Communications and Computer Engineering II:

Microprocessor 1: Instruction-Set Architecture

Lecturer : Tsuyoshi Isshiki

Dept. Communications and Computer Engineering,

Tokyo Institute of Technology

isshiki@ict.e.titech.ac.jp

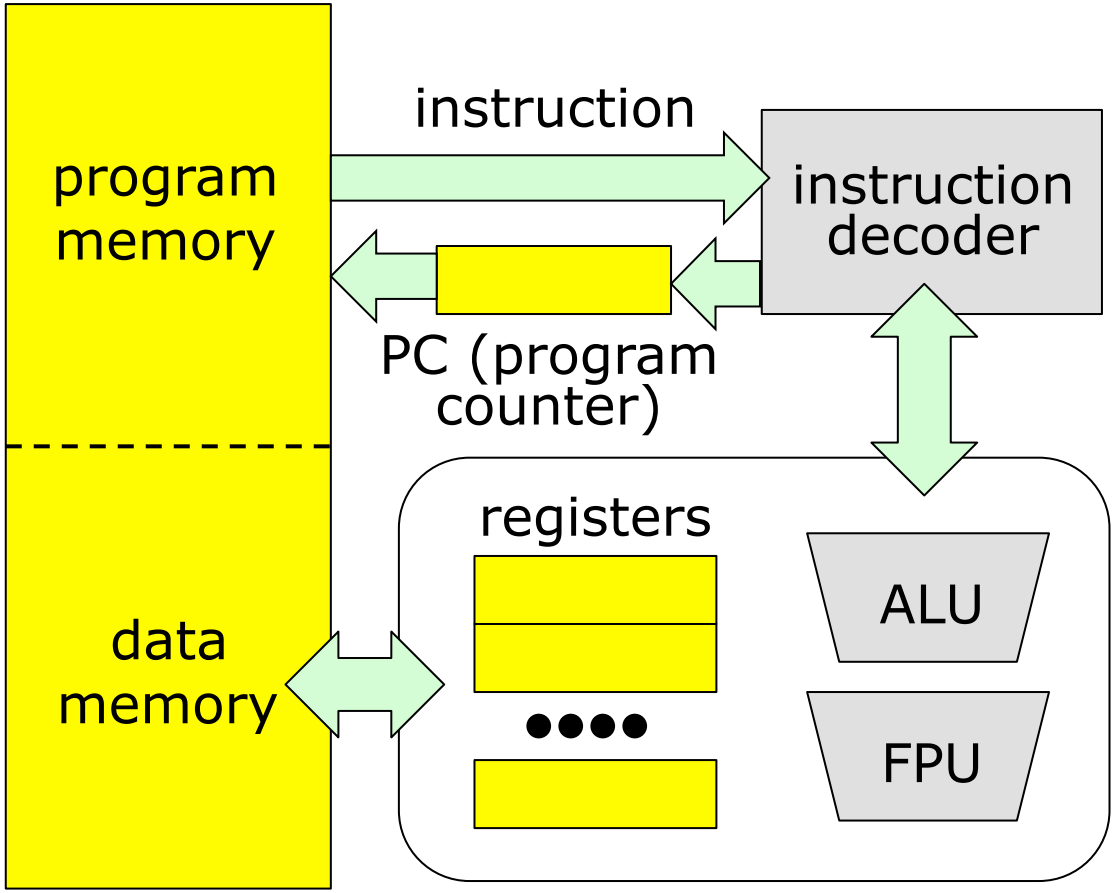
Lecture Outline

1. Microprocessor components : memory, registers, program counter, ALU (arithmetic logic unit), FPU (floating-point unit)
2. Instruction execution flow
3. Computer arithmetics using logic circuits
4. Instruction set classes : CISC vs. RISC
5. Instruction functionalities : data transfer (load, store), compute (add/sub/mult/etc), program control (branch, call/return)
6. Instruction formats : machine code (binary), assembler code, register transfer-level description
7. Instruction set examples: x86, MIPS, ARM

1. Microprocessor Components

- Memory : program / data storage
- Registers : temporal data storage
- Program counter (PC) : determines which instruction to execute
- Instruction decoder : determines what to do (data transfer, compute, program control)
- ALU (arithmetic logic unit) : add, sub, mult, div, and, or, xor, shifts
- FPU (floating-point unit) : fadd, fsub, fmult, fdiv

1. Microprocessor Components



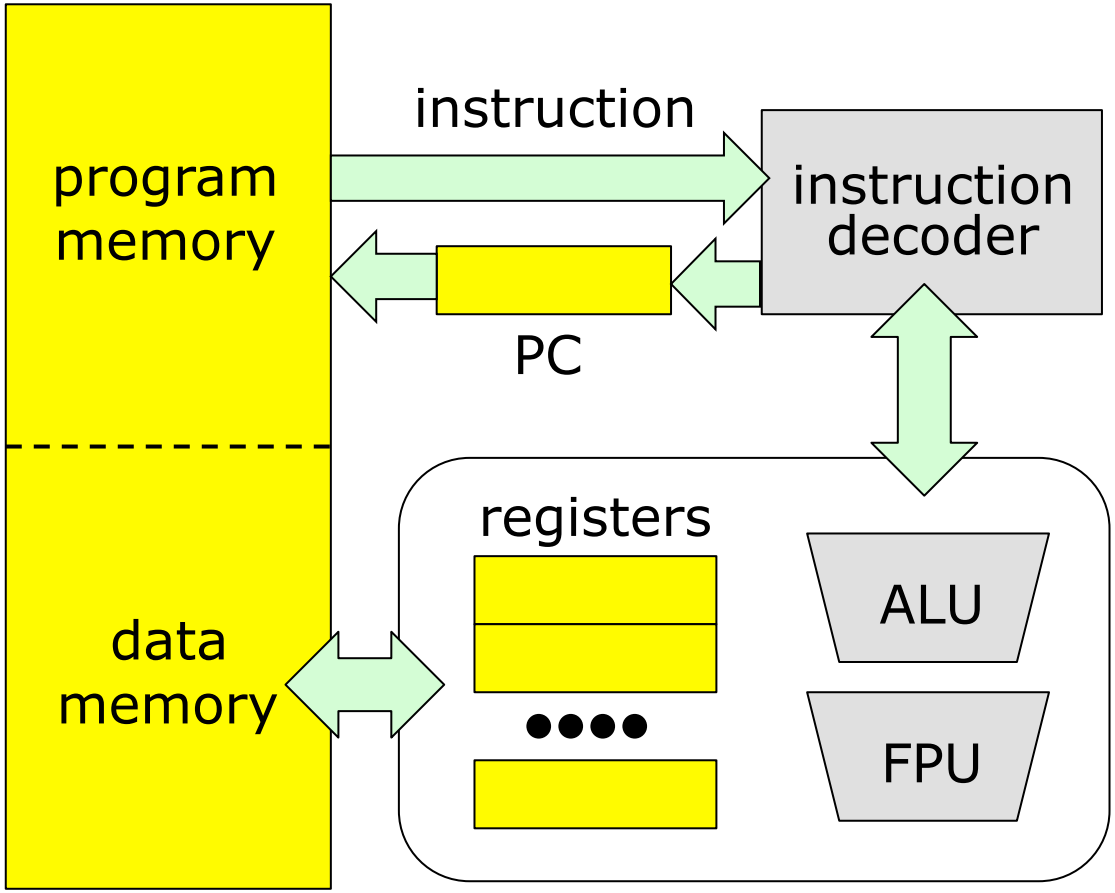
- Let's not worry about details here…

2. Instruction Execution Flow

1) Instruction fetch : read instruction from program memory pointed by PC
2) Instruction decode : determine what to do
3) Execute :
   a. Data transfer : load memory to register
   b. Compute : ALU/FPU operation
   c. Program control : compute next PC (jump, branch, call, return)
4) Store results : to register/data memory
5) Update PC

