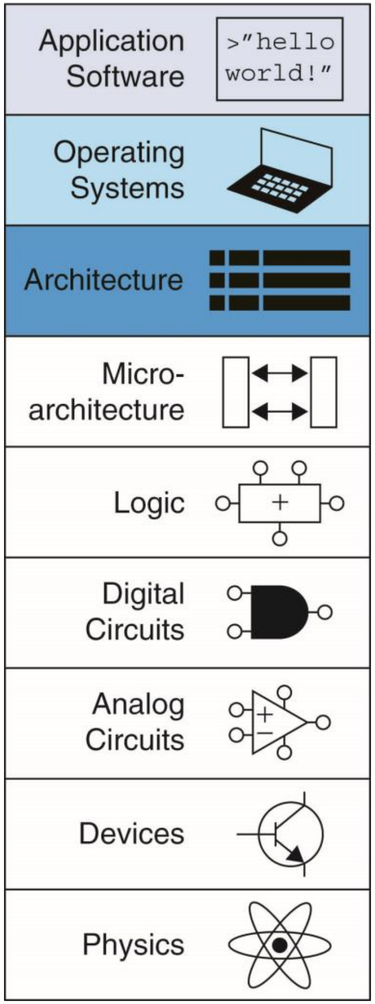
Chapter 6

***Digital Design and Computer Architecture*: ARM® Edition**

Sarah L. Harris and David Money Harris

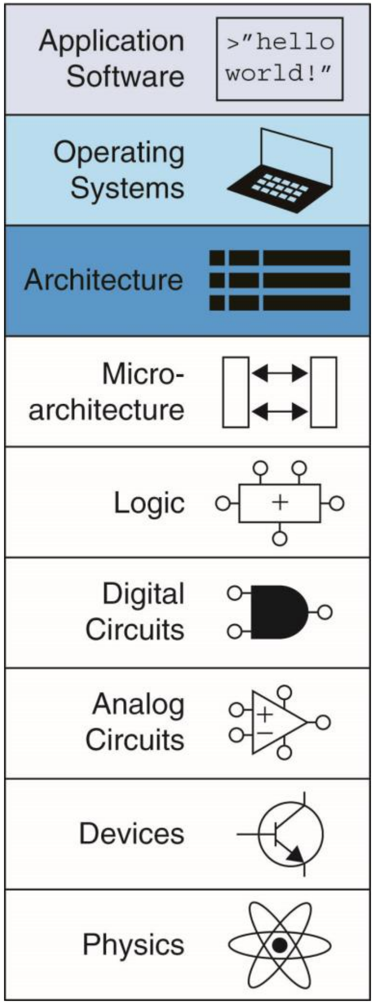
Chapter 6 :: Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**



Introduction

- **Jumping up a few levels of abstraction**
  - **Architecture:** programmer's view of computer
    - Defined by **instructions** & **operand locations**
  - **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Instructions

- **Commands in a computer's language**
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines,, etc.

ARM Architecture

- Developed in the 1980’s by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines,, etc.

Once you’ve learned one architecture, it’s easier to learn others

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Regularity supports design simplicity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** mnemonic – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Instruction: Subtraction

Similar to addition - only mnemonic changes

C Code

```
a = b - c;
```

ARM assembly code

```
SUB a, b, c
```

- **SUB:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

- Registers
- Constants (also called *immediates*)
- Memory

Operands: Registers

- ARM has 16 registers
- Registers are faster than memory
- Each register is 32 bits
- ARM is called a "32-bit architecture" because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

ARM Register Set

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Operands: Registers

- **Registers:**
  - R before number, all capitals
  - Example: “R0” or “register zero” or “register R0”

Operands: Registers

- **Registers used for specific purposes:**
  - **Saved registers:** R4-R11 hold variables
  - **Temporary registers:** R0-R3 and R12, hold intermediate values
  - Discuss others later

Instructions with Registers

Revisit ADD instruction

C Code

```
a = b + c
```

ARM Assembly Code

```
; R0 = a, R1 = b, R2 = c

ADD R0, R1, R2
```

Operands: Constants\Immediates

- Many instructions can use constants or *immediate* operands
- For example: ADD and SUB
- value is *immediate*ly available from instruction

C Code

```
a = a + 4;
b = a - 12;
```

ARM Assembly Code

```
; R0 = a, R1 = b
ADD R0, R0, #4
SUB R1, R0, #12
```

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Constant must have < 8 bits of precision

Generating Constants

Generating small constants using move (MOV):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

Constant must have < 8 bits of precision

Note: `MOV` can also use 2 registers: `MOV R7, R9`

Generating Constants

Generate larger constants using move (MOV) and or (ORR):

C Code

```
int a = 0x7EDC8765;
```

ARM Assembly Code

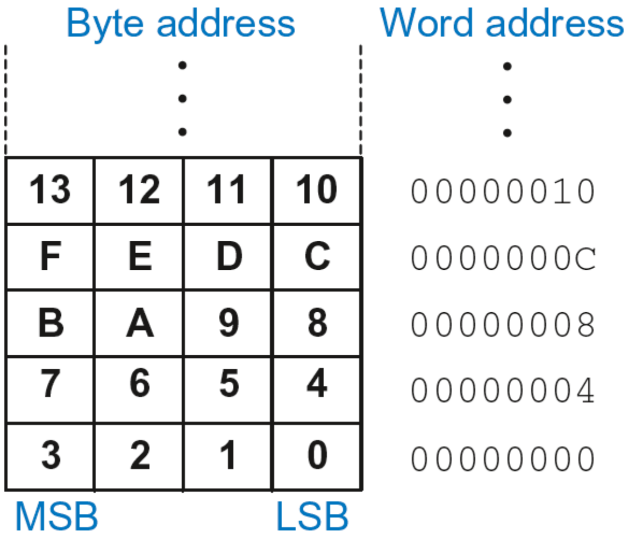
```
# R0 = a
MOV R0, #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```

Operands: Memory

- Too much data to fit in only 16 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables still kept in registers

Byte-Addressable Memory

- Each data **byte** has unique address
- 32-bit word = 4 bytes, so word address increments by 4



Byte address				Word address
⋮				⋮
13	12	11	10	00000010
F	E	D	C	0000000C
B	A	9	8	00000008
7	6	5	4	00000004
3	2	1	0	00000000
MSB			LSB	

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (`LDR`)
- **Format:**

```
LDR R0, [R1, #12]
```

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)

Reading Memory

- Memory read called ***load***
- **Mnemonic:** *load register* (LDR)
- **Format:**

```
LDR R0, [R1, #12]
```

Address calculation:

- add *base address* (R1) to the *offset* (12)
- address = (R1 + 12)

Result:

- R0 holds the data at memory address (R1 + 12)

Any register may be used as base address

Reading Memory

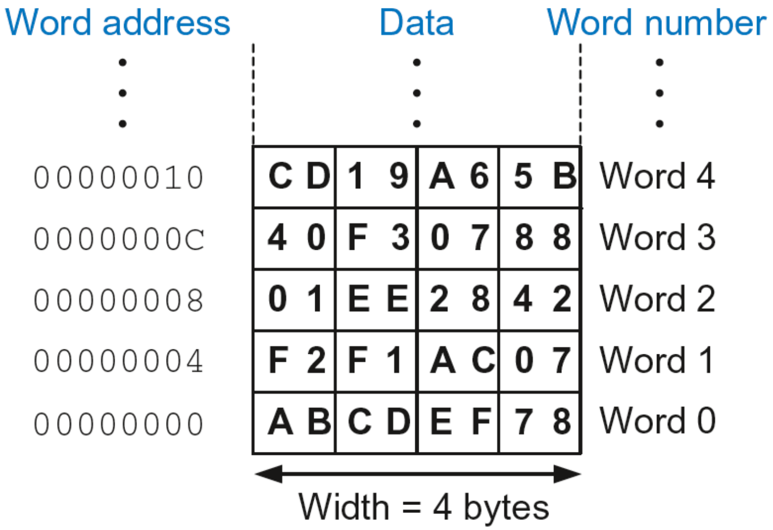
- **Example:** Read a word of data at memory address 8 into R3

Reading Memory

- **Example:** Read a word of data at memory address 8 into R3
  - Address = (R2 + 8) = 8
  - R3 = 0x01EE2842 after load

ARM Assembly Code

```
MOV R2, #0
LDR R3, [R2, #8]
```



Word address	Data	Word number
00000010	CD19A65B	Word 4
0000000C	40F30788	Word 3
00000008	01EE2842	Word 2
00000004	F2F1AC07	Word 1
00000000	ABCDEF78	Word 0

Writing Memory

- Memory write are called ***stores***
- **Mnemonic:** *store register* (`STR`)

Writing Memory

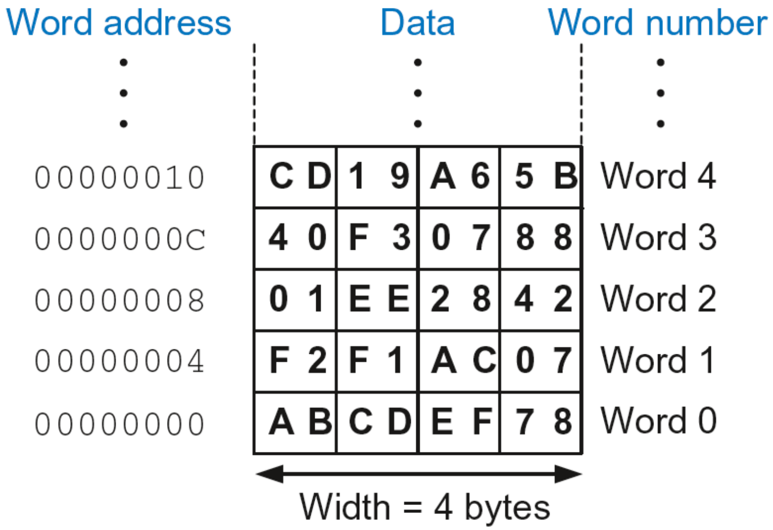
- **Example:** Store the value held in R7 into memory word 21.

Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```



Word address	Data	Word number
⋮	⋮	⋮
00000010	C D 1 9 A 6 5 B	Word 4
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

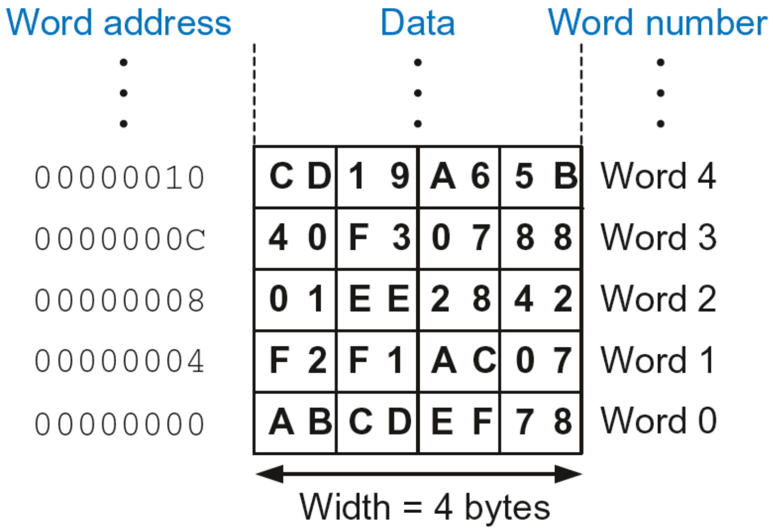
Writing Memory

- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```

The offset can be written in decimal or hexadecimal



Word address	Data	Word number
⋮	⋮	⋮
00000010	C D 1 9 A 6 5 B	Word 4
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Recap: Accessing Memory

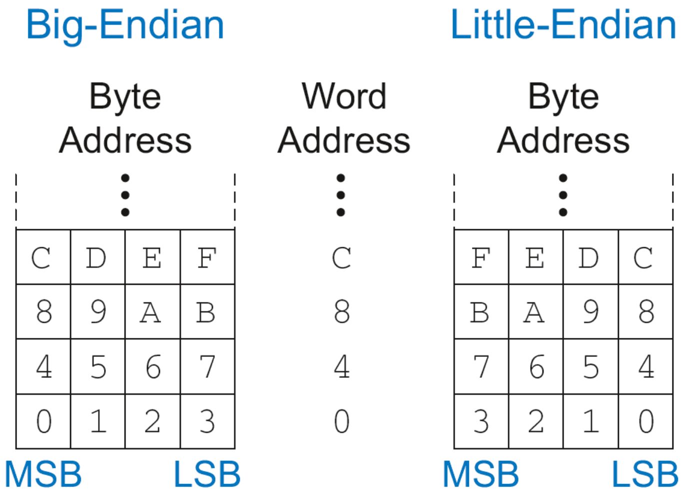
- Address of a memory **word** must be multiplied by 4
- **Examples:**
  - Address of memory word 2 = 2 × 4 = 8
  - Address of memory word 10 = 10 × 4 = 40

Big-Endian & Little-Endian Memory

- **How to number bytes within a word?**

Big-Endian & Little-Endian Memory

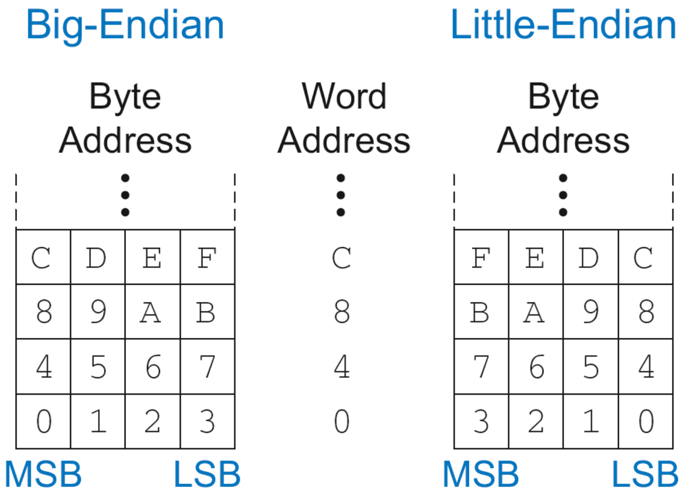
- **How to number bytes within a word?**
  - **Little-endian:** byte numbers start at the **little** (least significant) end
  - **Big-endian:** byte numbers start at the **big** (most significant) end



Big-Endian					Little-Endian			
Byte Address				Word Address	Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Memory

- **Jonathan Swift's *Gulliver's Travels*:** the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- **It doesn't really matter** which addressing type used – **except** when two systems **share data**



Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Example

Suppose R2 and R5 hold the values 8 and 0x23456789

- After following code runs on big-endian system, what value is in `R7`?
- In a little-endian system?

```
STR  R5, [R2, #0]
LDRB R7, [R2, #1]
```

Big-Endian & Little-Endian Example

Suppose R2 and R5 hold the values 8 and 0x23456789

- After following code runs on big-endian system, what value is in `R7`?
- In a little-endian system?

