ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

Università di Napoli Federico II
Corso di Laurea in Informatica

Docenti
Proff. Luigi Sauro gruppo 1 (A-G)
Silvia Rossi gruppo 2 (H-Z)

ARCHITETTURA ARM

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `SP` points to top of the stack

Stack expands by 2 words

How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `R4`, `R8`, `R9`

ARM Assembly Code

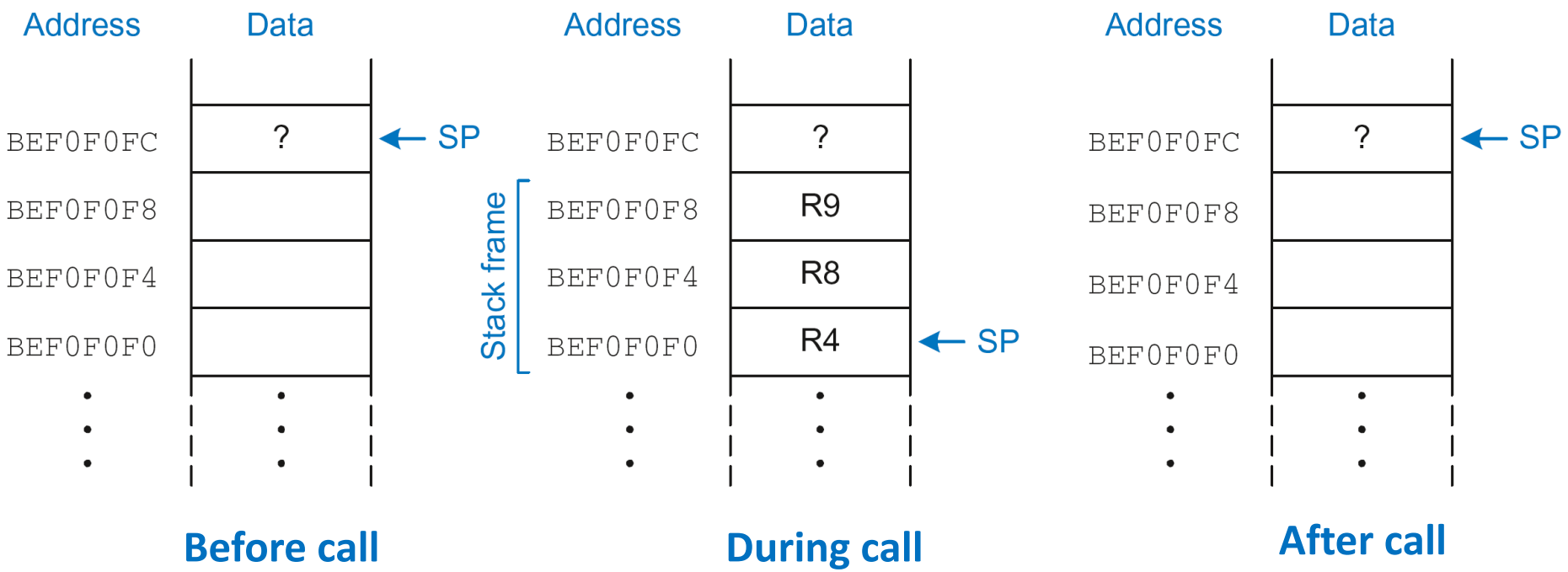
```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  MOV PC, LR          ; return to caller
```

