ISA: Ins. format

Computer Architecture

CS3501 – 2025I
PROF.: JGONZALEZ@UTEC.EDU.PE
SRC: HARRIS, HARRIS - DIGITAL DESIGN AND COMPUTER ARCHITECTURE
STALLINGS - COMPUTER ARCHITECTURE

Executive Summary

- Motivation: We need to move to a higher abstraction level between the software and detailed hardware.
- Problem: We need to understand the processor instructions to be able to give commands.
- Overview:
  - Instruction cycle
  - ARM instruction types
  - ARM instructions formats.
- Conclusion: We can command the processor using machine language code.

Outline

Recall: Instruction Execution

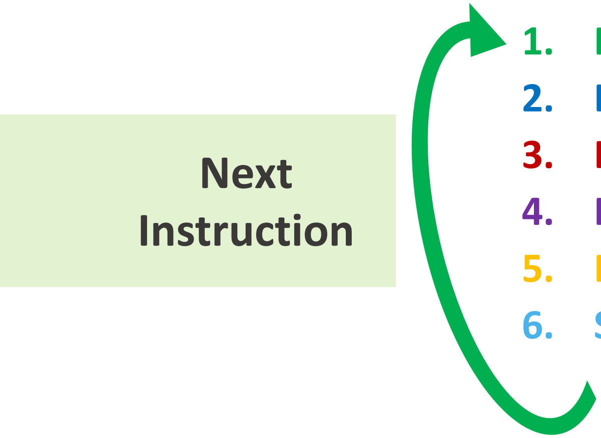
- By using instructions we can speak the language of the computer
- Thus, we now know how to tell the computer to
  - Execute computations in the ALU by using, for instance, an addition
  - Access operands from memory by using the load word instruction

How are the instructions executed on the computer?

The process of executing an instruction is called is **the instruction cycle or instruction phases.**

Instruction Cycle

- The instruction cycle is a sequence of steps or phases, that an instruction goes through to be executed



1. **FETCH:** Take the instruction
2. **DECODE:** What is the instruction about?
3. **EVALUATE ADDRESS:** Does it need something (data) from mem.?
4. **FETCH OPERANDS:** Take all the operands
5. **EXECUTE:** Perform the instruction real job.
6. **STORE RESULT:** Save the result (if it is needed …)

- **E.g.:** Intel x86 instruction ADD [eax], edx has six phases
- **Not all instructions have the six phases**. E.g:
  - LDR does not require EXECUTE
  - ADD does not require EVALUATE ADDRESS

Outline

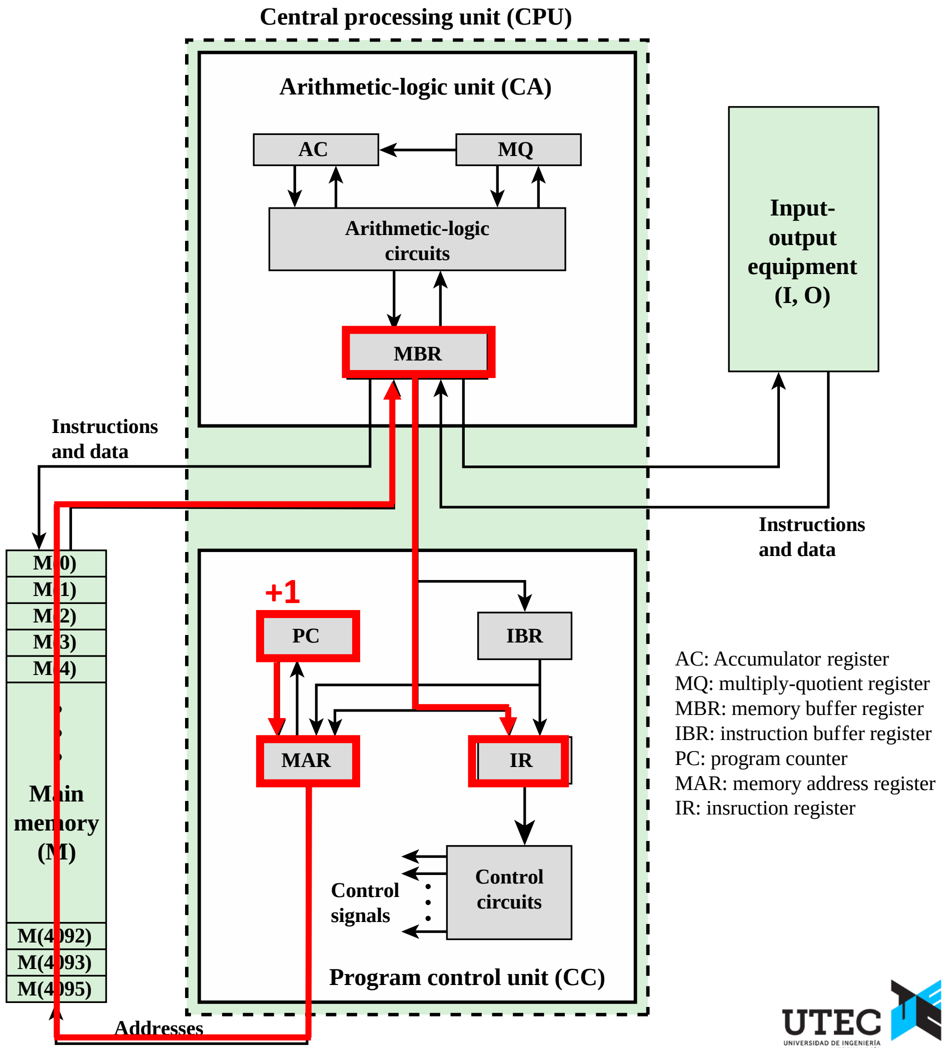
1. Fetch

- The FETCH phase **obtains the instruction from memory** and loads it into the **instruction register.**

