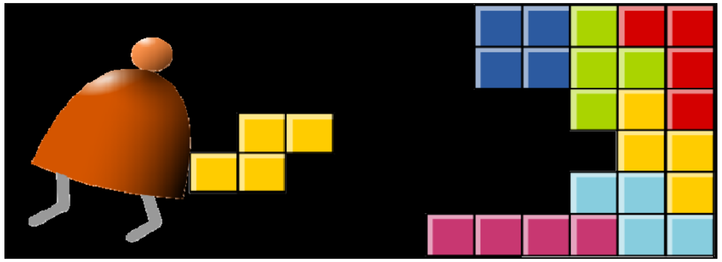
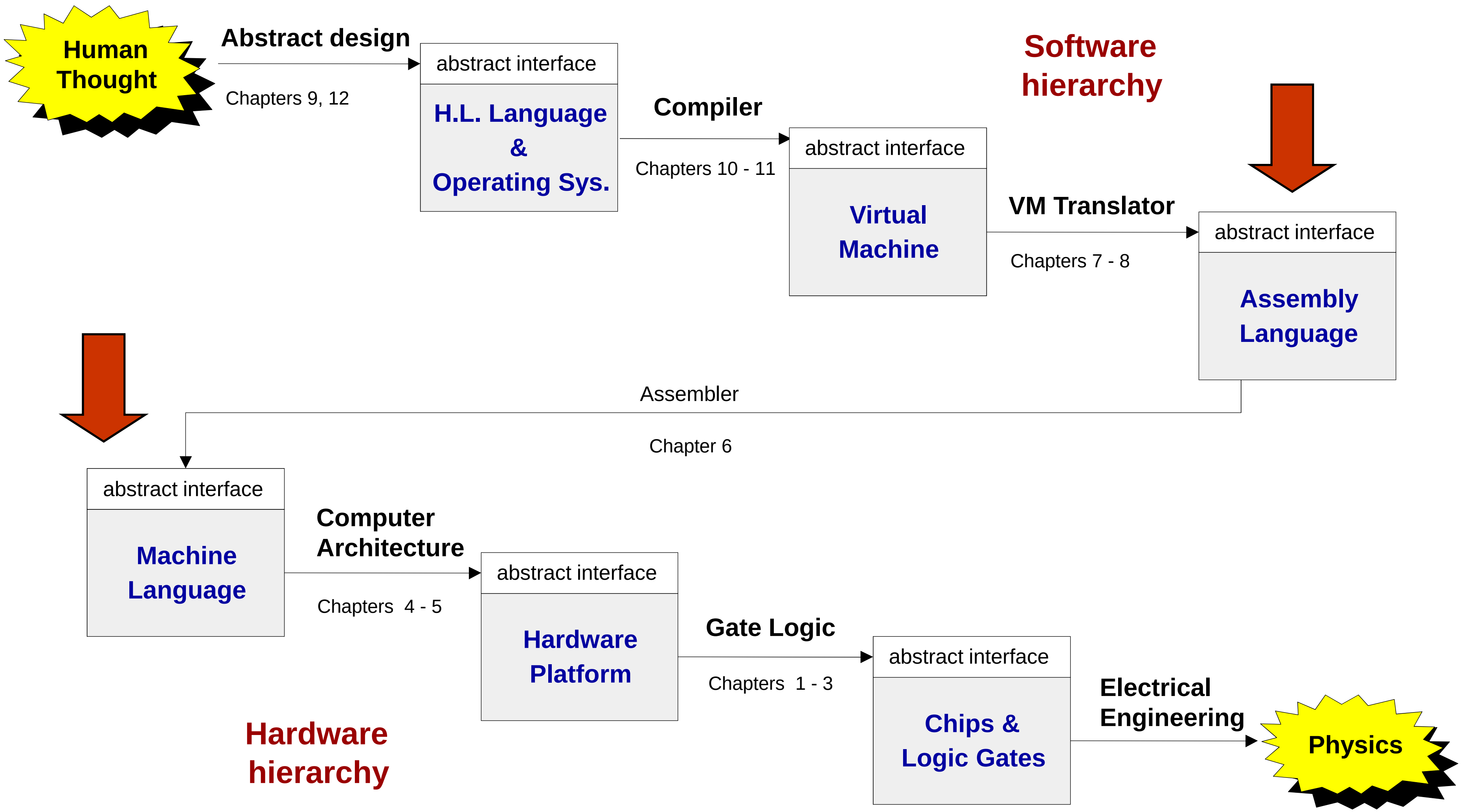
Machine (Assembly) Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