```
STR  R5, [R2, #0]
LDRB R7, [R2, #1]
```

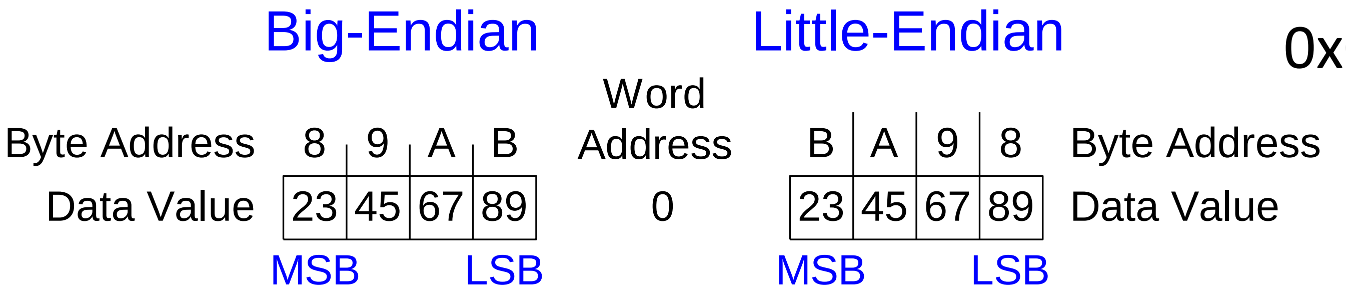
	Big-Endian					Little-Endian				
Byte Address	8	9	A	B	Word Address	B	A	9	8	Byte Address
Data Value	23	45	67	89	0	23	45	67	89	Data Value
	MSB			LSB		MSB			LSB	

Big-Endian & Little-Endian Example

Suppose R2 and R5 hold the values 8 and 0x23456789

- After following code runs on big-endian system, what value is in `R7`?
- In a little-endian system?

```
STR  R5, [R2, #0]
LDRB R7, [R2, #1]
```



	Big-Endian				Word	Little-Endian				
Byte Address	8	9	A	B	Address	B	A	9	8	Byte Address
Data Value	23	45	67	89	0	23	45	67	89	Data Value
	MSB			LSB		MSB			LSB	

Big-endian:
0x00000045

Little-endian:
0x00000067

Programming

High-level languages:

- e.g., C, Java, Python
- Written at higher level of abstraction

Ada Lovelace, 1815-1852

- British mathematician
- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was a child of the poet Lord Byron

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Data-processing Instructions

- Logical operations
- Shifts / rotate
- Multiplication

Logical Instructions

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

Logical Instructions: Examples

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

Logical Instructions: Uses

- AND or BIC: useful for **masking** bits

Example: Masking all but the least significant byte of a value

0xF234012F AND 0x000000FF = 0x0000002F

0xF234012F BIC 0xFFFFFF00 = 0x0000002F

Logical Instructions: Uses

- AND or BIC: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F AND 0x000000FF = 0x0000002F

  0xF234012F BIC 0xFFFFFF00   = 0x0000002F

- ORR: useful for **combining** bit fields

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F `AND` 0x000000FF = 0x0000002F

  0xF234012F `BIC` 0xFFFFFF00   = 0x0000002F

- `ORR`: useful for **combining** bit fields

  Example: Combine 0xF2340000 with 0x000012BC:

  0xF2340000 `ORR` 0x000012BC = 0xF23412BC

Shift Instructions

- `LSL`: logical shift left
- `LSR`: logical shift right
- `ASR`: arithmetic shift right
- `ROR`: rotate right

Shift Instructions

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5  ; R0=R7 << 5`

- `LSR`: logical shift right

- `ASR`: arithmetic shift right

- `ROR`: rotate right

Shift Instructions

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5  ; R0=R7 << 5`

- `LSR`: logical shift right

  Example: `LSR R3, R2, #31 ; R3=R2 >> 31`

- `ASR`: arithmetic shift right

- `ROR`: rotate right

Shift Instructions

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5  ; R0=R7 << 5`

- `LSR`: logical shift right

  Example: `LSR R3, R2, #31 ; R3=R2 >> 31`

- `ASR`: arithmetic shift right

  Example: `ASR R9, R11, R4 ; R9=R11 >>> R4`$_{7:0}$

- `ROR`: rotate right

Shift Instructions

- `LSL`: logical shift left

  Example: `LSL R0, R7, #5  ; R0=R7 << 5`

- `LSR`: logical shift right

  Example: `LSR R3, R2, #31 ; R3=R2 >> 31`

- `ASR`: arithmetic shift right

  Example: `ASR R9, R11, R4 ; R9=R11 >>>` $\mathtt{R4}_{7:0}$

- `ROR`: rotate right

  Example: `ROR R8, R1, #3  ; R8=R1 ROR 3`

Shift Instructions: Example 1

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31

Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Shift Instructions: Example 2

- **Register** shift amount (uses low 8 bits of register)
- Shift amount: 0-255

	Source registers			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Multiplication

- **`MUL`:** 32 × 32 multiplication, 32-bit result

- **`UMULL`:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result

- **`SMULL`:** Signed multiply long: 32 × 32 multiplication, 64-bit result

Multiplication

- **MUL:** 32 × 32 multiplication, 32-bit result

  ```
  MUL R1, R2, R3
  ```

  Result: `R1 = (R2 x R3)`$_{31:0}$

- **UMULL:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result

- **SMULL:** Signed multiply long: 32 × 32 multiplication, 64-bit result

Multiplication

- **MUL:** 32 × 32 multiplication, 32-bit result

  ```
  MUL R1, R2, R3
  ```

  Result: `R1 = (R2 x R3)` $_{31:0}$

- **UMULL:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result

  ```
  UMULL R1, R2, R3, R4
  ```

  Result: `{R1,R4} = R2 x R3` (`R2`, `R3` unsigned)

- **SMULL:** Signed multiply long: 32 × 32 multiplication, 64-bit result

Multiplication

- **MUL:** 32 × 32 multiplication, 32-bit result

  ```
  MUL R1, R2, R3
  ```

  Result: R1 = (R2 x R3)$_{31:0}$

- **UMULL:** Unsigned multiply long: 32 × 32 multiplication, 64-bit result

  ```
  UMULL R1, R2, R3, R4
  ```

  Result: {R1,R4} = R2 x R3 (R2, R3 unsigned)

- **SMULL:** Signed multiply long: 32 × 32 multiplication, 64-bit result

  ```
  SMULL R1, R2, R3, R4
  ```

  Result: {R1,R4} = R2 x R3 (R2, R3 signed)

Programming Building Blocks

- Data-processing Instructions
- **Conditional Execution**
- Branches
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Conditional Execution

Don’t always want to execute code sequentially

- For example:
  - if/else statements, while loops, etc.: only want to execute code *if* a condition is true
  - branching: jump to another portion of code *if* a condition is true

Conditional Execution

Don't always want to execute code sequentially

- For example:
  - if/else statements, while loops, etc.: only want to execute code *if* a condition is true
  - branching: jump to another portion of code *if* a condition is true
- ARM includes **condition flags** that can be:
  - set by an instruction
  - used to conditionally execute an instruction

ARM Condition Flags

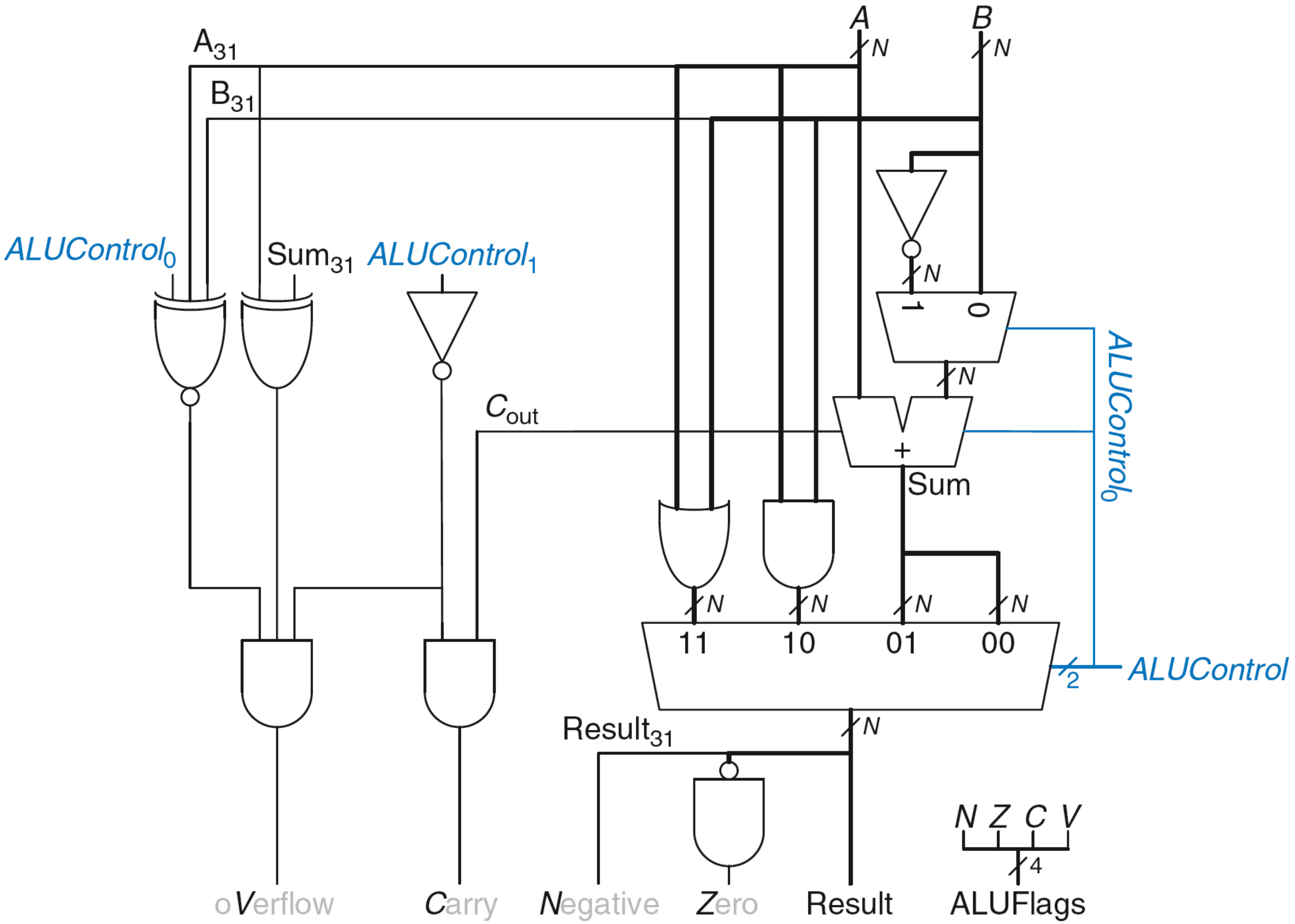
Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

ARM Condition Flags

Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

- Set by ALU (recall from Chapter 5)
- Held in *Current Program Status Register* (*CPSR*)

Review: ARM ALU



Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Does not save result
  - Sets flags

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Does not save result
  - Sets flags. If result:
    - Is 0, *Z*=1
    - Is negative, *N*=1
    - Causes a carry out, *C*=1
    - Causes a signed overflow, *V*=1

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result

- **Method 2:** Append instruction mnemonic with **S**

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`

  - Performs: R5-R6
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result

- **Method 2:** Append instruction mnemonic with **S**

  Example: `ADDS R1, R2, R3`

  - Performs: R2 + R3
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Saves result in R1

Condition Mnemonics

- Instruction may be ***conditionally executed*** based on the condition flags
- Condition of execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

  Example:

  ```
  CMP    R1, R2
  SUBNE  R3, R5, R8
  ```

  - **NE**: condition mnemonic
  - SUB will only execute if R1 ≠ R2 (i.e., Z = 0)

Condition Mnemonics

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

Conditional Execution

Example:

```
CMP   R5, R9             ; performs R5-R9
                         ; sets condition flags

SUBEQ R1, R2, R3         ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9         ; executes if R5-R9 is
                         ; negative (N=1)
```

Conditional Execution

Example:

```
CMP   R5, R9           ; performs R5-R9
                       ; sets condition flags

SUBEQ R1, R2, R3       ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9       ; executes if R5-R9 is
                       ; negative (N=1)
```

Suppose R5 = 17, R9 = 23:

`CMP` performs: 17 – 23 = -6 (Sets flags: *N*=1, *Z*=0, *C*=0, *V*=0)

`SUBEQ` **doesn’t execute** (they aren’t equal: *Z*=0)

`ORRMI` **executes** because the result was negative (*N*=1)

Programming Building Blocks

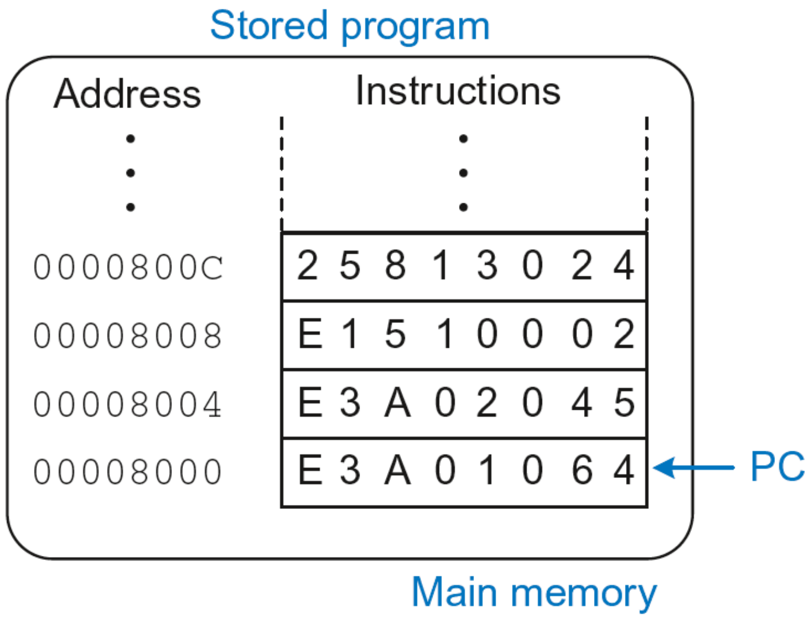
- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Branching

- Branches enable out of sequence instruction execution
- Types of branches:
  - **Branch (`B`)**
    - branches to another instruction
  - **Branch and link (`BL`)**
    - discussed later
- Both can be conditional or unconditional

The Stored Program

Assembly code	Machine code
`MOV   R1, #100`	0xE3A01064
`MOV   R2, #69`	0xE3A02045
`CMP   R1, R2`	0xE1510002
`STRHS R3, [R1, #0x24]`	0x25813024



Address	Instructions
⋮	⋮
0000800C	2 5 8 1 3 0 2 4
00008008	E 1 5 1 0 0 0 2
00008004	E 3 A 0 2 0 4 5
00008000	E 3 A 0 1 0 6 4

Unconditional Branching (B)

ARM assembly

```
  MOV R2, #17         ; R2 = 17
  B   TARGET          ; branch  to  target
  ORR R1, R1, #0x4    ; not executed

TARGET
     SUB R1, R1, #78       ; R1 = R1 + 78
```

Unconditional Branching (B)

ARM assembly

