



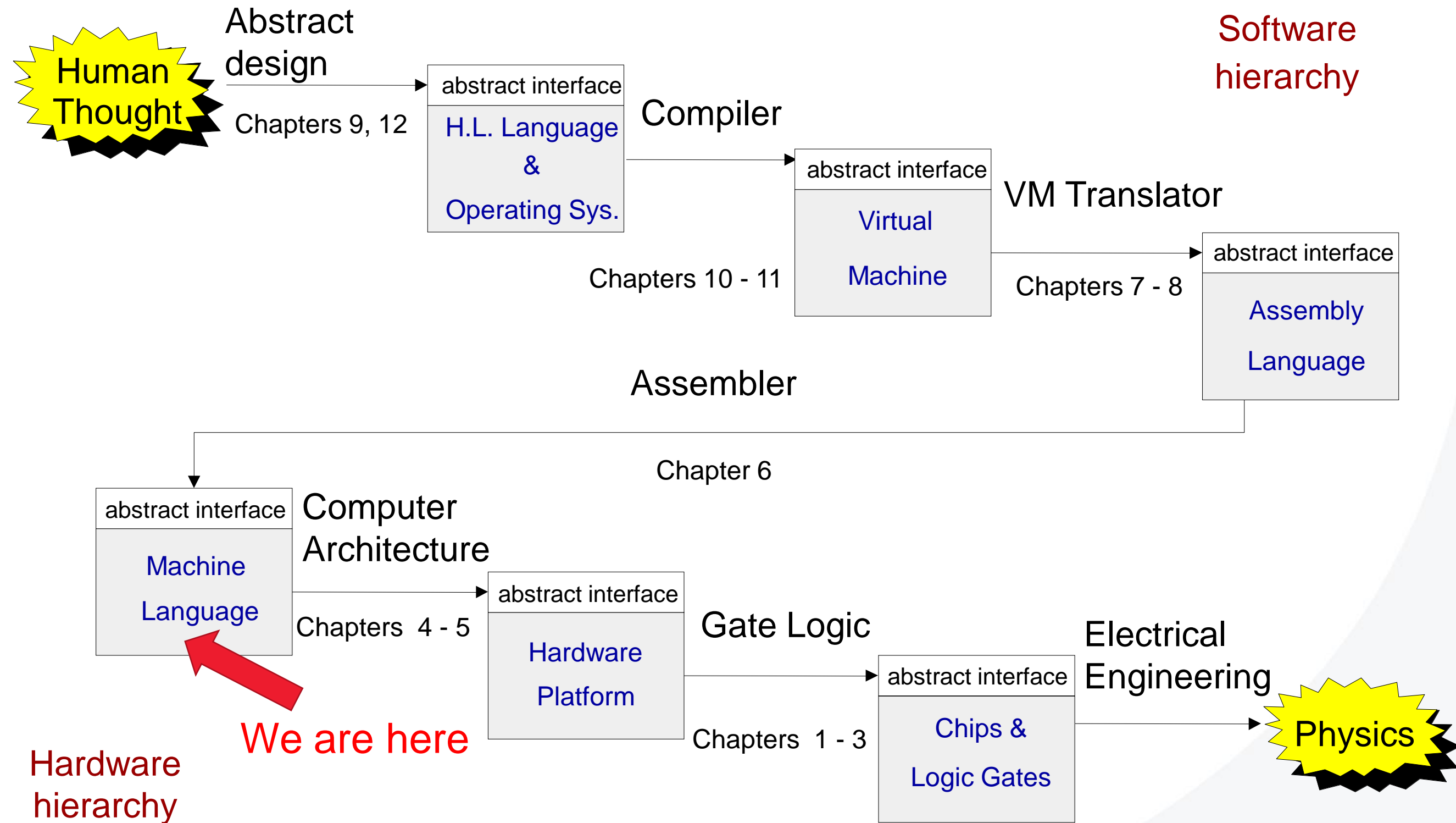
Computer Systems

Lecture 07: Machine Language Review and Exercise