Storing Register Values on the Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12     ; make space on stack for 3 registers
  STR R4, [SP, #-8]   ; save R4 on stack
  STR R8, [SP, #-4]   ; save R8 on stack
  STR R9, [SP]        ; save R9 on stack
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  LDR R9, [SP]        ; restore R9 from stack
  LDR R8, [SP, #-4]   ; restore R8 from stack
  LDR R4, [SP, #-8]   ; restore R4 from stack
  ADD SP, SP, #12     ; deallocate stack space
  MOV PC, LR          ; return to caller
```

The Stack during `diffofsums` Call



Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

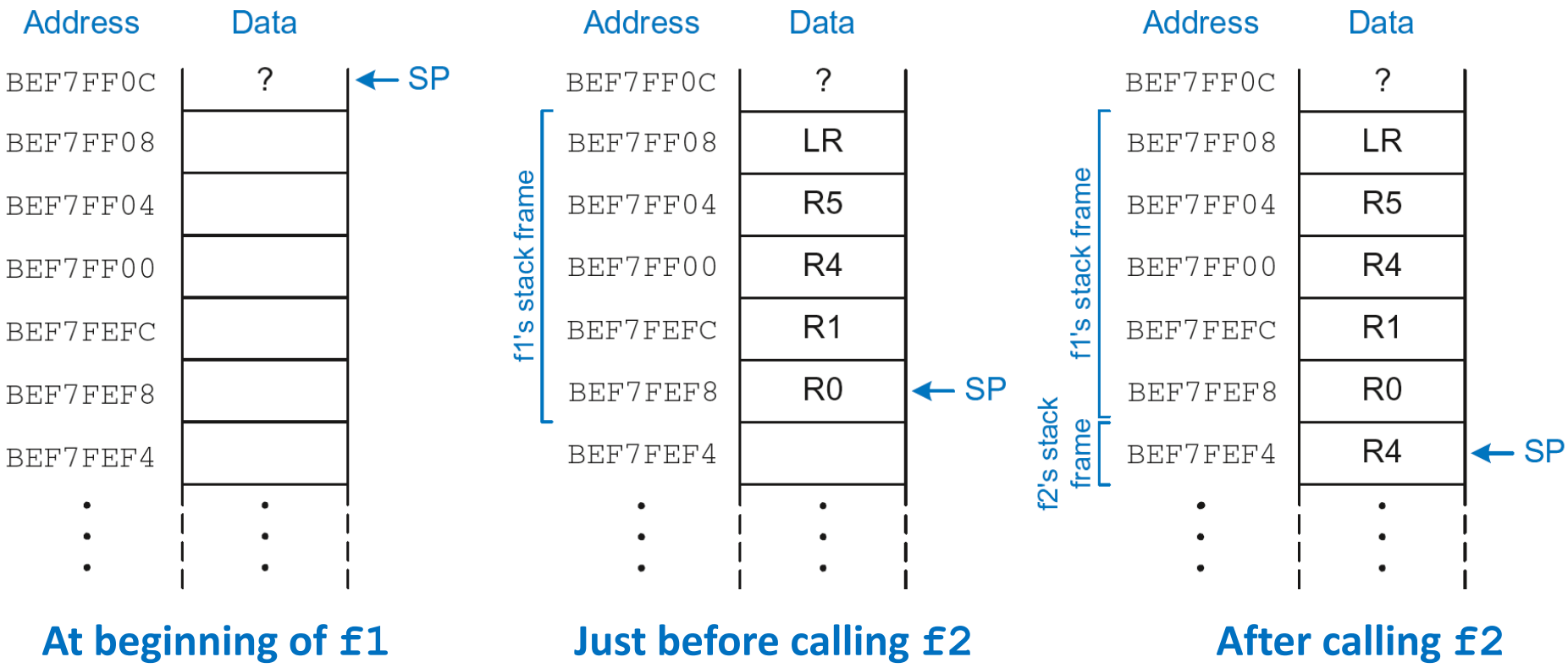
Nonleaf Function Example

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR} ; save regs
  ADD   R5,  R0, R1  ; x = (a+b)
  SUB   R12, R0, R1  ; temp = (a-b)
  MUL   R5,  R5, R12 ; x = x*temp
  MOV   R4,  #0      ; i = 0
FOR
  CMP   R4, R0       ; i < a?
  BGE   RETURN       ; no: exit loop
  PUSH {R0, R1}      ; save regs
  ADD   R0, R1, R4   ; arg is b+i
  BL    F2           ; call f2(b+i)
  ADD   R5, R5, R0   ; x = x+f2(b+i)
  POP  {R0, R1}      ; restore regs
  ADD   R4, R4, #1   ; i++
  B     FOR          ; repeat loop
RETURN
  MOV   R0, R5       ; return x
  POP  {R4, R5, LR}  ; restore regs
  MOV   PC, LR       ; return
```

```
; R0=p, R4=r
F2
  PUSH {R4}          ; save regs
  ADD   R4, R0, 5    ; r = p+5
  ADD   R0, R4, R0   ; return r+p
  POP  {R4}          ; restore regs
  MOV   PC, LR       ; return
```

Stack during Nonleaf Function



Recursive Function Call

C Code

- Funzione non foglia che chiama se stessa
  - Si comporta da chiamato e da chiamante
  - Salve registri preservati e anche quelli non preservati

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Recursive Function Call

ARM Assembly Code

```
0x94 FACTORIAL  STR R0, [SP, #-4]!    ;store R0 on stack
0x98            STR LR, [SP, #-4]!    ;store LR on stack
0x9C            CMP R0, #2            ;set flags with R0-2
0xA0            BHS ELSE              ;if (r0>=2) branch to else
0xA4            MOV R0, #1            ; otherwise return 1
0xA8            ADD SP, SP, #8        ; restore SP 1
0xAC            MOV PC, LR            ; return
0xB0 ELSE       SUB R0, R0, #1        ; n = n - 1
0xB4            BL  FACTORIAL         ; recursive call
0xB8            LDR LR, [SP], #4      ; restore LR
0xBC            LDR R1, [SP], #4      ; restore R0 (n) into R1
0xC0            MUL R0, R1, R0        ; R0 = n*factorial(n-1)
0xC4            MOV PC, LR            ; return
```

