中国科学院大学
University of Chinese Academy of Sciences

CS101

Systems Thinking

Modularization-2:
Instruction Set and Instruction Pipeline

zxu@ict.ac.cn
zhangjialin@ict.ac.cn

Outline

- What is systems thinking?
- Three objectives of systems thinking
- Abstraction
- Modularization
  - Modularization and modules
  - Combinational circuits
  - Sequential circuits
  - Instruction Set and Instruction Pipeline
    - Design a simple instruction set
    - Executing instructions by an instruction pipeline
  - Software Stack
- Seamless transition

These slides acknowledge sources for additional data not cited in the textbook

4.4 Instruction set and instruction pipeline

- 1-minute quiz
  - Q1: Combinational circuits vs. Boolean expressions
    What are the relationships between combinational circuits and Boolean expressions?

4.4 Instruction set and instruction pipeline

- 1-minute quiz
  - Q1: Combinational circuits vs. Boolean expressions
    What are the relationships between combinational circuits and Boolean expressions?
  - A1: Combinational circuits are equivalent to and implement Boolean expressions

4.4 Instruction set and instruction pipeline

- 1-minute quiz
  - Q1: Combinational circuits vs. Boolean expressions
    What are the relationships between combinational circuits and Boolean expressions?
  - A1: Combinational circuits are equivalent to and implement Boolean expressions

- Remarks
  - For any combinational circuit, there is an equivalent Boolean expression
  - And vice versa
    - Here, equivalence means they both have the same truth table
  - A combinational circuit implements a Boolean expression
    - By a logic diagram of gates

4.4 Instruction set and instruction pipeline

- 1-minute quiz
  - Q2: Sequential circuits vs. automata
    What are the relationships between sequential circuits and automata?

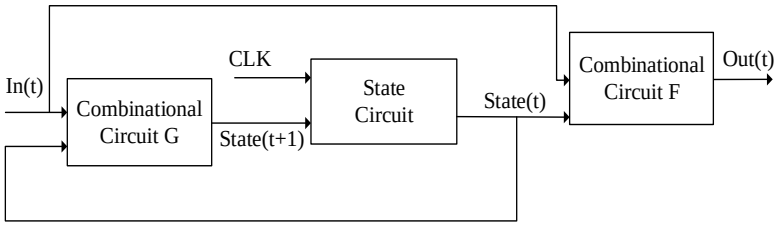
4.4 Instruction set and instruction pipeline

- 1-minute quiz
  - Q2: Sequential circuits vs. automata
    What are the relationships between sequential circuits and automata?
  - A2: Sequential circuits are equivalent to and implement automata

4.4 Instruction set and instruction pipeline

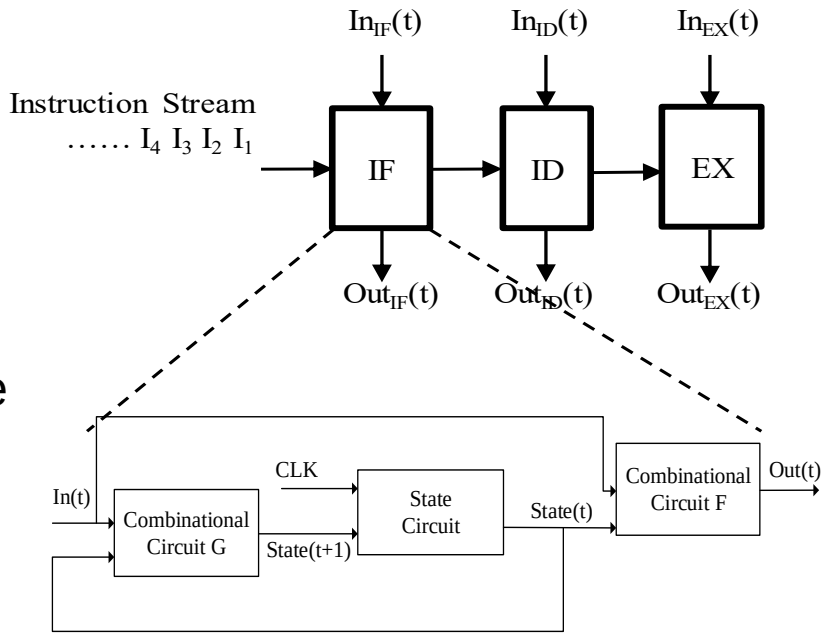
- 1-minute quiz
  - Q2: Sequential circuits vs. automata
    What are the relationships between sequential circuits and automata?
  - A2: Sequential circuits are equivalent to and implement automata