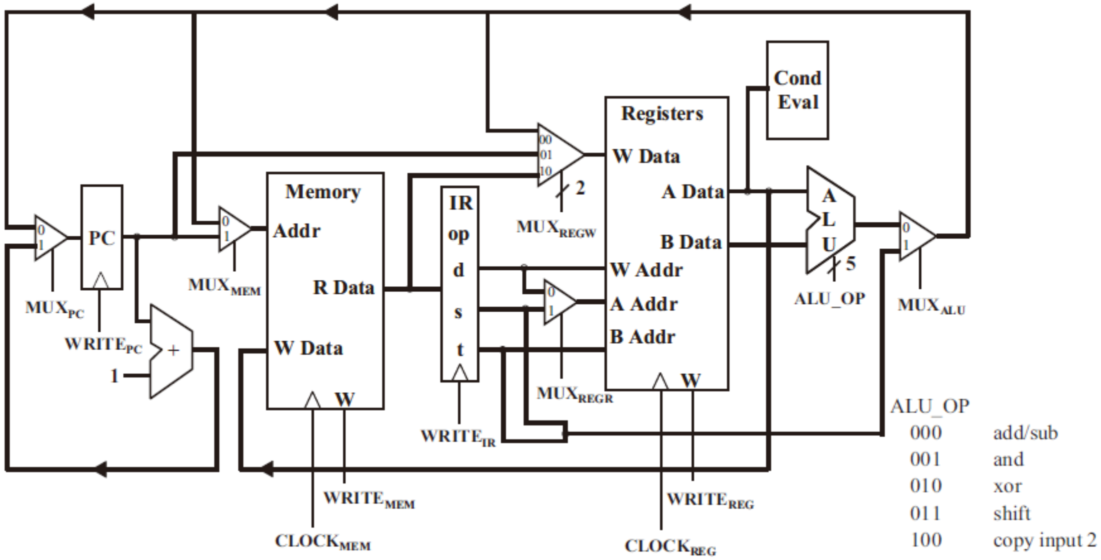
Machine language

Abstraction – implementation duality:

- Machine language (= instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform

- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	1	pc ← R[t]
F:	jump and link	2	R[d] ← pc; pc ← addr



Machine language

Abstraction – implementation duality:

- Machine language (= instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform
- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

Another duality:

- Binary version: 0001 0001 0010 0011 (machine code)
- Symbolic version ADD R1, R2, R3 (assembly)

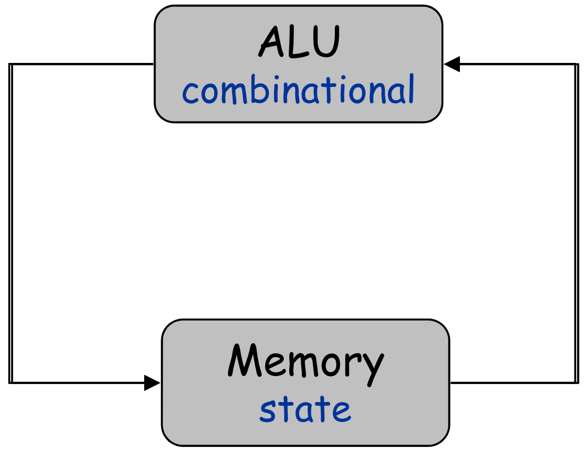
Machine language

Abstraction – implementation duality:

- Machine language (= instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform
- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

Another duality:

- Binary version
- Symbolic version



Loose definition:

- Machine language = an agreed-upon formalism for manipulating a *memory* using a *processor* and a set of *registers*
- Same spirit but different syntax across different hardware platforms.

Lecture plan

- Machine languages at a glance
- The Hack machine language:
  - Symbolic version
  - Binary version
- Perspective

(The assembler will be covered in chapter 6).

Typical machine language commands (3 types)

- ALU operations
- Memory access operations

  (addressing mode: how to specify operands)

  - Immediate addressing, LDA R1, 67 // R1=67
  - Direct addressing, LD R1, 67 // R1=M[67]
  - Indirect addressing, LDI R1, R2 // R1=M[R2]
- Flow control operations

Typical machine language commands (a small sample)

```
// In what follows R1,R2,R3 are registers, PC is program counter,
// and addr is some value.

ADD R1,R2,R3       // R1 ← R2 + R3

ADDI R1,R2,addr    // R1 ← R2 + addr

AND R1,R1,R2       // R1 ← R1 and R2 (bit-wise)

JMP addr           // PC ← addr

JEQ R1,R2,addr     // IF R1 == R2 THEN PC ← addr ELSE PC++

LOAD R1, addr      // R1 ← RAM[addr]

STORE R1, addr     // RAM[addr] ← R1

NOP                // Do nothing

// Etc. – some 50-300 command variants
```

The Hack computer

A 16-bit machine consisting of the following elements:

Data memory: **RAM** – an addressable sequence of registers

Instruction memory: **ROM** – an addressable sequence of registers

Registers: **D**, **A**, **M**, where **M** stands for **RAM[A]**

Processing: **ALU**, capable of computing various functions

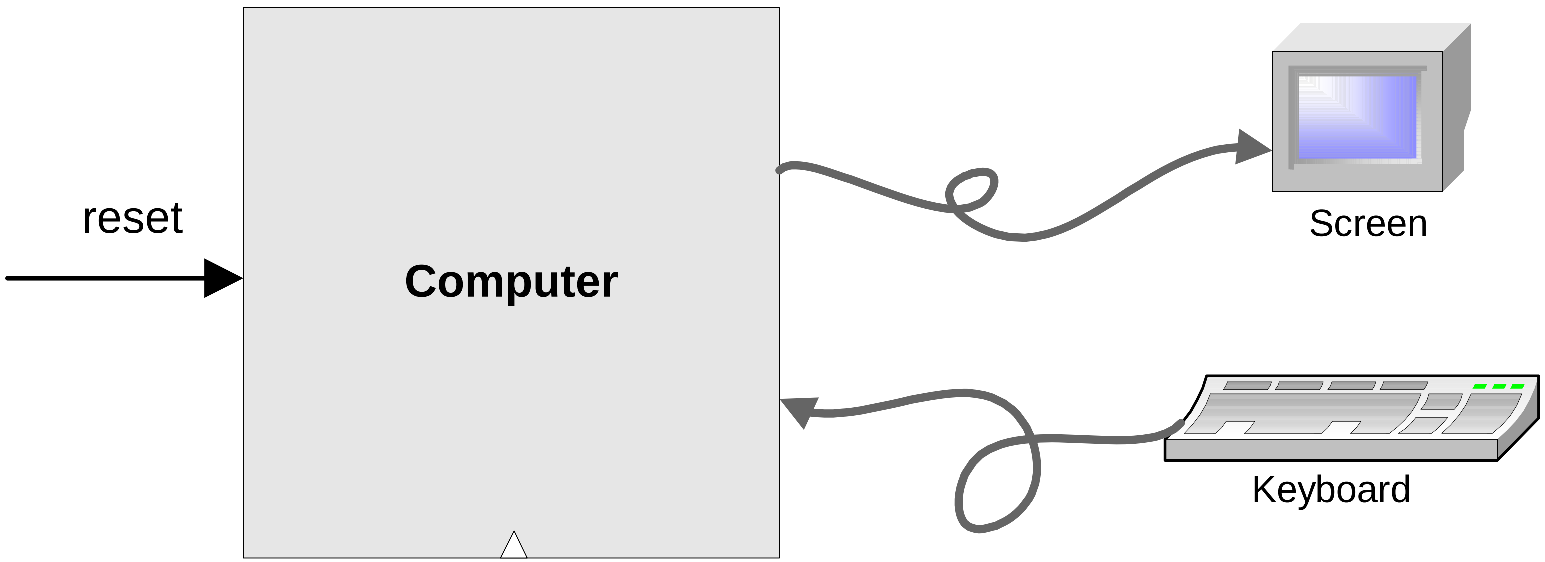
Program counter: **PC**, holding an address

Control: The **ROM** is loaded with a sequence of 16-bit instructions, one per memory location, beginning at address 0. Fetch-execute cycle: later

Instruction set: Two instructions: A-instruction, C-instruction.