Recursive Function Call

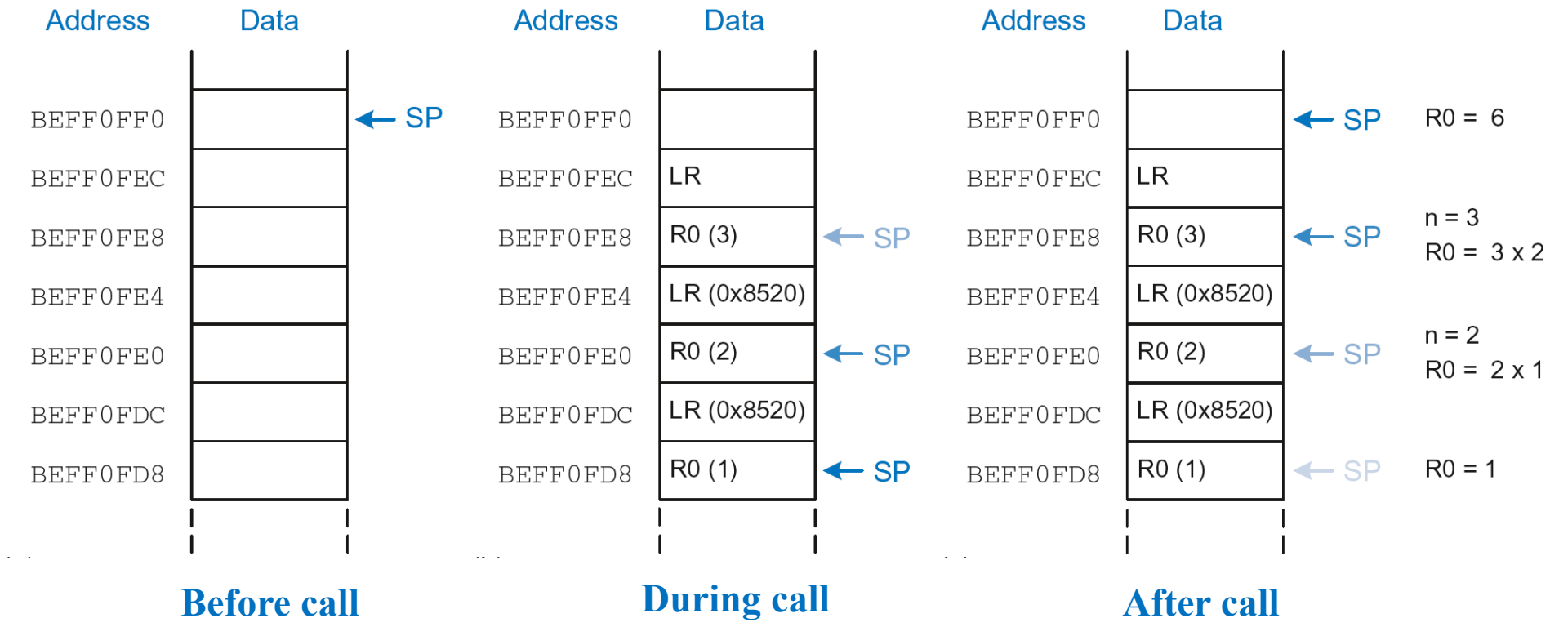
C Code

```
int factorial(int n) {

  if (n <= 1)
    return 1;



  else
    return (n * factorial(n-1));
}
```

ARM Assembly Code

```
0x94 FACTORIAL  STR R0, [SP, #-4]!
0x98            STR LR, [SP, #-4]!
0x9C            CMP R0, #2
0xA0            BHS ELSE
0xA4            MOV R0, #1
0xA8            ADD SP, SP, #8
0xAC            MOV PC, LR
0xB0 ELSE       SUB R0, R0, #1
0xB4            BL  FACTORIAL
0xB8            LDR LR, [SP], #4
0xBC            LDR R1, [SP], #4
0xC0            MUL R0, R1, R0
0xC4            MOV PC, LR
```