- Remarks
  - For any sequential circuit, there is an equivalent automaton
  - And vice versa
    - Here, equivalence means they both have the same state transition table and initial conditions
  - A sequential circuit implements a automaton
    - By a logic diagram of a state circuit and two combinational circuits



4.4 Instruction set and instruction pipeline

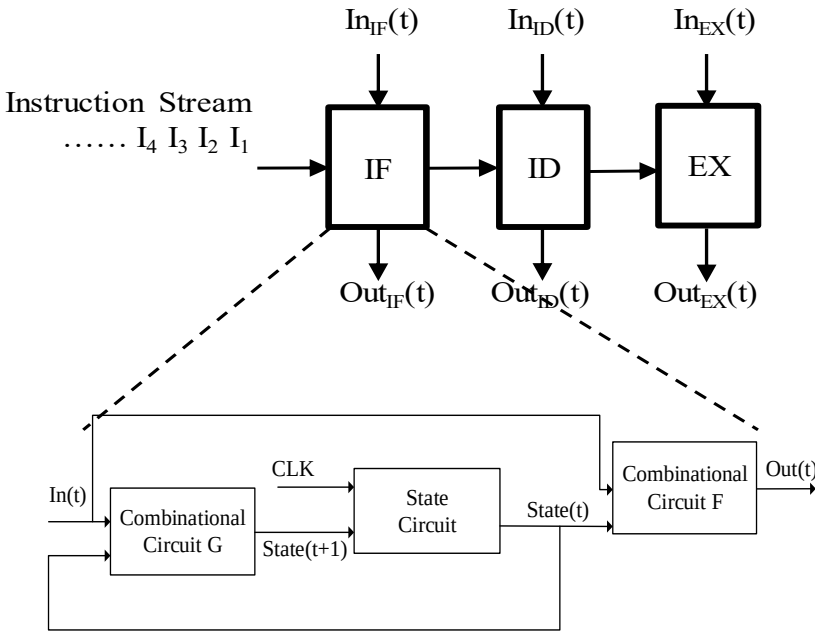
- Automata (sequential circuits) are basic concepts, widely used in computers and application systems
- A processor (CPU) is implemented as a group of sequential circuits
  - Instruction pipeline is the basic hardware abstraction to execute instructions
  - Each stage of the instruction pipeline is a sequential circuit
- Example
  - The 3-stage instruction pipeline (when executing instruction MOV 0, R1)
    - Instruction Fetch (IF) stage: IR ← M[PC]
      - Fetch an instruction from the memory cell M[PC] to the Instruction Register of CPU
    - Instruction Decode (ID): Control Signals = Decode(IR)
      - Decode the instruction to generate control signals
    - Instruction Execute (EX): R1 ← 0; PC ← PC+1
      - Execute the instruction according to the control signals, and increment PC



A 3-stage instruction pipeline is implemented as 3 sequential circuits

4.4 Instruction set and instruction pipeline

- Automata (sequential circuits) are basic concepts, widely used in computers and application systems
- A processor (CPU) is implemented as a group of sequential circuits
  - Instruction pipeline is the basic hardware abstraction to execute instructions
  - Each stage of the instruction pipeline is a sequential circuit
- The 3-stage instruction pipeline can execute instructions in overlapped mode, thus increase clock frequency
  - E. g., 1 GHz → 3 GHz
  - See pipelining when discussing Amdahl's law



A 3-stage instruction pipeline is implemented as 3 sequential circuits
Practical CPUs have 5~31 pipeline stages