THE UNIVERSITY
of ADELAIDE

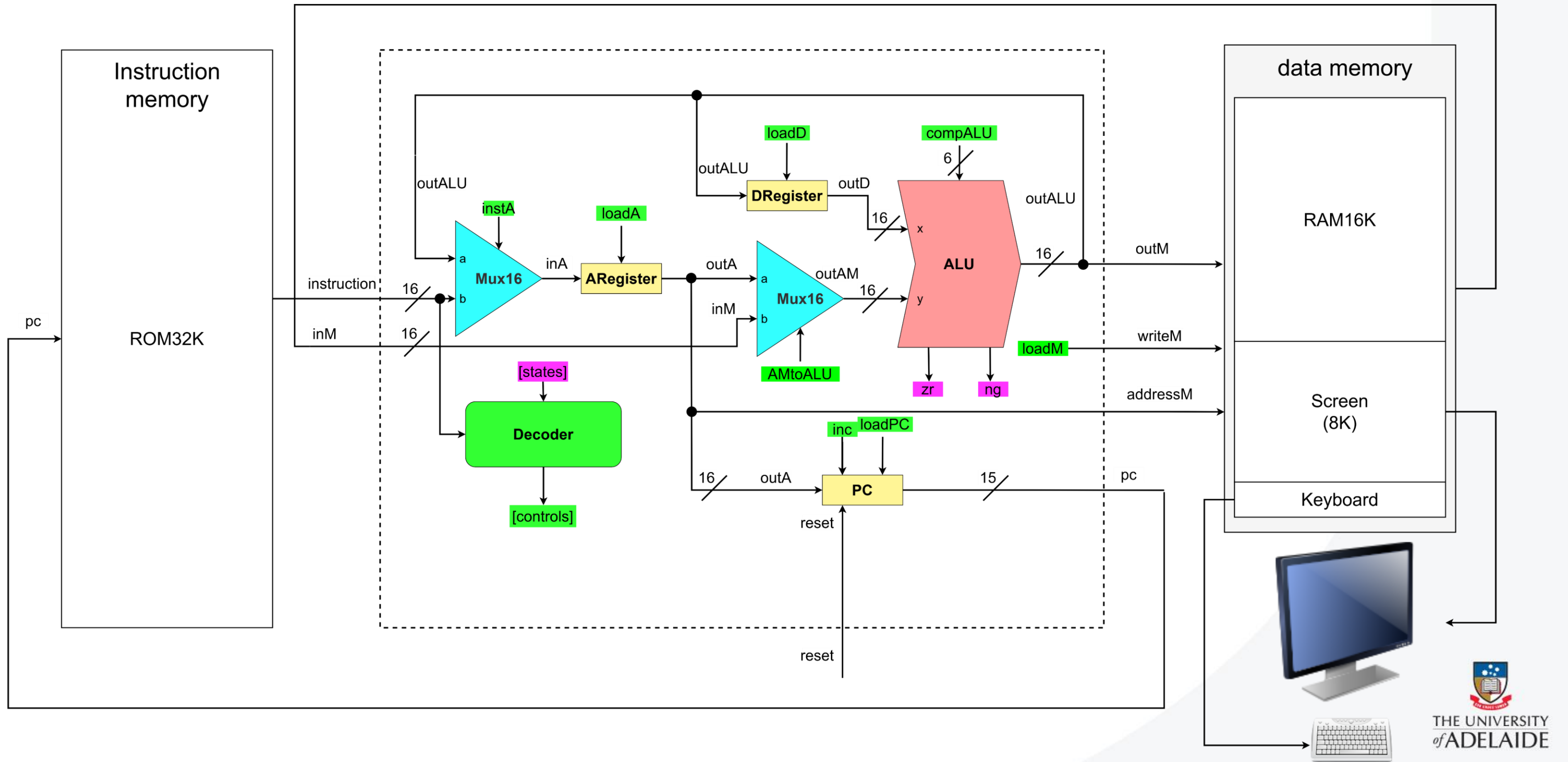
Our Journey



(Abstraction–implementation paradigm)