Stack during Recursive Call



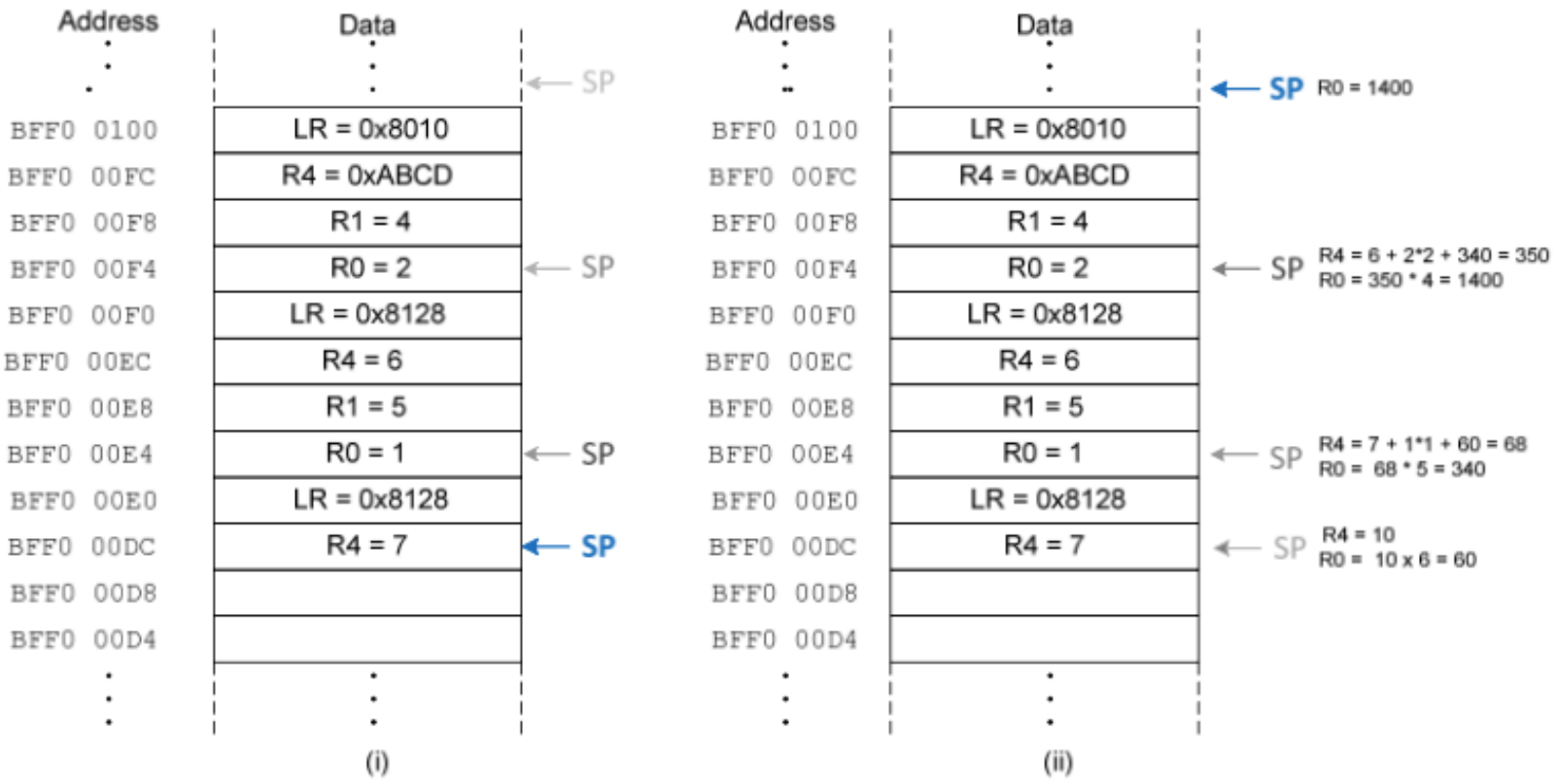
Function Call Summary

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`LR`, maybe `R0-R3, R8-R12`)
  - Calls function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`
- **Callee**
  - Saves registers that might be disturbed `(R4-R7)`
  - Performs function
  - Puts result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

Esercizio 6.34 Si consideri la seguente funzione di alto livello.

```
// codice C
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