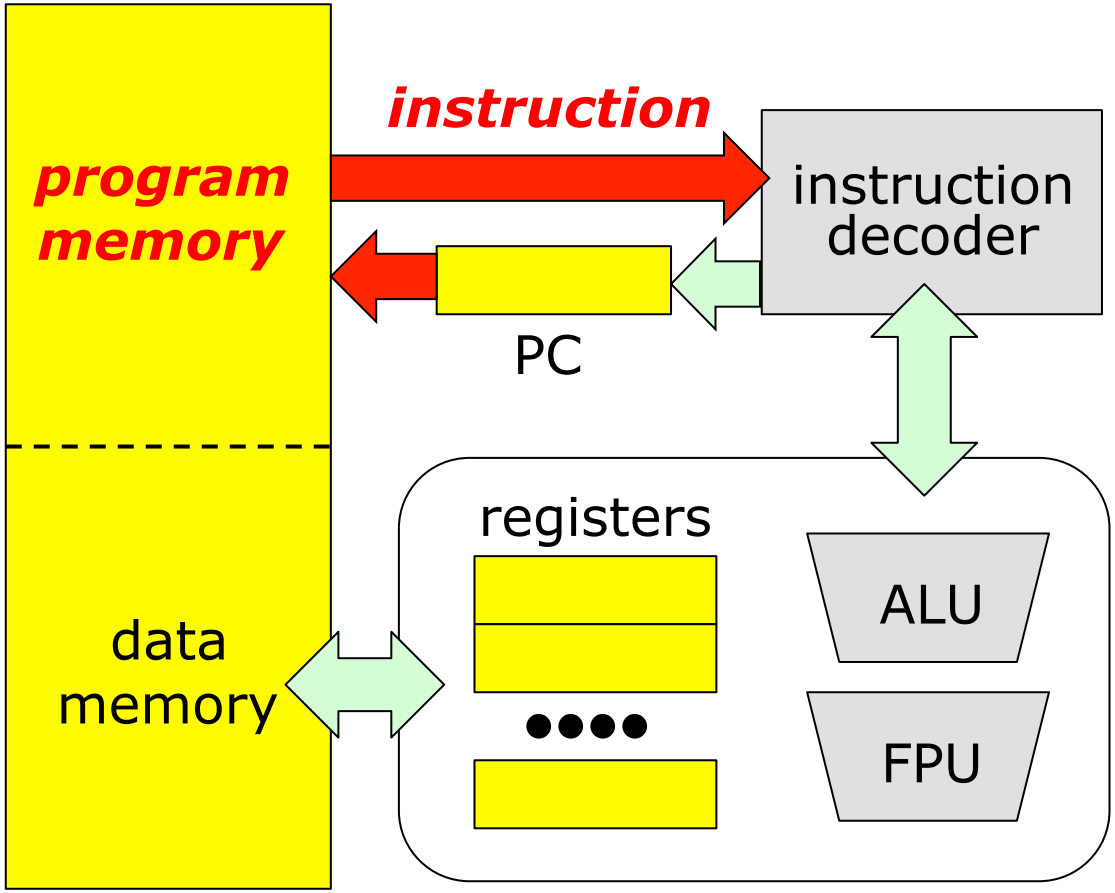
2. Instruction Execution Flow

1) Instruction fetch : read instruction from program memory pointed by PC



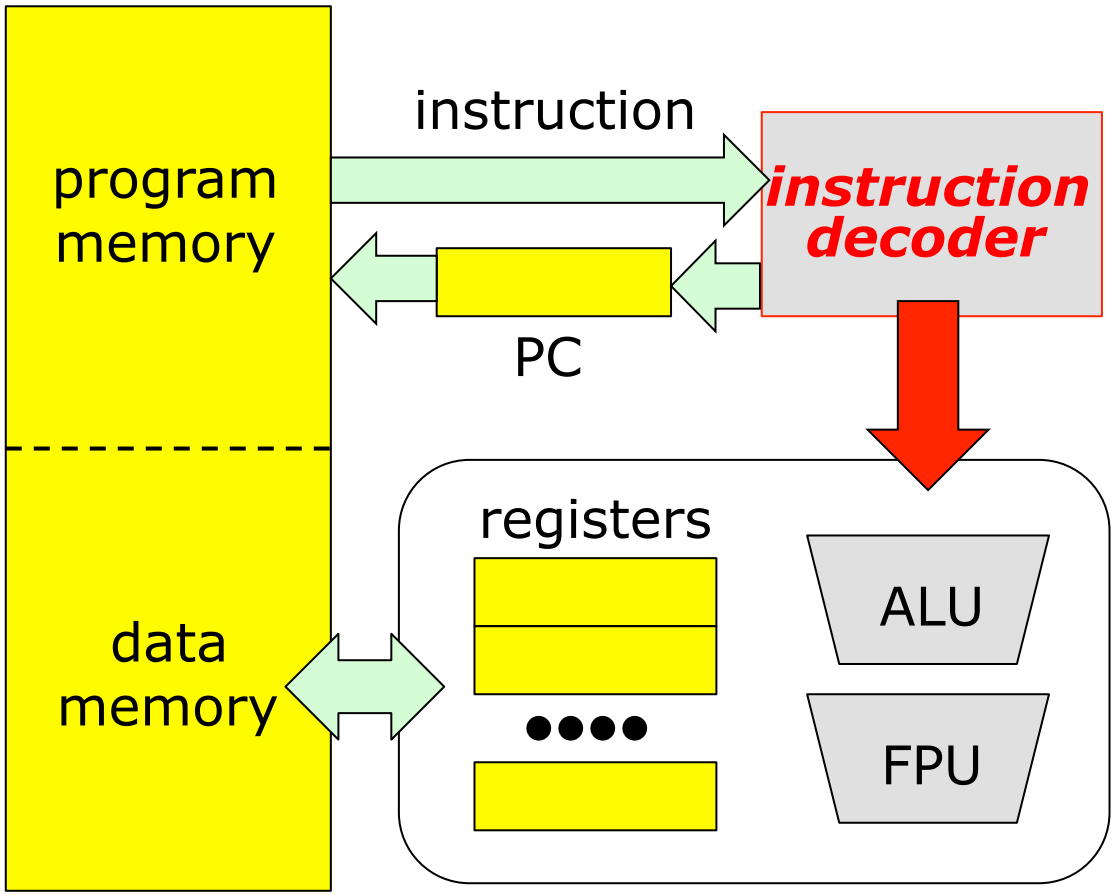
2. Instruction Execution Flow

1) Instruction fetch : read instruction from program memory pointed by PC



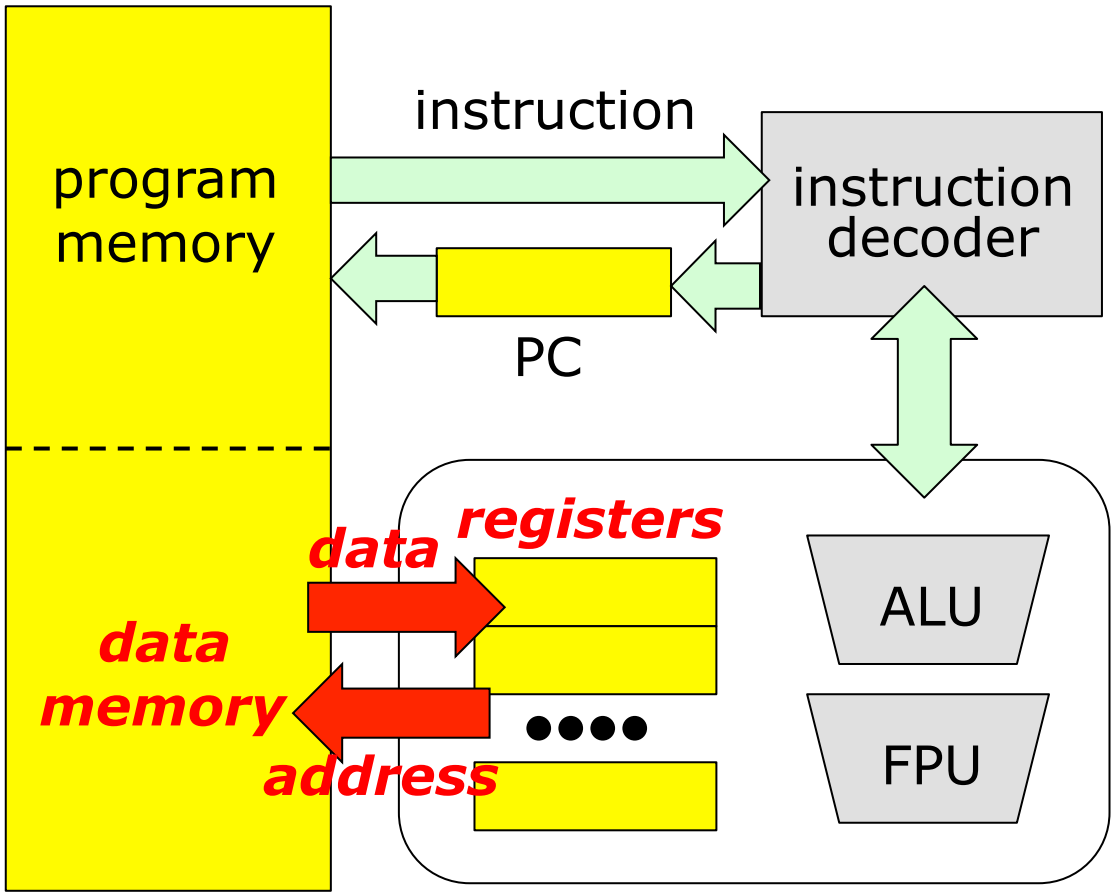
2. Instruction Execution Flow

2) Instruction decode : determine what to do