Step 1: Load MAR and increment PC

Step 2: Access memory

Step 3: Load IR with the content of MBR

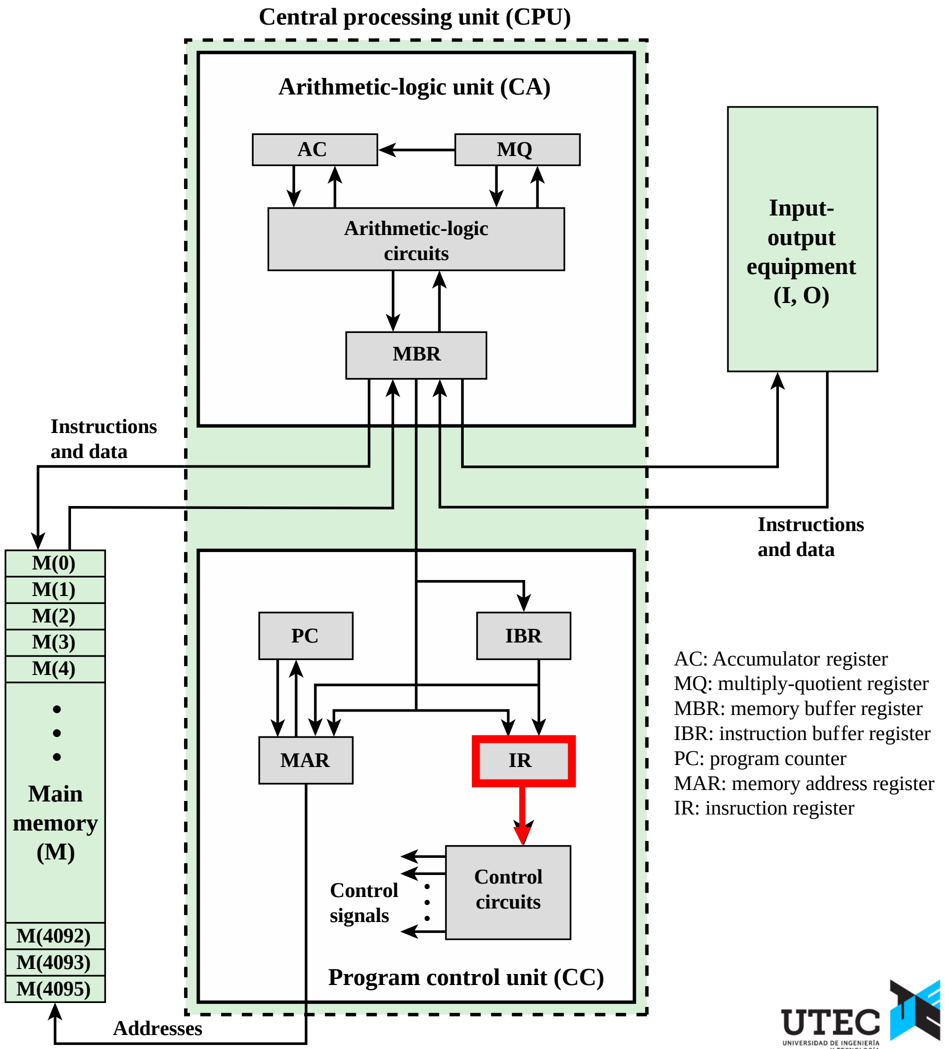


AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

2. Decode

- The DECODE phase **identifies the instruction.**
- A **4-to-16 decoder** identifies which of the **16 opcodes** is going to be processed.
  - A **decoder**, the input is the four bits IR[24:21]

DECODE identifies the instruction to be processed

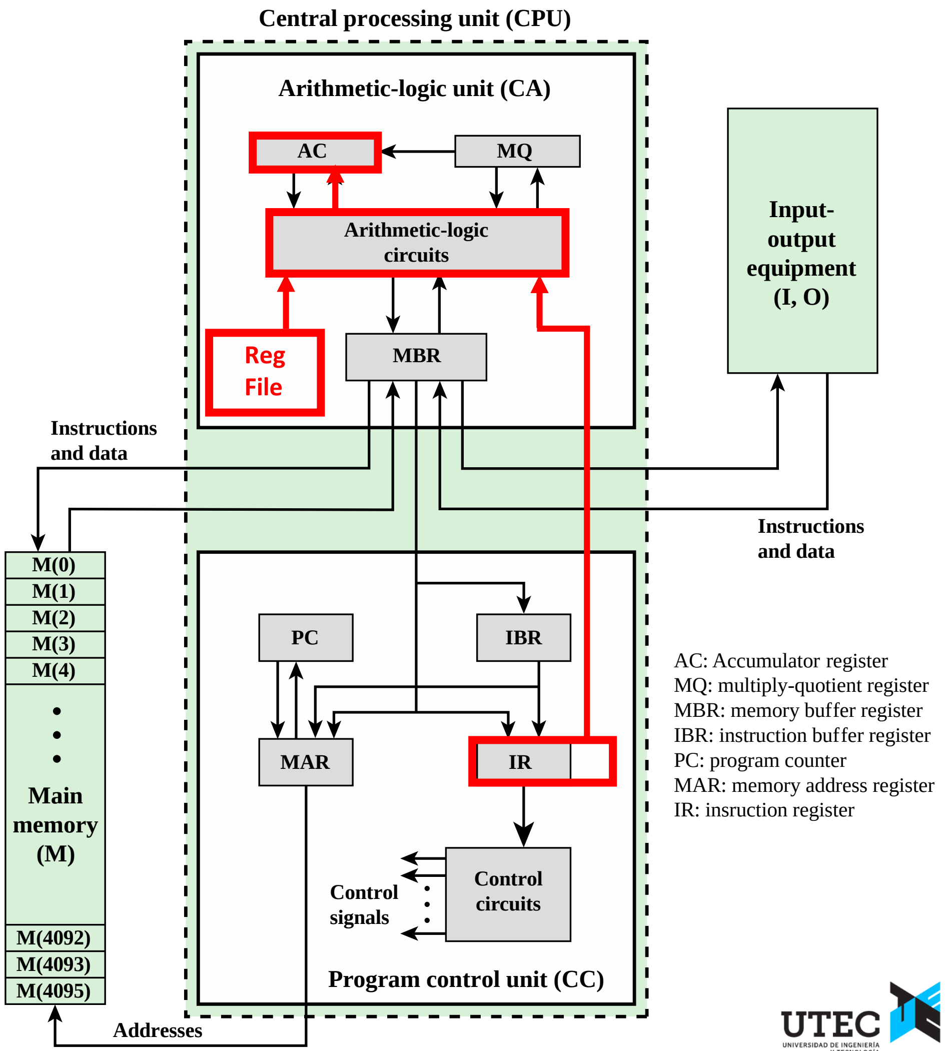


AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

3. Evaluate Address

- The EVALUATE ADDRESS phase **computes the address of the memory location** that is needed to process the instruction
- **Examples:**
  1. This phase **is necessary in LDR**
     - It computes the **address of the data word** that is to be read from memory **by adding an offset** to the content of a register
  2. But **not necessary in ADD**