```
;R4 = b
;Address   ARM Assembly

0x8100     F     PUSH {R4, LR}          ; store R4 and LR on stack
0x8104           ADD  R4, R1, #2        ; b = k + 2
0x8108           CMP  R0, #0            ; n == 0?
0x810c           BNE  ELSE
0x8110           MOV  R4, #10           ; if yes, b = 10
0x8114           B    DONE              ; branch to end of function
0x8118     ELSE  PUSH {R0, R1}          ; store n and k on stack
0x811c           SUB  R0, R0, #1        ; set up args: n = n-1
0x8120           ADD  R1, R1, #1        ; k = k+1
0x8124           BL   F                 ; recursively call F
0x8128           MOV  R2, R0            ; move return value to R2
0x812c           POP  {R0, R1}          ; restore values of n and k
0x8130           MUL  R3, R0, R0        ; R3 = n*n
0x8134           ADD  R2, R2, R3        ; R2 = (n*n)+f(n-1,k+1)
0x8138           ADD  R4, R2, R4        ; b = b+(n*n)+f(n-1,k+1)
0x813c     DONE  MUL  R0, R4, R1        ; R0 = b*k
0x8140           POP  {R4, LR}          ; restore R4 and LR
0x8144           MOV  PC, LR            ; return to point of call
```


Address
Data
BFF0 0100
BFF0 00FC
BFF0 00F8
BFF0 00F4
BFF0 00F0
BFF0 00EC
BFF0 00E8
BFF0 00E4
BFF0 00E0
BFF0 00DC
BFF0 00D8
BFF0 00D4
LR = 0x8010
R4 = 0xABCD
R1 = 4
R0 = 2
LR = 0x8128
R4 = 6
R1 = 5
R0 = 1
LR = 0x8128
R4 = 7
SP
SP
SP
SP
(i)
Address
Data
BFF0 0100
BFF0 00FC
BFF0 00F8
BFF0 00F4
BFF0 00F0
BFF0 00EC
BFF0 00E8
BFF0 00E4
BFF0 00E0
BFF0 00DC
BFF0 00D8
BFF0 00D4
LR = 0x8010
R4 = 0xABCD
R1 = 4
R0 = 2
LR = 0x8128
R4 = 6
R1 = 5
R0 = 1
LR = 0x8128
R4 = 7
SP
R0 = 1400
SP
R4 = 6 + 2*2 + 340 = 350
R0 = 350 * 4 = 1400
SP
R4 = 7 + 1*1 + 60 = 68
R0 = 68 * 5 = 340
SP
R4 = 10
R0 = 10 x 6 = 60
(ii)

How to Encode Instructions?

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  - 32-bit data, 32-bit instructions
  - For design simplicity, would prefer a single instruction format but…

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  - 32-bit data, 32-bit instructions
  - For design simplicity, would prefer a single instruction format but…
  - Instructions have different needs

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `ADD`, `SUB`: use 3 register operands
  - `LDR`, `STR`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (regularity supports design simplicity and smaller is faster)

Machine Language

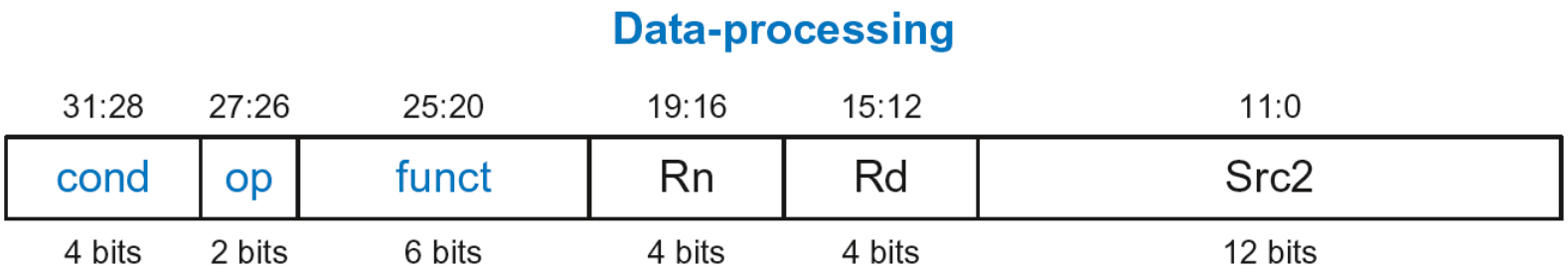
- **Binary representation of instructions**
- Computers only understand **1's and 0's**
- **32-bit instructions**