Review: The Hack Computer



System Architecture

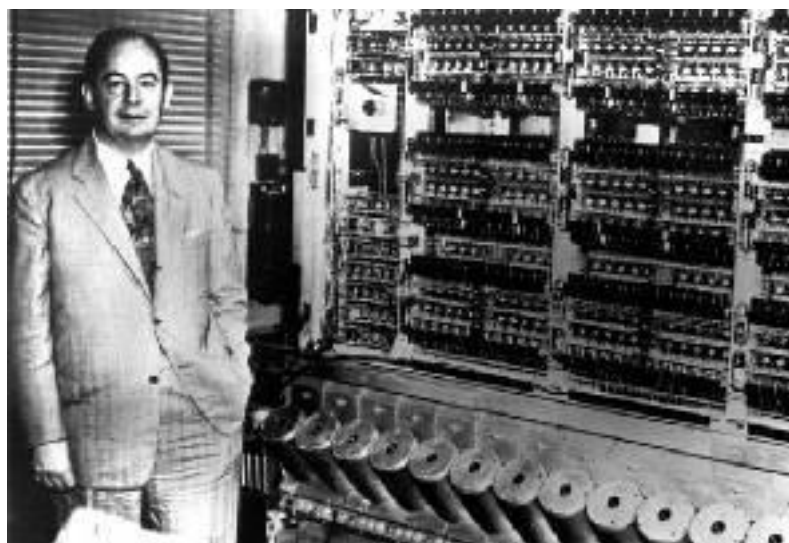
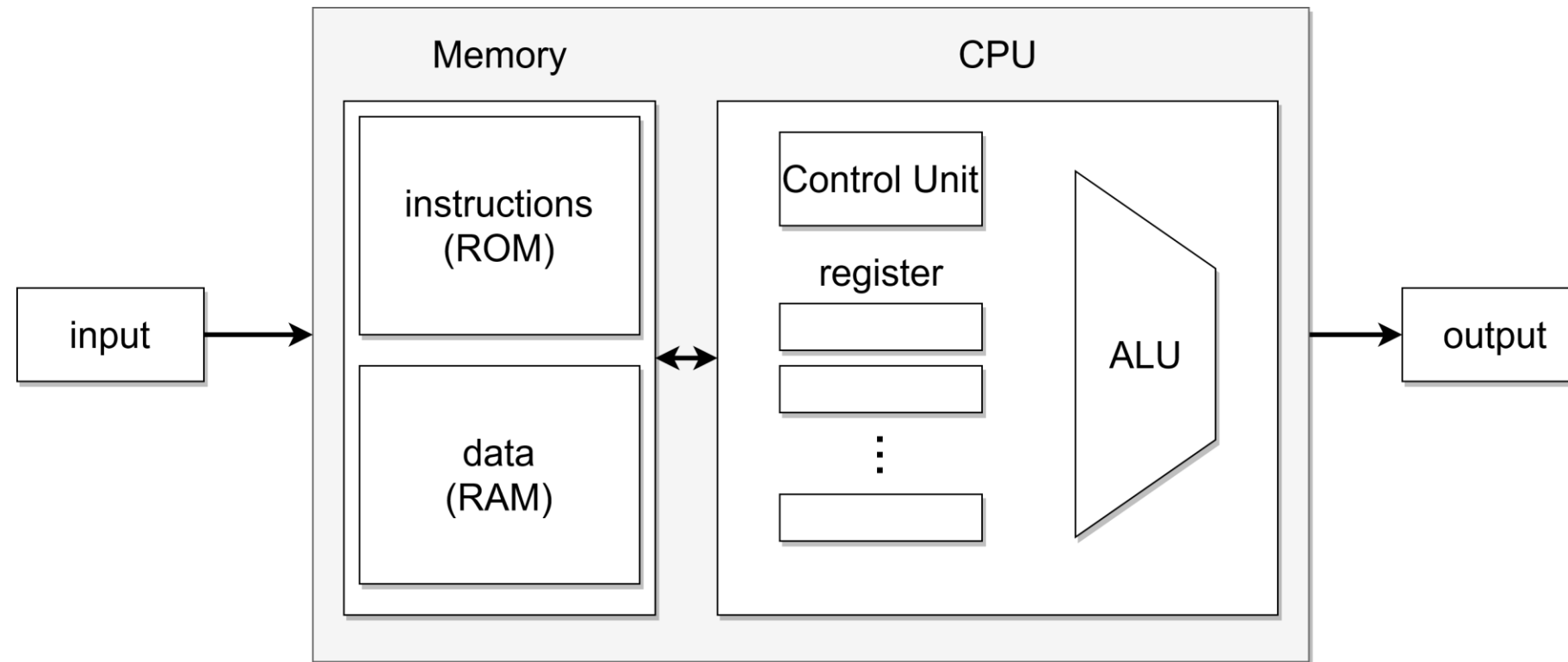