```
   MOV R2, #17          ; R2 = 17
   B   TARGET           ; branch to target
   ORR R1, R1, #0x4     ; not executed

TARGET
       SUB R1, R1, #78       ; R1 = R1 + 78
```

Labels (like TARGET) indicate instruction location. Labels can't be reserved words (like ADD, ORR, etc.)

The Branch Not Taken

ARM Assembly

```
   MOV  R0, #4          ; R0 = 4
   ADD  R1, R0, R0      ; R1 = R0+R0 = 8
   CMP  R0, R1          ; sets flags with R0-R1
   BEQ  THERE           ; branch not taken (Z=0)
   ORR  R1, R1, #1      ; R1 = R1 OR R1 = 9
THERE
   ADD R1, R1, 78       ; R1 = R1 + 78 = 87
```

Programming Building Blocks

- Data-processing Instructions
- Conditional Execution
- Branches
- **High-level Constructs:**
  - **if/else statements**
  - **for loops**
  - **while loops**
  - arrays
  - function calls

if Statement

C Code

```
if (i == j)
  f = g + h;



f = f - i;
```

if Statement

C Code / ARM Assembly Code

```
              ;R0=f, R1=g, R2=h, R3=i, R4=j

if (i == j)    CMP R3, R4      ; set flags with R3-R4
  f = g + h;   BNE L1          ; if i!=j, skip if block
               ADD R0, R1, R2  ; f = g + h

              L1
f = f - i;     SUB R0, R0, R2  ; f = f - i
```

if Statement

C Code   ARM Assembly Code

```
                 ;R0=f, R1=g, R2=h, R3=i, R4=j

if (i == j)       CMP R3, R4        ; set flags with R3-R4
   f = g + h;     BNE L1            ; if i!=j, skip if block
                  ADD R0, R1, R2    ; f = g + h

                 L1
f = f - i;        SUB R0, R0, R2    ; f = f - i
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

if Statement: Alternate Code

C Code

```
if (i == j)
  f = g + h;
f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4       ; set flags with R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

if Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
L1
 SUB R0, R0, R2
```

Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP    R3, R4        ; set flags with R3-R4
 ADDEQ R0, R1, R2    ; if (i==j) f = g + h
 SUB    R0, R0, R2    ; f = f - i
```

if Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
L1
 SUB R0, R0, R2
```

Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4       ; set flags with R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

Useful for **short** conditional blocks of code

if/else Statement

C Code

```
if (i == j)
  f = g + h;


else
  f = f - i;
```

ARM Assembly Code

if/else Statement

C Code

```
if (i == j)
  f = g + h;


else
  f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4        ; set flags with R3-R4
 BNE L1            ; if i!=j, skip if block
 ADD R0, R1, R2    ; f = g + h
 B   L2            ; branch past else block
L1
 SUB R0, R0, R2    ; f = f - i
L2
```

if/else Statement: Alternate Code

C Code

```
if (i == j)
  f = g + h;
else
   f = f – i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4       ; set flags with R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h

 SUBNE R0, R0, R2   ; else f = f - i
```

if/else Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
 B   L2
L1
 SUB R0, R0, R2
L2
```

Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4      ; set flags with R3-R4
 ADDEQ R0, R1, R2  ; if (i==j) f = g + h

 SUBNE R0, R0, R2  ; else f = f - i
```

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV   R0, #1          ; pow = 1
  MOV   R1, #0          ; x = 0

WHILE
  CMP R0, #128          ; R0-128
  BEQ DONE              ; if (pow==128)
                        ; exit loop
  LSL R0, R0, #1        ; pow=pow*2
  ADD R1, R1, #1        ; x=x+1
  B   WHILE             ; repeat loop

DONE
```

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;



while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV   R0, #1            ; pow = 1
  MOV   R1, #0            ; x = 0

WHILE
  CMP R0, #128            ; R0-128
  BEQ DONE                ; if (pow==128)
                          ; exit loop
  LSL R0, R0, #1          ; pow=pow*2
  ADD R1, R1, #1          ; x=x+1
  B   WHILE               ; repeat loop

DONE
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

for Loops

```
for (initialization; condition; loop operation)
    statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

for Loops

C Code

```c
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1           ; i = 1
  MOV   R1, #0           ; sum = 0

FOR
  CMP R0, #10            ; R0-10
  BEQ DONE               ; if (i==10)
                         ; exit loop
  ADD R1, R1, R0         ; sum=sum + i
  ADD R0, R0, #1         ; i = i + 1
  B   FOR                ; repeat loop

DONE
```

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
   MOV   R0, #9            ; i = 9
   MOV   R1, #0            ; sum = 0

FOR
   ADD   R1, R1, R0        ; sum=sum + i
   SUBS  R0, R0, #1        ; i = i - 1
                           ; and set flags

   BNE   FOR               ; if (i!=0)
                           ; repeat loop
```

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #9          ; i = 9
  MOV   R1, #0          ; sum = 0

FOR
  ADD   R1, R1, R0      ; sum=sum + i
  SUBS R0, R0, #1       ; i = i – 1
                        ; and set flags
  BNE   FOR             ; if (i!=0)
                        ; repeat loop
```

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch – instead of 2

Programming Building Blocks

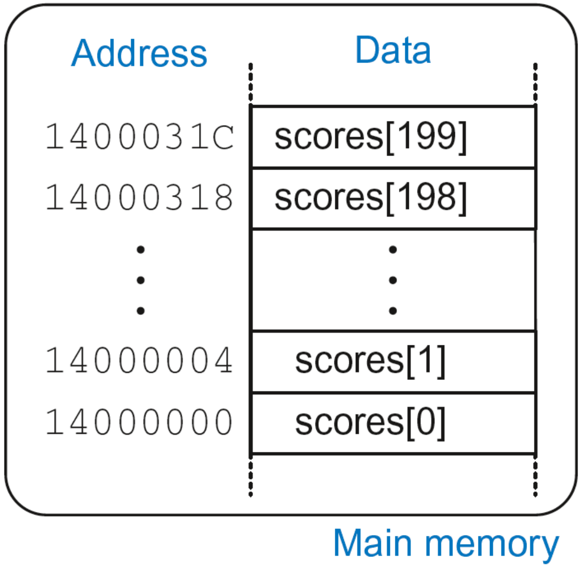
- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - **arrays**
  - function calls

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements

Arrays

- 5-element array
  - **Base address** = 0x14000000 (address of first element, scores[0])
  - Array elements accessed relative to base address



Accessing Arrays

C Code

```
   int array[5];
   array[0] = array[0] * 8;
   array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
```

Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000            ; R0 = 0x60000000

  LDR R1, [R0]                   ; R1 = array[0]
  LSL R1, R1, 3                  ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                   ; array[0] = R1

  LDR R1, [R0, #4]               ; R1 = array[1]
  LSL R1, R1, 3                  ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]               ; array[1] = R1
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
```

Arrays using for Loops

C Code

```
   int array[200];
   int i;

   for (i=199; i >= 0; i = i - 1)
        array[i] = array[i] * 8;
```

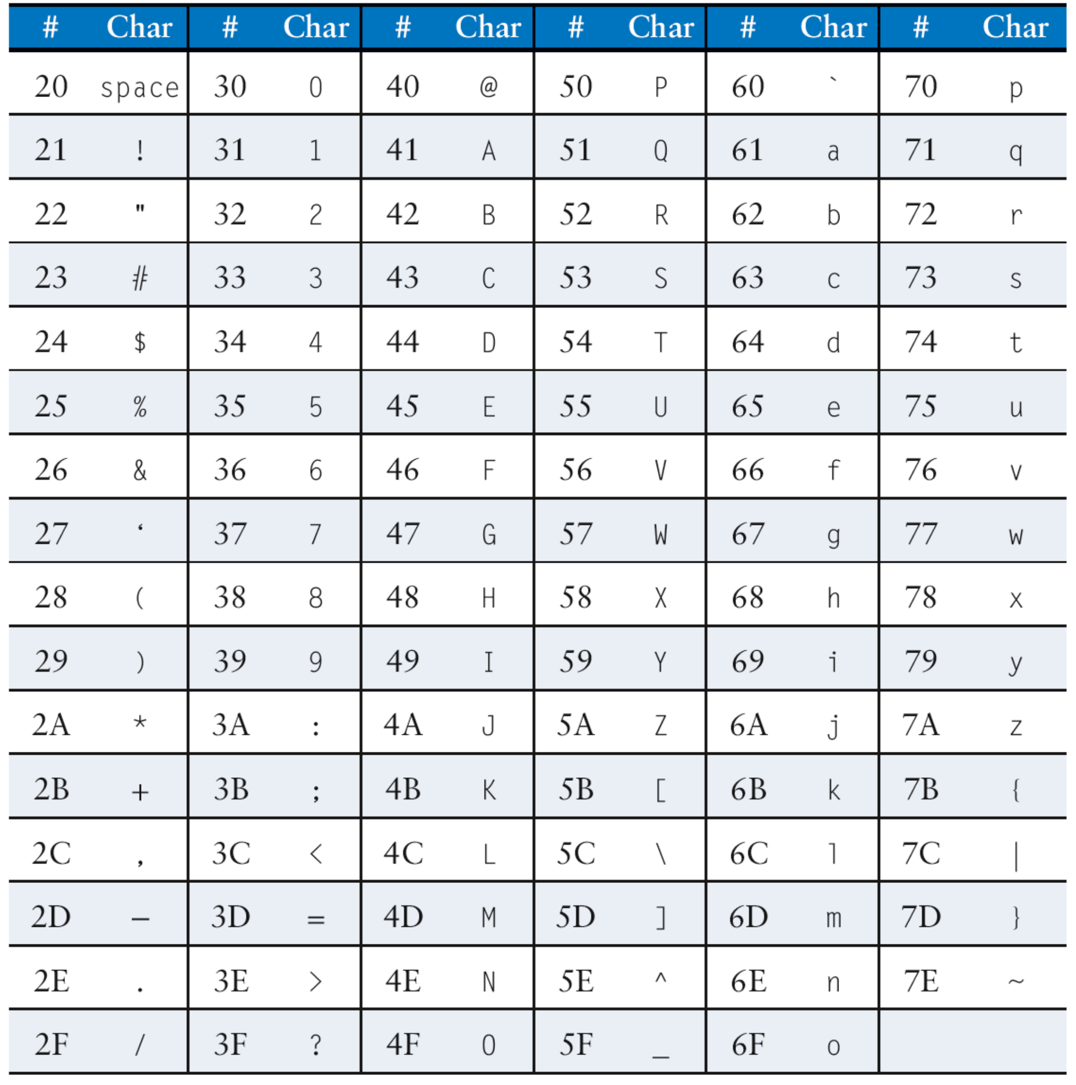
ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]     ; R2 = array(i)
  LSL  R2, R2, #3               ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]     ; array(i) = R2
  SUBS R0, R0, #1               ; i = i - 1
                                ; and set flags
  BPL  FOR                      ; if (i>=0) repeat loop
```

ASCII Code

- American Standard Code for Information Interchange
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters



#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - **function calls**

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

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

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

ARM Function Conventions

- **Call Function:** branch and link
  `BL`
- **Return** from function: move the link register to PC: `MOV PC, LR`
- **Arguments:** `R0-R3`
- **Return value:** `R0`

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN    BL  SIMPLE
0x00000204         ADD R4, R5, R6
...

0x00401020 SIMPLE  MOV PC, LR
```

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL   SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

`void` means that `simple` doesn’t return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

BL branches to `SIMPLE`
`LR` = `PC` + 4 = 0x00000204

MOV PC, LR makes `PC` = `LR`
(the next instruction executed is at 0x00000200)

Input Arguments and Return Value

ARM conventions:

- Argument values: `R0` - `R3`
- Return value: `R0`

Input Arguments and Return Value

C Code

```c
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;               // return value
}
```

Input Arguments and Return Value

ARM Assembly Code

```
; R4 = y
MAIN
  ...
  MOV R0, #2          ; argument 0 = 2
  MOV R1, #3          ; argument 1 = 3
  MOV R2, #4          ; argument 2 = 4
  MOV R3, #5          ; argument 3 = 5
  BL DIFFOFSUMS       ; call function
  MOV R4, R0          ; y = returned value
  ...
; 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
```

Input Arguments and Return Value

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
```

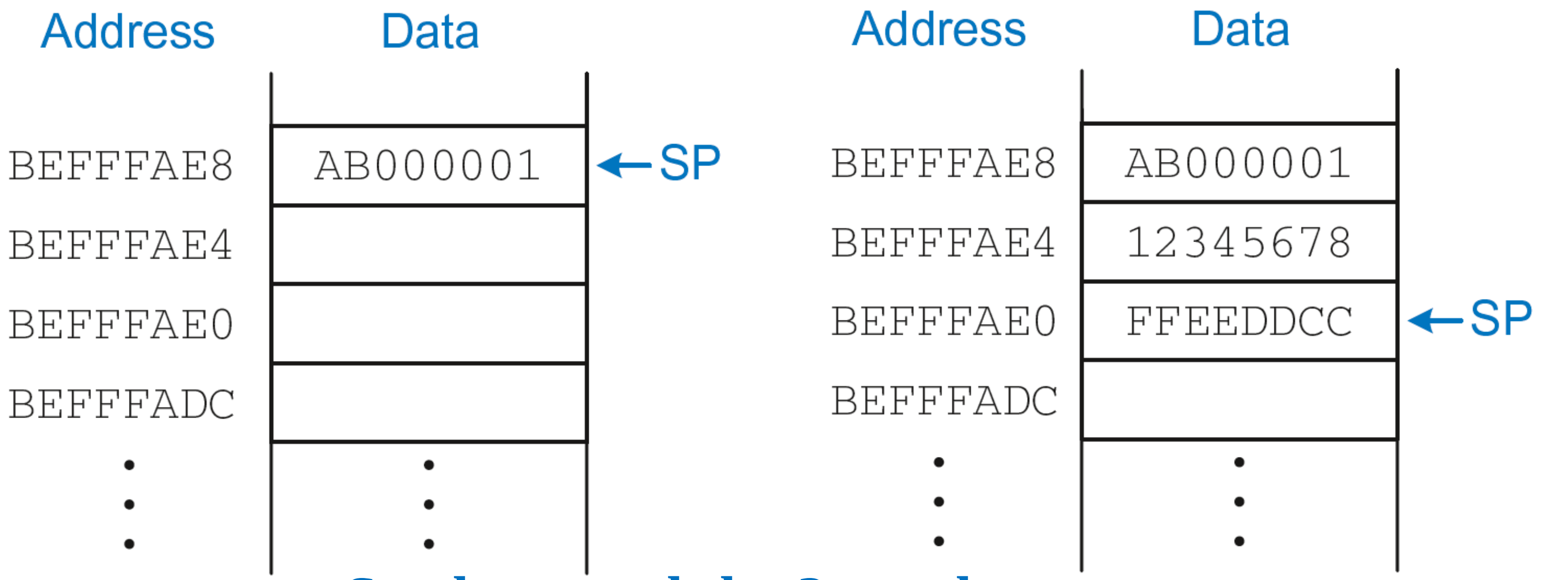
- `diffofsums` overwrote 3 registers: `R4`, `R8`, `R9`
- `diffofsums` can use *stack* to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space no longer needed