LDR calculates the address by adding a register and an immediate



AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

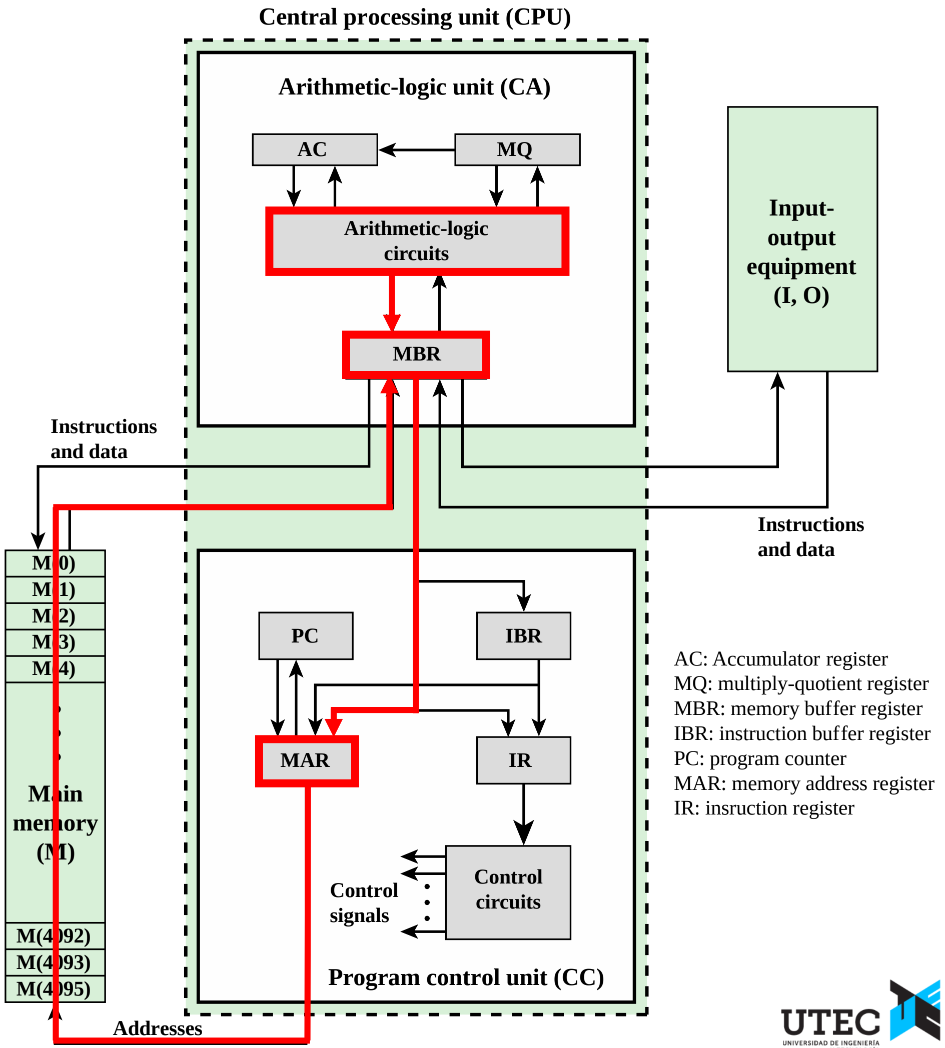
4. Fetch Operands

- The FETCH OPERANDS phase **obtains the source operands** needed to process the instruction

- **Example:** In LDR
  - Step 1: **Load MAR** with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing **source operand in MBR**

- **Example:** In ADD
  - Obtain the source operands **from the register file**
  - In most current microprocessors, this phase can be done **at the same time the instruction is being decoded**

LDR loads MAR (step 1), and places the results in MBR (step 2)



AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

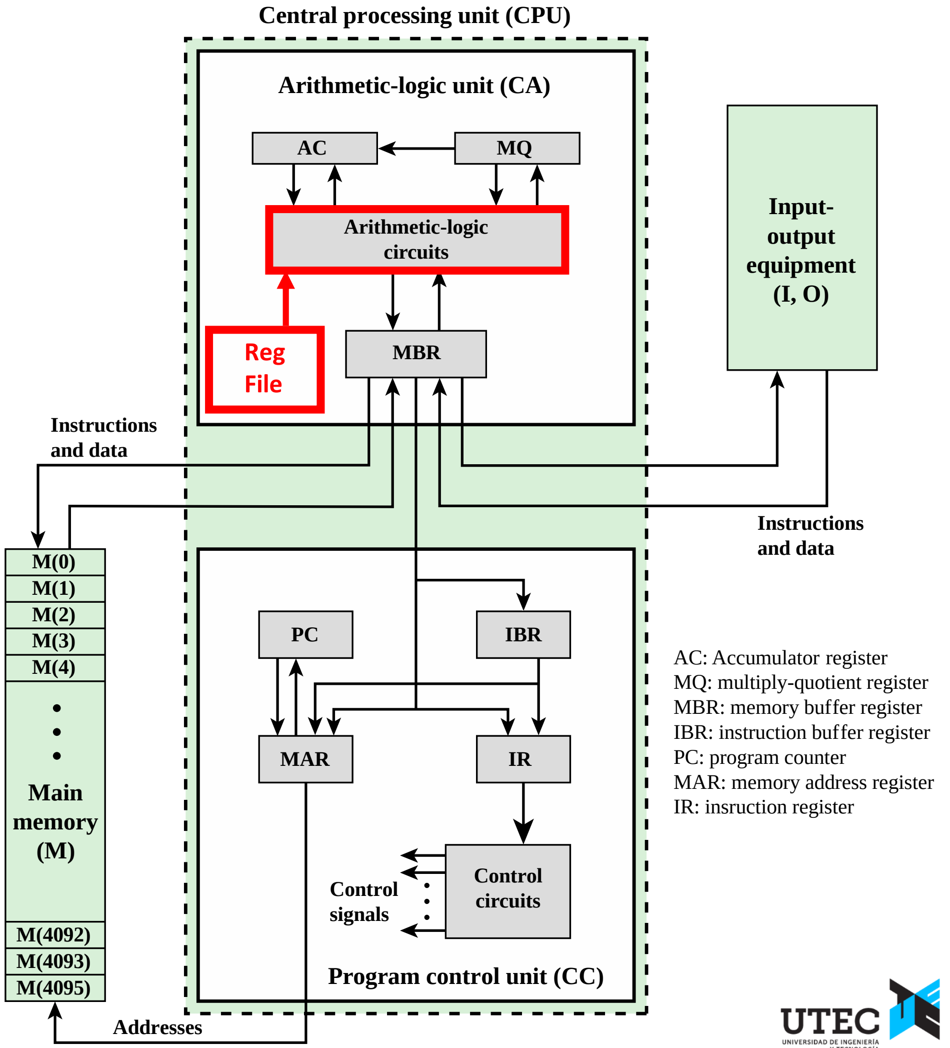
4. Fetch Operands

- The FETCH OPERANDS phase **obtains the source operands** needed to process the instruction

- **Example:** In LDR
  - Step 1: **Load MAR** with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing **source operand in MBR**

- **Example:** In ADD
  - Obtain the source operands **from the register file**
  - In most current microprocessors, this phase can be done **at the same time the instruction is being decoded**

ADD operands from Register File.