4.4.1 Design the instruction set of FC

- The Fibonacci Computer (FC) executes only the following code (shown in both Go and assembly language notations)
  - Recall Section 2.3 in textbook

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]          //R0=12 initially
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2              // i:=2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1              // label Loop
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2                 // i++
                                    CMP 51, R2             // i < 51?
}                                   JL Loop                // if Yes, goto Loop
```

- Design an instruction set for FC
  - Any instruction consists of an opcode and one or more operands
    - E.g., **opcode operand operand**
  - In mnemonics, e.g., **MOV 0, R1**
  - In binary, e.g., **000 000000 01**

Design Process

- FC has a memory and five registers
  - FLAGS: CPU status register
    - Holding status value of instruction execution, such as if the result is overflow, zero, less than, etc.
    - Only “less than” is used in this example
  - PC: program counter
    - Holding the address of the next instruction to be executed
  - R0, R1, and R2: general purpose registers
    - Holding operands of instructions
- Determine the types of instructions and decide the opcodes (one for a type)
  - Merge similar instructions into a type
  - E.g., There are three distinct instructions moving an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes (one for a type)
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2 √
                                    CMP 51, R2
}                                   JL Loop
```

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2 √
                                    CMP 51, R2 √
}                                   JL Loop
```

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
  - Merge similar instructions into a type
  - E.g., 3 instructions move an immediate value to a register
    - MOV 0, R1; MOV 1, R1; MOV 2, R2
    - They belong to one type of instruction
- There are six types of instructions