The Stack

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



Address	Data	
BEFFFAE8	AB000001	← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
⋮	⋮	

Address	Data	
BEFFFAE8	AB000001	
BEFFFAE4	12345678	
BEFFFAE0	FFEEDDCC	← SP
BEFFFADC		
⋮	⋮	

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

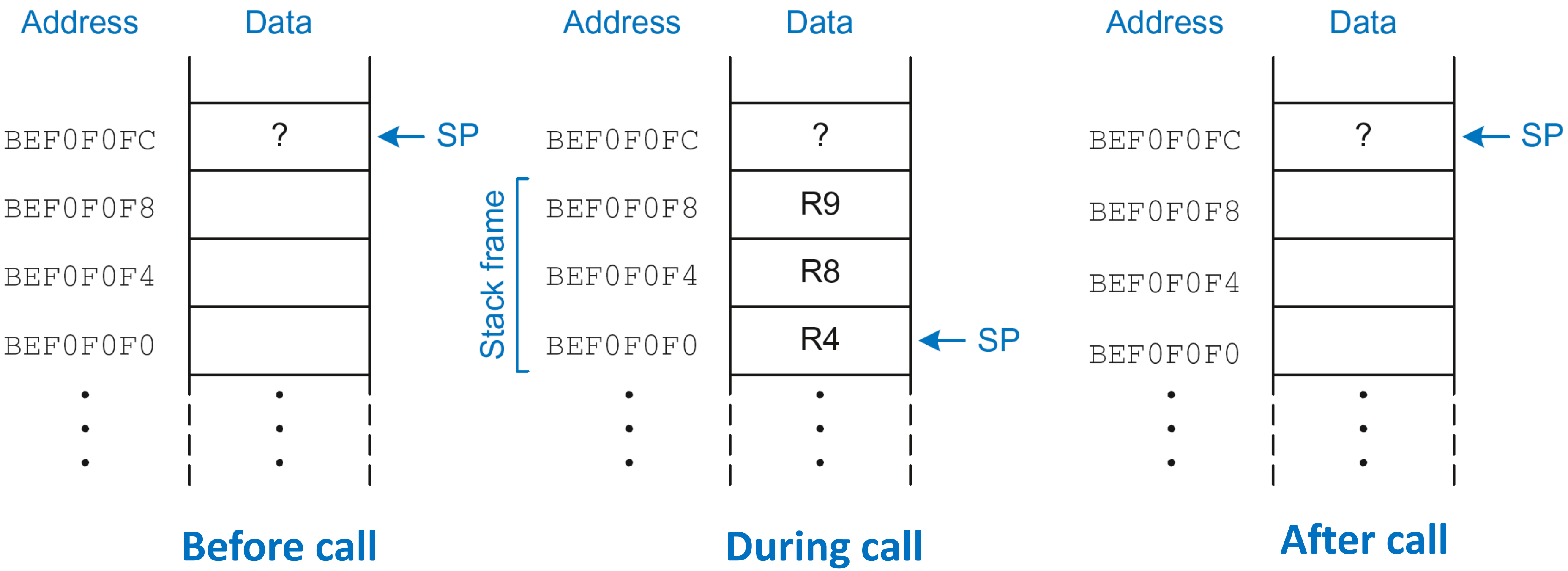
```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1        ; R8 = f + g
  ADD R9, R2, R3        ; R9 = h + i
  SUB R4, R8, R9        ; result = (f + g) - (h + i)
  MOV R0, R4            ; put return value in R0
  MOV PC, LR            ; return to caller
```

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



Before call

Address	Data	
BEF0F0FC	?	← SP
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

During call

	Address	Data	
	BEF0F0FC	?	
Stack frame	BEF0F0F8	R9	
	BEF0F0F4	R8	
	BEF0F0F0	R4	← SP
	⋮	⋮	

After call

Address	Data	
BEF0F0FC	?	← SP
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

Registers

Preserved *Callee-Saved*	**Nonpreserved** *Caller-Saved*
R4-R11	**R12**
R14 (LR)	**R0-R3**
R13 (SP)	**CPSR**
stack above SP	**stack below SP**

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
   STR R4, [SP, #-4]!  ; save R4 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 R4, [SP], #4    ; restore R4 from stack
   MOV PC, LR          ; return to caller
```

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  STR R4, [SP, #-4]! ; save R4 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 R4, [SP], #4   ; restore R4 from stack
  MOV PC, LR         ; return to caller
```

Notice code optimization for expanding/contracting stack

Nonleaf Function

ARM Assembly Code

```
STR LR, [SP, #-4]!      ; store LR on stack
BL  PROC2               ; call another function
...
LDR LR, [SP], #4        ; restore LR from stack
jr  $ra                 ; return to caller
```

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

```c
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
```

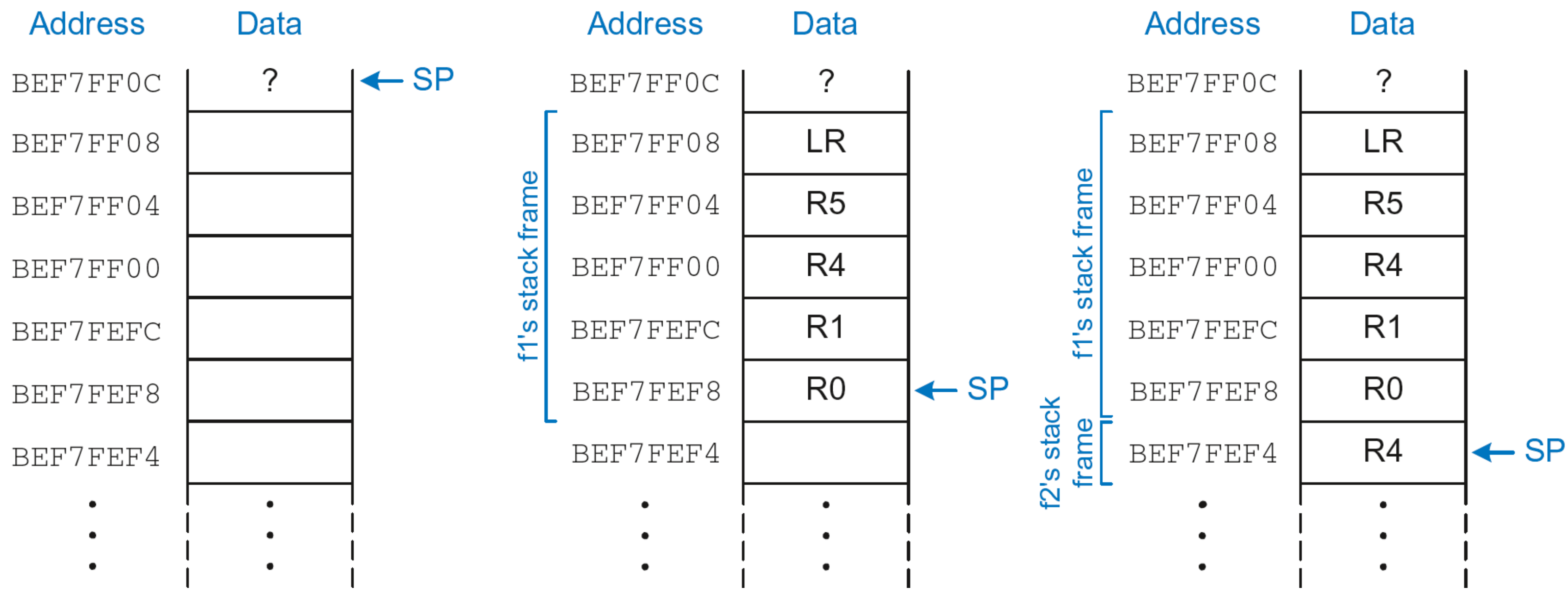
Nonleaf Function Example

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



Address	Data	
BEF7FF0C	?	← SP
BEF7FF08		
BEF7FF04		
BEF7FF00		
BEF7FEFC		
BEF7FEF8		
BEF7FEF4		
⋮	⋮	

At beginning of f1

Address	Data	
BEF7FF0C	?	
BEF7FF08	LR	
BEF7FF04	R5	
BEF7FF00	R4	
BEF7FEFC	R1	
BEF7FEF8	R0	← SP
BEF7FEF4		
⋮	⋮	

Just before calling f2

Address	Data	
BEF7FF0C	?	
BEF7FF08	LR	
BEF7FF04	R5	
BEF7FF00	R4	
BEF7FEFC	R1	
BEF7FEF8	R0	
BEF7FEF4	R4	← SP
⋮	⋮	

After calling f2

Recursive Function Call

C Code

```
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

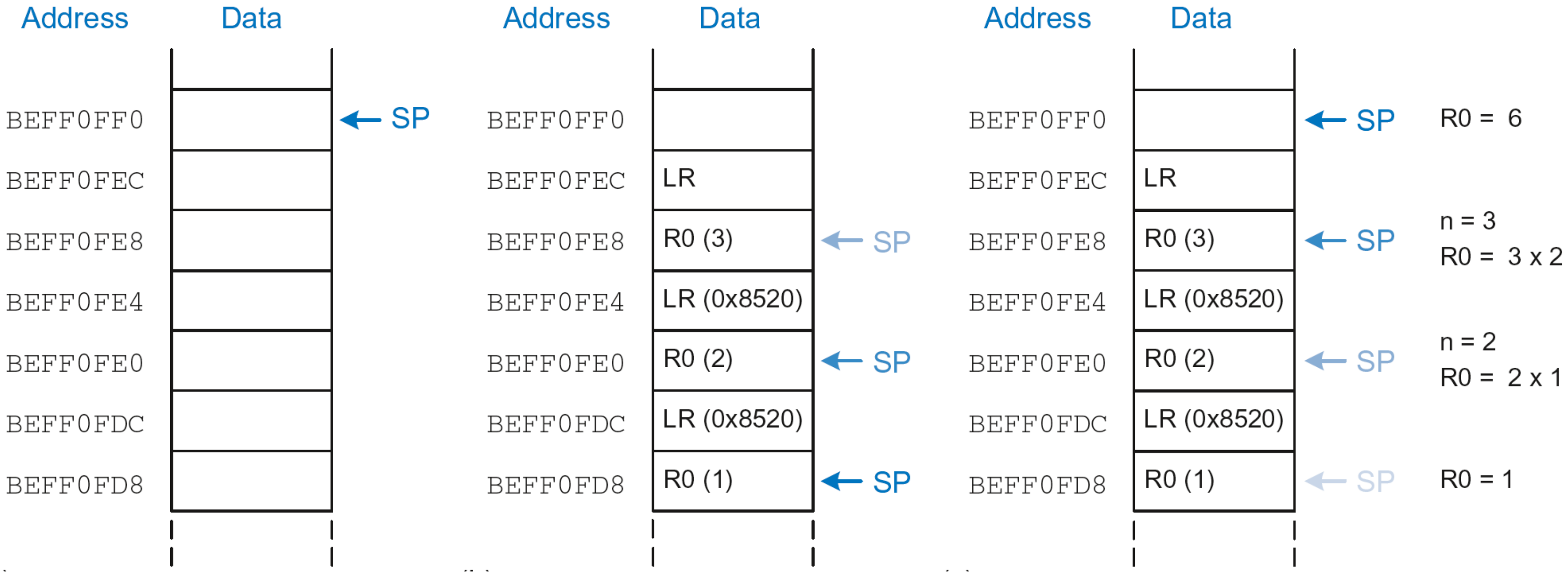
C Code

```
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



Address	Data	
BEFF0FF0		← SP
BEFF0FEC		
BEFF0FE8		
BEFF0FE4		
BEFF0FE0		
BEFF0FDC		
BEFF0FD8		

Before call

Address	Data	
BEFF0FF0		
BEFF0FEC	LR	
BEFF0FE8	R0 (3)	← SP
BEFF0FE4	LR (0x8520)	
BEFF0FE0	R0 (2)	← SP
BEFF0FDC	LR (0x8520)	
BEFF0FD8	R0 (1)	← SP

During call

Address	Data		
BEFF0FF0		← SP	R0 = 6
BEFF0FEC	LR		

BEFF0FE8	R0 (3)	← SP	n = 3
			R0 = 3 x 2

BEFF0FE4	LR (0x8520)		

BEFF0FE0	R0 (2)	← SP	n = 2
			R0 = 2 x 1

BEFF0FDC	LR (0x8520)		
BEFF0FD8	R0 (1)	← SP	R0 = 1

After 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`

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

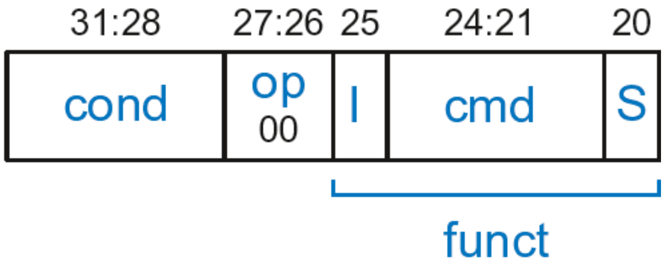
- **Operands:**
  - ***Rn*****:** first source register
  - ***Src2*****:** second source – register or immediate
  - ***Rd*****:** destination register
- **Control fields:**
  - ***cond*****:** specifies conditional execution
  - ***op*****:** the *operation code* or *opcode*
  - ***funct*****:** the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