Early Computers

Just good to know

- **We have been using computers for a very very long time but not in the form we are using today.**
 - Early machines were used to predict astronomical positions for calendars and astrology.
- **The Antikythera mechanism, possibly 100 to 200 BC**
- its sophistication suggests such devices existed much earlier.
- **The first programmable analogue computer?**
 - Al-Jazari built a programmable water powered clock c. 1206.
- **Slide rules**
 - invented around 1620 with the discovery of logarithms.
 - heavily used by nearly everyone until the 1970s replaced by electronic calculators.

Von Neumann machine (circa 1940)



John Von Neumann (and others) ... made it possible

Stored
program
concept!



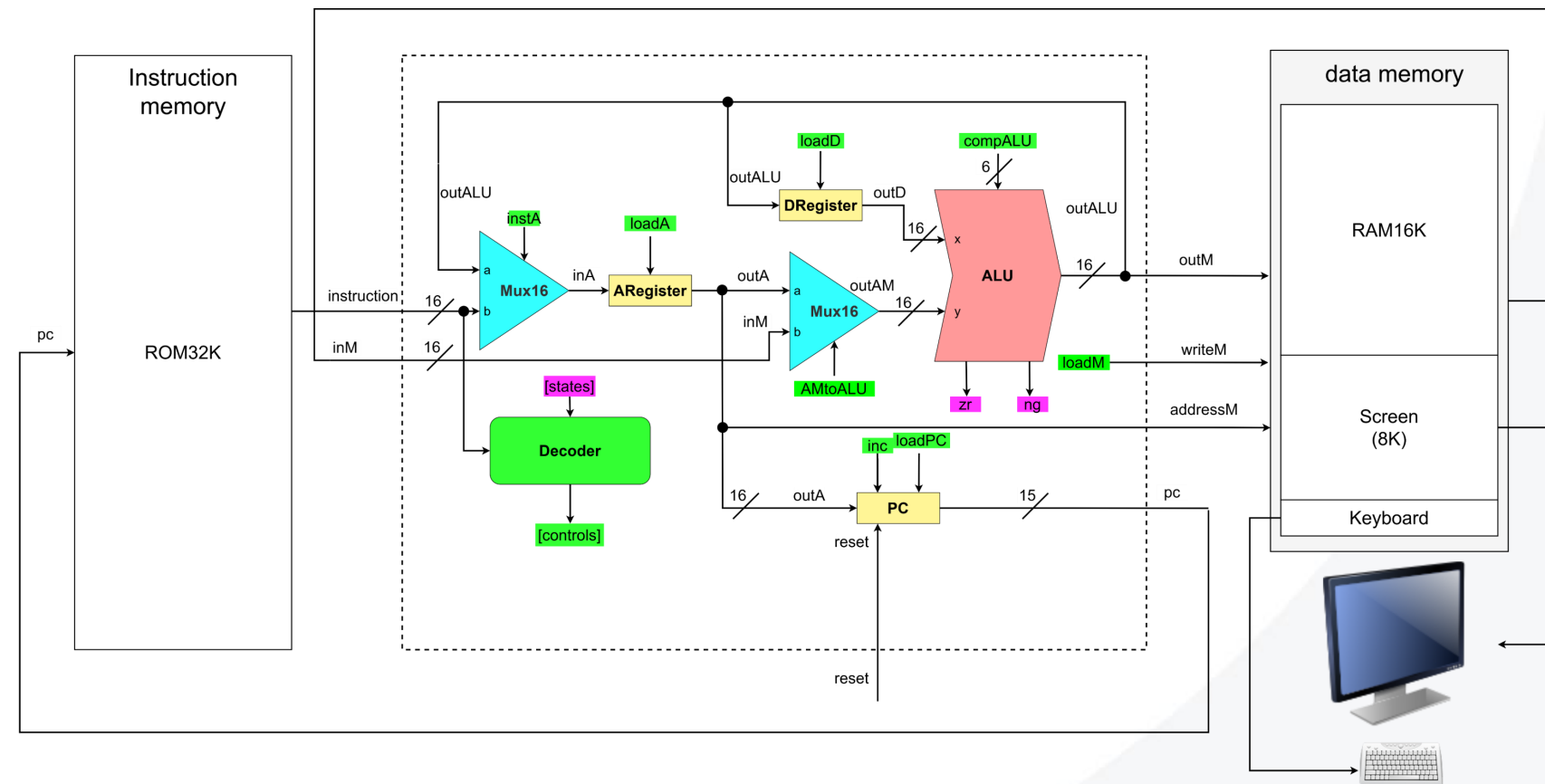
Andy Grove (and others) ... made it small and fast.