```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]  √
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8] √
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1√
                                    ADD M[R0+R2*8-8], R1√
                                    MOV R1, M[R0+R2*8-0] √
                                    INC R2 √
                                    CMP 51, R2 √
}                                   JL Loop √
```

Need 3 bits for the opcodes

Instruction Type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Design Process

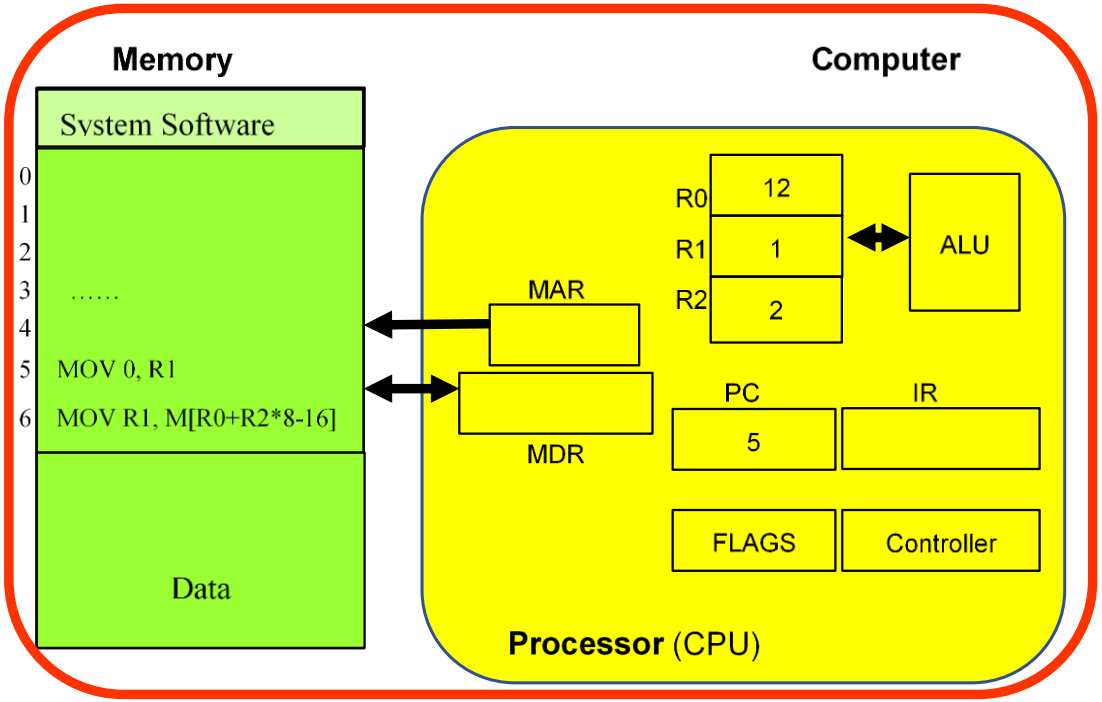
```
fib[0] = 0                          MOV 0, R1
                                    MOV R1, M[R0]
fib[1] = 1                          MOV 1, R1
                                    MOV R1, M[R0+8]
for i := 2; i < 51; i++ {           MOV 2, R2
   fib[i] = fib[i-1] + fib[i-2]     MOV 0, R1
                                    ADD M[R0+R2*8-16], R1
                                    ADD M[R0+R2*8-8], R1
                                    MOV R1, M[R0+R2*8-0]
                                    INC R2
                                    CMP 51, R2