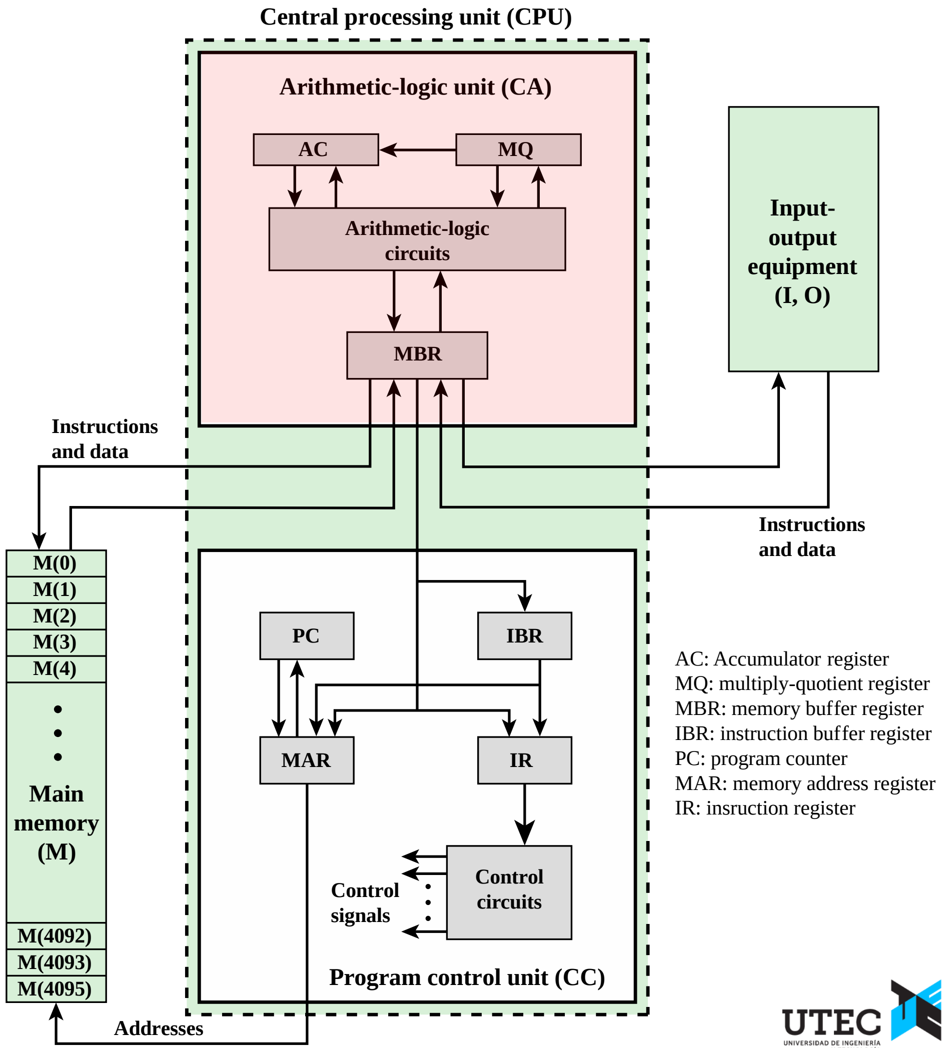


AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

5. Execute

- The EXECUTE phase **executes the instruction**
- **Example:** In ADD, it **performs addition in the ALU**

ADD adds SR1 and SR2

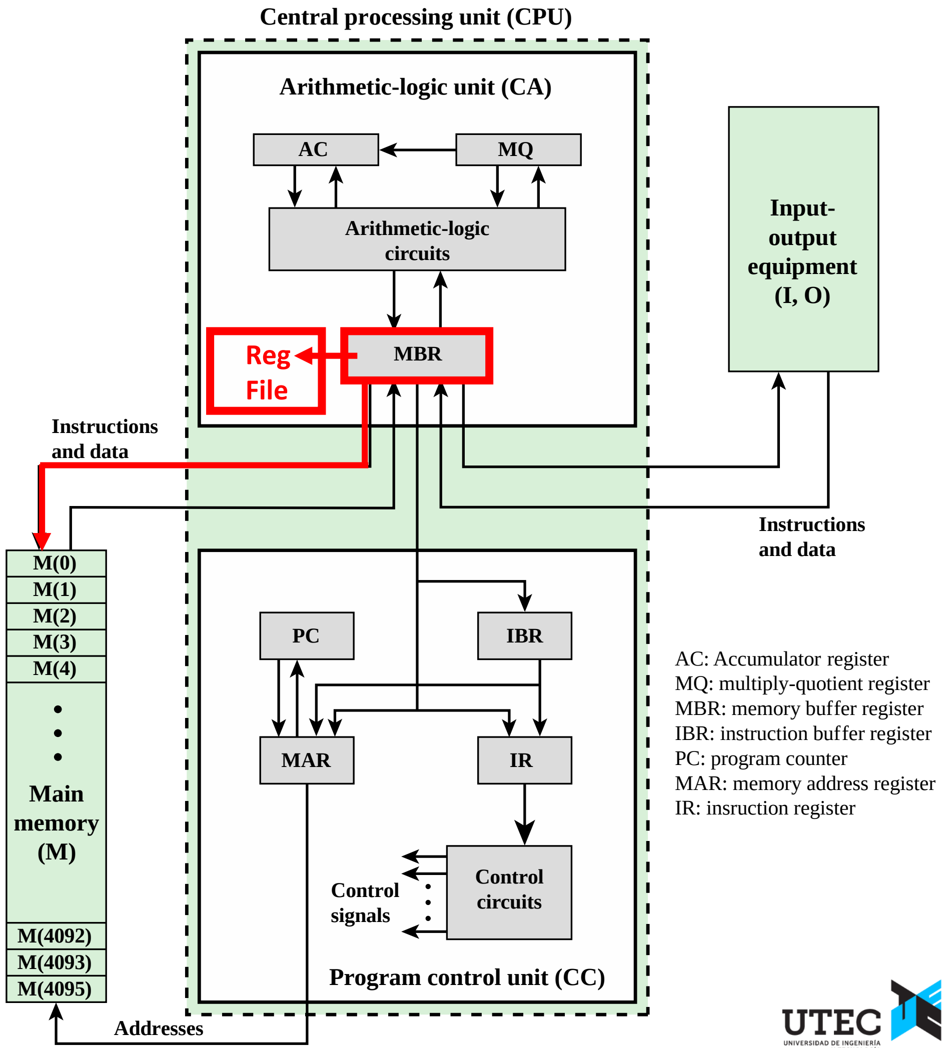


AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

6. Store Results

- The STORE RESULT phase **writes to the designated destination**
- Once STORE RESULT is completed, **a new instruction cycle** starts (with the FETCH phase)

Instruction writes to RegFile or Memory



AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

Outline

ARM Types of Instructions

Three main types of instructions:

1. **Data-processing Instructions**
   - A. Logical operations
   - B. Shifts / rotate
   - C. Arithmetic
   - **1.1 with conditional Execution**
2. **Branches**
3. **Memory**

Outline

ARM Instructions Formats

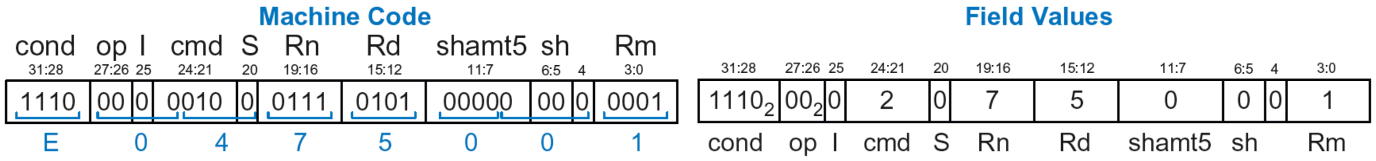
- Computers only understand **1's and 0's**
- **Binary (32-bit) representation of instructions.**
- **Three main formats:**
  1. Data-processing
  2. Memory
  3. Branch

See extra slides for Data-processing and Branch formats.

How? Interpreting Machine Code

0xE0475001

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



Interpreting Machine Code

- **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***
- ***Obs:***
  - **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:**

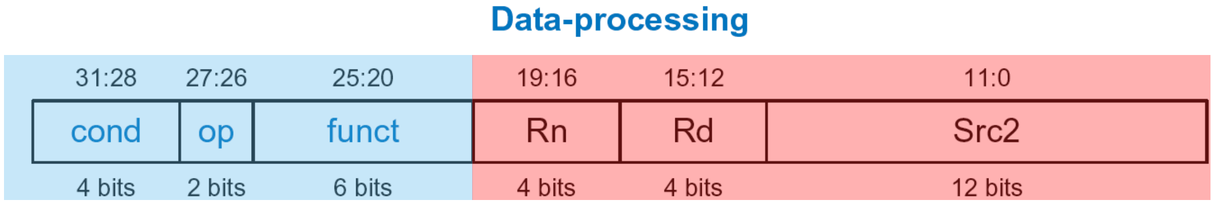
    Examine ***funct*** bits for indexing mode, instruction, and add or subtract offset

ARM Instructions Formats

- Computers only understand **1's and 0's**
- **Binary (32-bit) representation of instructions.**
- **Three main formats:**
  1. Data-processing
  2. Memory
  3. Branch