  - Simplicity favors regularity: 32-bit data & instructions
- **3 instruction formats:**
  - Data-processing
  - Memory
  - Branch

Instruction Formats

- **Data-processing**
- Memory
- Branch

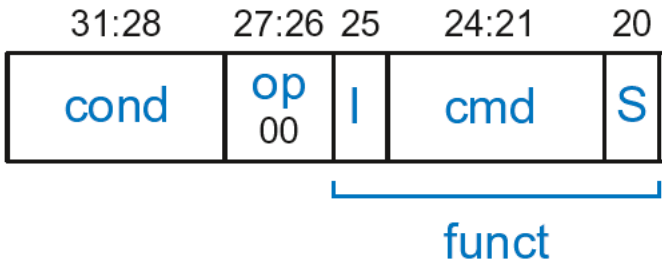
Data-processing Instruction Format

- **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*/operation to perform



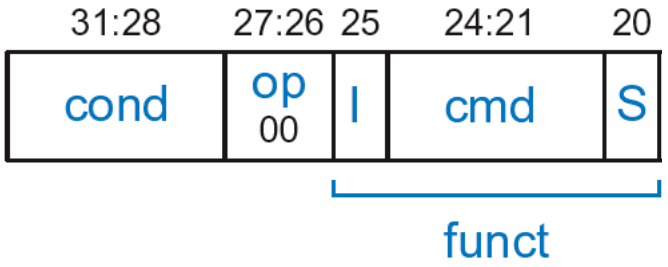
Data-processing Control Fields

- ***op*** **= 00_2** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit



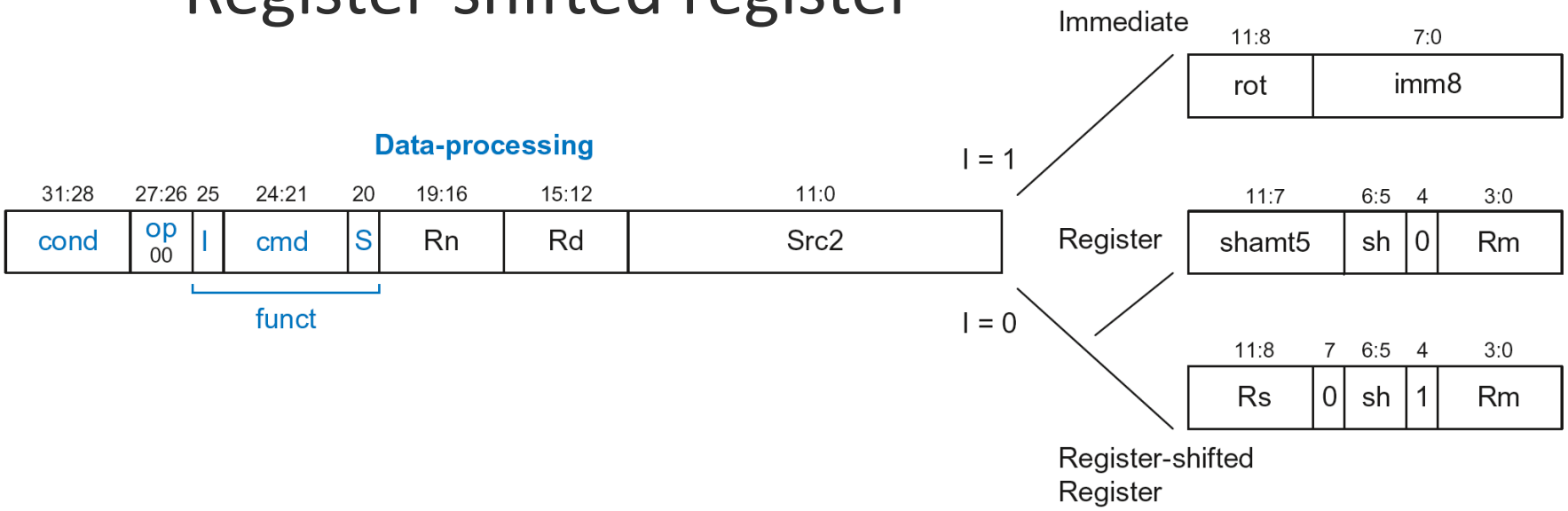
Data-processing Control Fields

- ***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

- *Src2* can be:
  - Immediate
  - Register
  - Register-shifted register



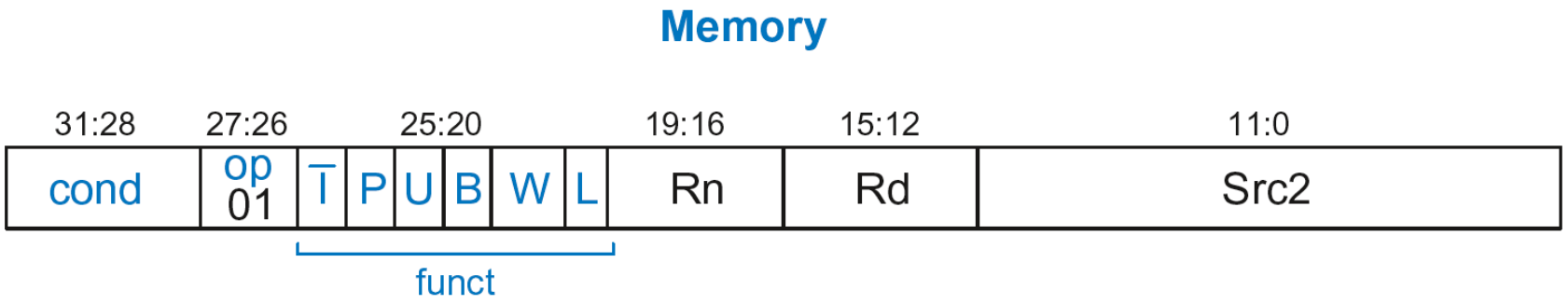
Instruction Formats

- Data-processing
- **Memory**
- Branch

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = offset
- ***funct*** = 6 control bits



Offset Options

Recall: Address = Base Address + Offset

Example: `LDR R1, [R2, #4]`

Base Address = R2, Offset = 4

Address = (R2 + 4)

- Base address always in a register
- The offset can be:
  - an immediate
  - a register
  - or a scaled (shifted) register

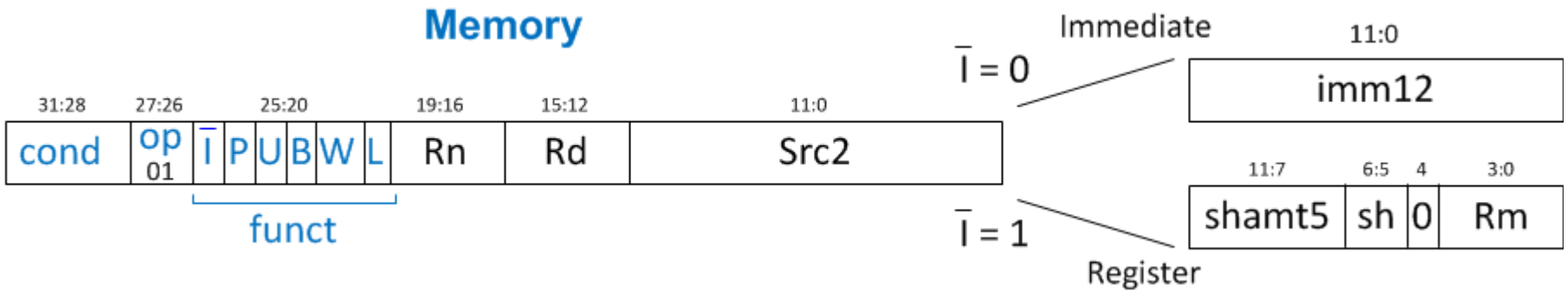
Offset Examples

ARM Assembly	Memory Address
`LDR R0, [R3, #4]`	R3 + 4
`LDR R0, [R5, #-16]`	R5 – 16
`LDR R1, [R6, R7]`	R6 + R7
`LDR R2, [R8, -R9]`	R8 – R9
`LDR R3, [R10, R11, LSL #2]`	R10 + (R11 << 2)
`LDR R4, [R1, -R12, ASR #4]`	R1 – (R12 >>> 4)
`LDR R0, [R9]`	R9

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = **offset: register (optionally shifted) or immediate**
- ***funct*** = 6 control bits



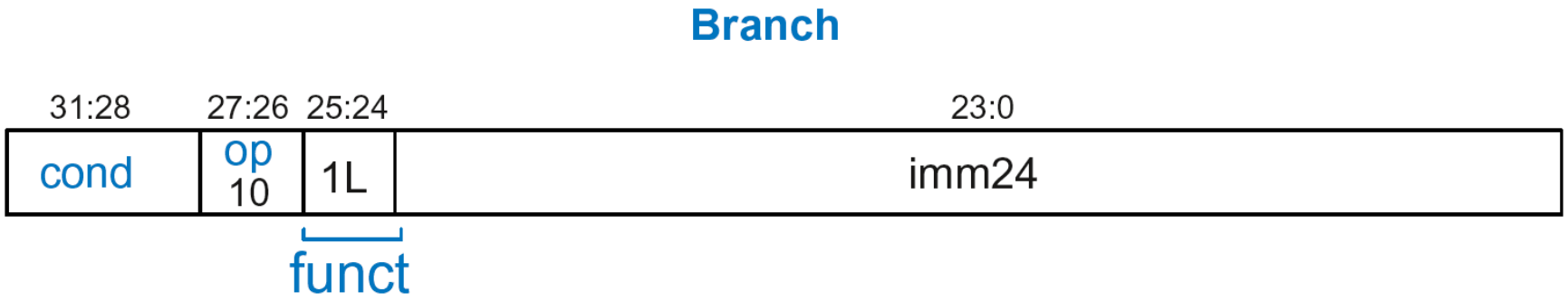
Instruction Formats

- Data-processing
- Memory
- **Branch**

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)*:*** Next PC when branch taken
- BTA is relative to current PC + 8
- *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

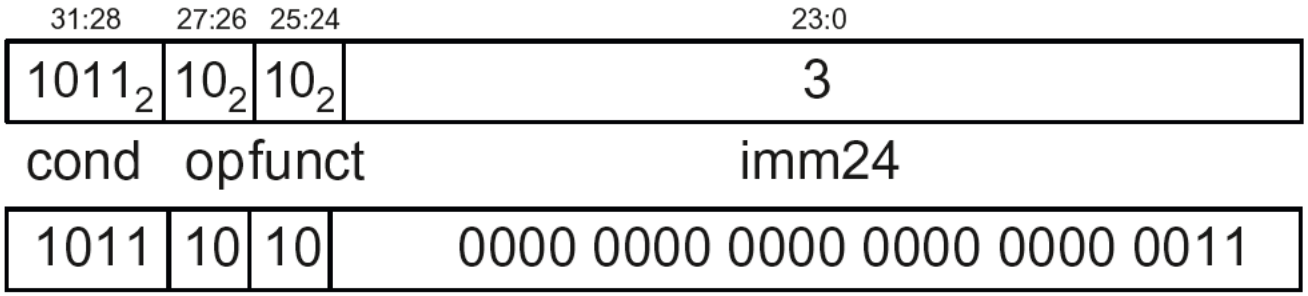
Branch Instruction: Example 1

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

Field Values



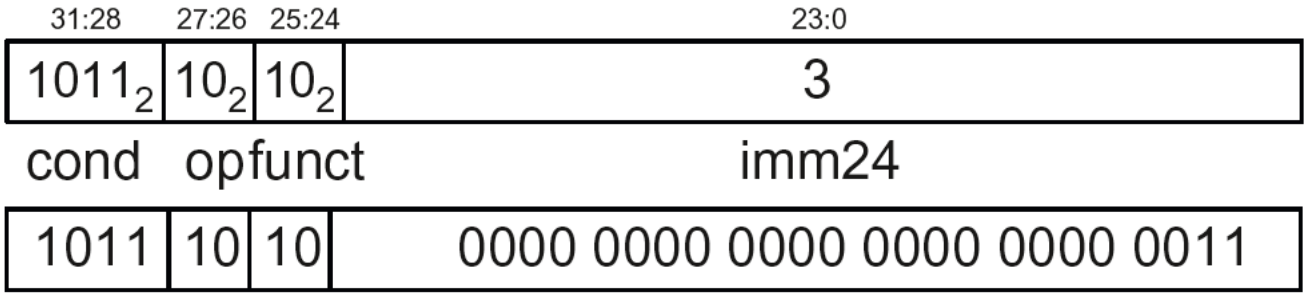
Branch Instruction: Example 1

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

Field Values



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

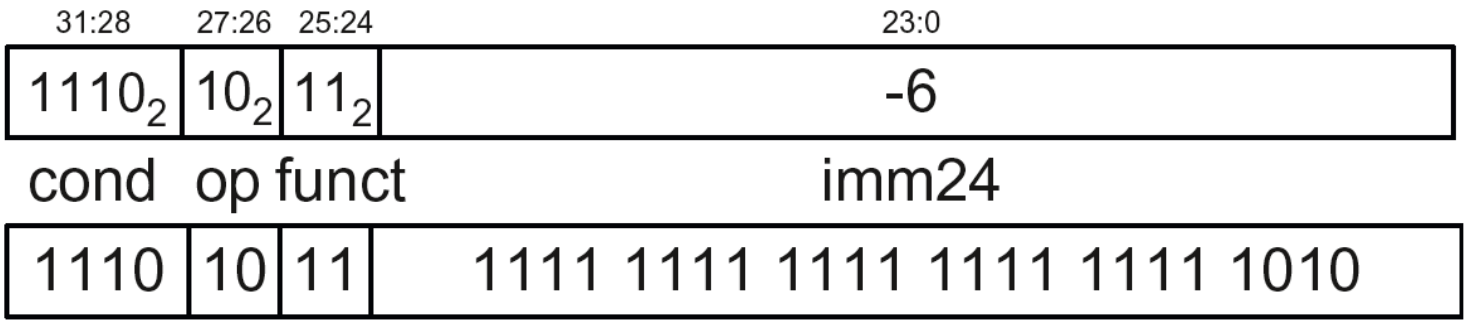
Branch Instruction: Example 2

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

Field Values



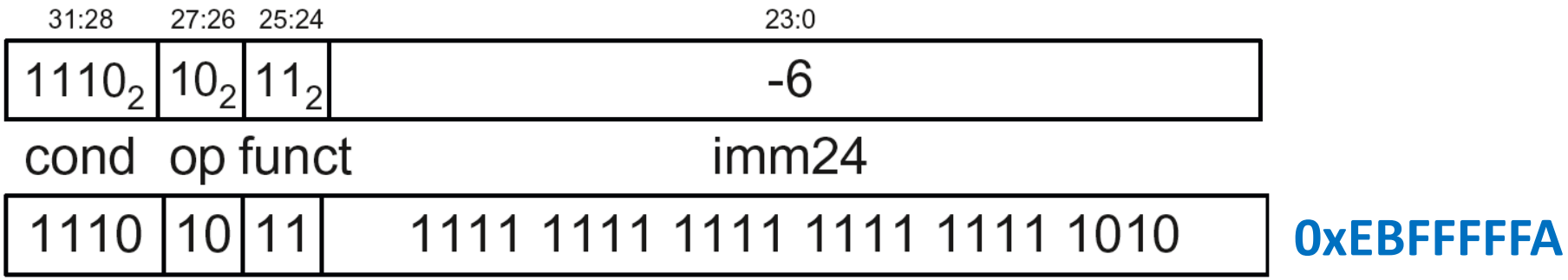
Branch Instruction: Example 2

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
```