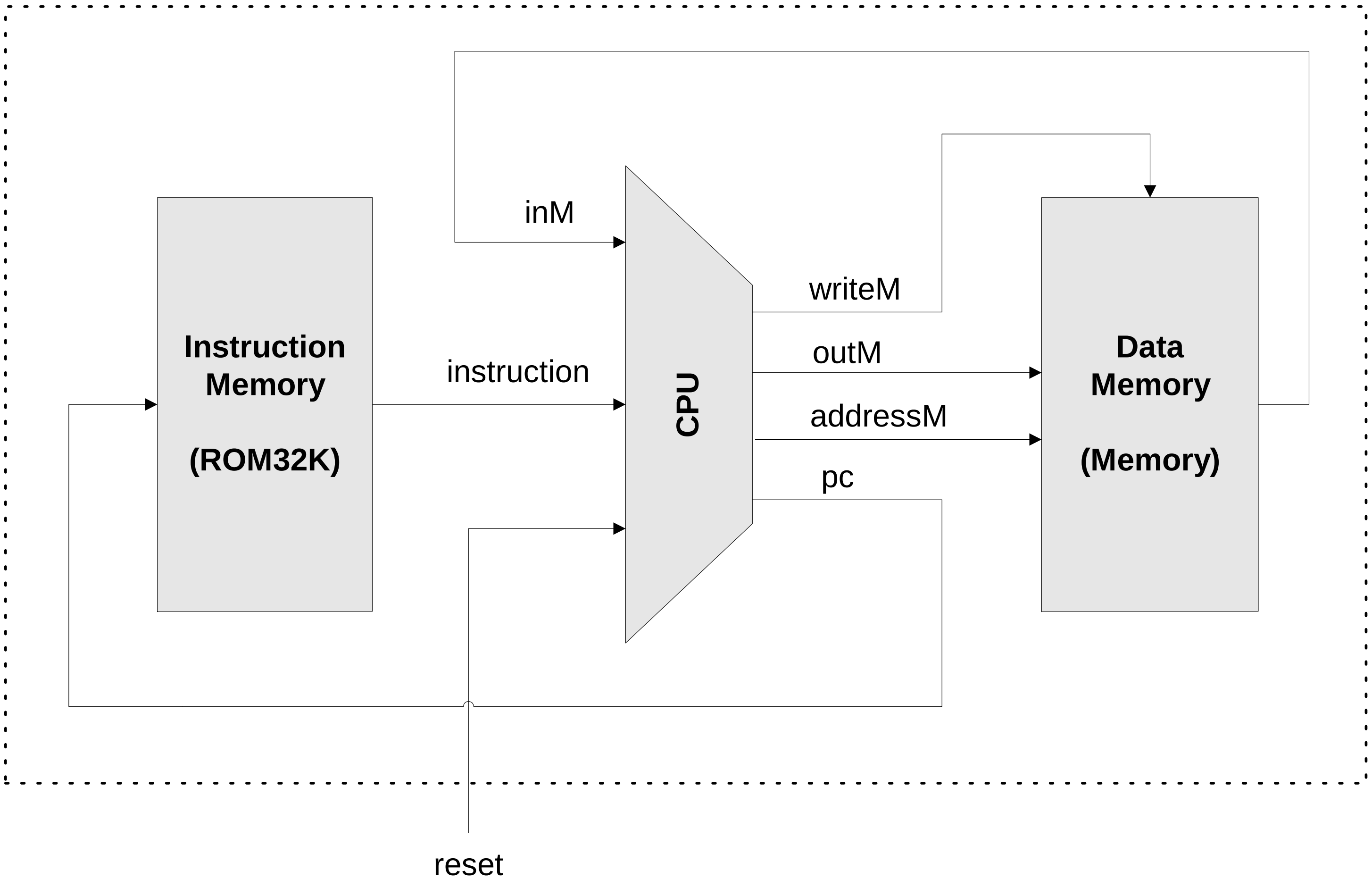
The Hack computer

A 16-bit machine consisting of the following elements:



The Hack computer

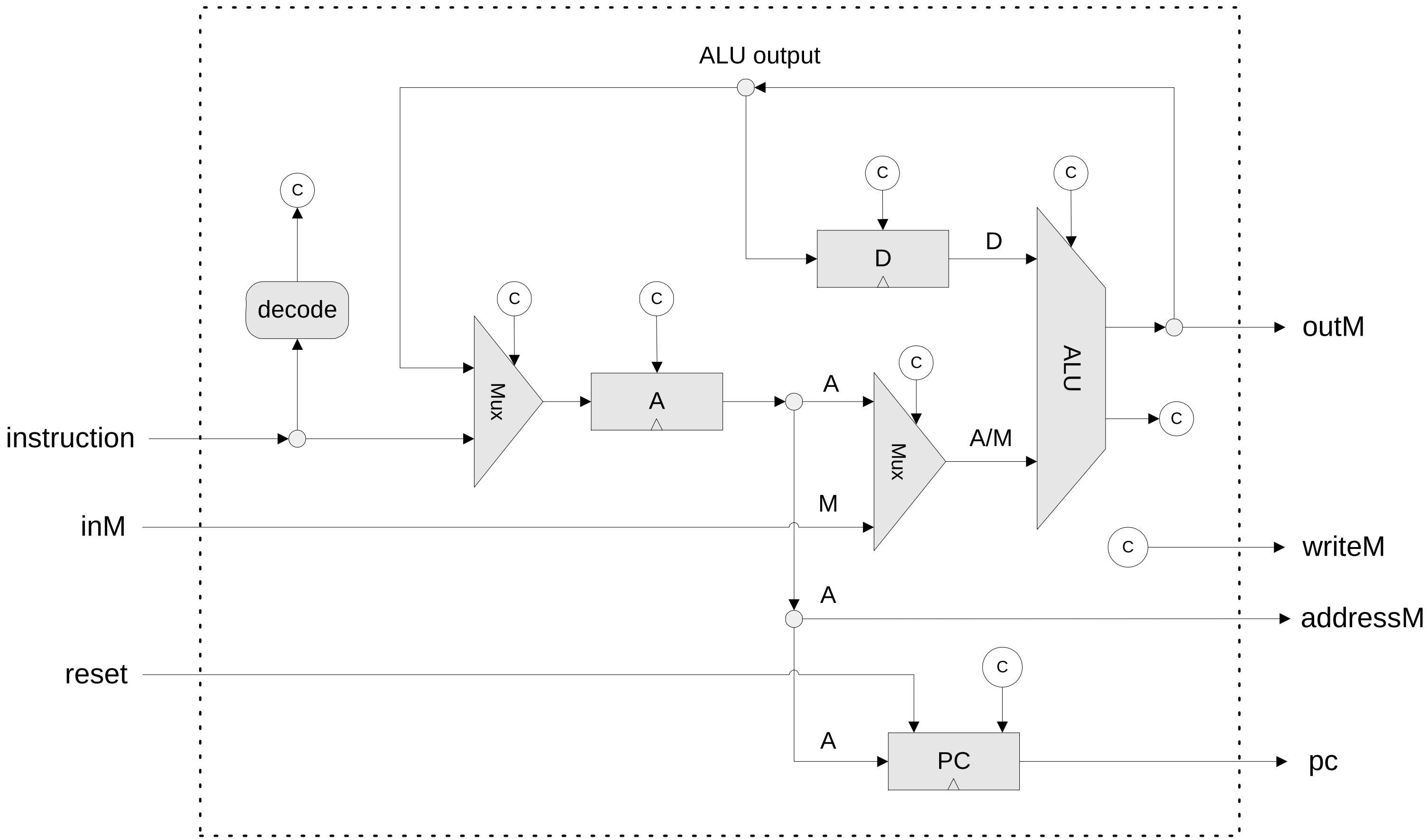
A 16-bit machine consisting of the following elements:



Both memory chips are 16-bit wide and have 15-bit address space.

The Hack computer

A 16-bit machine consisting of the following elements:



The A-instruction

```
@value        // A ← value
```

Where *value* is either a number or a symbol referring to some number.

Why A-instruction? It is impossible to pack both addr and instr into 16 bits.

<u>Used for:</u>

- Entering a constant value
  (**A** = **value**)

<u>Coding example:</u>

```
@17     // A = 17
D = A   // D = 17
```

- Selecting a **RAM** location
  (**register** = **RAM[A]**)

```
@17     // A = 17
D = M   // D = RAM[17]
```

- Selecting a **ROM** location
  (**PC** = **A**)

```
@17     // A = 17
JMP     // fetch the instruction
        // stored in ROM[17]
```

The C-instruction (first approximation)

dest = *x* + *y*

dest = *x* - *y*

dest = *x*

dest = 0

dest = 1

dest = -1

x = {A, D, M}

y = {A, D, M, 1}

dest = {A, D, M, MD, AM, AD, AMD, null}

Exercise: Implement the following tasks using Hack commands:

- Set **D** to **A-1**
- Set both **A** and **D** to **A + 1**
- Set **D** to **19**
- Set both **A** and **D** to **A + D**
- Set **RAM[5034]** to **D - 1**
- Set **RAM[53]** to **171**
- Add 1 to **RAM[7]**, and store the result in **D**.

The C-instruction (first approximation)

Exercise: Implement the following tasks using Hack commands:

1. Set **D** to **A-1**
2. Set both **A** and **D** to **A + 1**
3. Set **D** to **19**
4. Set both **A** and **D** to **A + D**
5. Set **RAM[5034]** to **D - 1**
6. Set **RAM[53]** to **171**
7. Add **1** to **RAM[7]**, and store the result in **D**.

1. D = A-1
2. AD=A+1
3. @19
   D=A
4. AD=A+D
5. @5034
   M=D-1
6. @171
   D=A
   @53
   M=D
7. @7
   D=M+1

The C-instruction (first approximation)

dest = *x* + *y*

dest = *x* - *y*

dest = *x*

dest = 0

dest = 1

dest = -1

x = {A, D, M}

y = {A, D, M, 1}

dest = {A, D, M, MD, AM, AD, AMD, null}

Symbol table:

j	3012
sum	4500
q	3812
arr	20561

(All symbols and values are arbitrary examples)

Exercise: Implement the following tasks using Hack commands:

- `sum = 0`
- `j = j + 1`
- `q = sum + 12 – j`
- `arr[3] = -1`
- `arr[j] = 0`
- `arr[j] = 17`
- `etc.`

The C-instruction (first approximation)

Exercise: Implement the following tasks using Hack commands:

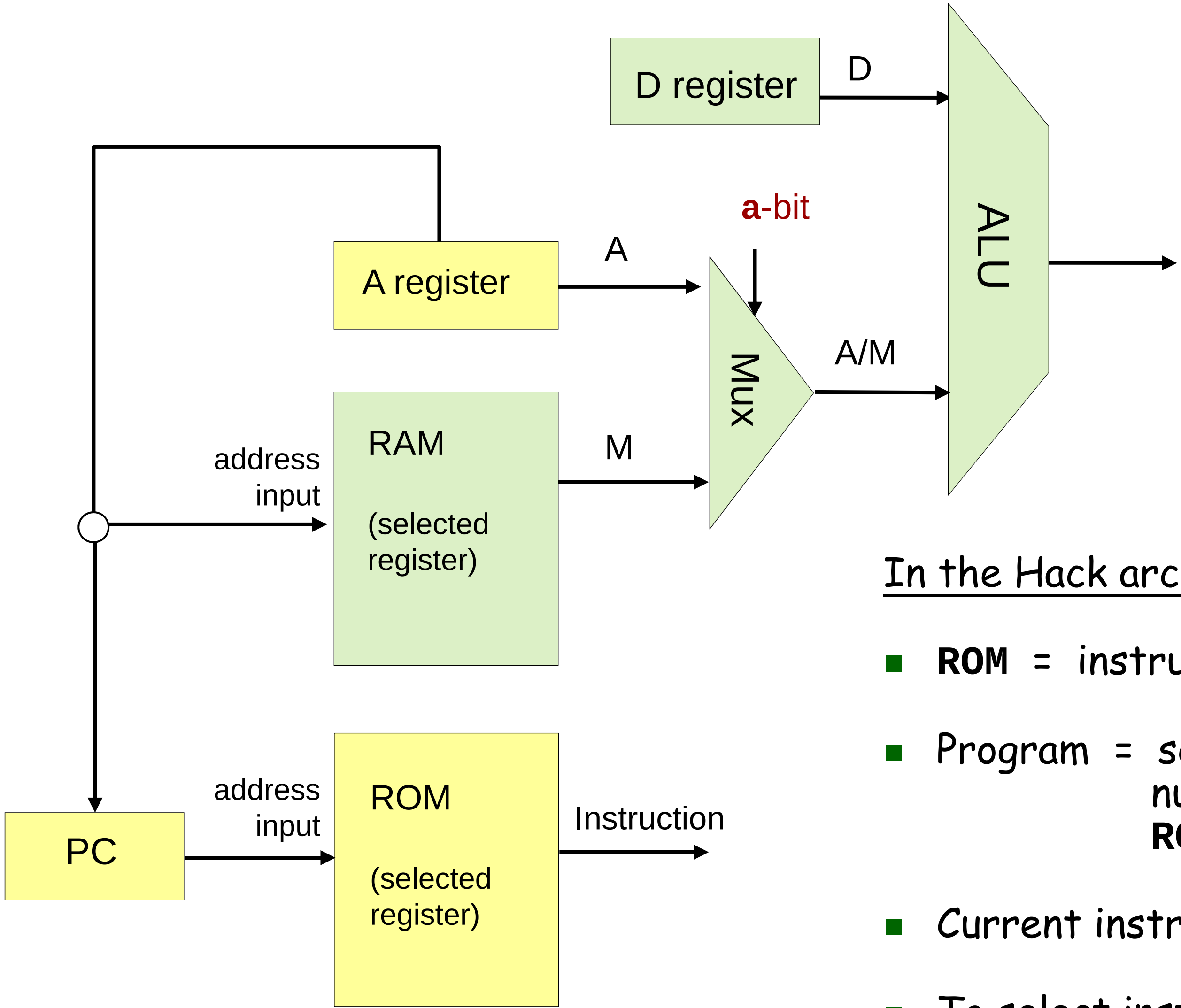
1. `sum = 0`
2. `j = j + 1`
3. `q = sum + 12 – j`
4. `arr[3] = -1`
5. `arr[j] = 0`
6. `arr[j] = 17`
7. `etc.`

```
1. @sum
   M=0
2. @j
   M=M+1
3. @sum
   D=M
   @12
   D=D+A
   @j
   D=D-M
   @q
   M=D
```

```
4. @arr
   D=A
   @3
   A=D+A
   M=-1
5. @j
   D=M
   @arr
   A=A+D
   M=0
```

```
6. @j
   D=M
   @arr
   D=A+D
   @ptr
   M=D
   @17
   D=A
   @ptr
   A=M
   M=D
```

Control (focus on the yellow chips only)



In the Hack architecture:

- **ROM** = instruction memory
- Program = sequence of 16-bit numbers, starting at **ROM[0]**
- Current instruction = **ROM[PC]**
- To select instruction *n* from the **ROM**, we set **A** to *n*, using the instruction **@n**

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

- goto 50
- if D==0 goto 112
- if D<9 goto 507
- if RAM[12] > 0 goto 50
- if sum>0 goto END
- if x[i]<=0 goto NEXT.