}                                   JL Loop
```

- FC has a memory and five registers
  - FLAGS, PC, R0, R1, and R2
- Determine the types of instructions and decide the opcodes
- For each opcode, determine its operands
  - Assuming the instruction length = 11 bits In practice, assume 8, 16, 32 or 64 bits
  - There are 3 data registers, needing 2 bits
  - Leave 6 bits for immediate value
  - The base+index+offset mode
    - Address = R0 + R2*I + J, where
      R0, R2 are fixed
      I = 0, 1, 2, 4, 8
      J = 0, ± 4, ± 8, ± 16
    - 5×7 = 35 possible (I, J) pairs
    - $35 < 2^6$, 6 bits are enough
- Notes
  - For INC R2, operand 1 can be any value
  - JL has only one operand

Opcode 3-bit	Operand 1 Immediate Value, 6-bit	Operand 2 Register, 2-bit	Instruction
000	000000	01	MOV 0, R1
000	000001	01	MOV 1, R1
000	000010	10	MOV 2, R2
011	111111	10	INC R2
100	110011	10	CMP 51, R2
101	00000101		JL Loop
Opcode 3-bit	**Operand 1 Memory Address, 6-bit**	**Operand 2 Register, 2-bit**	**Instruction**
001	R0+R2*0+0	01	MOV R1, M[R0]
001	R0+R2*0+8	01	MOV R1, M[R0+8]
001	R0+R2*0-0	01	MOV R1, M[R0+R2*8-0]