The Hack computer

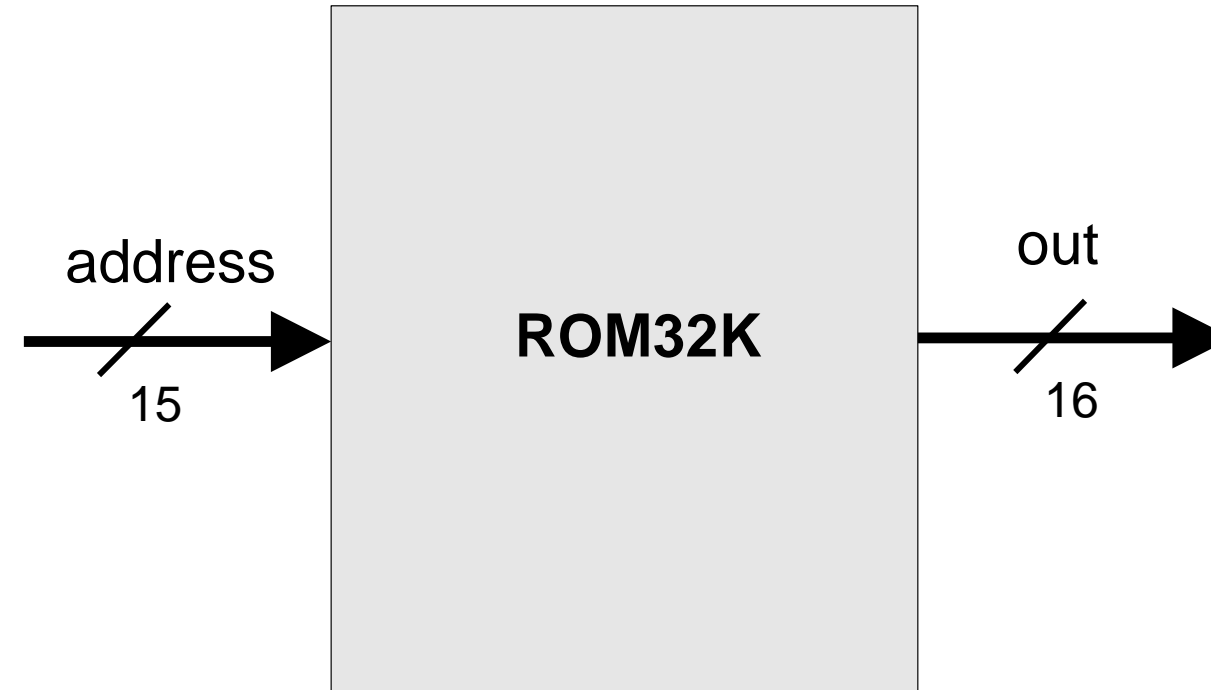
- A 16-bit Von Neumann platform
- The *instruction memory* and the *data memory* are physically separate
- Screen: 512 rows by 256 columns, black and white
- Keyboard: standard ANSI 104 keyboard
- Designed to execute programs written in the Hack machine language
- Can be easily built from the chip-set that we built so far in the course