Hack commands:

A-command: @value // set A to value

C-command: dest = comp ; jump // dest = and ;jump
// are optional

Where:

comp = 0 , 1 , -1 , D , A , !D , !A , -D , -A , D+1 ,
A+1 , D-1, A-1 , D+A , D-A , A-D , D&A ,
D|A , M , !M , -M ,M+1, M-1 , D+M , D-M ,
M-D , D&M , D|M

dest = M, D, A, MD, AM, AD, AMD, or null

jump = JGT , JEQ , JGE , JLT , JNE , JLE , JMP, or null

In the command dest = comp; jump, the jump materialzes if (comp jump 0) is true. For example, in D=D+1,JLT, we jump if D+1 < 0.

Hack convention:

- True is represented by -1
- False is represented by 0

Symbol table:

sum	2200
x	4000
i	6151
END	50
NEXT	120

(All symbols and values in are arbitrary examples)

Coding examples (practice)

<u>Exercise: Implement the following tasks using Hack commands:</u>

1. goto 50
2. if D==0 goto 112
3. if D<9 goto 507
4. if RAM[12] > 0 goto 50
5. if sum>0 goto END
6. if x[i]<=0 goto NEXT.

```
1. @50
   0; JMP
2. @112
   D; JEQ
3. @9
   D=D-A
   @507
   D; JLT
4. @12
   D=M
   @50
   D; JGT
5. @sum
   D=M
   @END
   D: JGT
6. @i
   D=M
   @x
   A=A+D
   D=M
   @NEXT
   D; JLE
```

IF logic – Hack style

High level:

```
if condition {
   code block 1
} else {
   code block 2
}
code block 3
```

Hack:

```
   D ← condition
   @IF_TRUE
   D;JEQ
   code block 2
   @END
   0;JMP
(IF_TRUE)
   code block 1
(END)
    code block 3
```

Hack convention:

- True is represented by -1
- False is represented by 0

WHILE logic – Hack style

High level:

```
while condition {
   code block 1
}
Code block 2
```

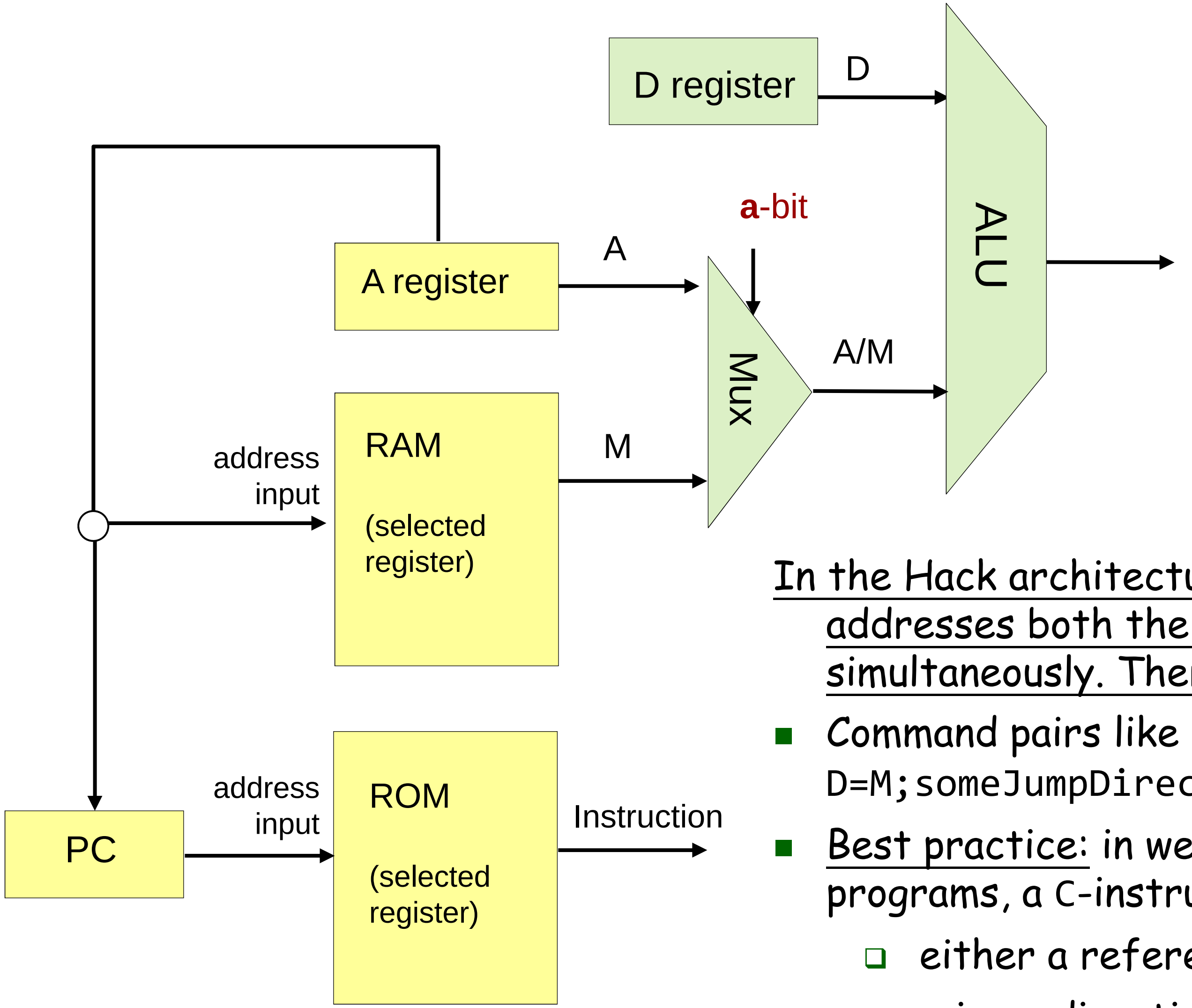
Hack:

```
(LOOP)
     D ← condition
     @END
     D;JNE
     code block 1
     @LOOP
     0;JMP
(END)
     code block 2
```