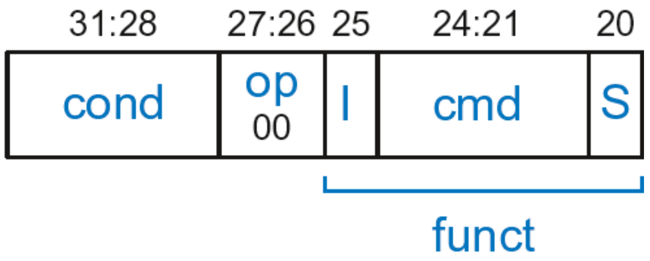
Data-processing Instruction Format

- **Operands:**
  - ***Rn*:** first source register
  - ***Src2*:** second source can be: a) Immediate, b) Register or c)Register-shifted register
  - ***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

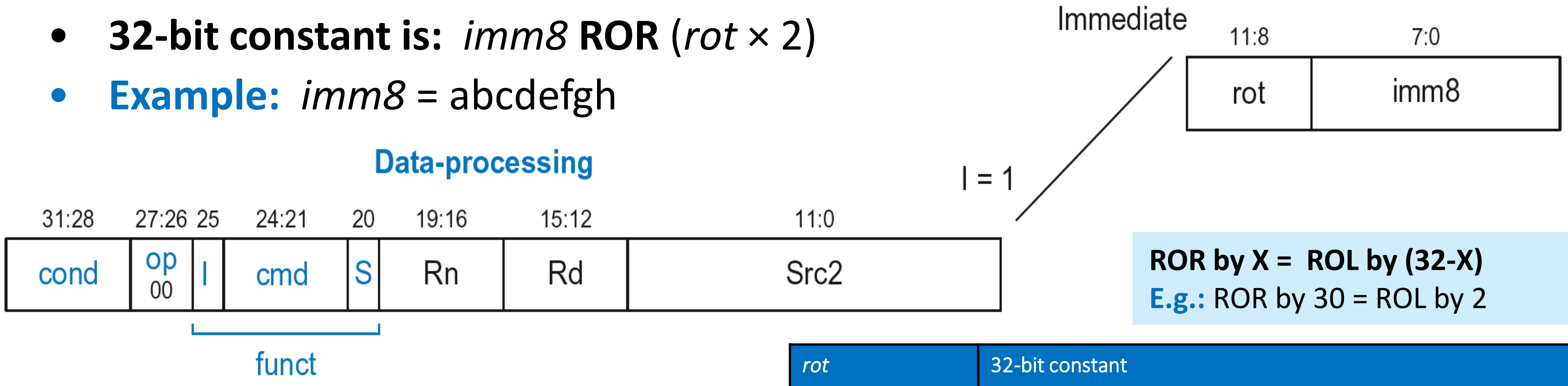
- ***op*** **= 00_2** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit
  - ***cmd:*** specifies the specific data-processing instruction. For example,
    - ***cmd*** = 0100_2 for `ADD`
    - ***cmd*** = 0010_2 for `SUB`
  - ***I***-bit
    - ***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: Immediate

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

IMPORTANT! ARM has a unique encoding please check: https://minhhua.com/arm_immed_encoding/index.html



ROR by X = ROL by (32-X)
E.g.: ROR by 30 = ROL by 2

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

Example: ADD 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

Example: SUB with Immediate *Src2*

IMPORTANT! ARM has a unique encoding please check: https://minhhua.com/arm_immed_encoding/index.html

```
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

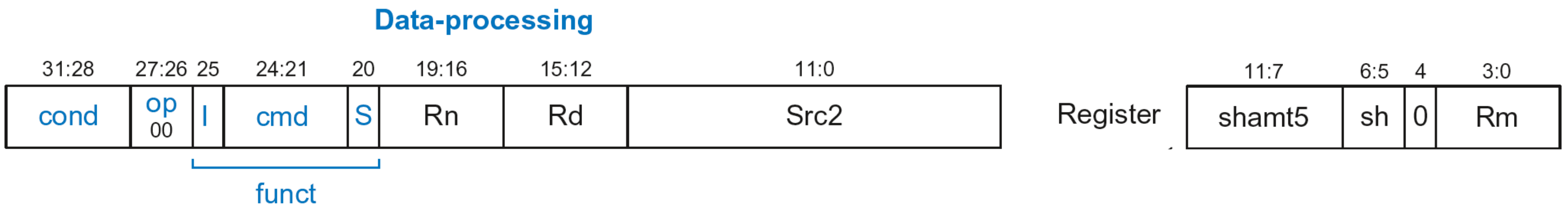
31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

0xE2432EFF

Data-processing *Src2* Variations: Register

- ***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)

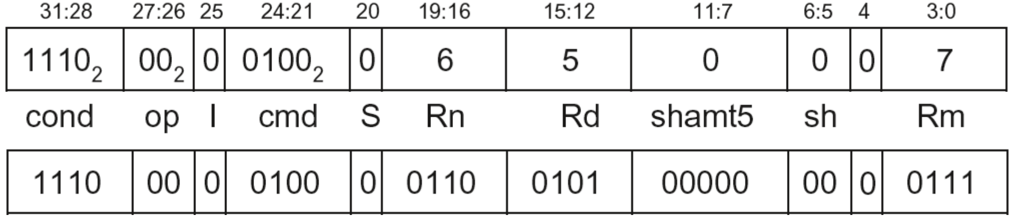


Example: ADD with Register *Src2*

```
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



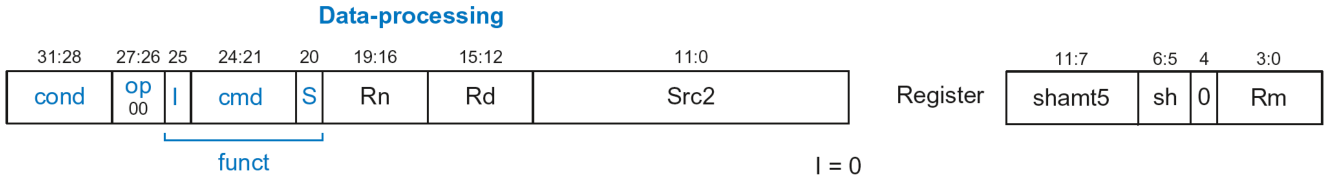
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	0	0100	0	0110	0101	00000	00	0	0111

0xE0865007

Example: ORR 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)



1110 00 0 1100 0 0101 1001 00010 01 0 0011

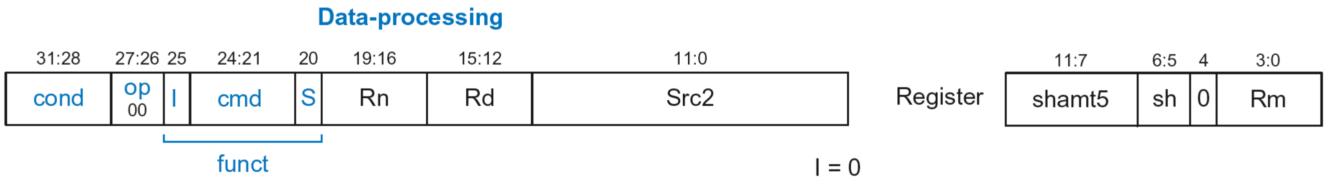
0xE1859123

Shifting 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.