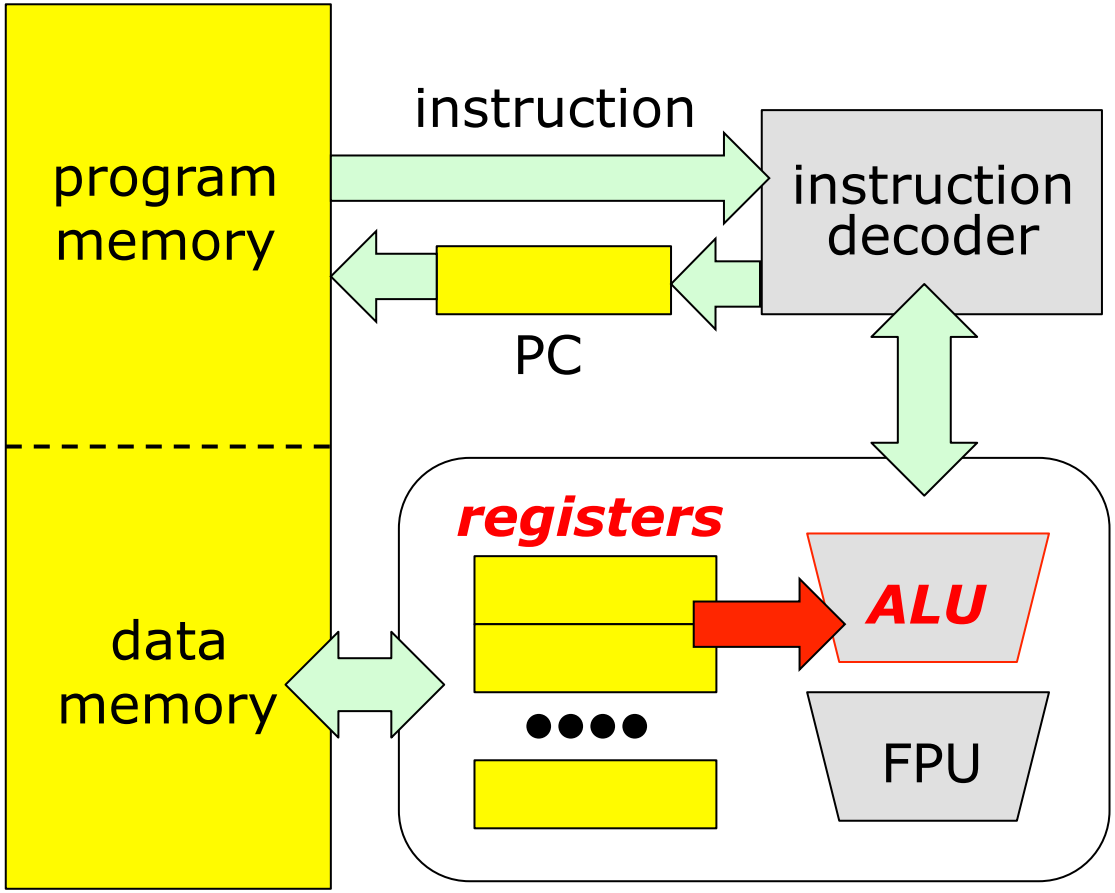
2. Instruction Execution Flow

3) Execute : (a) load memory to register



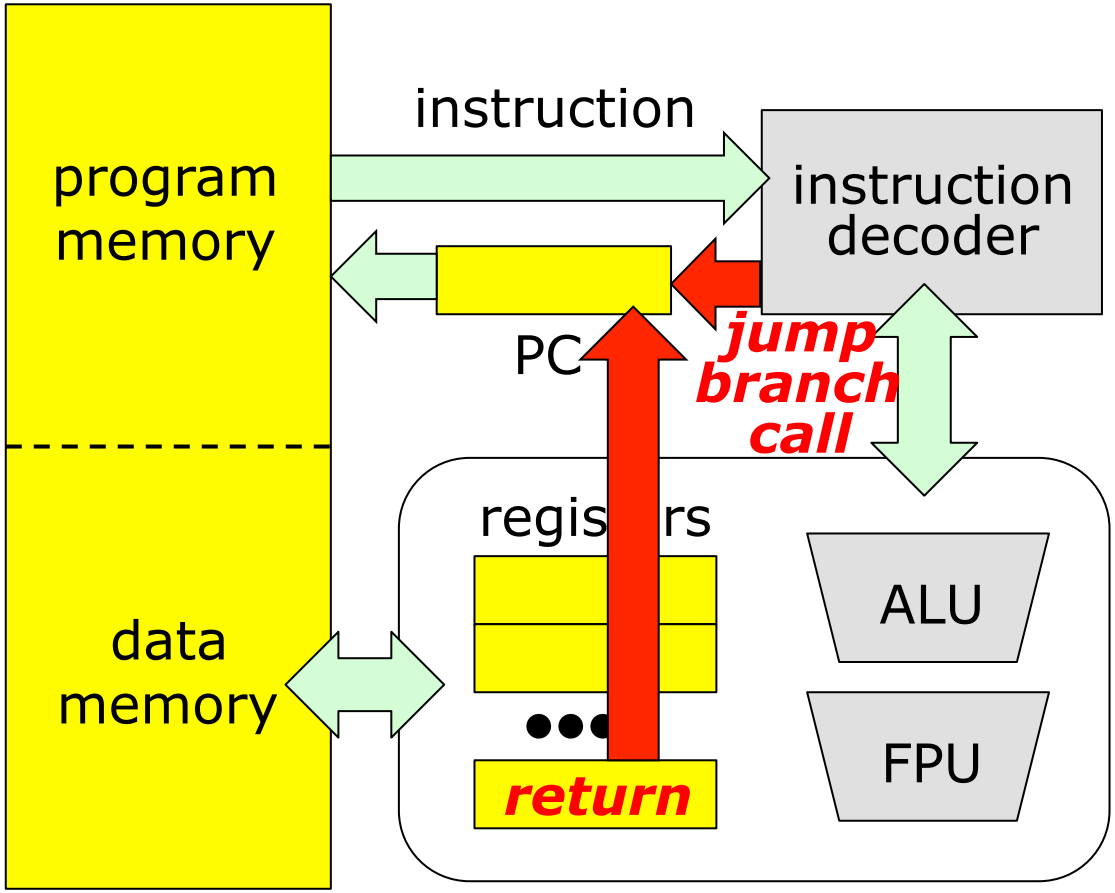
2. Instruction Execution Flow

3) Execute : (b) compute : ALU/FPU operation



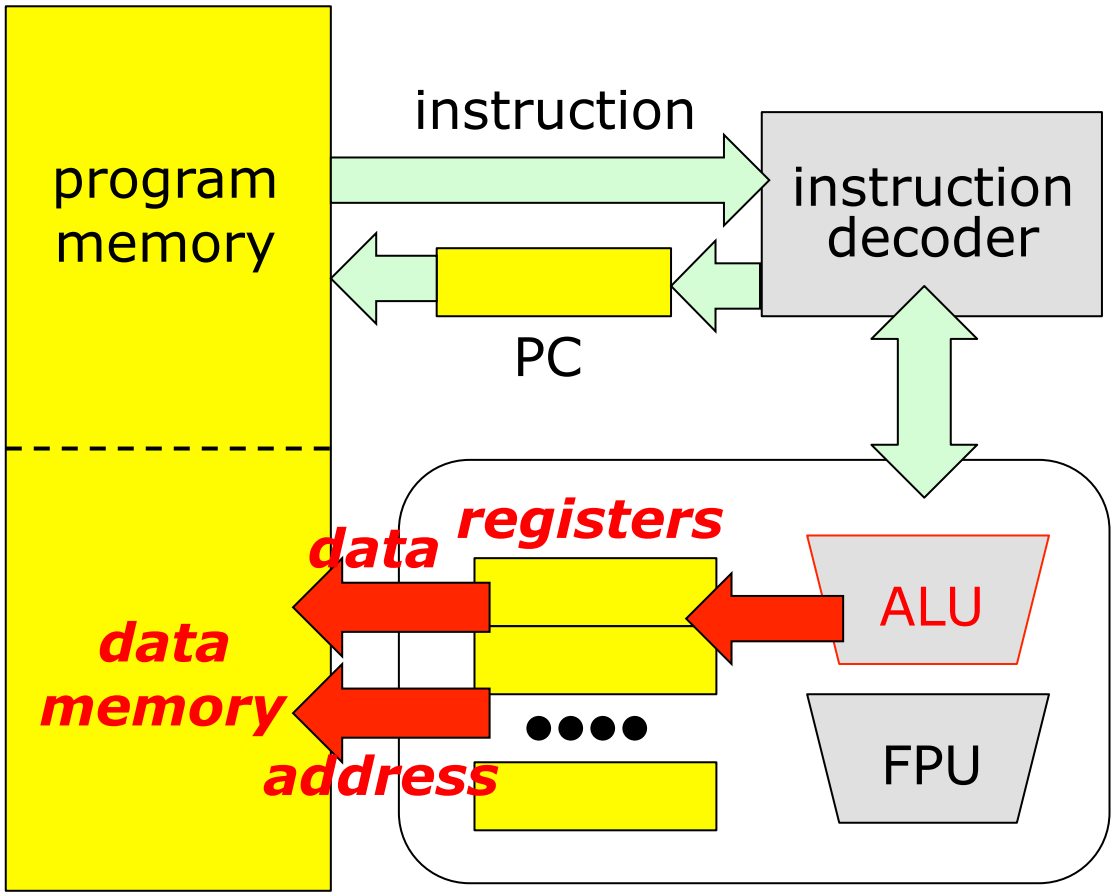
2. Instruction Execution Flow

3) Execute : (c) compute next PC (jump, branch, call, return)