Hack convention:

- True is represented by -1
- False is represented by 0

Side note (focus on the yellow chip parts only)



In the Hack architecture, the `A` register addresses both the RAM and the ROM, simultaneously. Therefore:

- Command pairs like `@addr` followed by `D=M;someJumpDirective` make no sense
- Best practice: in well-written Hack programs, a `C`-instruction should contain
  - either a reference to `M`, or
  - a jump directive,

  but not both.

Complete program example

C language code:

```
// Adds 1+...+100.
 int i = 1;
 int sum = 0;
 while (i <= 100){
    sum += i;
    i++;
 }
```

Hack assembly code:

```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM location
      M=0     // sum=0
(LOOP)
      @i
      D=M      // D = i
      @100
      D=D-A    // D = i - 100
      @END
      D;JGT    // If (i-100) > 0 goto END
      @i
      D=M      // D = i
      @sum
      M=D+M    // sum += i
      @i
      M=M+1    // i++
      @LOOP
      0;JMP    // Got LOOP
 (END)
      @END
      0;JMP    // Infinite loop
```

Hack assembly convention:

- Variables: lower-case
- Labels: upper-case
- Commands: upper-case

Demo
CPU emulator

Symbols in Hack assembly programs

Symbols created by Hack programmers and code generators:

- Label symbols: Used to label destinations of goto commands. Declared by the pseudo command **(XXX)**. This directive defines the symbol **XXX** to refer to the instruction memory location holding the next command in the program (within the program, **XXX** is called "label")
- Variable symbols: Any user-defined symbol **xxx** appearing in an assembly program that is not defined elsewhere using the **(xxx)** directive is treated as a variable, and is "automatically" assigned a unique RAM address, starting at RAM address 16

  By convention, Hack programmers use lower-case and upper-case letters for variable names and labels, respectively.

Predefined symbols:

- I/O pointers: The symbols **SCREEN** and **KBD** are "automatically" predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the Hack platform's *screen* and *keyboard* memory maps)
- Virtual registers: covered in future lectures.
- VM control registers: covered in future lectures.

Q: Who does all the "automatic" assignments of symbols to RAM addresses?

A: The *assembler*, which is the program that translates symbolic Hack programs into binary Hack program. As part of the translation process, the symbols are resolved to RAM addresses. (more about this in future lectures)

```
// Typical symbolic
// Hack code, meaning
// not important
   @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @addr
   M=D
(LOOP)
   @addr
   A=M
   M=-1
   @addr
   D=M
   @32
   D=D+A
   @addr
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP
```

Perspective

- Hack is a simple machine language
- User friendly syntax: **D=D+A** instead of **ADD D,D,A**
- Hack is a "½-address machine": any operation that needs to operate on the RAM must be specified using two commands: an **A**-command to address the RAM, and a subsequent **C**-command to operate on it
- A Macro-language can be easily developed
  - D=D+M[XXX] => @XXX followed by D=D+M
  - GOTO YYY => @YYY followed by 0; JMP
- A Hack assembler is needed and will be discusses and developed later in the course.