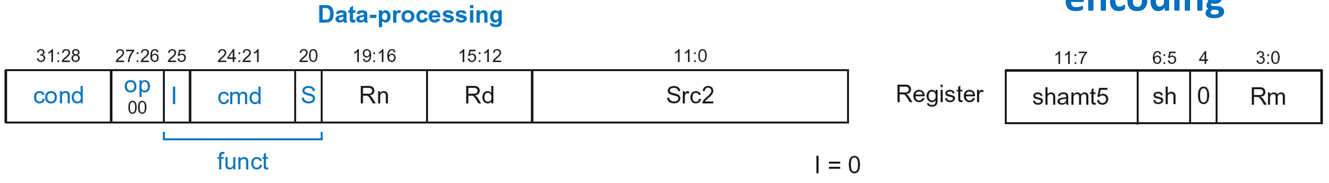


Example: Shift (ROR) Instructions Immediate shamt

```
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



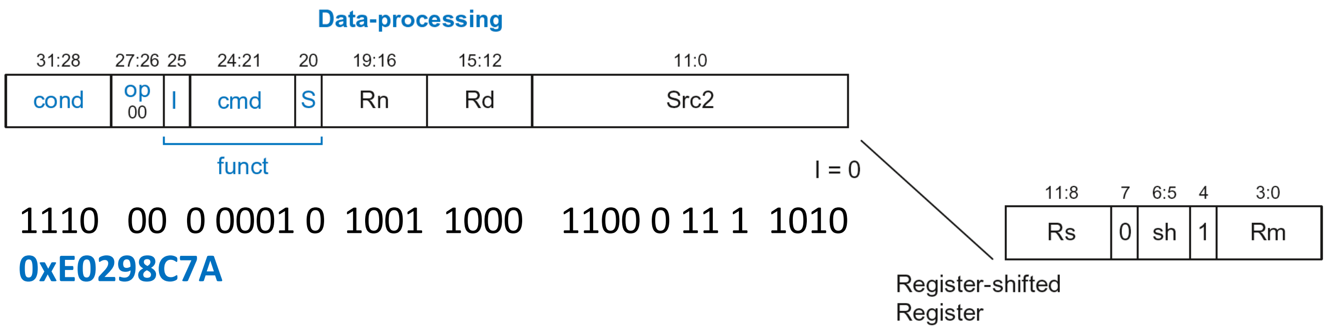
1110 00 0 1101 0 0000 0001 10111 11 0 0010

0xE1A01BE2

Example: EOR 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)



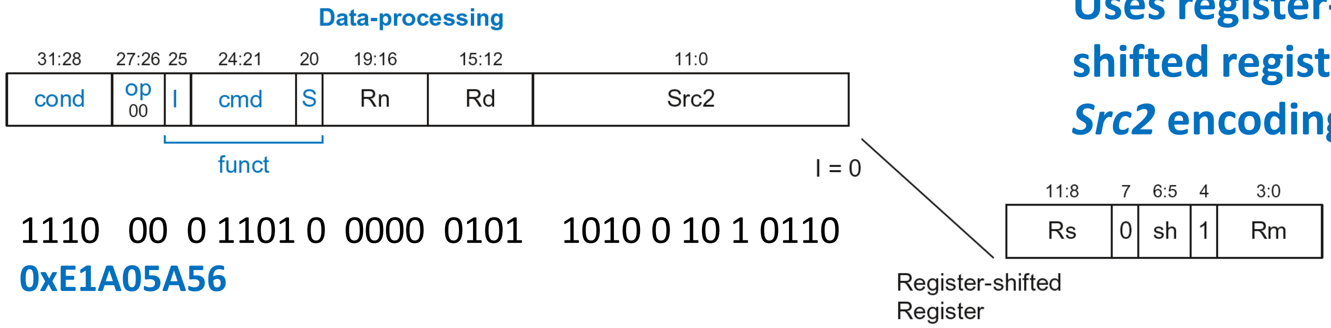
1110 00 0 0001 0 1001 1000 1100 0 11 1 1010

0xE0298C7A

Example: Shift Instructions with 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)



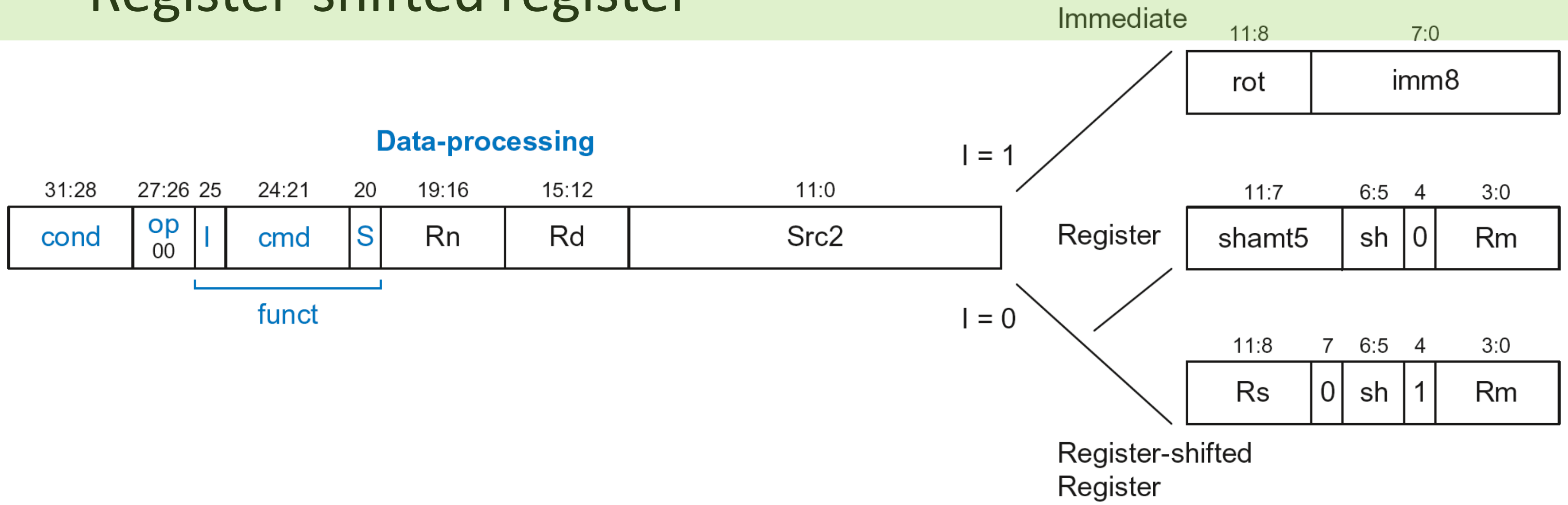
1110 00 0 1101 0 0000 0101 1010 0 10 1 0110

0xE1A05A56

Uses register-shifted register *Src2* encoding

Summary: Data-processing Format

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



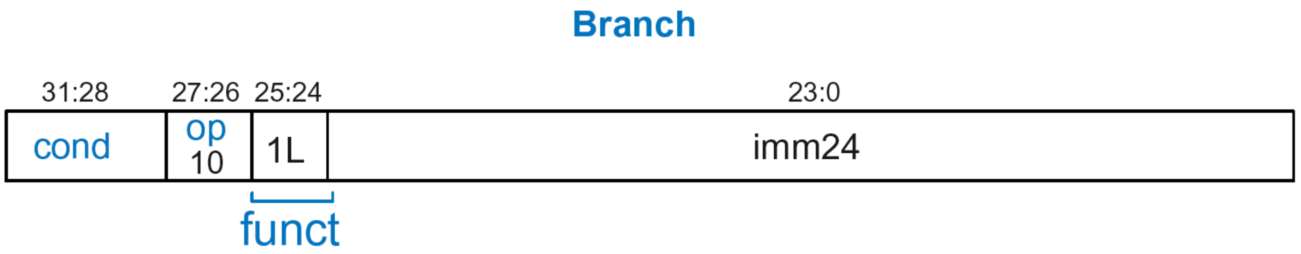
ARM Instructions Formats

- Computers only understand **1's and 0's**
- **Binary (32-bit) representation of instructions.**
- **Three main formats:**
  1. Data-processing
  2. Memory
  3. 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

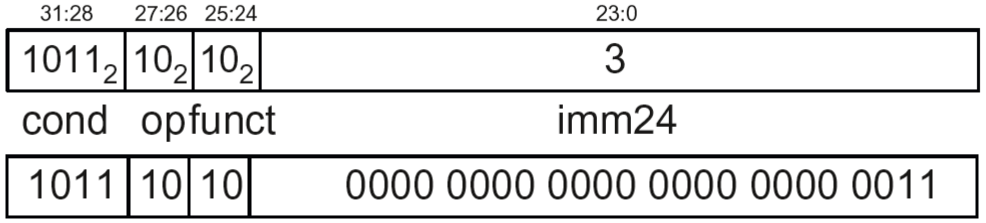
Example 1: Branch Instruction

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

Example 2: Branch Instruction

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
```