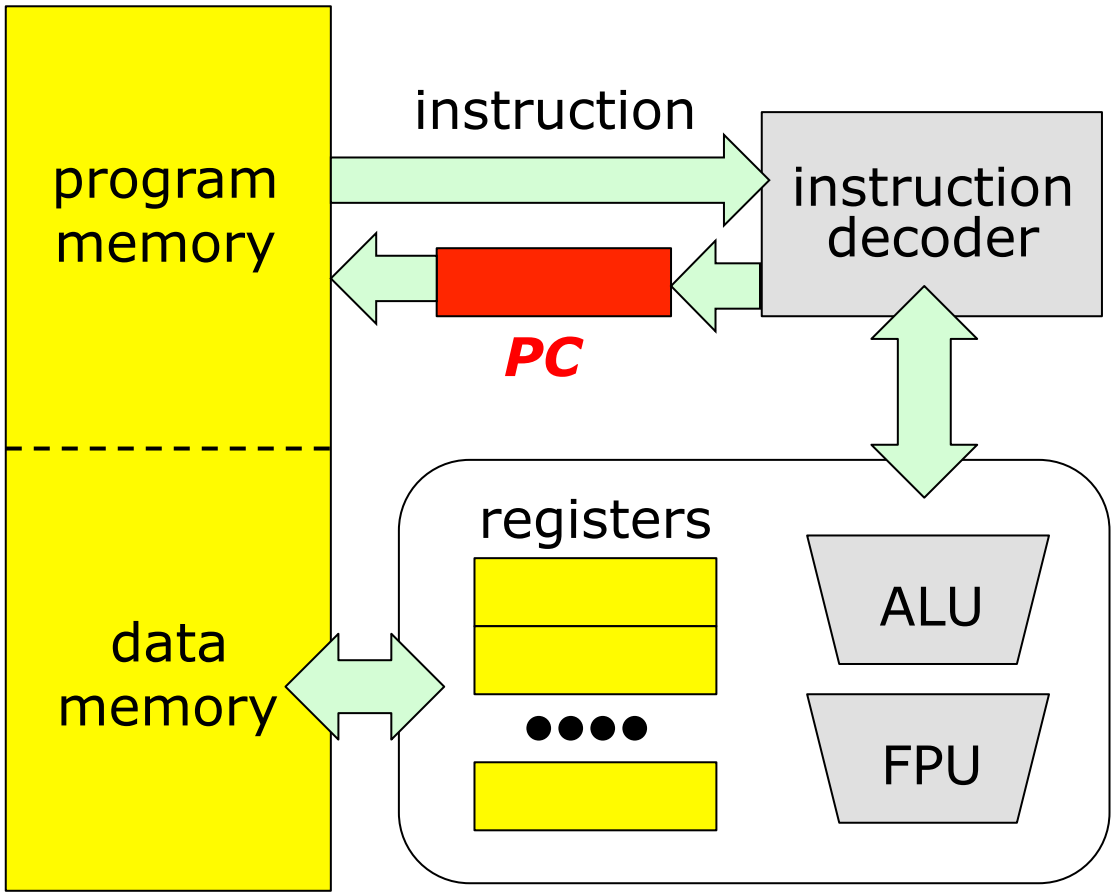
2. Instruction Execution Flow

4) Store result : to register/memory



2. Instruction Execution Flow

5) Update PC



Lecture Outline

1. Microprocessor components : memory, registers, program counter, ALU (arithmetic logic unit), FPU (floating-point unit)
2. Instruction execution flow
3. Computer arithmetics using logic circuits
4. Instruction set classes : CISC vs. RISC
5. Instruction functionalities : data transfer (load, store), compute (add/sub/mult/etc), program control (branch, call/return)
6. Instruction formats : machine code (binary), assembler code, register transfer-level description
7. Instruction set examples: x86, MIPS, ARM

3. Computer Arithmetics using Logic Circuits

Binary data representation

- Unsigned integer

$$value = \sum_{i=0}^{N-1} b_i \cdot 2^i \quad (b_i : i^{\text{th}} \text{ bit})$$

- Signed integer (2's complement)

$$value = -b_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i \cdot 2^i$$

  - Leftmost bit (most significant bit: MSB) is the "sign" bit

sign = 1 : negative integer

sign = 0 : non-negative integer

binary	unsigned	signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Binary Number Calculation (Computer Arithmetic)

- Calculation can be done on binary numbers similarly to "paper & pencil" method on decimal numbers
- Binary number calculation is very convenient for "machine calculators" since each digit is 0 or 1

→ Binary number calculation is one of the key functions for "logic circuits" (*computer arithmetic*)

```
  7   00111
+ 5   00101
 12   01100
```

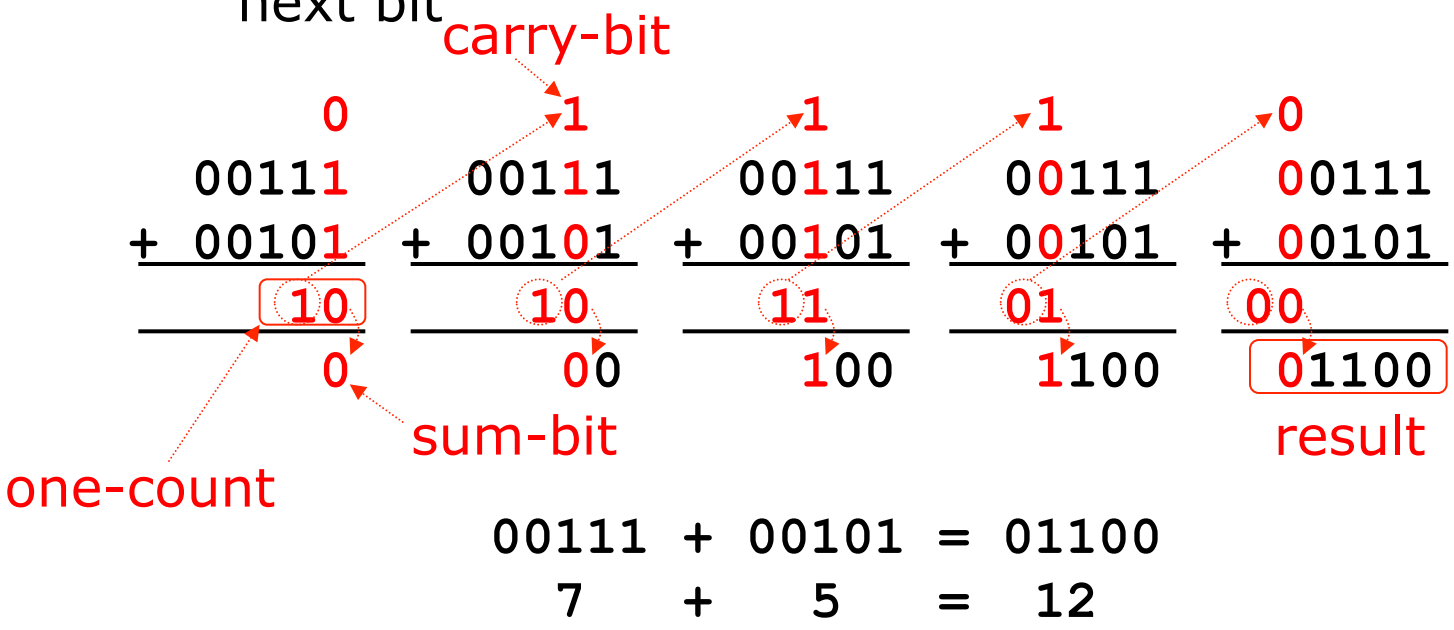
```
  7   00111
- 5   00101
  2   00010
```

```
  12        01100
x 13      x 01101
  36        01100
 12         00000
 156       01100
           01100
          00000
          010011100
```

```
   22          00010110
5 ) 114   101 ) 01110010
  10          101
  14          01000
  10            101
   4            0111
                101
                100
```

Binary Addition

- At each bit position (from right to left):
  - Count the number of 1s ("one-count") and write it in binary numbers
  - Carry the upper bit of the "one-count" to the next bit

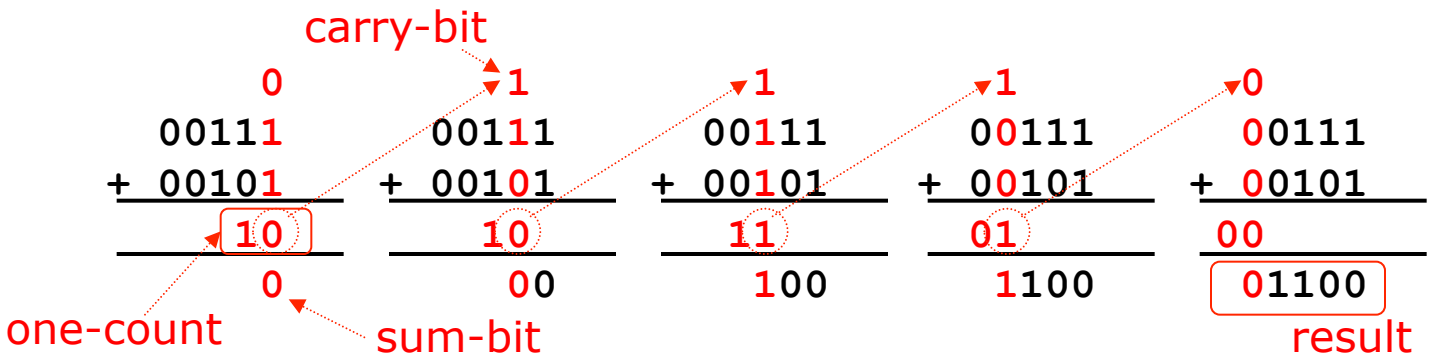


00111 + 00101 = 01100

7 + 5 = 12

1-bit Adder (Full Adder)

- Full adder counts the number of 1s on the 3-bit input



3-bit input

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

one-count (2-bit output)