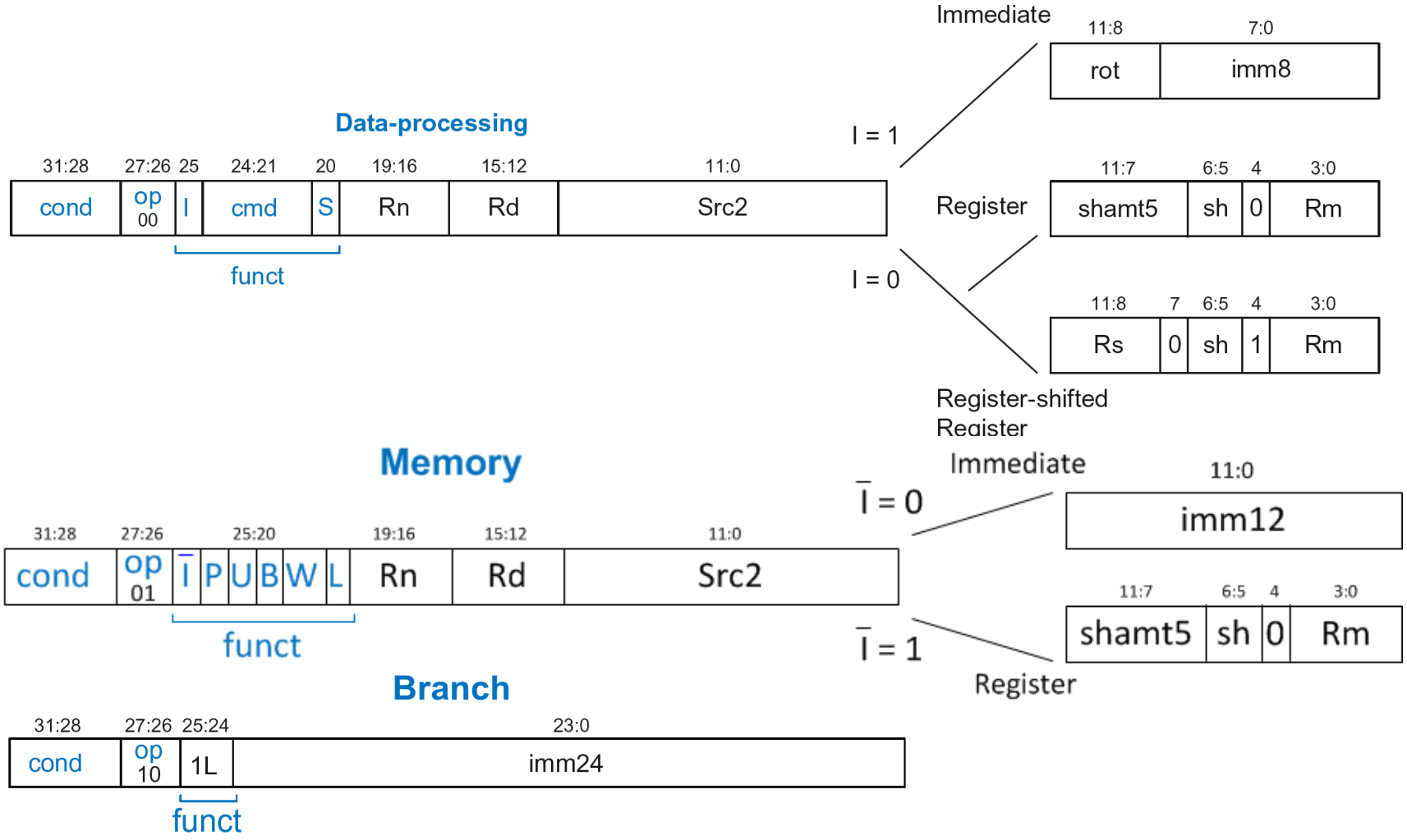
- PC = 0x8050
- PC + 8 = 0x8058
- `TEST` label is 6 instructions before PC+8
- So, *imm24* = -6