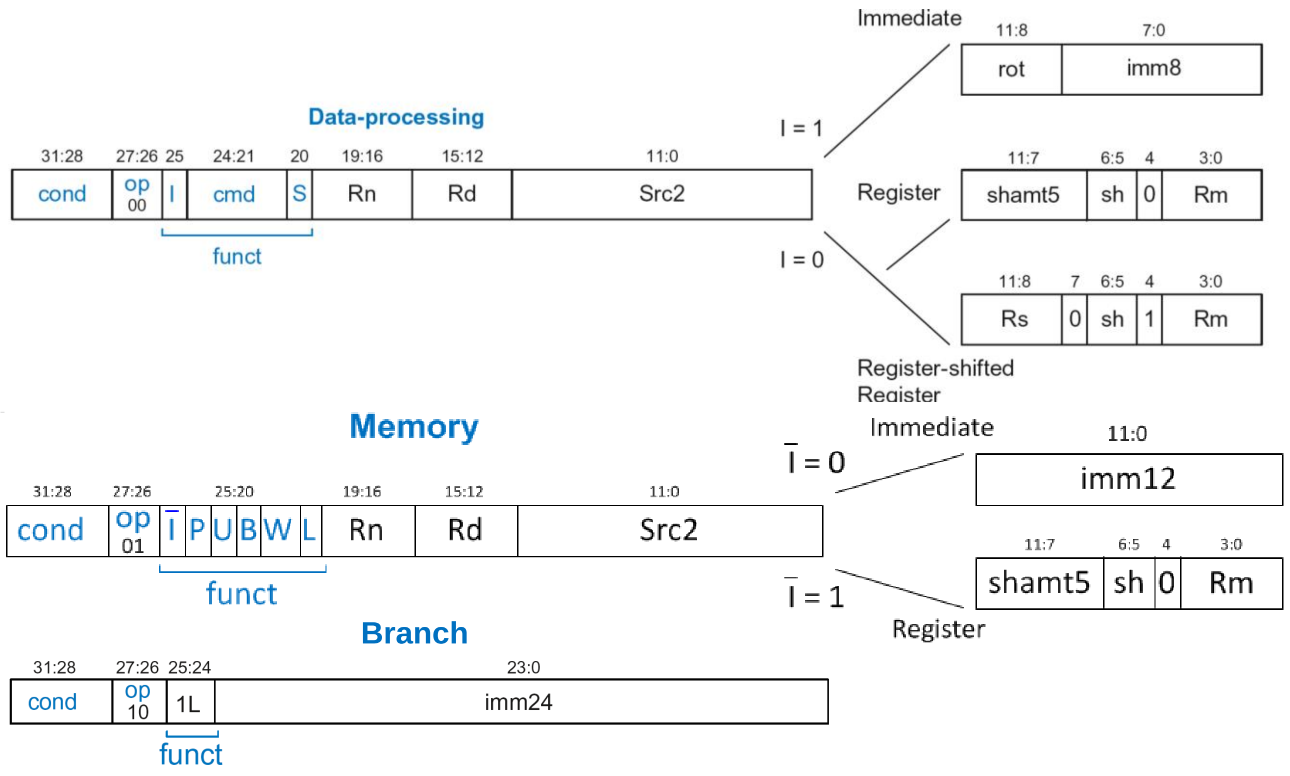
- PC = 0x8050
- PC + 8 = 0x8058
- `TEST` label is 6 instructions before PC+8
- So, *imm24* = -6

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

Summary: Instruction Formats



ARM Instructions Formats

- Computers only understand **1’s and 0’s**
- **Binary (32-bit) representation of instructions.**
- **Three main formats:**
    1. Data-processing
    2. Memory
    3. Branch

See extra slides for Data-processing and Branch formats.

Reading Memory

Important: Memory Instructions

- 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

Writing Memory

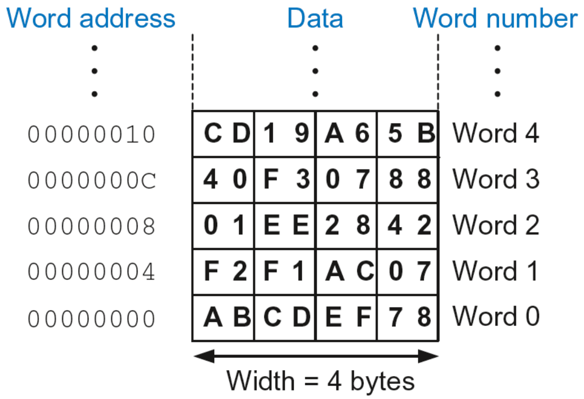
Important: Memory Instructions

- Memory write are called ***stores***
- **Mnemonic:** *store register* (`STR`)
- **Example:** Store the value held in R7 into memory word 21.
- Memory address = 4 x 21 = 84 = 0x54

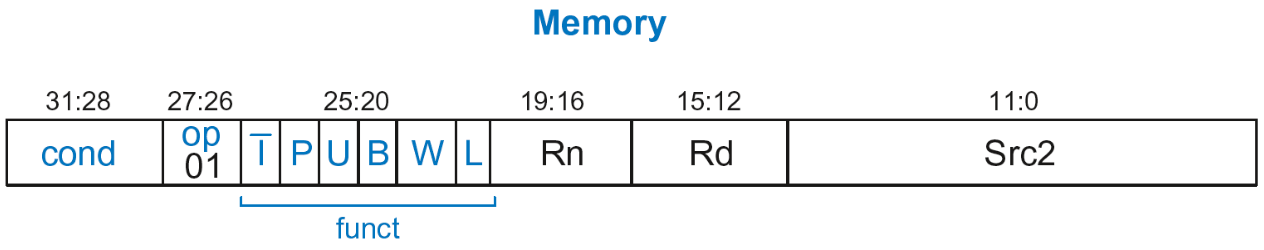
ARM assembly code

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

The offset can be written in decimal or hexadecimal



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

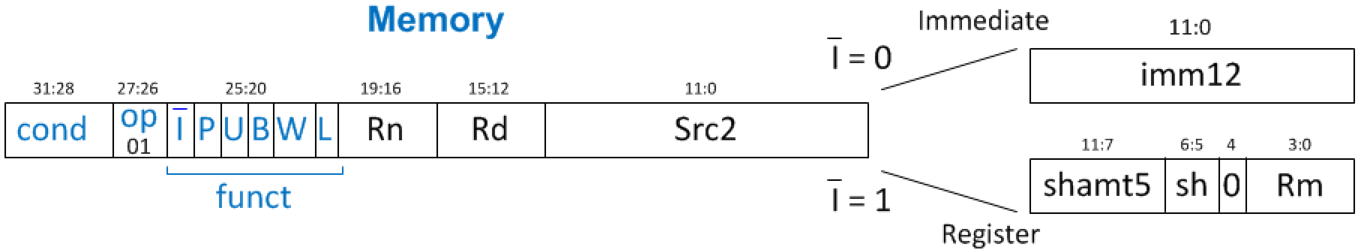
    Address = (R2 + 4)
- **Base address always in a register**
- **The offset can be:**
  - an immediate
  - a register
  - or a scaled (shifted) register