$c = xy + yz + zx$

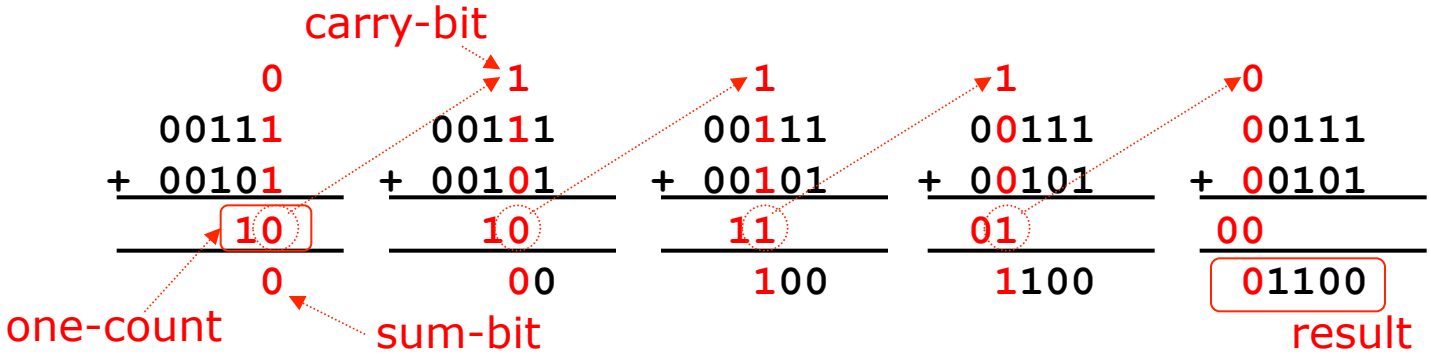
$s = x \oplus y \oplus z$

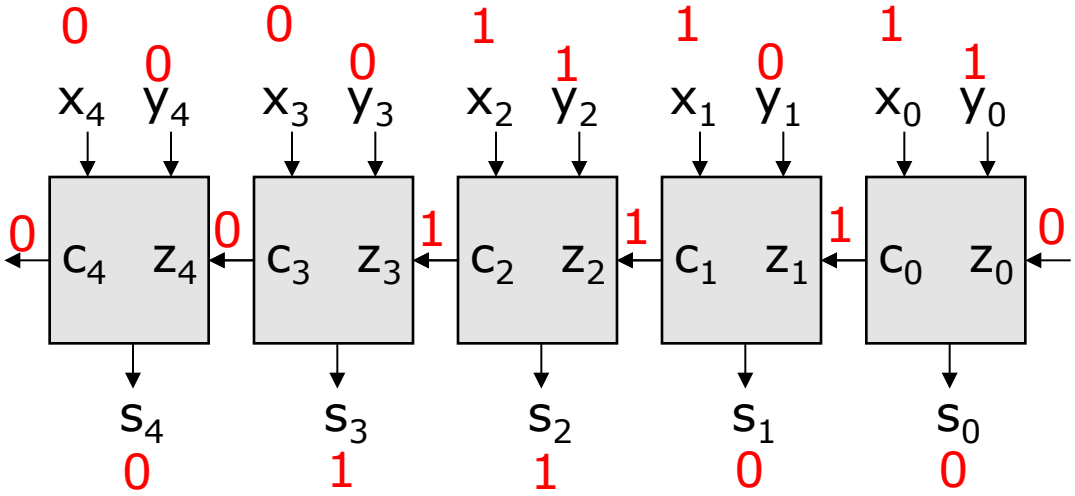
$= (\sim x\ \sim y\ z) + (\sim x\ y\ \sim z)$

$+ (x\ \sim y\ \sim z) + (x\ y\ z)$

N-bit Adder

- N-bit adder can be implemented by cascading N full adders





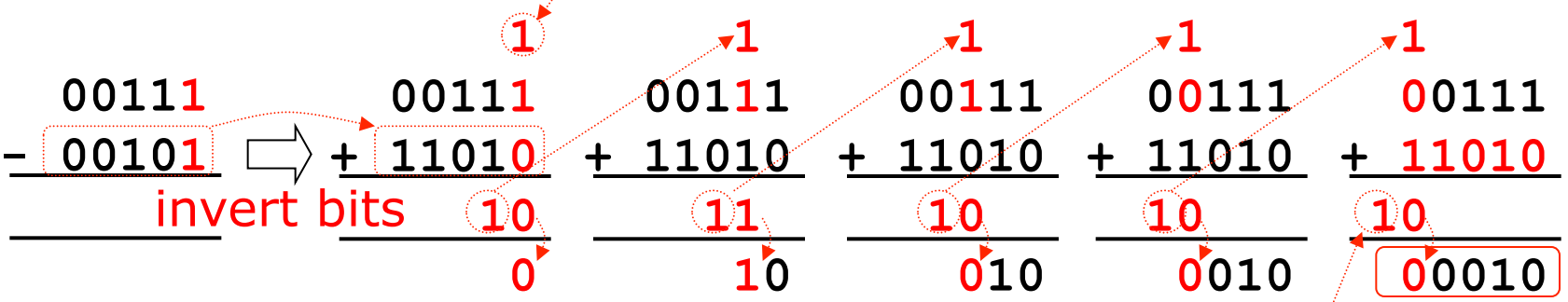
x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Binary Subtraction

- $Z = X - Y$
  - Invert the bits of the 2nd operand ($Y \rightarrow Y'$)
    - $Y + Y' = 2^N - 1$
      $-Y = Y' + 1 - 2^N$
  - $Z = X + Y' + 1 - 2^N$

Ex: 00101 + 11010 = 11111

-2^N has no effects on the N-bit representation

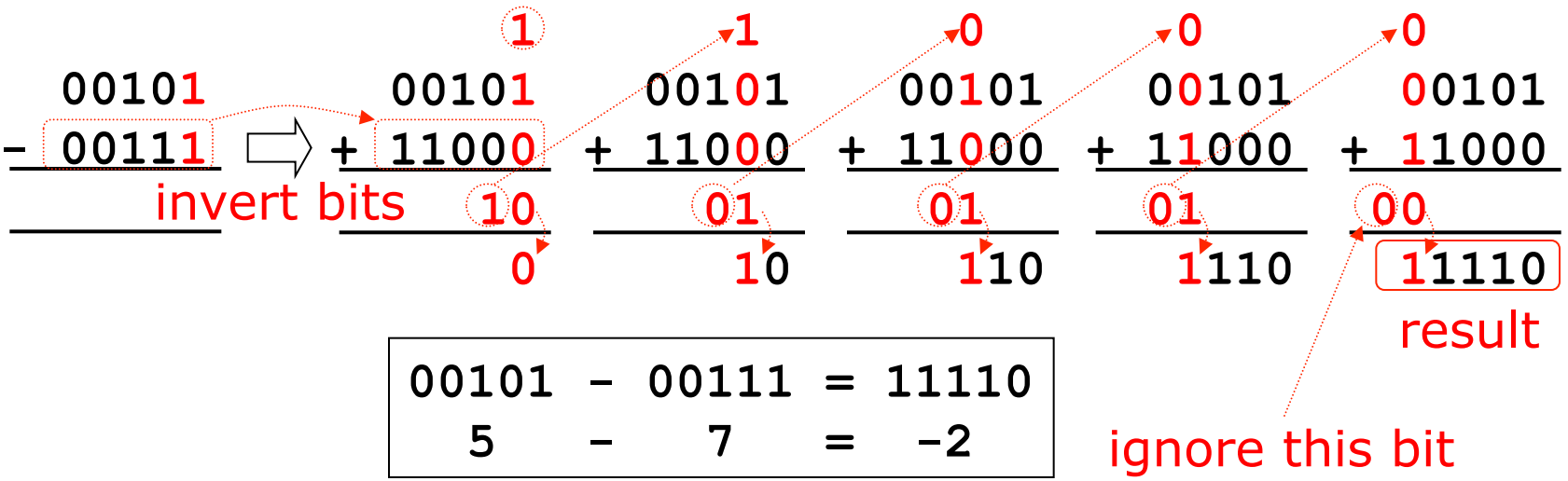


00111 - 00101 = 00010
7 - 5 = 2

ignore this bit

Binary Subtraction

- If the subtraction is negative, the result is represented in 2's complement



00101	–	00111	=	11110
5	–	7	=	–2

Floating-Point Data Representation (IEEE 754 standard)

- Single precision (32-bit, 4-byte)
  - sign(**S**:1-bit), exponent(**E**:8-bit), fraction(**F**:23-bit → *unsigned*)
  - Exponent value biased by –127 ($2^8 - 1$)
  - Normalized numbers (common case)
    - Magnitude range: [2^{-126} , 2^{127}] ("0.0" NOT included)
    - Most significant bit of fraction is always '1' (so omitted)
  - normalized_num = $(-1)^{\mathbf{S}} * 2^{(\mathbf{E} - 127)} * (1 + \mathbf{F} * 2^{-23})$

    1.00 → (0 01111111 00000000000000000000000)

    1.50 → (0 01111111 10000000000000000000000)

    2.25 → (0 10000000 00100000000000000000000)
- Double precision (64-bit, 8-byte)
  - sign(**S**:1-bit), exponent(**E**:11-bit), fraction(**F**:52-bit)
  - normalized_num = $(-1)^{\mathbf{S}} * 2^{(\mathbf{E} - 1023)} * (1 + \mathbf{F} * 2^{-52})$

Floating-Point Data Representation (IEEE 754 standard)