Data-processing

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

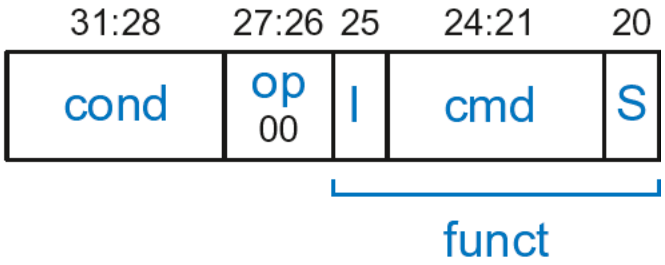
Data-processing Control Fields

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



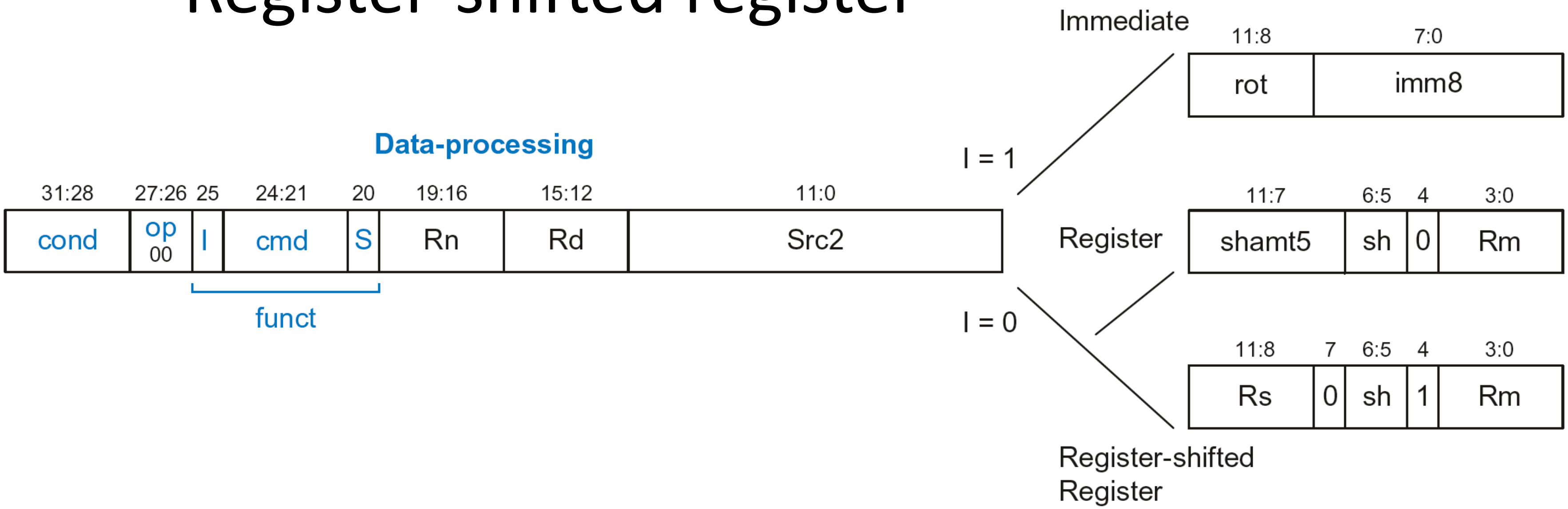
Data-processing Control Fields

- ***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`



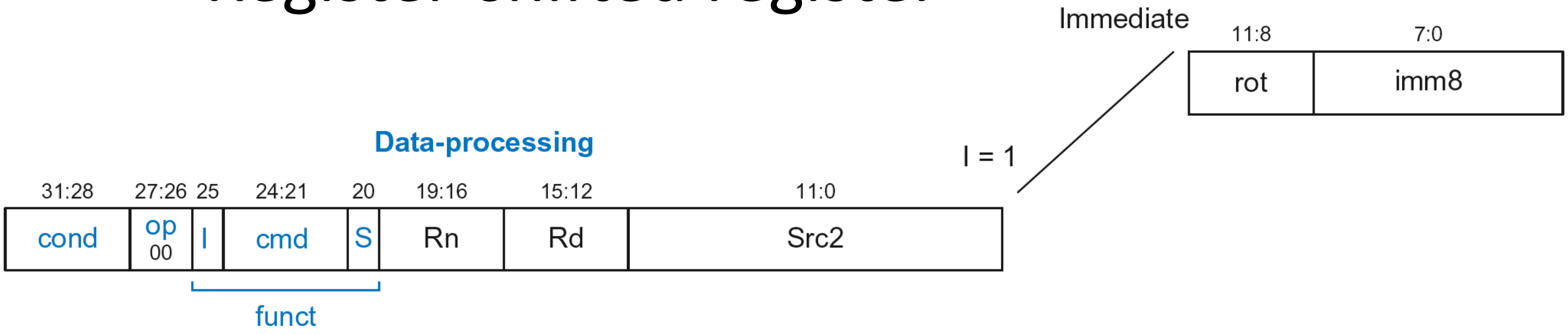
Data-processing *Src2* Variations

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



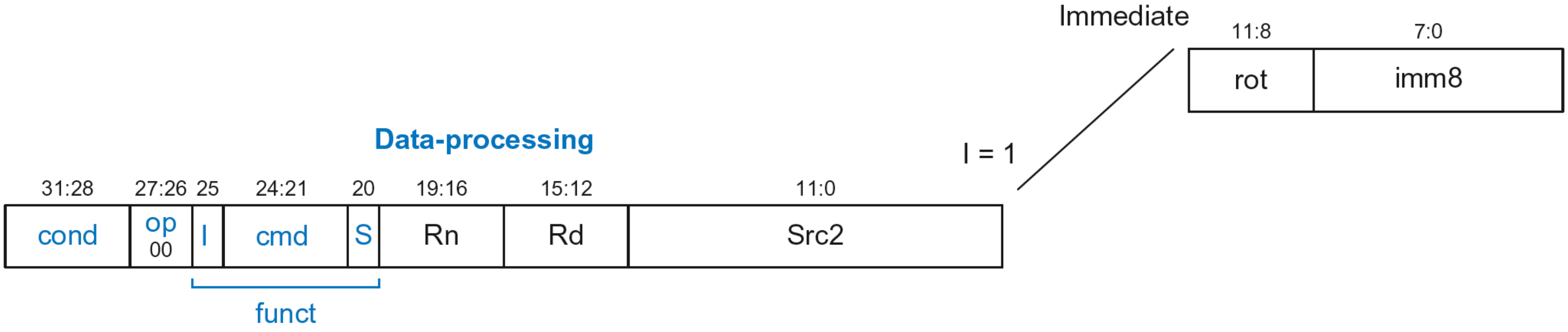
Data-processing *Src2* Variations

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



Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)



Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefgh

rot	**32-bit constant**
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value

ROR by X = ROL by (32-X)
Ex: ROR by 30 = ROL by 2

- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefgh

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

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
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

DP Instruction with Immediate *Src2*

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

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
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

0xE281002A

DP Instruction with Immediate *Src2*

```
SUB R2, R3, #0xFF0
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0010_2 (2) for SUB
- ***Src2*** is an immediate so ***I***=1
- ***Rd*** = 2, ***Rn*** = 3
- ***imm8*** = 0xFF
- ***imm8*** must be rotated right by 28 to produce 0xFF0, so ***rot*** = 14

DP Instruction with Immediate *Src2*

```
SUB R2, R3, #0xFF0
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0010_2 (2) for SUB
- ***Src2*** is an immediate so ***I***=1
- ***Rd*** = 2, ***Rn*** = 3
- ***imm8*** = 0xFF
- ***imm8*** must be rotated right by 28 to produce 0xFF0, so ***rot*** = 14

ROR by 28 =

ROL by (32-28) = 4

DP Instruction with Immediate *Src2*

```
SUB R2, R3, #0xFF0
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0010_2 (2) for SUB
- ***Src2*** is an immediate so ***I***=1
- ***Rd*** = 2, ***Rn*** = 3
- ***imm8*** = 0xFF
- ***imm8*** must be rotated right by 28 to produce 0xFF0, so ***rot*** = 14

ROR by 28 =

ROL by (32-28) = 4

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

DP Instruction with Immediate *Src2*

```
SUB R2, R3, #0xFF0
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0010_2 (2) for SUB
- ***Src2*** is an immediate so ***I***=1
- ***Rd*** = 2, ***Rn*** = 3
- ***imm8*** = 0xFF
- ***imm8*** must be rotated right by 28 to produce 0xFF0, so ***rot*** = 14

ROR by 28 =

ROL by (32-28) = 4

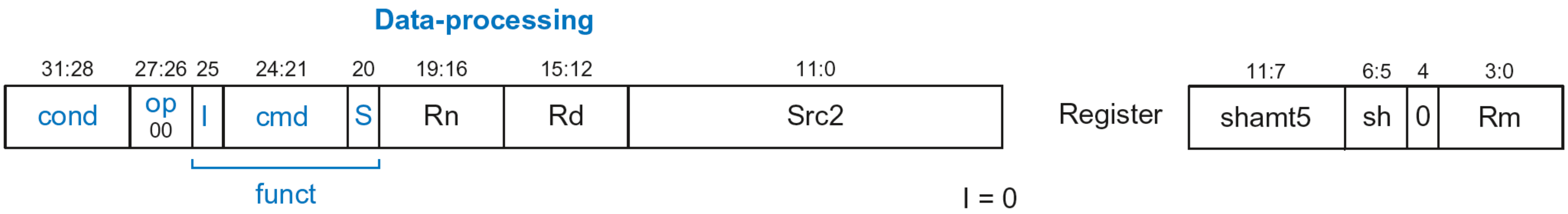
Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

0xE2432EFF

DP Instruction with Register *Src2*

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



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

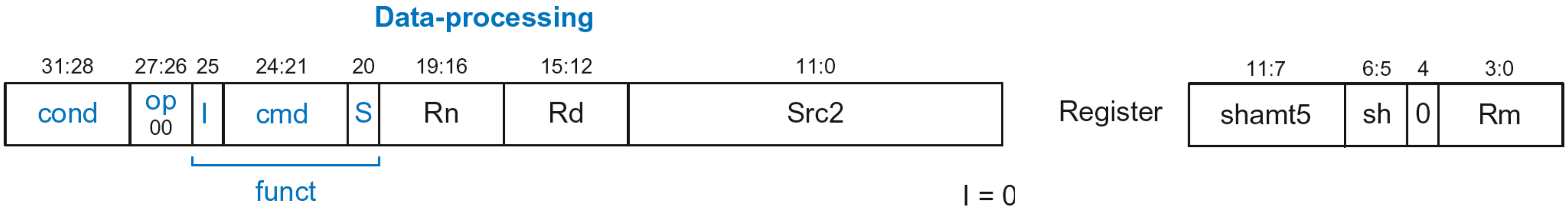
I = 0

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

DP Instruction with Register *Src2*

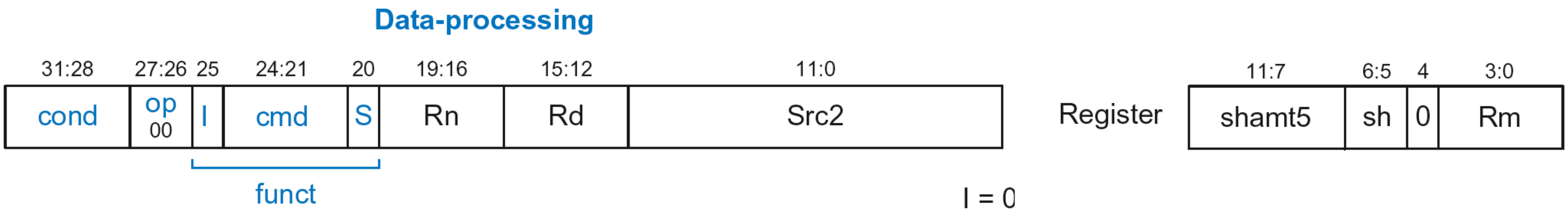
- ***Rm***: the second source operand
- ***shamt5***: the amount Rm is shifted
- ***sh***: the type of shift (i.e., >>, <<, >>>, ROR)



DP Instruction with Register *Src2*

- ***Rm***: the second source operand
- ***shamt5***: the amount rm is shifted
- ***sh***: the type of shift (i.e., >>, <<, >>>, ROR)

First, consider unshifted versions of *Rm* (*shamt5*=0, *sh*=0)



DP Instruction with Register *Src2*

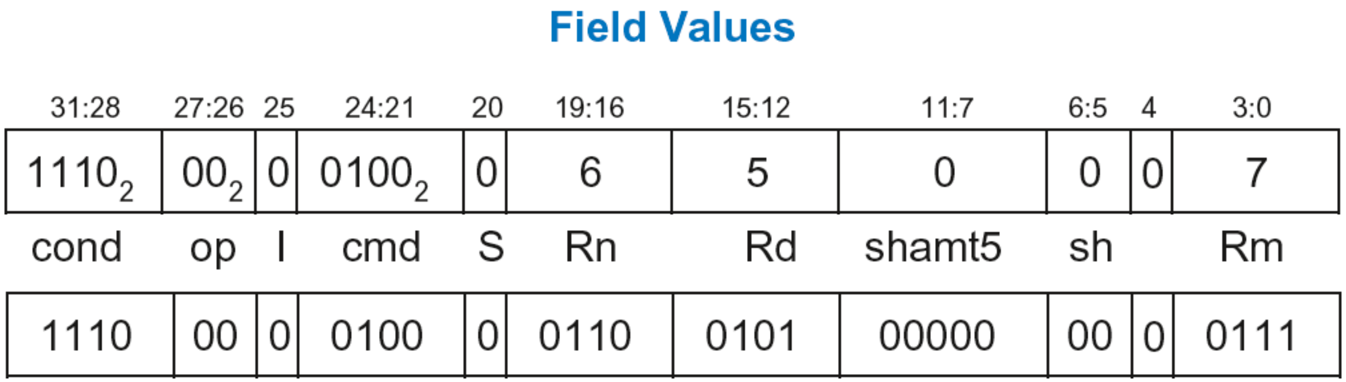
`ADD R5, R6, R7`

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for `ADD`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 6, ***Rm*** = 7
- ***shamt*** = 0, ***sh*** = 0

DP Instruction with Register *Src2*

`ADD R5, R6, R7`

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for `ADD`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 6, ***Rm*** = 7
- ***shamt*** = 0, ***sh*** = 0



Field Values

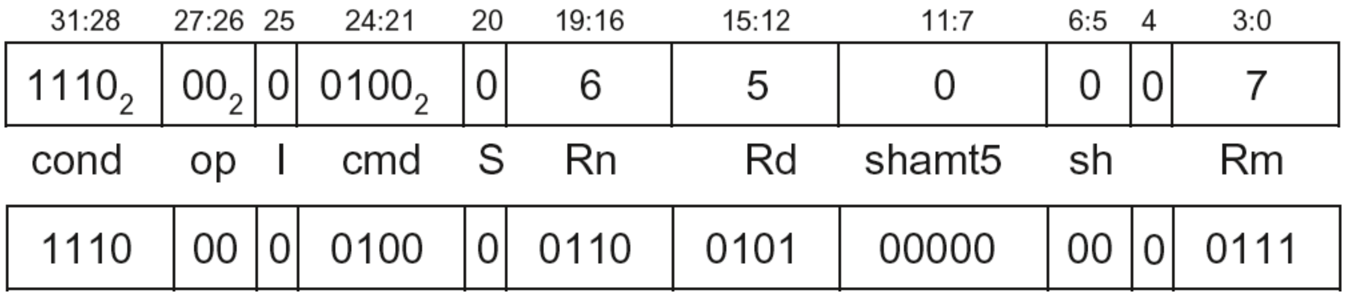
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	0	0100	0	0110	0101	00000	00	0	0111

DP Instruction with Register *Src2*

`ADD R5, R6, R7`

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for `ADD`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 6, ***Rm*** = 7
- ***shamt*** = 0, ***sh*** = 0

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
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	0	0100	0	0110	0101	00000	00	0	0111

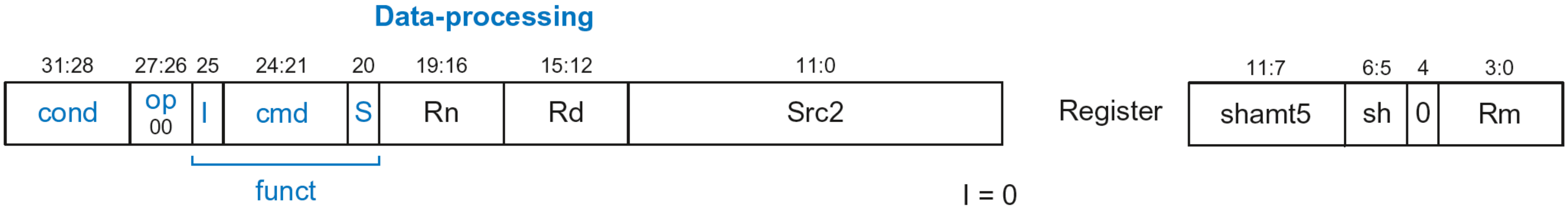
0xE0865007

DP Instruction with Register *Src2*

- ***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

Now, consider shifted versions.