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



ARM 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
```

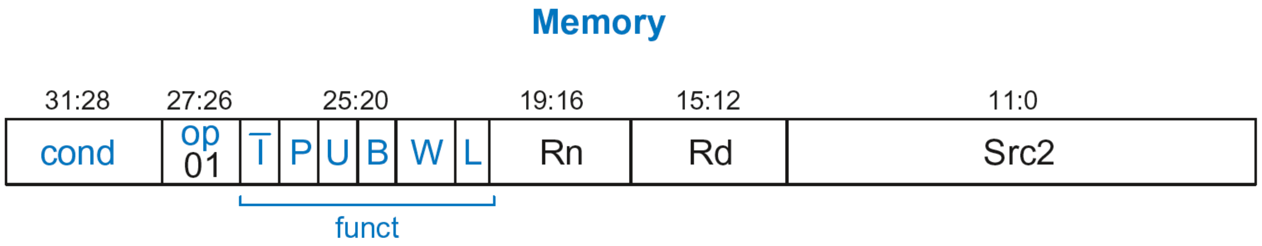
Memory Instruction Format

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

funct:

I:	Immediate bar
P:	Preindex
U:	Add
B:	Byte
W:	Writeback
L:	Load



Memory Format *funct* Encodings

Type of Operation

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

Indexing Mode

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

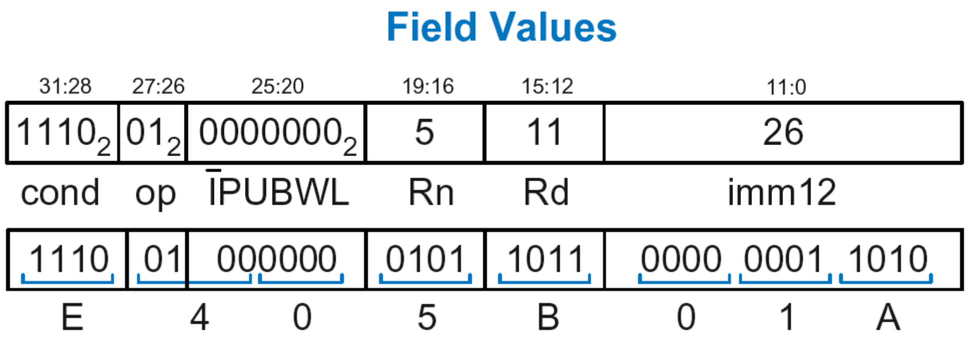
Add/Subtract Immediate/Register Offset

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

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
- *f~~u~~nct* = 0000000_2 (0)
  I = 0 (immediate offset), *P* = 0 (postindex),
  U = 0 (subtract), *B* = 0 (store word), *W* = 0 (postindex),
  L = 0 (store)
- *Rd* = 11, *Rn* = 5, *imm12* = 26

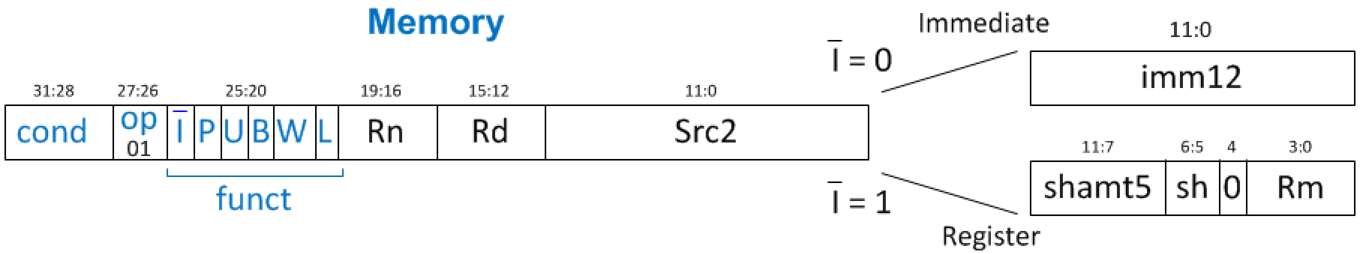


Memory Instr. with Register *Src2*

```
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)**
  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**

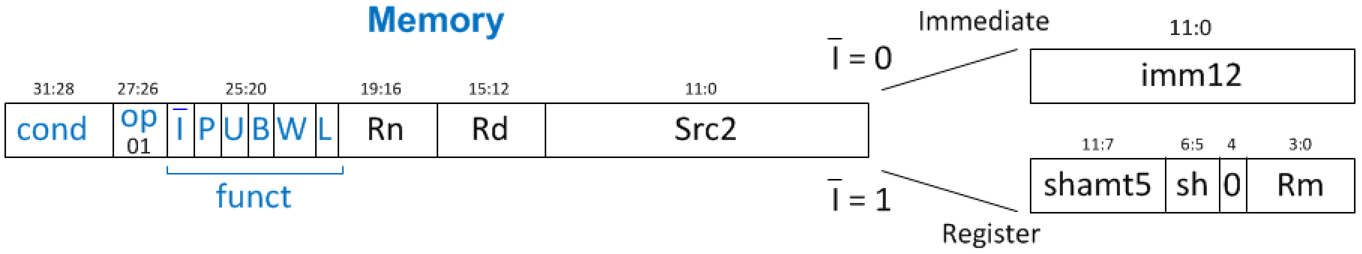


Memory Instr. with Scaled Reg. *Src2*

```
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)**
  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**



Addressing Modes

- An addressing mode **is a mechanism** for **specifying where an operand is located**

- There four addressing modes in ARM: Register, Immediate, Base, PC-Relative.
  - Two of them are **memory addressing modes**
    - PC-relative
    - Base+offset

Addressing Modes

- **Register:**
  - Source and destination operands found in registers
  - Used by data-processing instructions

- **Three submodes:**
  - 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`

- **Immediate:**
  - 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

- **Immediate:**
  - Address of operand is:
    base register + offset

  Three submodes:
  - **12-bit 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)])

- **PC-relative:**
  - 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)

Outline

Conclusions

- **We detailed the instruction cycle for program execution.**
- **We analyzed the instruction types and the formats for adequate operation.**
- We conclude that a **processor operates through a defined set of instructions that allows the programmer to control the processor and execute tasks.**

ISA: Ins. format

Computer Architecture

CS3501 – 2025I

PROF.: JGONZALEZ@UTEC.EDU.PE