- Outside normalized representation range
  - Denormalized number : very small values ($< 2^{-126}$)
    - **E** = 0 : represents 2^{-126} (not 2^{-127} !)
    - Most significant bit of fraction part is always '0'
    - denormalized_num = $(-1)^{\mathbf{S}} * 2^{-126} * (\mathbf{F} * 2^{-23})$
  - "Zero" (two types!)
    - **E** = 0, **F** = 0

    S = 0 → +0
    S = 1 → −0
  - "Inf" (infinity)
    - **E** = 255, **F** = 0

    S = 0 → +∞
    S = 1 → −∞
  - "NaN" (not-a-number)
    - **E** = 255, **F** ≠ 0
    - x / 0.0 is "NaN" (divided by 0)

Lecture Outline

1. Microprocessor components : memory, registers, program counter, ALU (arithmetic logic unit), FPU (floating-point unit)
2. Instruction execution flow
3. Computer arithmetics using logic circuits
4. Instruction set classes : CISC vs. RISC
5. Instruction functionalities : data transfer (load, store), compute (add/sub/mult/etc), program control (branch, call/return)
6. Instruction formats : machine code (binary), assembler code, register transfer-level description
7. Instruction set examples: x86, MIPS, ARM

3. Instruction-Set Classes

1) **CISC:** Complex Instruction-Set Computer (Intel x86)
   - ***Variable*** instruction length : 1 byte ~ 17 bytes → complex instruction decoder
   - Compute operands can come from *memory*
   - Compute result can be stored to *memory*
2) **RISC:** Reduced Instruction-Set Computer (MIPS, ARM)
   - ***Fixed*** instruction length : 2 bytes, 4 bytes → simple instruction decoder
   - ***Memory access*** : load or store only
   - Compute operands & results : registers

→ ***Why do we have these two different approaches in the instruction set design?***

5. Instruction Functionalities

1) **Data transfer (load/store)**
   - How to specify memory address?
     - Absolute address
     - Base-reg + displacement
     - Base-reg + index-reg * scale + displacement
2) **Compute (add/sub/mult/div/…)**
   - How to specify operands?
     - Registers and immediates → RISC
     - Registers, immedates and memory → CISC
3) **Program control (branch, call/return)**
   - How to specify branch condition?
   - How to specify jump target address
     - Absolute address
     - PC-relative address

6. Instruction Formats

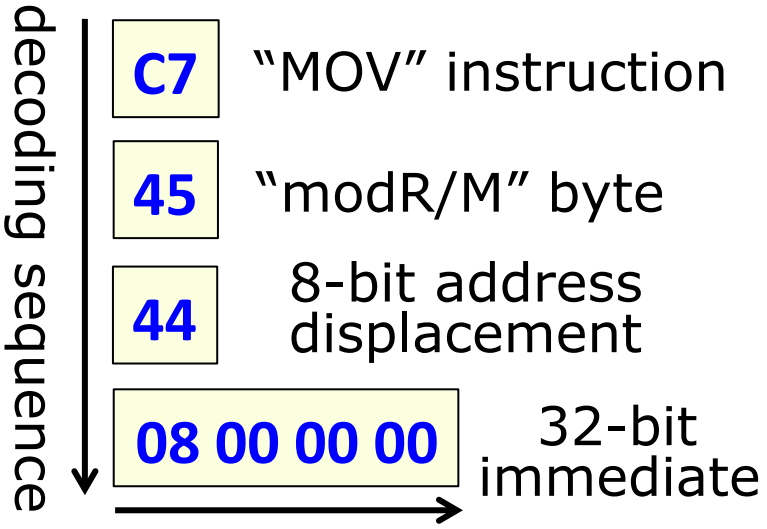
1) **Machine code** (binary) : logic circuit "understands"…
   - Instruction length (if variable-length in CISC)
   - Required actions (fetch operands, compute, store results, update PC)
2) **Assembly code** (text) : language to "specify" …
   - Location of operands (registers, memory, constants)
   - What to do (compute) with the operands
   - Location to store the result (registers, memory)
   - Location of the next instruction (jump, branch, call, return)
3) **Register-transfer description** : describes the movement of data between registers and memory
   - Usually described as (non-formal) expressions
   - Register-transfer level (RTL) description : hardware description language → can describe not only instruction behavior but also any kinds of logic circuits

6. Instruction Formats (x86)

1) Machine code

C7 45 44 08 00 00 00

Instruction byte sequence



2) Assembly code

mov dword ptr [rbp+44h],8

3) Register-transfer description

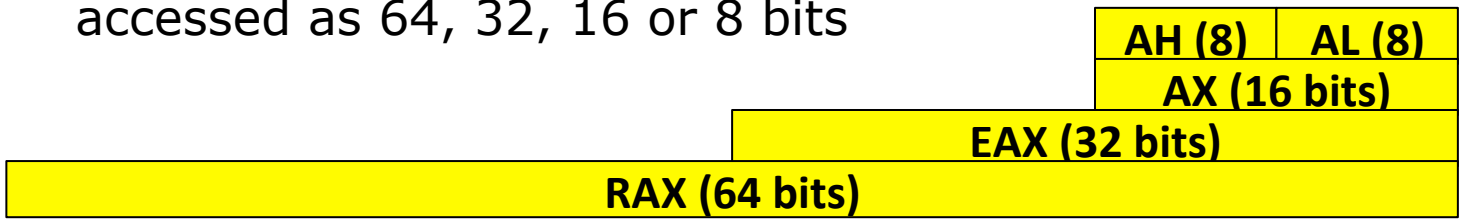
M_{32}[rbp+44h] ← 00000008h

Lecture Outline

1. Microprocessor components : memory, registers, program counter, ALU (arithmetic logic unit), FPU (floating-point unit)
2. Instruction execution flow
3. Computer arithmetics using logic circuits
4. Instruction set classes : CISC vs. RISC
5. Instruction functionalities : data transfer (load, store), compute (add/sub/mult/etc), program control (branch, call/return)
6. Instruction formats : machine code (binary), assembler code, register transfer-level description
7. Instruction set examples: x86, MIPS, ARM

Intel x86-64 (CISC) [1978~]

- **Registers :**
  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 ~ R15 → accessed as 64, 32, 16 or 8 bits



  - RFLAGS: status register (overflow, sign, zero, carry, etc)
  - Segment registers: DS, SS, CS ... → not used often in 64-bit mode
- **Instruction length :** 1 byte ~ 17 bytes
- **Compute instruction :** A = A op B
  - Destination is same as 1st source operand
  - Memory operand (destination) allowed
- **Operand types :**
  - Immediate (constant encoded in instruction)
  - Register
  - Memory : base-reg (+ index-reg*scale)(+ displacement)

"r"-operand: reg

"r/m"-operand: reg or mem

x86-64 Instruction Format

	prefix	REX	opcode	modR/M	SIB	disp	imm
#bytes	(0~3)	(0~1)	(1~3)	(0~1)	(0~1)	(0~4)	(0~4)

- **prefix** (optional) :
  - F0 : atomic instruction (exclusive mem R/W)
  - F2, F3 : Repeat instruction (string instr., IO instr.)
  - 2E, 36, 3E, 26, 64, 65 : segment override
  - 66, 67 : operand/address size override
- **REX** (opt) : extended registers (R8~R15), operand size extension (default size or 64-bit)
- **opcode** (mandantory) : 1 ~ 3 bytes
- **modR/M** (opt) : operand mode (reg, mem addr)
  - reg-IDs: "r"-operand, "r/m" operand
- **SIB** (opt) : base-reg-ID, index-reg-ID, scale
  - ***base-reg*** + ***index-reg*** * ***scale*** + displacement
- **disp** (opt) : address displacement (1, 2, 4 bytes)
- **imm** (opt) : immediate (constant) (1, 2, 4 bytes)

"r"-operand: reg

"r/m"-operand: reg or mem

x86-64 MOV Instructions (Data Transfer : Load/Store)

88 + 1~6B	: r/m8	← r8
89 + 1~6B	: r/m32	← r32
8A + 1~6B	: r8	← r/m8
8B + 1~6B	: r32	← r/m32
B0 + 1B	: r8	← imm8
B8 + 4B	: r32	← imm32
C6 + 2~7B	: r/m8	← imm8
C7 + 5~10B	: r/m32	← imm32

16-bit/64-bit modes indicated in "prefix" "REX" bytes

modR/M (1 byte)

mod	reg-ID	r/m-ID
2-bits	3-bits	3-bits

00 : r/m = M[reg]
01 : r/m = M[reg + disp8]
10 : r/m = M[reg + disp32]
11 : r/m = reg

1st byte

additional # bytes

reg: specified by r/m-ID

r/m-ID = 100 && mod != 11 → *SIB field*

- **imm8, imm32 : 8-bit/32-bit immediate**
- **r/m8, r/m32 : 8-bit/32-bit reg-or-mem operand**
- **r8, r32 : 8-bit/32-bit reg operand**

x86-64 MOV Instructions (Data Transfer : Load/Store)

88 + 1~6B	: r/m8	← r8
89 + 1~6B	: r/m32	← r32
8A + 1~6B	: r8	← r/m8
8B + 1~6B	: r32	← r/m32
B0 + 1B	: r8	← imm8
B8 + 4B	: r32	← imm32
C6 + 2~7B	: r/m8	← imm8
C7 + 5~10B	: r/m32	← imm32