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

I = 0

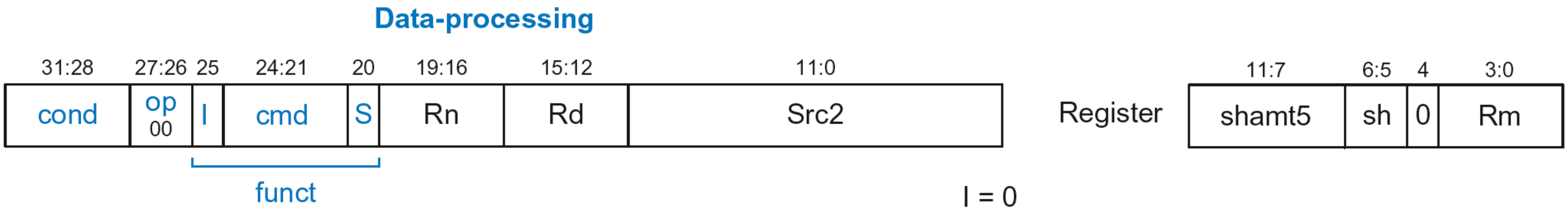
Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

DP Instruction with Register *Src2*

`ORR R9, R5, R3,` **`LSR #2`**

- **Operation:** R9 = R5 OR (R3 >> 2)
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1100_2 (12) for `ORR`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 9, ***Rn*** = 5, ***Rm*** = 3
- ***shamt5*** = 2, ***sh*** = 01_2 (LSR)



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct

I = 0

Register

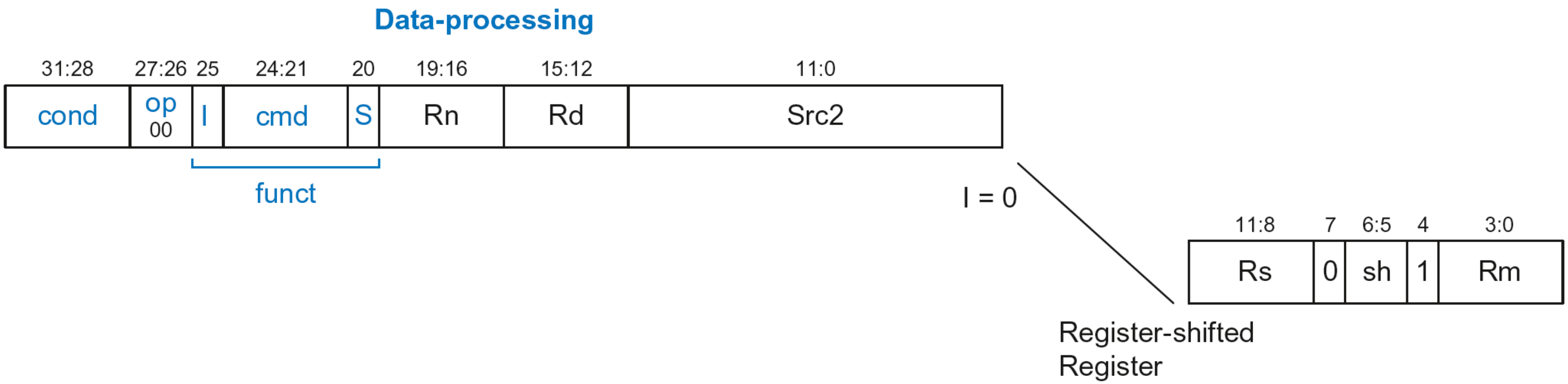
11:7	6:5	4	3:0
shamt5	sh	0	Rm

1110 00 0 1100 0 0101 1001 00010 01 0 0011

0xE1859123

DP with Register-shifted Reg. *Src2*

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



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

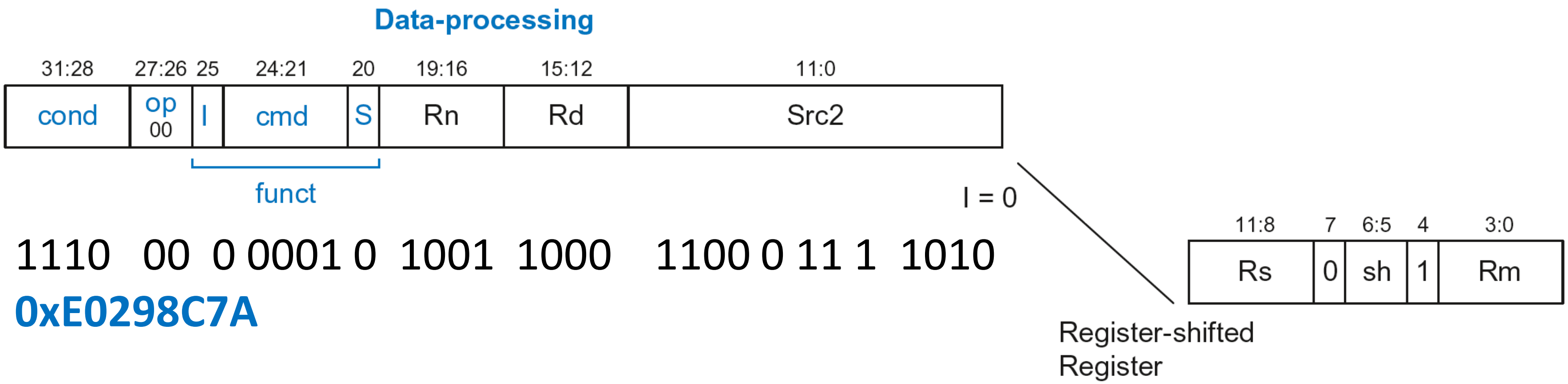
11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

Register-shifted Register

DP with Register-shifted Reg. *Src2*

```
EOR R8, R9, R10, ROR R12
```

- **Operation:** R8 = R9 XOR (R10 ROR R12)
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0001_2 (1) for `EOR`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 8, ***Rn*** = 9, ***Rm*** = 10, ***Rs*** = 12
- ***sh*** = 11_2 (ROR)



1110   00 0 0001 0 1001 1000   1100 0 11 1 1010

0xE0298C7A

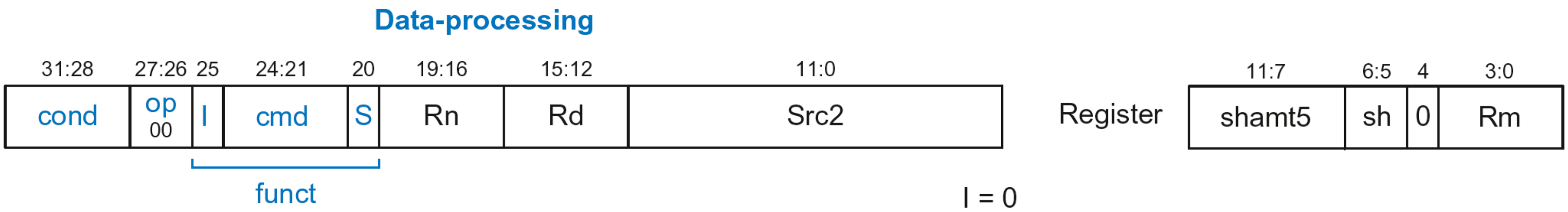
Shift Instructions Encoding

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

Shift Instructions: Immediate shamt

```
ROR R1, R2, #23
```

- **Operation:** R1 = R2 ROR 23
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is an immediate-shifted register so ***I***=0
- ***Rd*** = 1, ***Rn*** = 0, ***Rm*** = 2
- ***shamt5*** = 23, ***sh*** = 11_2 (ROR)



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2
		funct					I = 0

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

1110 00 0 1101 0 0000 0001 10111 11 0 0010

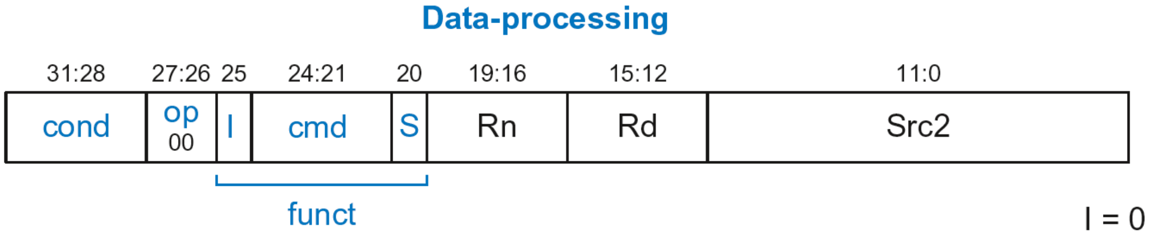
0xE1A01BE2

Shift Instructions: Immediate shamt

```
ROR R1, R2, #23
```

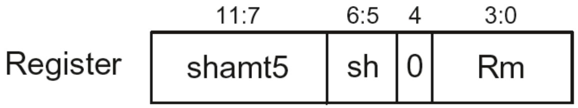
- **Operation:** R1 = R2 ROR 23
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is an immediate-shifted register so ***I***=0
- ***Rd*** = 1, ***Rn*** = 0, ***Rm*** = 2
- ***shamt5*** = 23, ***sh*** = 11_2 (ROR)

Uses (immediate-shifted) register *Src2* encoding



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2



	11:7	6:5	4	3:0
Register	shamt5	sh	0	Rm

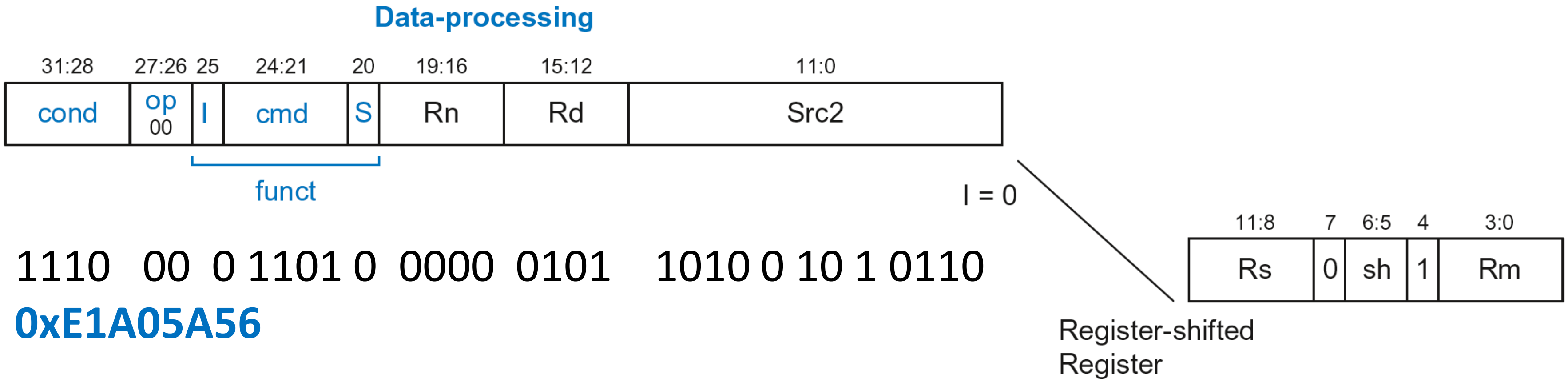
1110   00 0 1101 0 0000 0001   10111 11 0 0010

0xE1A01BE2

Shift Instructions: Register shamt

```
ASR R5, R6, R10
```

- **Operation:** R5 = R6 >>> $R10_{7:0}$
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 0, ***Rm*** = 6, ***Rs*** = 10
- ***sh*** = 10_2 (ASR)



1110 00 0 1101 0 0000 0101 1010 0 10 1 0110

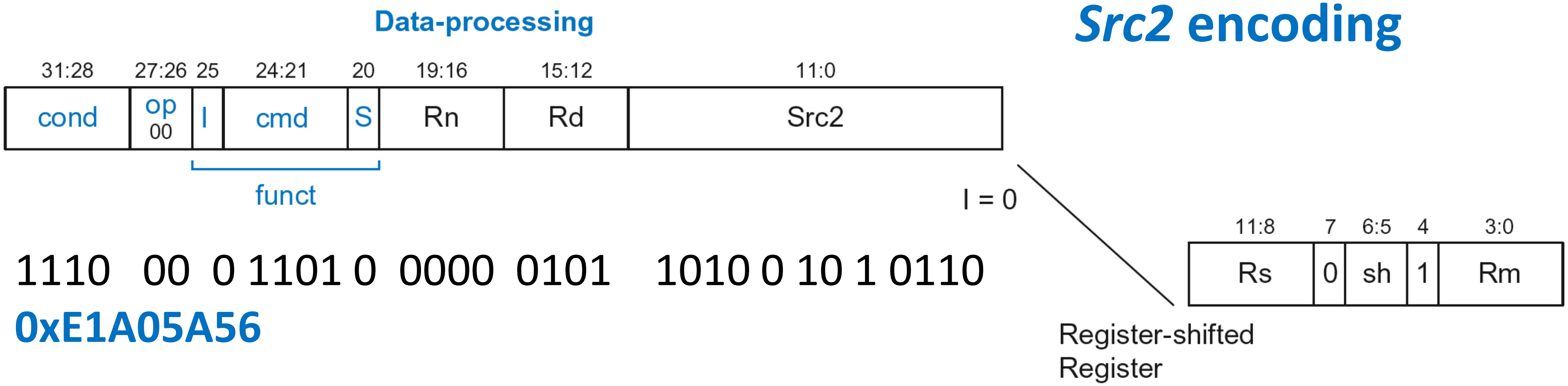
0xE1A05A56

Shift Instructions: Register shamt

```
ASR R5, R6, R10
```

- **Operation:** R5 = R6 >>> $R10_{7:0}$
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 0, ***Rm*** = 6, ***Rs*** = 10
- ***sh*** = 10_2 (ASR)