010	R0+R2*8-8	01	ADD M[R0+R2*8-8], R1
010	R0+R2*8-16	01	ADD M[R0+R2*8-16], R1

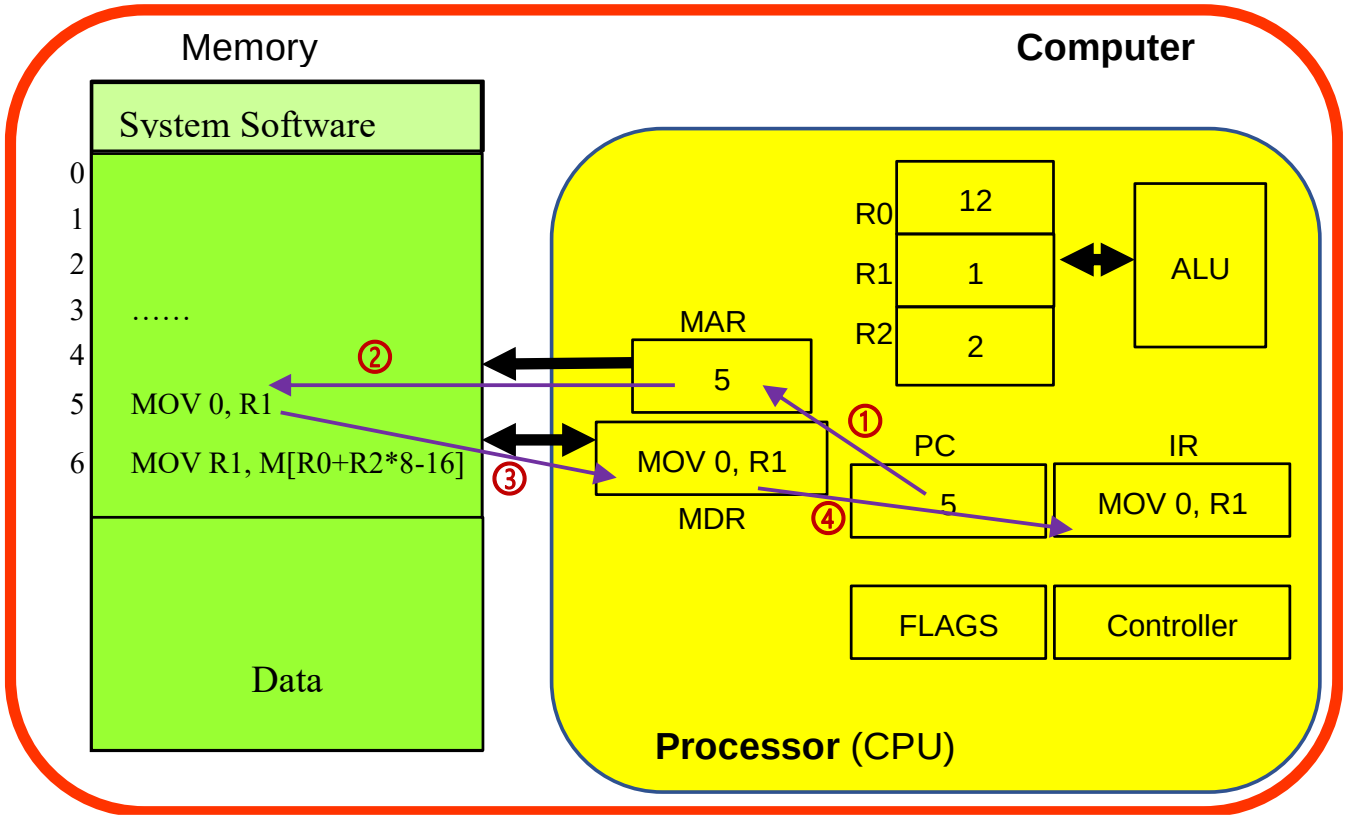
4.4.2 Look inside a processor

- To see an example of executing instruction MOV 0, R1
  - by a 3-stage instruction pipeline
    - Instruction Fetch (IF): IR←M[PC]
    - Instruction Decode (ID): Signals = Decode(IR)
    - Instruction Execute (EX): R1 ← 0; PC ← PC+1
- Internal components not visible to user
  - **IR**: **Instruction Register** holding the instruction being executed
  - **MAR**: **Memory Address Register**, holding the memory address used
  - **MDR**: **Memory Data Register**, holding the data for a load or store
  - Controller: control circuit to generate control signals



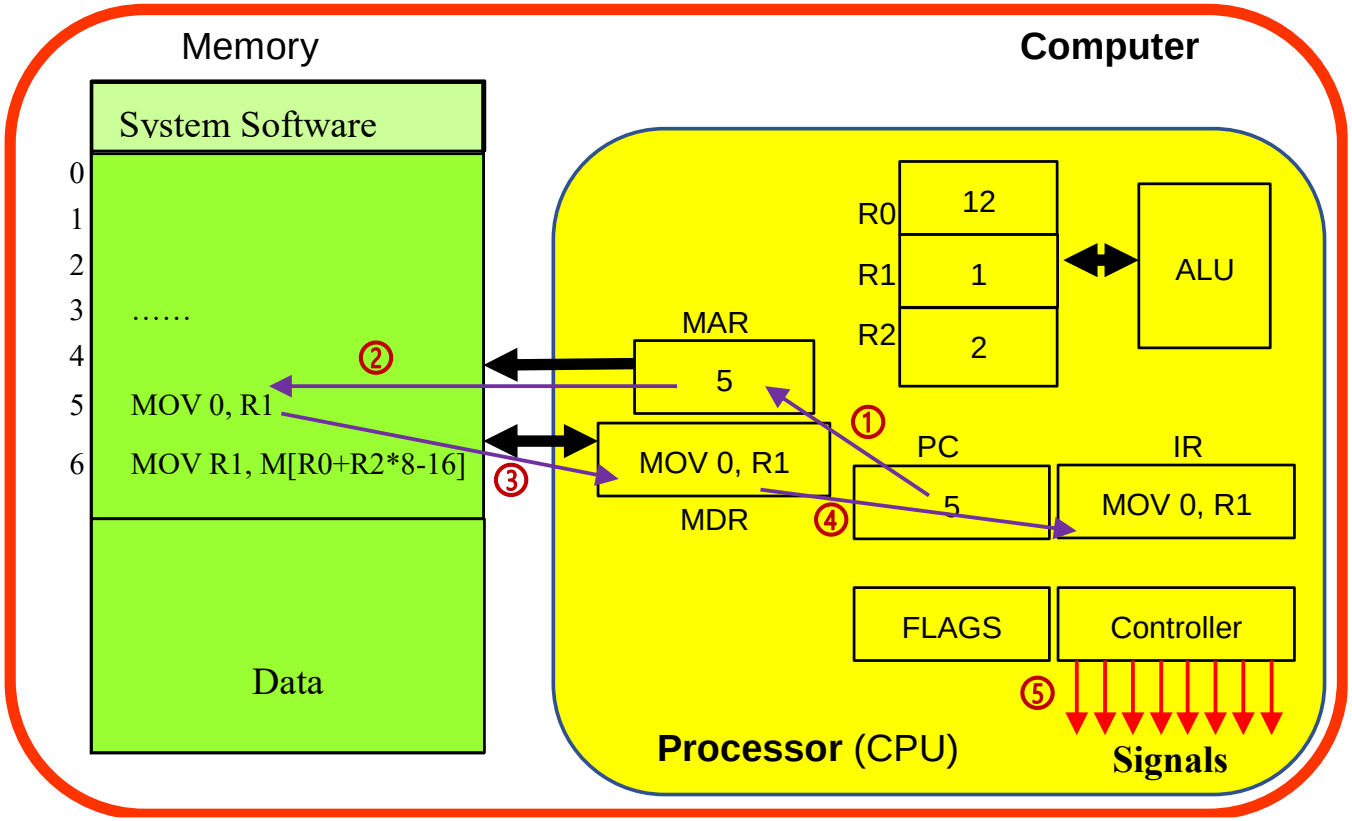
Execution details

- After IF stage (micro operations ① ② ③ ④)



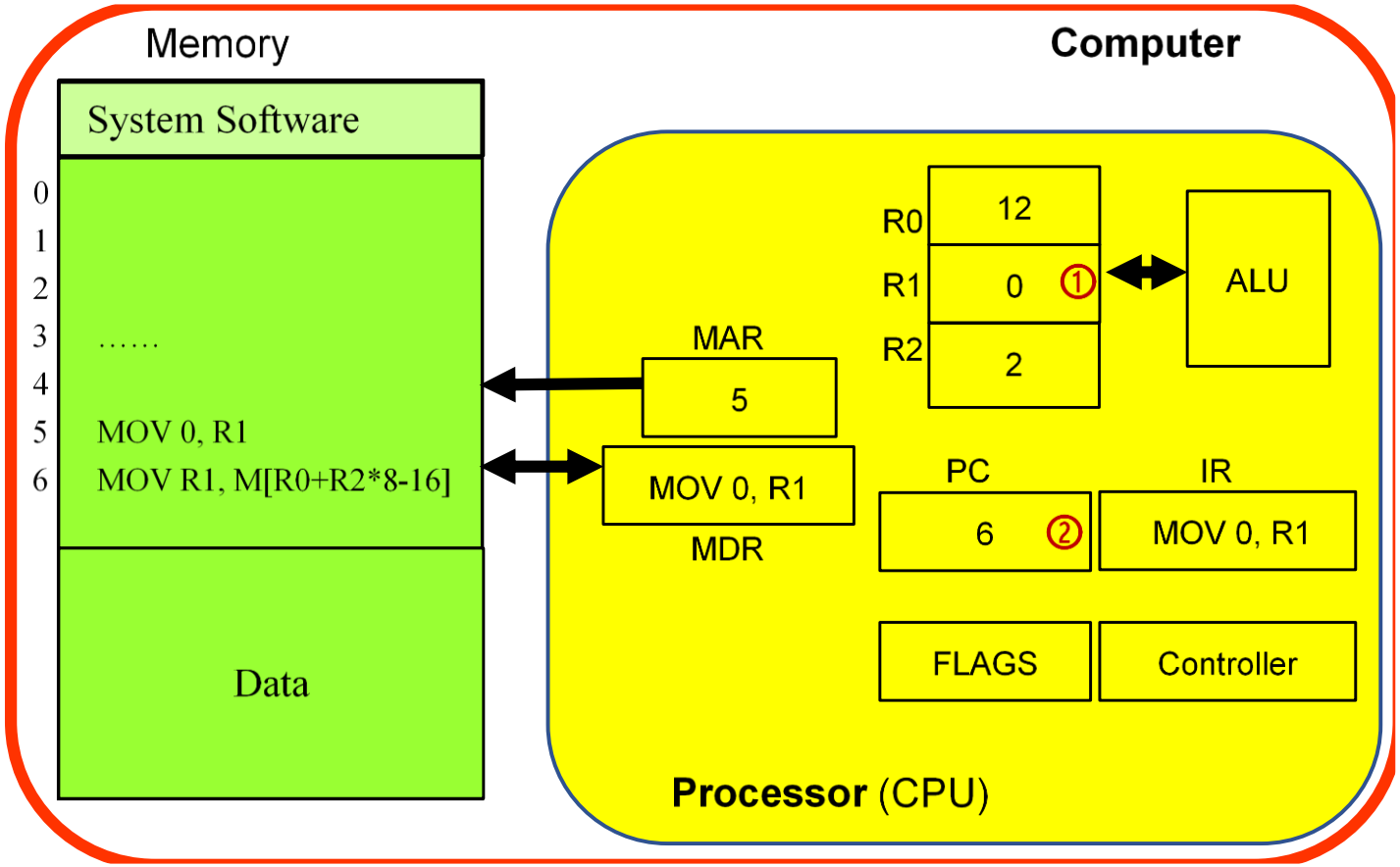
Execution details

- After IF stage (micro operations ① ② ③ ④)
- After ID stage (micro operation ⑤)



Execution details

- After EX stage (micro operations ① ②)
  - 0→R1; PC+1→PC



4.5 Software Stack on a von Neumann Computer

- Software is organized as a layered structure, called software stack
  - Upper layers use lower layers, utilizing the modularization and reuse advantages
- Notes
  - Middleware: between application software and system software
  - Firmware stored in ROM (**why**), e.g., BIOS (the Basic Input/Output System)
  - Software comes in source code form binary code form

<table>
<tr><th colspan="3">Software Type</th><th>Example</th></tr>
<tr><td colspan="3">Application Software</td><td>Scientific computing, Business computing, Personal productivity software; fib.dp.go, myPage.html PDF, Search Engine, TikTok, WeChat</td></tr>
<tr><td rowspan="4">Infrastructure Software</td><td>Middleware</td><td>Databases, Web servers, Web Browsers</td><td>MySQL, Nginx, WebServer.go Chrome, Safari</td></tr>
<tr><td rowspan="3">System Software</td><td>Languages, Compilers, Interpreters</td><td>C, Go, JavaScript, Python Shell</td></tr>
<tr><td>Operating Systems</td><td>Linux, Android, iOS, Windows</td></tr>
<tr><td>Firmware</td><td>BIOS</td></tr>
<tr><td colspan="4">von Neumann Architecture</td></tr>
<tr><td colspan="4">Hardware</td></tr>
</table>

Students use software in blue and create software in red