16-bit/64-bit modes indicated in "prefix" "REX" bytes

modR/M (1 byte)

mod	reg-ID	r/m-ID
2-bits	3-bits	3-bits

00 : r/m = M[reg]
01 : r/m = M[reg + disp8]
10 : r/m = M[reg + disp32]
11 : r/m = reg

1st byte

additional # bytes

reg: specified by r/m-ID

- **1st byte (opcode) determines operand size and types**
- **Source operands: immediate, register, memory**
- **Destination operand: register, memory**

r/m-ID = 100 && mod != 11 → *SIB field*

x86-64 ADD Instructions

04 + 1B	: AL	← AL + imm8
05 + 4B	: EAX	← EAX + imm32
81 + 5~10B	: r/m32	← r/m32 + imm32
83 + 2~7B	: r/m32	← r/m32 + imm8
00 + 1~6B	: r/m8	← r/m8 + r8
01 + 1~6B	: r/m32	← r/m32 + r32
02 + 1~6B	: r8	← r8 + r/m8
03 + 1~6B	: r32	← r32 + r/m32

1st byte

additional # bytes

- **1st byte (opcode) determines operand size and types**
- **Source operands: immediate, register, memory**
- **Destination operand: same as one of the source operands**

x86-64 Jcc Instructions
(Program Control : Conditional Jump)

77 + 1B : JA rel8 (if "above")
73 + 1B : JAE rel8 (if "above or equal")
72 + 1B : JB rel8 (if "below")
76 + 1B : JBE rel8 (if "below or equal")
74 + 1B : JE rel8 (if "equal")
75 + 1B : JNE rel8 (if "not equal")
7F + 1B : JG rel8 (if "greater")
7D + 1B : JGE rel8 (if "greater or equal")
7C + 1B : JL rel8 (if "less")
7E + 1B : JLE rel8 (if "less or equal")

→ Same Jcc exists for rel32 (32-bit relative address)

rel8 : 8-bit relative address on cur-PC

"above", "below" : unsigned compare

"greater", "less" : signed compare

- **RFLAGS register stores condition flags**
- **cur-PC : address of NEXT instruction**
- **Jump target address = cur-PC + rel8**

x86-64 Code Example

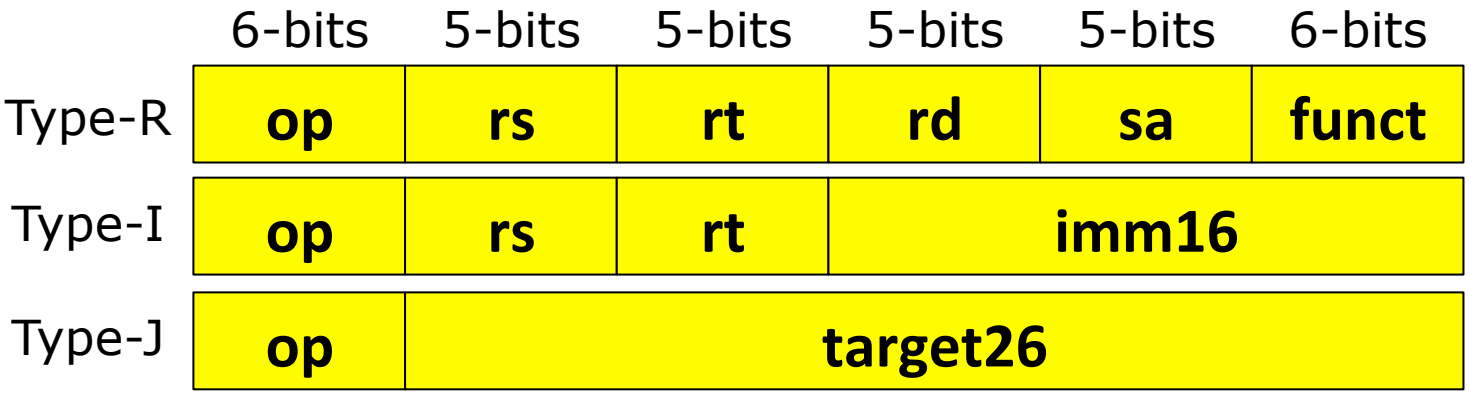
```
013FA58770   40 55                    push    rbp
013FA58772   57                       push    rdi
013FA58773   48 81 EC D8 02 00 00     sub     rsp,2D8h
013FA5877A   48 8D 6C 24 30          lea     rbp,[rsp+30h]
013FA5877F   48 8B FC                mov     rdi,rsp
013FA58782   B9 B6 00 00 00          mov     ecx,0B6h
013FA58787   B8 CC CC CC CC          mov     eax,0CCCCCCCCh
013FA5878C   F3 AB                   rep stos dword ptr [rdi]
013FA5878E   C7 45 04 00 00 00 00    mov     dword ptr [rbp+04h],0
013FA58795   C7 45 24 10 00 00 00    mov     dword ptr [rbp+24h],10h
013FA5879C   C7 45 44 08 00 00 00    mov     dword ptr [rbp+44h],8
013FA587A3   B9 80 00 00 00          mov     ecx,80h
013FA587A8   E8 C9 8A FF FF          call    013FA51276h
013FA587AD   48 89 85 A8 01 00 00    mov     qword ptr [rbp+1A8h],rax
.........
```

- **Variable-length instructions** → complex instruction fetch/decoder logic
- Large number of instruction types (~170) and many addressing mode combinations → compiler design can be ***very difficult***

MIPS (RISC) [1985~]

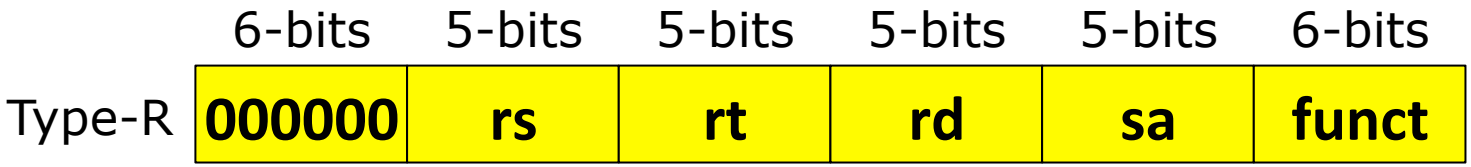
- **Motivation :** simplify processor architecture for faster processing speed, efficient circuit implementation, and easier compiler design → *textbook architecture for research and education*
- **Registers :**
  - 32 x 32-bit general purpose registers
  - R0 : "zero" register
  - Dedicated LO/HI registers (mult/div result)
- **Instruction length :** 32 bits
- **Compute instruction :** A = B op C
  - Source operands: register or immediate
  - Destination operand: register
- **Memory access :** load/store only

MIPS Instruction Format



- **op** : determines format type (R/I/J) and operation
- **rs, rt, rd** : reg-IDs
- **sa** : shift amount
- **funct** : determines operation on Type-R
- **imm16** : 16-bit immediate
- **target26** : 26-bit target address (relative)

MIPS Instruction Format



rd ← funct(rs, rt);

rd ← funct(rs, sa);

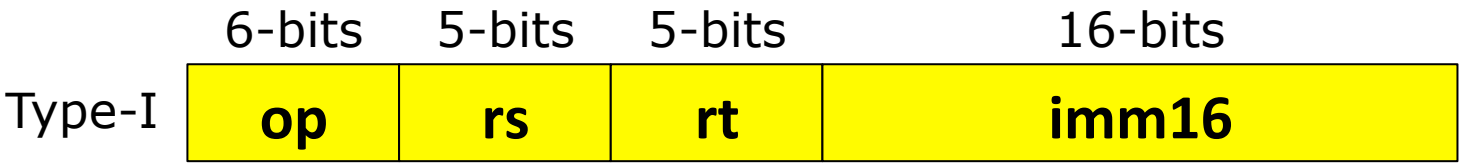
funct :

100000(add), **100001**(addu), **100010**(sub), **100011**(subu)
100100(and), **100101**(or), **100110**(xor)
011010(div), **011011**(divu), **011000**(mult), **011001**(multu)
010000(move from HI : rd ← HI)
010010(move from LO : rd ← LO)
000000(shift-left : rd ← rt << sa)
000100(shift-left : rd ← rt << rs)
000011(shift right arithmetic : rd ← rt >> sa)
000010(shift right logical : rd ← rt >> sa)
101010(set on less than : rd = (rs < rt))
101011(set on less than : rd = (rs < rt) : unsigned)
001000(jump register : PC ← rs)

add, sub : overflow exception

addu, subu : unsigned, no overflow exception

MIPS Instruction Format



op :

[Compute]

001000(addi), **001001**(addiu)

001100(andi), **001101**(ori), **001110**(xori)

rt ← op(rs, imm16)

[Load]

100011(word), **100010**(half), **100101**(half unsigned),
100000(byte), **100100**(byte unsigned)

rt ← M[rs + imm16]

[Store]

101011(store word), **101001**(store half), **101000**(store byte)

M[rs + imm16] ← rt

[Branch]

000100(beq), **000101**(bneq)

if (rs op rt) PC ← PC + imm16

==, !=

MIPS Instruction Format

op :

000010(jump), **000011**(jump and link)

PC ← (PC & F0000000) | (target26 << 2)

R31 ← PC + 8 (jump and link)