Uses register-shifted register *Src2* encoding

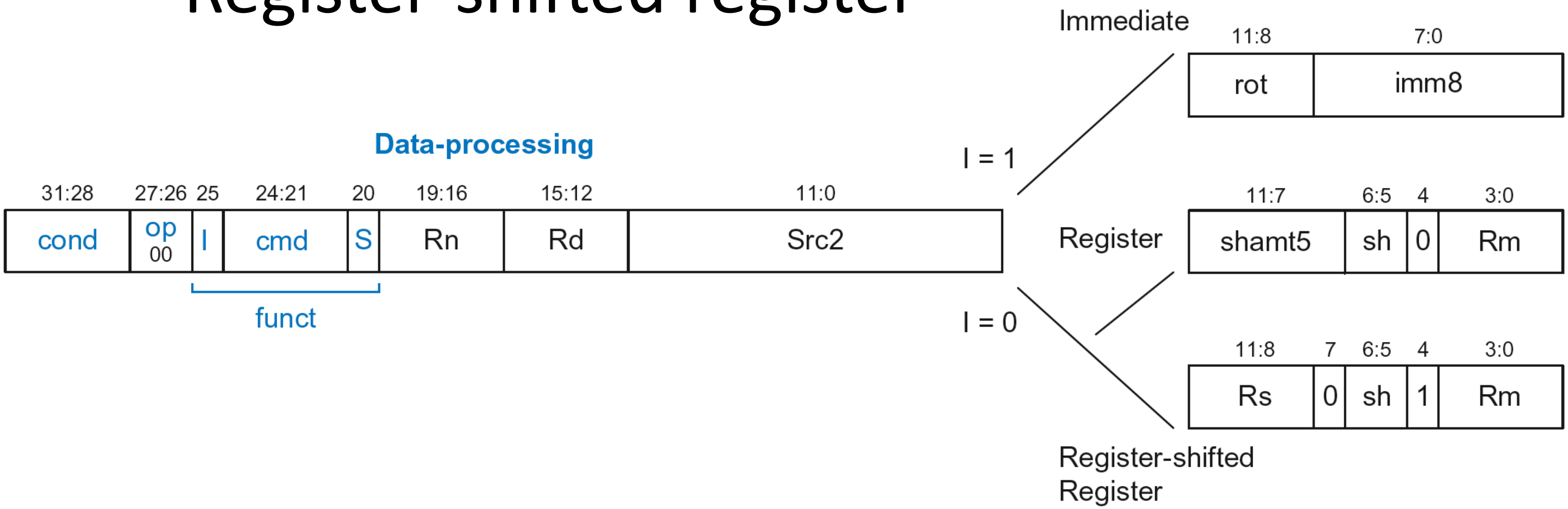


1110 00 0 1101 0 0000 0101 1010 0 10 1 0110

0xE1A05A56

Review: Data-processing Format

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



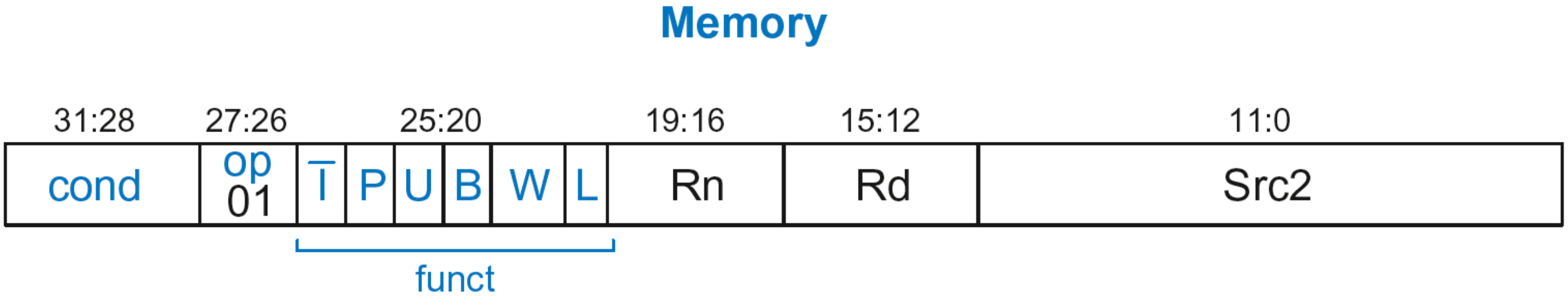
Instruction Formats

- 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

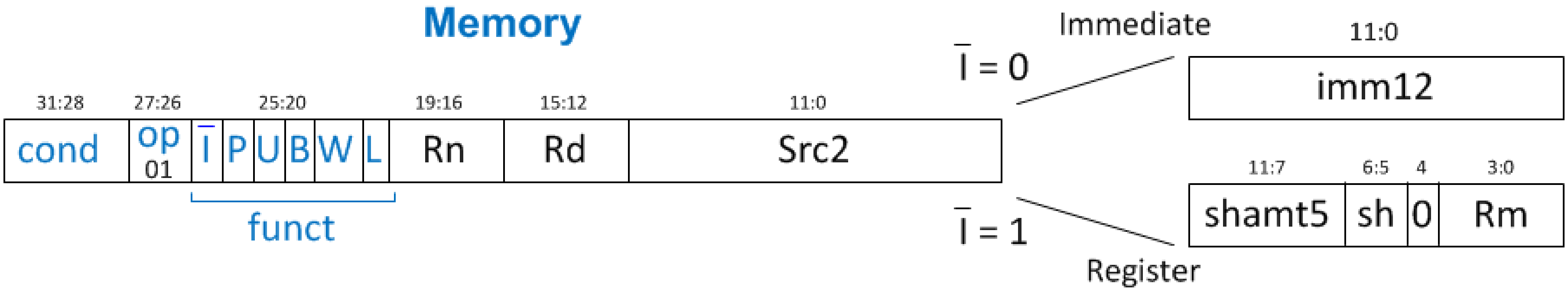
Offset Examples

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



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:**

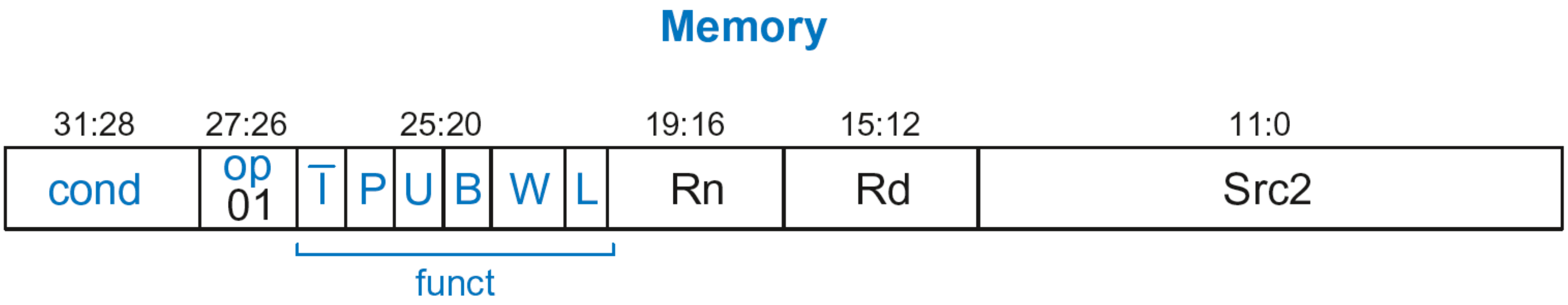
```
LDR R3, [R5, #16]!   ; R3 = mem[R5+16]
                     ; R5 = R5 + 16
```

- **Postindex:**

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

Memory Instruction Format

- ***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

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

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

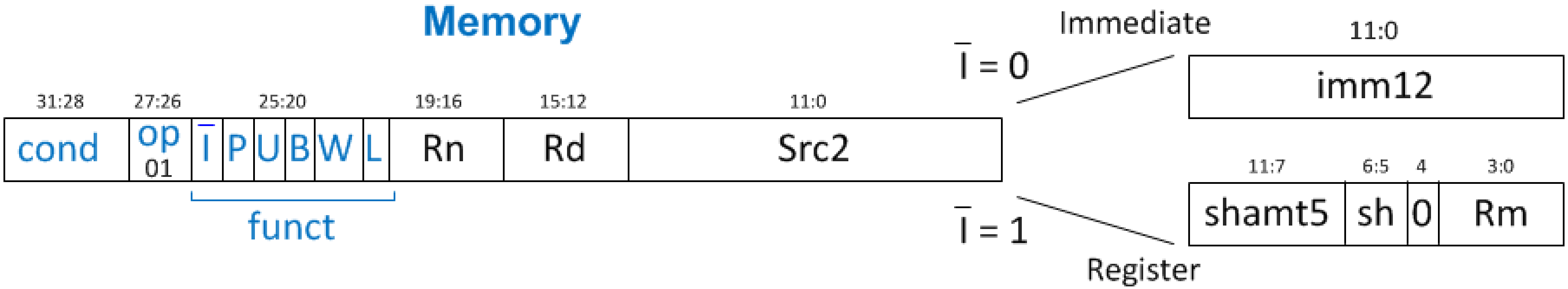
Add/Subtract Immediate/Register Offset

Value	$\overline{I}$	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

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 Instr. with Immediate *Src2*

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

- **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)

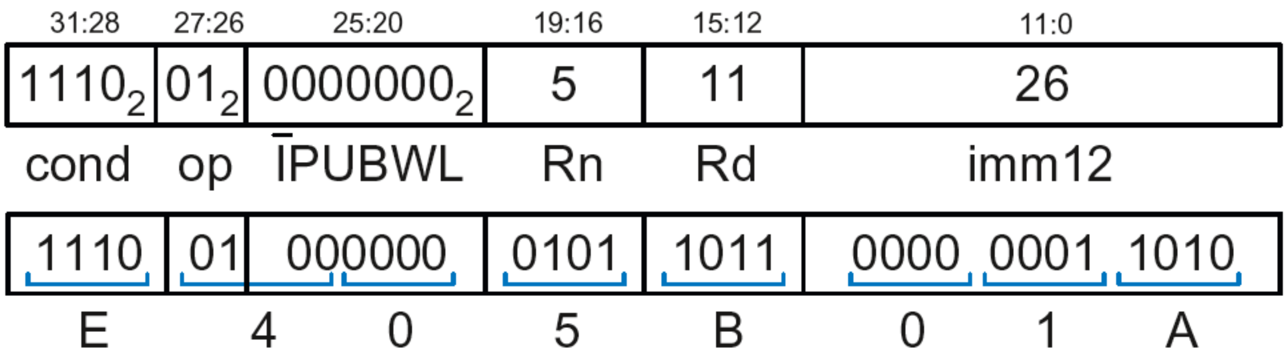
  $\overline{I}$ = 0 (immediate offset), ***P*** = 0 (postindex),
  U = 0 (subtract), ***B*** = 0 (store word), ***W*** = 0 (postindex),
  L = 0 (store)
- ***Rd*** = 11, ***Rn*** = 5, ***imm12*** = 26

Memory Instr. with Immediate *Src2*

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

- **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)
  $\bar{I}$ = 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

Field Values



31:28	27:26	25:20	19:16	15:12	11:0
1110_2	01_2	0000000_2	5	11	26
cond	op	$\bar{I}$PUBWL	Rn	Rd	imm12
1110	01	000000	0101	1011	0000 0001 1010

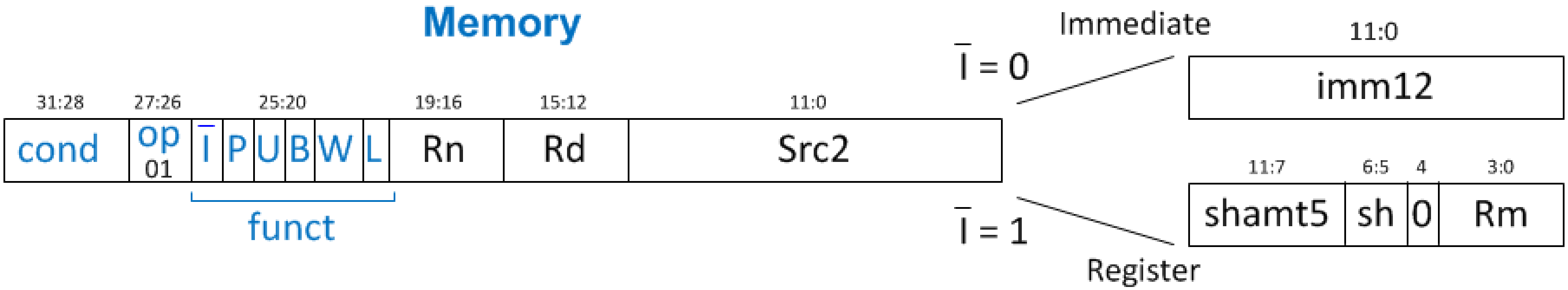
E 4 0 5 B 0 1 A

Memory Instr. with Register *Src2*

```
LDR R3, [R4, R5]
```

- **Operation:** R3 <= mem[R4 + R5]
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111001_2 (57)
  $\overline{I}$ = 1 (register offset), ***P*** = 1 (offset indexing),
  U = 1 (add), ***B*** = 0 (load **word**), ***W*** = 0 (offset indexing),
  L = 1 (load)
- ***Rd*** = 3, ***Rn*** = 4, ***Rm*** = 5 (***shamt5*** = 0, ***sh*** = 0)

1110 01 111001 0100 0011 00000 00 0 0101 = **0xE7943005**



Memory

31:28 | 27:26 | 25:20 | 19:16 | 15:12 | 11:0

cond | op 01 | $\overline{I}$ P U B W L | Rn | Rd | Src2

funct

$\overline{I}$ = 0 Immediate: 11:0 imm12

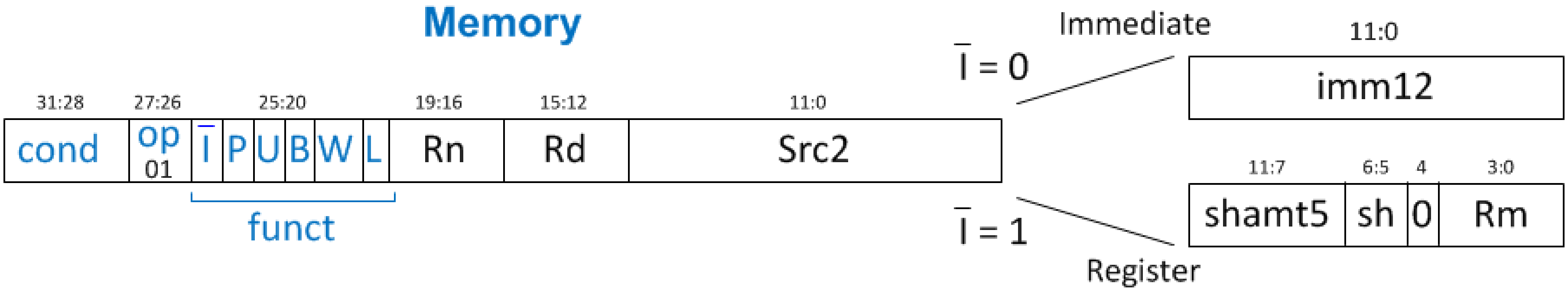
$\overline{I}$ = 1 Register: 11:7 shamt5 | 6:5 sh | 4 0 | 3:0 Rm

Memory Instr. with Scaled Reg. *Src2*

```
STR R9, [R1, R3, LSL #2]
```

- **Operation:** mem[R1 + (R3 << 2)] <= R9
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111000_2 (0)
  $\overline{I}$ = 1 (register offset), ***P*** = 1 (offset indexing),
  U = 1 (add), ***B*** = 0 (store **word**), ***W*** = 0 (offset indexing),
  L = 0 (store)
- ***Rd*** = 9, ***Rn*** = 1, ***Rm*** = 3, ***shamt*** = 2, ***sh*** = 00_2 (LSL)

