and the heap (dynamically allocated and deallocated)
  - and segments for exception handlers, the operating system (OS) and input/output (I/O).



Compiling a High-Level Program

High-Level Code

```
int f, g, y; // global variables


int sum(int a, int b) {
   return (a + b);
}


int main(void)
{
   f = 2;
   g = 3;
   y = sum(f, g);
   return y;
}
```

ARM Assembly Code

```
   .text
   .global  sum
   .type    sum, %function
sum:
   add      r0, r0, r1
   bx       lr
   .global  main
   .type    main, %function
main:
   push     {r3, lr}
   mov      r0, #2
   ldr      r3, .L3
   str      r0, [r3, #0]
   mov      r1, #3
   ldr      r3, .L3+4
   str      r1, [r3, #0]
   bl       sum
   ldr      r3, .L3+8
   str      r0, [r3, #0]
   pop      {r3, pc}
.L3:
   .word    f
   .word    g
   .word    y
```

Further Reading

- You can read **Chapter 6** of your book
  - **Section 6.4**