School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 7

adelaide.edu.au

seek LIGHT

Review – Languages

- We have to translate higher level languages to the machine level.
- Machine languages manipulate memory using the registers and processor.
- There are lots of machine language commands
  - The more we add, the more complicated our chip becomes.
  - RISC vs CISC

What we're doing now

- This week we're going to talk about:
  - Programming languages
    - Quick overview
    - History
    - High level languages can be very ear identical) across different hardware platforms.
  - Machine languages
    - Manipulating memory using a processor and a set of registers
    - Will be different across different hardware platforms
  - Machine language in HACK

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 A-instruction

```
@value          // A ← value
```

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

Used for:

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

```
@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]
```

Coding example:

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, A, AM, AD, AMD, null}

Exercise: In small groups implement the following tasks using Hack :

- 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)

```
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, A, AM, AD, AMD, null}

Symbol table:

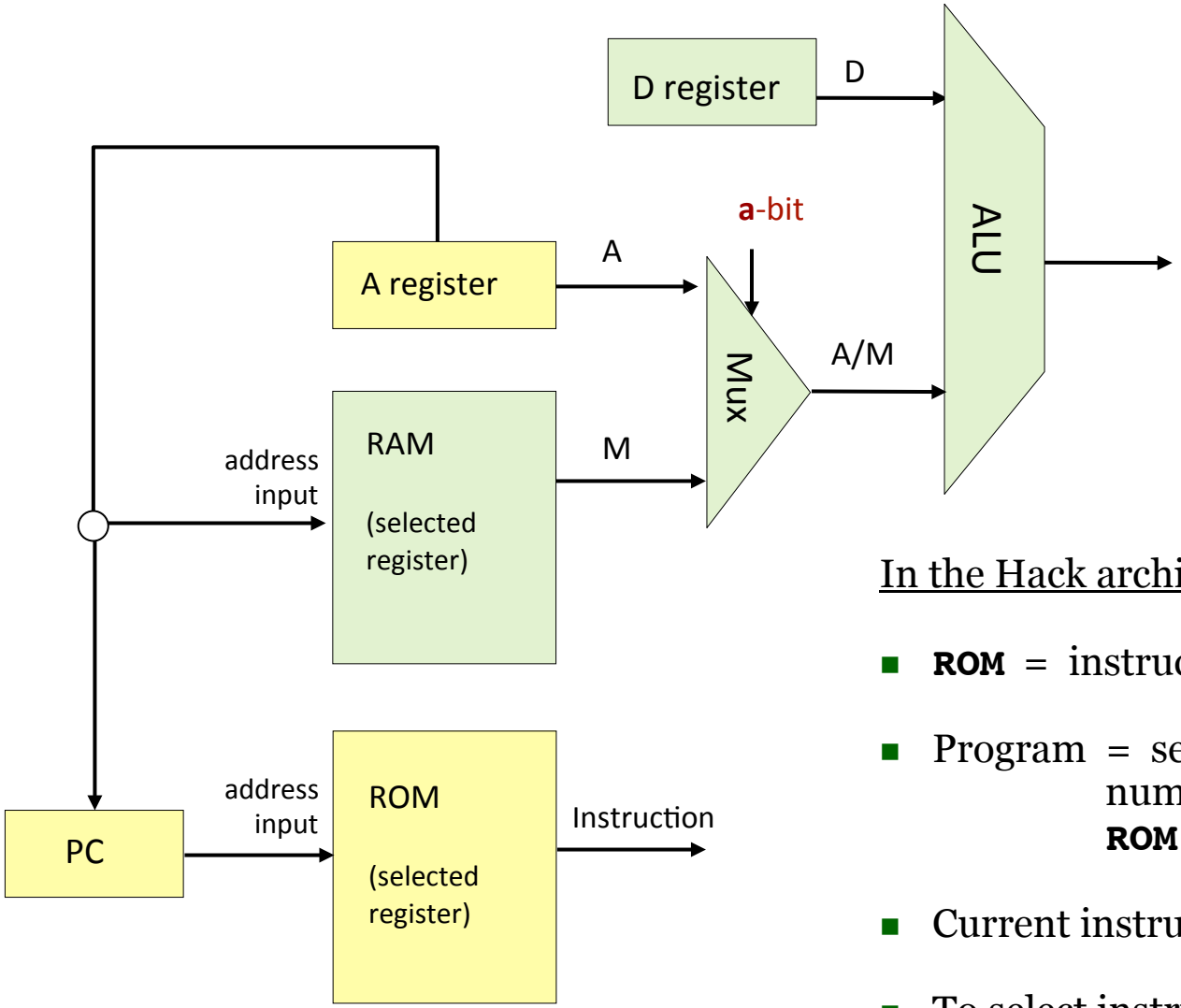
j	3012
sum	4500
q	3812
arr	20561

(All symbols and values are arbitrary examples)

Exercise: In small groups, implement the following tasks using Hack:

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

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

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 , MD , A , 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)

IF logic – Hack style

High level:

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

Hack:

```
    D ← not 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
```

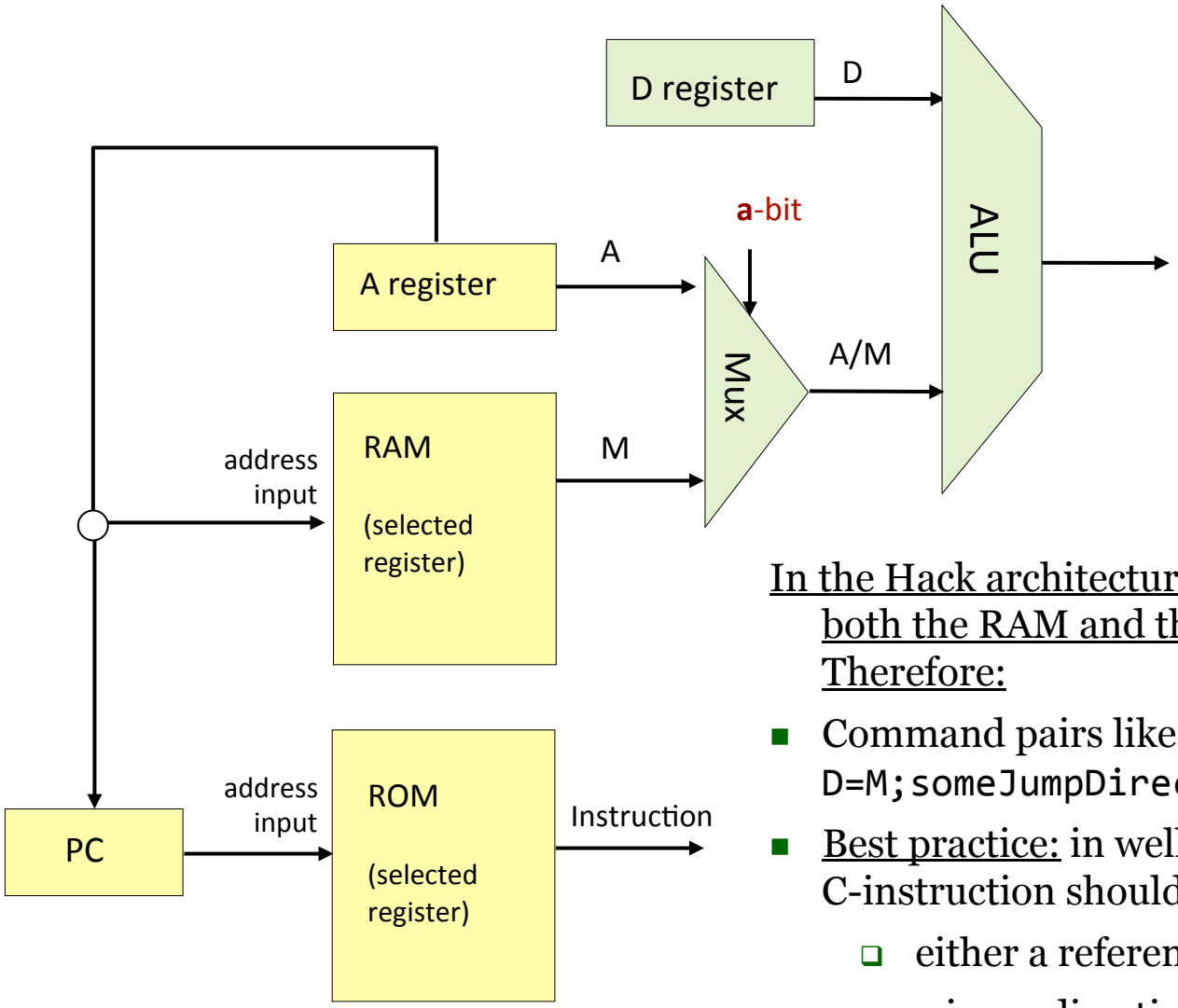
Hack:

```
(LOOP)
      D ← not condition)
      @END
      D;JEQ
      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.
 into i = 1;
 into 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

Complete program example

C language code:

```
// Adds 1+...+100.
 into i = 1;
 into 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

What's this?

Symbols in Hack assembly prog

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 *very simple* machine language
- User friendly syntax: **`D=D+A`** instead of **`ADD D,D,A`**
  - A return to mathematical syntax!
- 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
- A Hack assembler is needed, it also turns symbols and labels into numbers, and will be discussed and developed later in the course.

Next week

- There is a lecture on Monday!
- There is a tutorial next week.
- You should read “Chapter 5” from the forums and start looking at Assignment 2.
  - Remember there is a milestone due!
- Any questions? Ask on the forum or right now!