Field Values



0xEBFFFFFA

Review: Instruction Formats



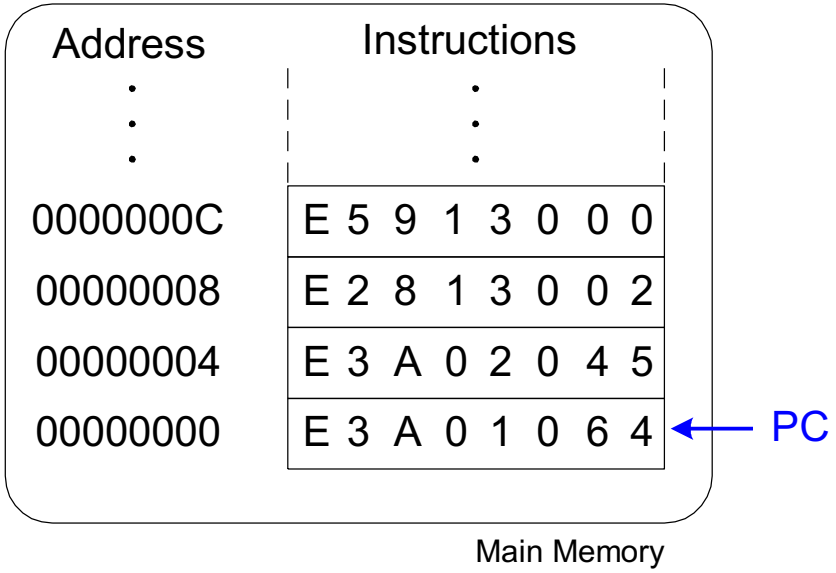
Power of the Stored Program

- **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