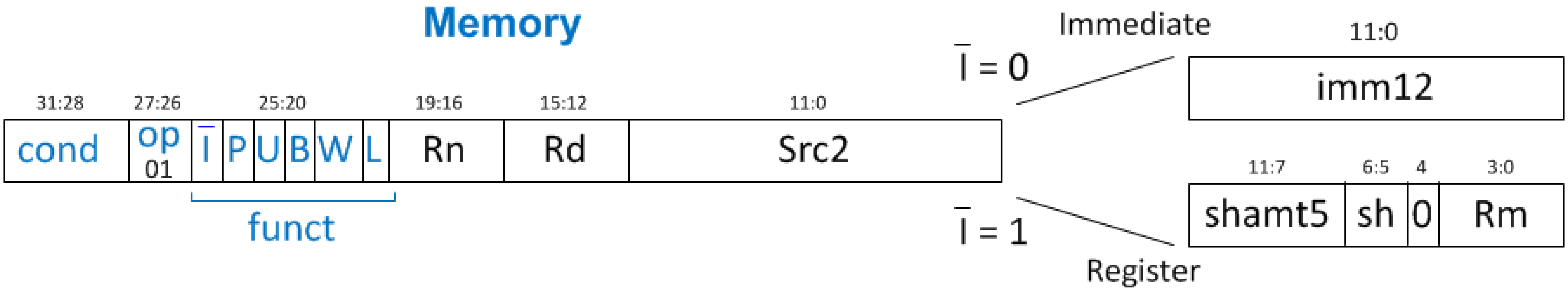
1110 01 111000 0001 1001 00010 00 0 0011 = **0xE7819103**



Review: 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*** = $\bar{I}$ (immediate bar), *P* (preindex), *U* (add), *B* (byte), *W* (writeback), *L* (load)



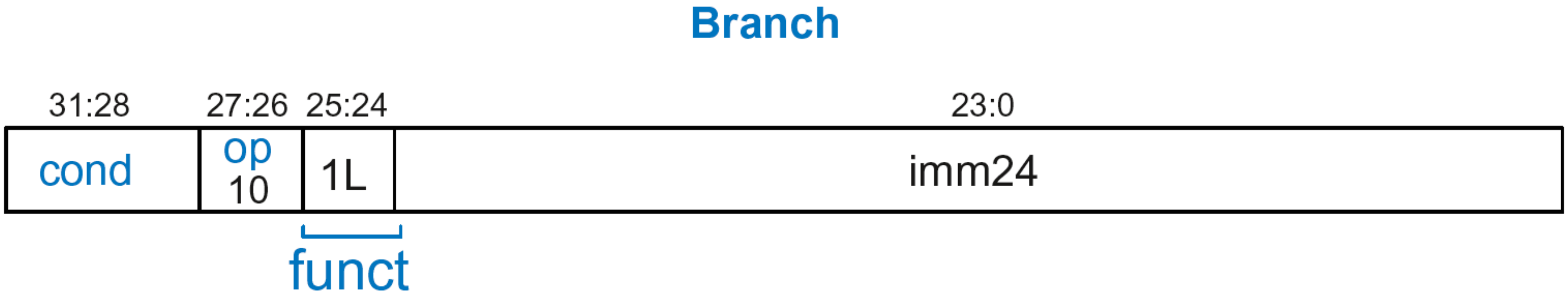
Instruction Formats

- 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

31:28	27:26	25:24	23:0
1011_2	10_2	10_2	3
cond	opfunct		imm24
1011	10	10	0000 0000 0000 0000 0000 0011

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

31:28	27:26	25:24	23:0
1011_2	10_2	10_2	3
cond	opfunct		imm24
1011	10	10	0000 0000 0000 0000 0000 0011

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

31:28	27:26	25:24	23:0
1110_2	10_2	11_2	-6
cond	op	funct	imm24
1110	10	11	1111 1111 1111 1111 1111 1010

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
```

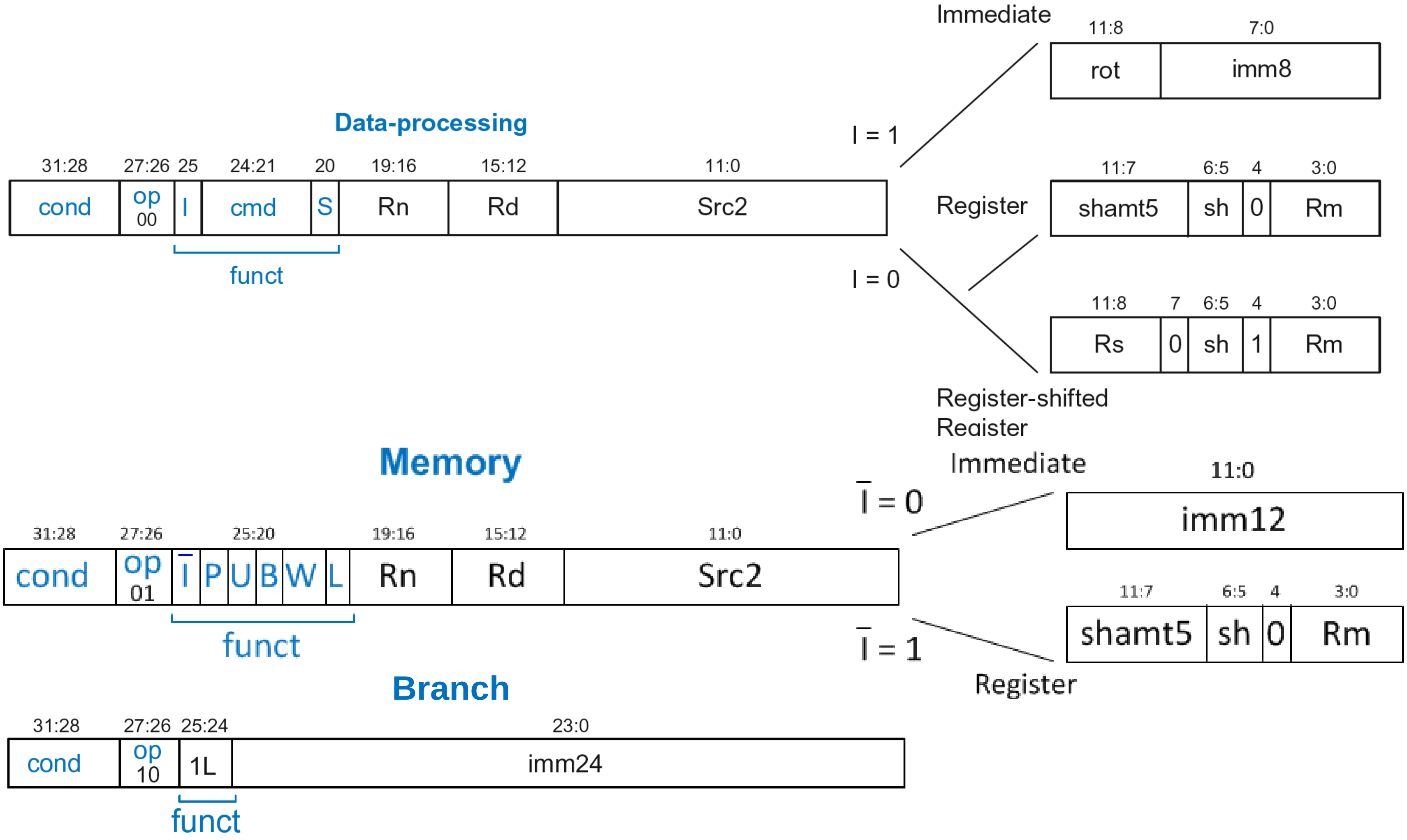
- PC = 0x8050
- PC + 8 = 0x8058
- `TEST` label is 6 instructions before PC+8
- So, *imm24* = -6

Field Values

31:28	27:26	25:24	23:0
1110_2	10_2	11_2	-6
cond	op	funct	imm24

1110	10	11	1111 1111 1111 1111 1111 1010	**0xEBFFFFFA**

Review: Instruction Formats



Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2
		funct					

I = 1

Immediate

11:8	7:0
rot	imm8

I = 0

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

Register-shifted Register

Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2
		funct								

$\bar{I}$ = 0

Immediate

11:0
imm12

$\bar{I}$ = 1

11:7	6:5	4	3:0
shamt5	sh	0	Rm

Register

Branch

31:28	27:26	25:24	23:0
cond	op 10	1L	imm24
		funct	

Conditional Execution

Encode in *cond* bits of machine instruction

For example,

```
ANDEQ R1, R2, R3        (cond = 0000)
ORRMI R4, R5, #0xF      (cond = 0100)
SUBLT R9, R3, R8        (cond = 1011)
```

Review: Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z \ OR \ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z \ OR \ (N \oplus V)$
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

Interpreting Machine Code: Example 1

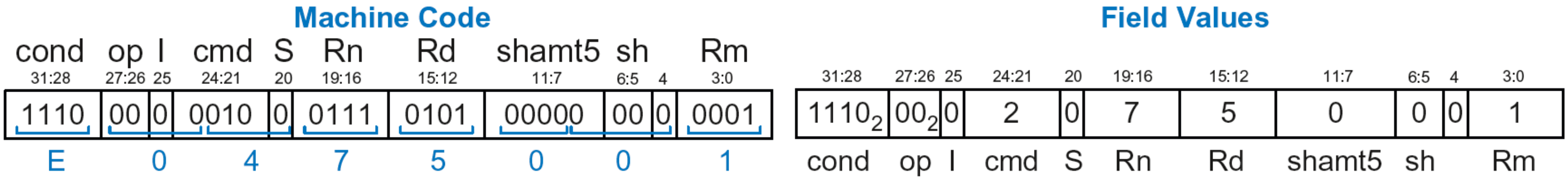
0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction

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*)



Machine Code

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0010	0	0111	0101	00000	00	0	0001
E	0		4		7	5	0	0		1

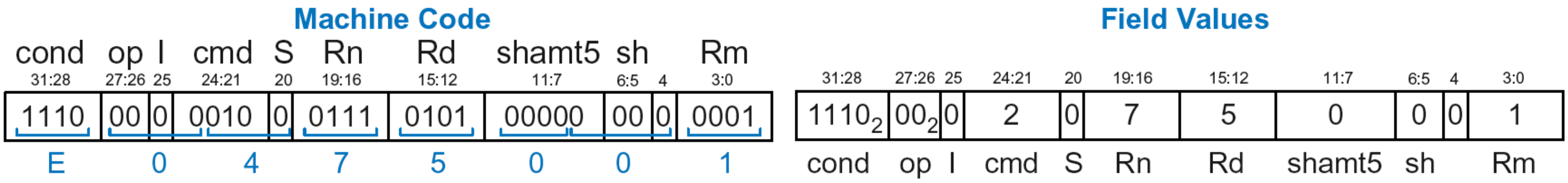
Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	2	0	7	5	0	0	0	1
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

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`**



Machine Code

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0010	0	0111	0101	00000	00	0	0001

E 0 4 7 5 0 0 1

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	2	0	7	5	0	0	0	1
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Interpreting Machine Code: Example 2

0xE5949010

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	$\overline{I}$PUBWL	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	$\overline{I}$PUBWL	Rn	Rd	imm12

Addressing Modes

How do we address operands?

- Register
- Immediate
- Base
- PC-Relative

Addressing Modes

How do we address operands?

- **Register Only**
- Immediate
- Base
- PC-Relative

Register Addressing

- Source and destination operands found in registers
- Used by data-processing instructions
- **Three submodes:**
  - Register-only
  - Immediate-shifted register
  - Register-shifted register

Register Addressing Examples

- **Register-only**

  Example: `ADD R0, R2, R7`

- **Immediate-shifted register**

  Example: `ORR R5, R1, R3, LSL #1`

- **Register-shifted register**

  Example: `SUB R12, R9, R0, ASR R1`

Addressing Modes

How do we address operands?

- Register Only
- **Immediate**
- Base
- PC-Relative

Immediate Addressing

- Source and destination operands found in registers **and** immediates

  Example: `ADD R9, R1, #14`

- Uses data-processing format with *I*=1
- Immediate is encoded as
  - 8-bit immediate (*imm8*)
  - 4-bit rotation (*rot*)
- 32-bit immediate = *imm8* ROR (*rot* x 2)

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- **Base**
- PC-Relative

Base Addressing

- Address of operand is:

  base register + offset

- Offset can be a:
  - 12-bit Immediate
  - Register
  - Immediate-shifted Register

Base Addressing Examples

- **Immediate offset**

  Example: `LDR R0, [R8, #-11]`

  (R0 = mem[R8 - 11])

- **Register offset**

  Example: `LDR R1, [R7, R9]`

  (R1 = mem[R7 + R9])

- **Immediate-shifted register offset**

  Example: `STR R5, [R3, R2, LSL #4]`

  (R5 = mem[R3 + (R2 << 4)])

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- Base
- **PC-Relative**

PC-Relative Addressing

- Used for branches
- Branch instruction format:
  - Operands are PC and a signed 24-bit immediate (*imm24*)
  - Changes the PC
  - New PC is relative to the old PC
  - *imm24* indicates the number of words away from PC+8
- PC = (PC+8) + (SignExtended(*imm24*) x 4)

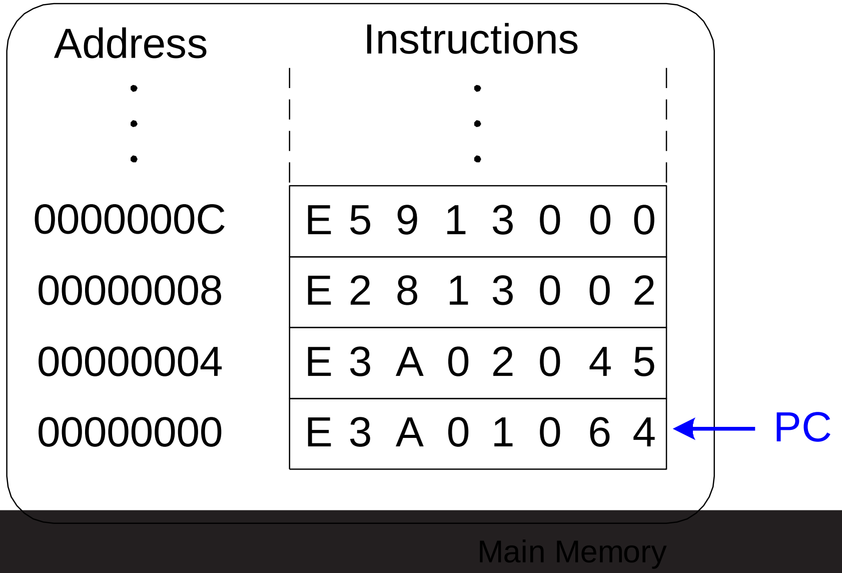
Power of the Stored Program

- **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



Address	Instructions
⋮	⋮
0000000C	E 5 9 1 3 0 0 0
00000008	E 2 8 1 3 0 0 2
00000004	E 3 A 0 2 0 4 5
00000000	E 3 A 0 1 0 6 4

Program Counter (PC): keeps track of current instruction

Up Next

How to implement the ARM Instruction Set Architecture in Hardware

Microarchitecture