- **NOT PC-relative addressing → cannot directly support "PIC"**
- **PIC (Position-independent code): can execute regardless of where the code is loaded in the memory → *shared libraries***
- **Global offset table (GOT) contains the address list of PIC functions → need extra work (instructions) for PIC execution**

MIPS Code Example

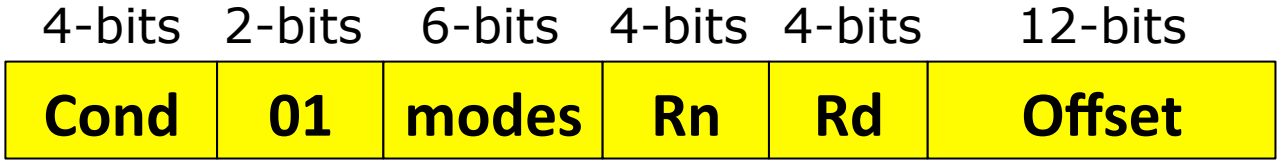
```
80000180    0001D821    addu $27, $0, $1
80000184    3C019000    lui $1, -28672
80000188    AC220200    sw $2, 512($1)
8000018C    3C019000    lui $1, -28672
80000190    AC240204    sw $4, 516($1)
80000194    401A6800    mfc0 $26, $13
80000198    001A2082    srl $4, 516($1)
8000019C    3084001F    andi $4, $4, 31
800001A0    34020004    ori $2, $0, 4
. . . . . .
```

- Fixed-length instructions → simple instruction fetch/ decoder logic
- Small number of instruction types (~110) and simple addressing mode
- Most operands are registers → compiler design become easier

ARM (RISC) [1985~]

- **History :**
  - Acorn Computers (UK) released ARM1/2 (1985~86)
  - Apple/Acorn → spin-off company "Advanced RISC Machines (ARM)" (1990)
  - Wide market adoptions (late 1990's) → PDAs, cell phone
  - 2013 : ~100% world-wide share in smart-phones
- **Normal RISC features :**
  - Fixed instruction length : 32-bit (Thumb: 16-bit)
  - 16 x 32-bit general purpose registers
  - Load/store : "compute" operands from reg/imm only
- **"Advanced" RISC features :**
  - **Predicated instructions :** 4-bit conditional field
  - **Rich addressing mode :** auto-increment, shifted-reg, ...
  - **PC accessible as register** → PC-relative address
  - **Fast context switching :** separate register-sets for different contexts (User, fast-interrupt, interrupt, etc)

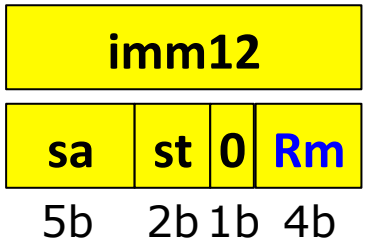
ARM Instruction Format (Single Data Transfer : Load/Store)



- **Cond:** 15 execute conditions
  - **EQ, NE :** equal, not-equal
  - **CS, CC, HI, LS :** unsigned comparison
  - **GE, LT, GT, LE :** signed comparison
  - **MI, PL, VS, VC :** neg/pos, overflow
  - **AL :** unconditionally execute
- **modes:** addressing information (next page)
- **Rn:** base reg-ID
- **Rd:** src/dest data reg-ID
- **Offset:** imm12 or shifted index-reg

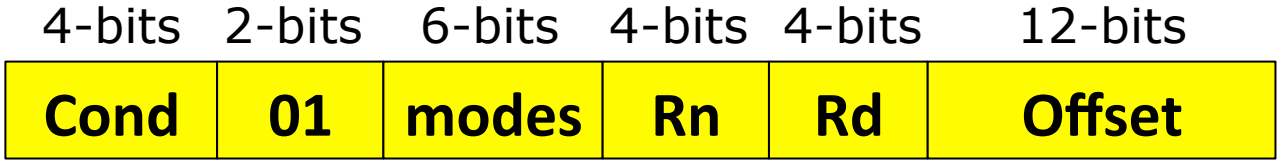
CPSR register stores cond flags

sa : shift amount
Rm : index reg-ID



st : shift type
- **Logical left/right**
- **arithmetic right**
- **rotate right**

ARM Instruction Format (Single Data Transfer : Load/Store)

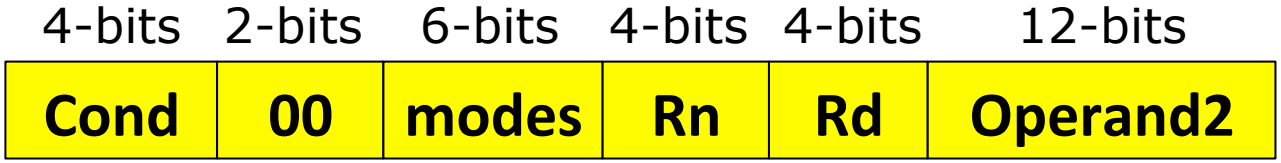


- **modes:** addressing information
  - offset type (1-bit) : imm12 or shifted index-reg
    - **offset = imm12** or **(Rm << sa)**
  - increment mode (3-bits)
    - post/pre : post-increment or pre-increment
    - up/down : positive or negative offset
    - writeback : update base-reg after increment
  - byte/word (1-bit) → *halfword (16-bit) transfer uses different code format*
  - load/store (1-bit)

Rd = M[Rn + imm12]; (pre-inc, no writeback)

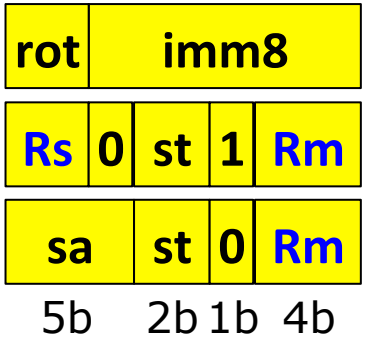
Rd = M[Rn]; Rn += imm12; (post-inc, writeback)

ARM Instruction Format (Data Processing : "Compute")



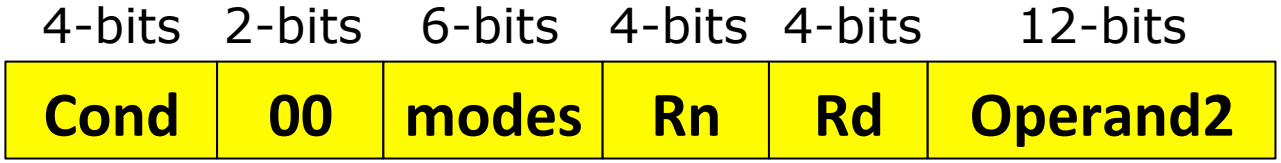
- **Cond:** 15 execute conditions (same)
- **modes:**
  - Operand2-type (1-bit) : rotated-imm8 or shifted-reg
  - **OpCode** (4-bit)
  - condition flag update at **CPSR** reg (1-bit)
- **Rn:** 1st operand reg-ID
- **Rd:** dest reg-ID
- **Operand2:** rotated-imm8 or shifted-reg

sa : shift amount
Rm : index reg-ID
Rs : shift amount reg

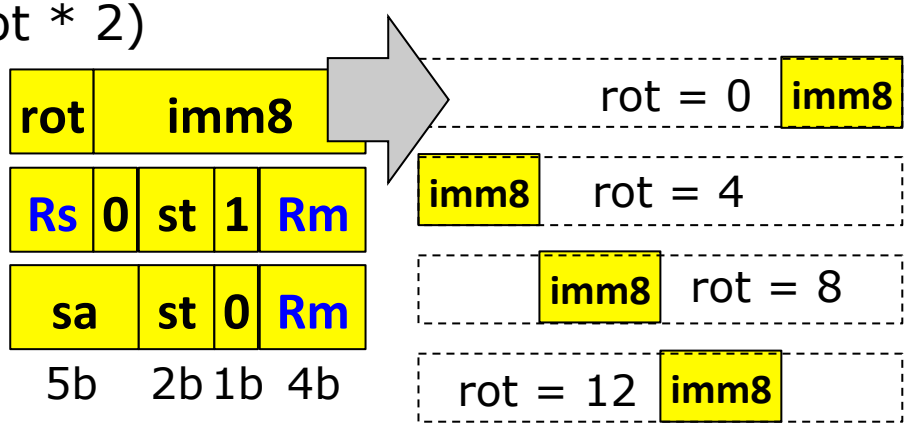


st : shift type
- **Logical left/right**
- **arithmetic right**
- **rotate right**

ARM Instruction Format (Data Processing : "Compute")



- **OpCode** (4-bits inside **modes**):
  - Add/Sub : SUB, RSB, ADD, ADC, SBC, RSC
  - Logical/Move : AND, EOR, ORR, MOV, BIC, MVN
  - Compare/Test : TST, TEQ, CMP, CMN
- **Operand2 format**
  - rotate(zeroExt(imm8), rot * 2)
  - Rm << sa
  - Rm << Rs
- **Compute format**
  - Rd ← Rn op Operand2



Summary

1. Microprocessor components : memory, registers, program counter, ALU, FPU
2. Instruction execution flow
3. Computer arithmetics
4. Instruction sets : CISC vs. RISC
   - Functionalities : data transfer, compute, program control
   - Formats : machine code, assembler code, register transfer description

- **THINK ABOUT:** (not a homework question)
  - Why are there so many different instruction sets?
  - Why are most of them not used anymore?
  - Why are some of them still being used?