Main parts of the Hack computer:

- Instruction memory (ROM)
 - Memory (RAM):
 - Data memory
 - Screen (memory map)
 - Keyboard (memory map)
 - CPU
 - Computer (**the logic that holds everything together**).
-
- The diagram illustrates a computer system architecture. A box labeled 'pc' (program counter) has an arrow pointing to a larger box labeled 'ROM32K'. To the right of the 'ROM32K' box is a vertical stack of two boxes: the top one is labeled 'instruction' and the bottom one is labeled 'inM'. A line connects the bottom of the 'ROM32K' box to the bottom of the 'instruction' box, and another line connects the bottom of the 'inM' box back to the 'pc' box, forming a loop.



Instruction memory



Function:

The ROM is pre-loaded with a program written in the Hack machine language

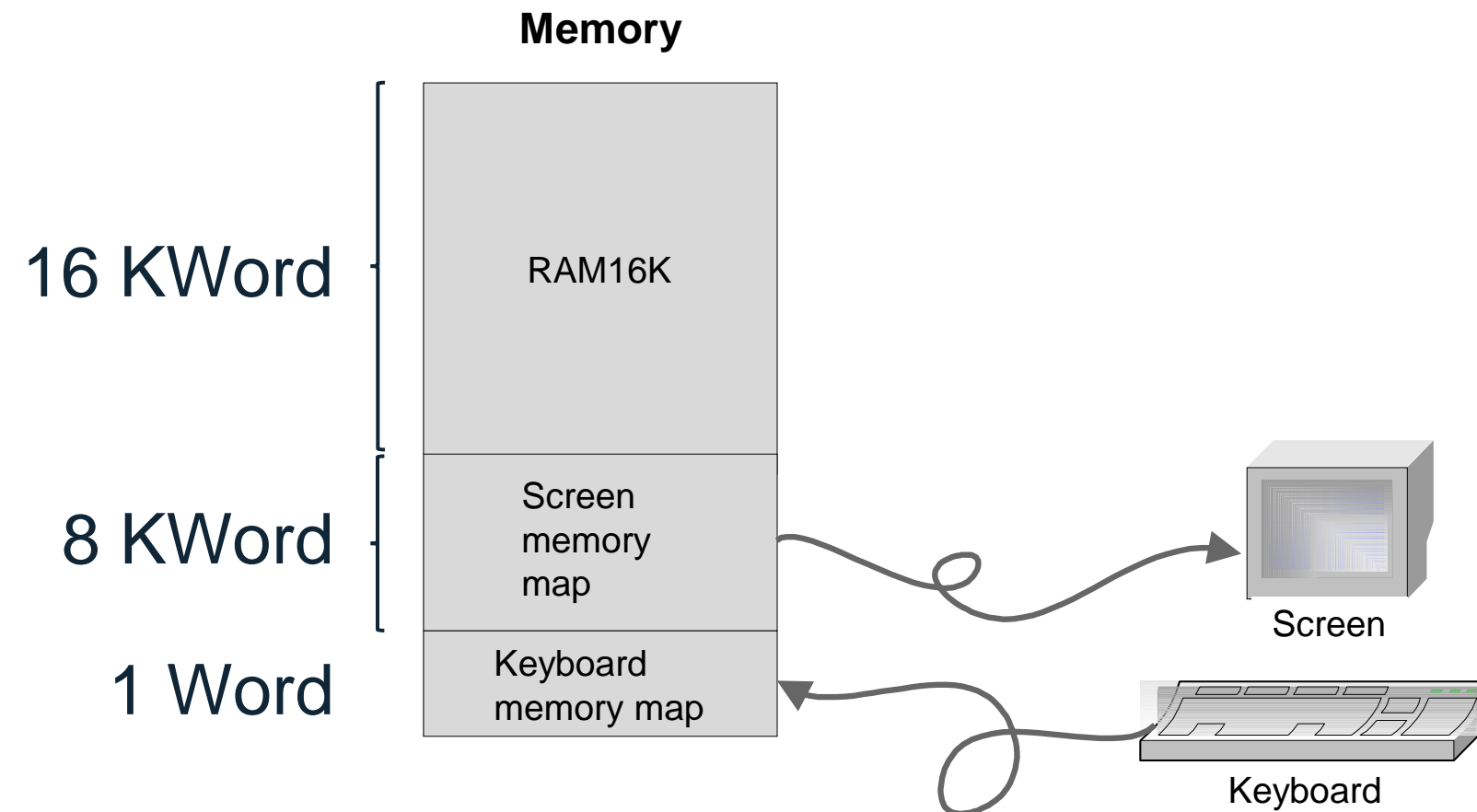
The ROM chip always emits a 16-bit number:

`out = ROM32K[address]`

This number is interpreted as the current instruction.



Memory and Memory Mapped Input and Output (MMIO)



From the program's point of view our RAM is just 32K words of read/write memory

Using the memory:

- To record or recall values (e.g. variables, objects, arrays), use the first 16K words of the memory
- To write to the screen (or read the screen), use the next 8K words of the memory
- To read which key is currently pressed, use the next word of the memory.

Data memory

Low-level (hardware) read/write logic:

To read $\text{RAM}[k]$: set address to k ,
probe out

To write $\text{RAM}[k]=x$: set address to k ,
set in to x ,
set load to 1,
run the clock

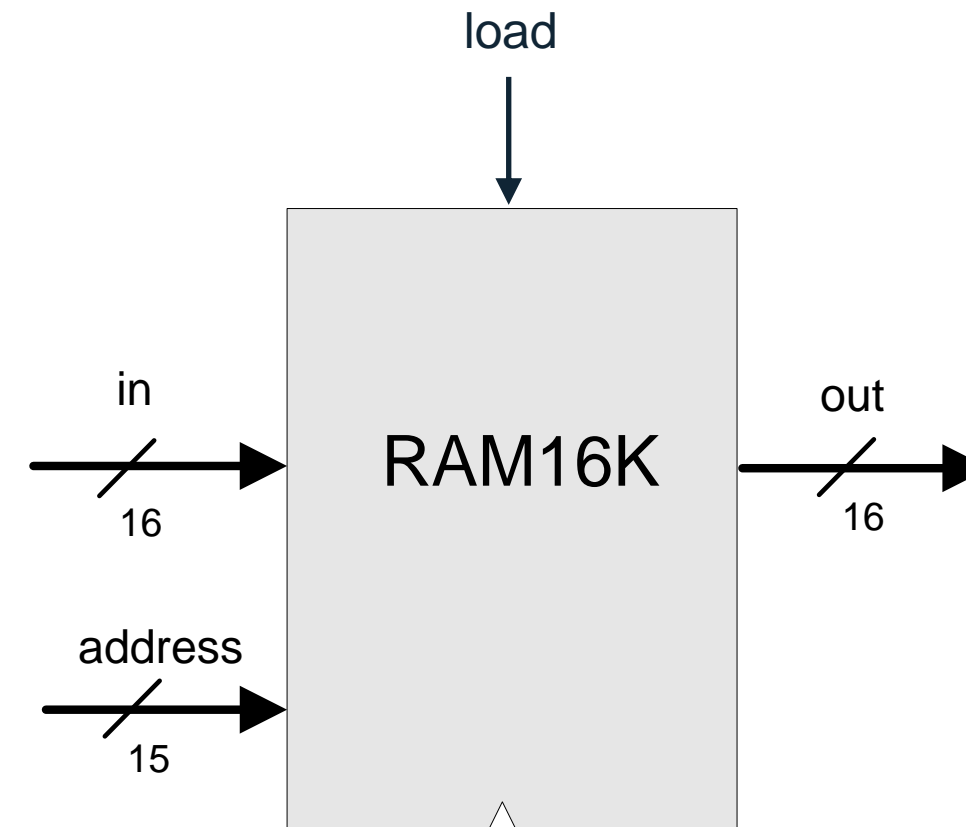
High-level (OS) read/write logic:

To read $\text{RAM}[k]$: use the OS function $\text{out} = \text{peek}(k)$

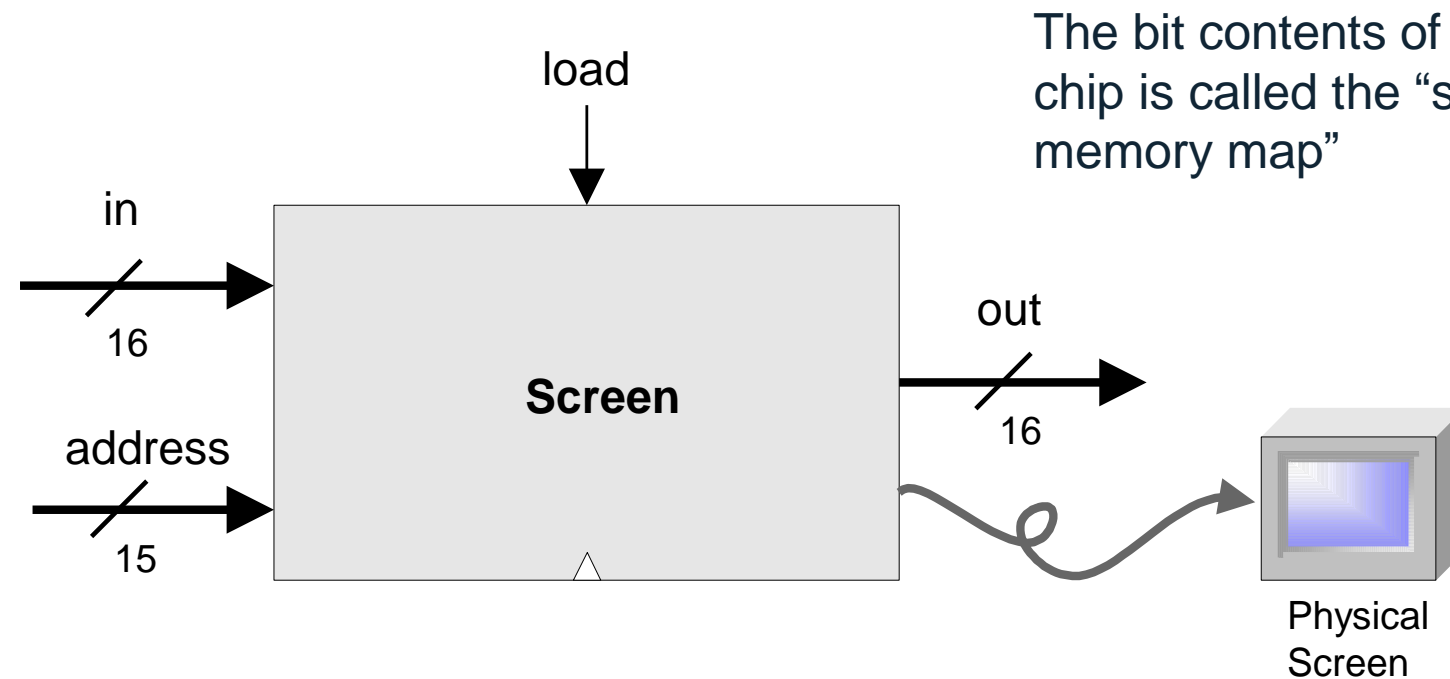
To write $\text{RAM}[k]=x$: use the OS function $\text{poke}(k, x)$

peek and poke are OS commands whose implementation should have the same behavior as the low-level commands

More about peek and poke this later in the course, when we'll write the OS.



Screen



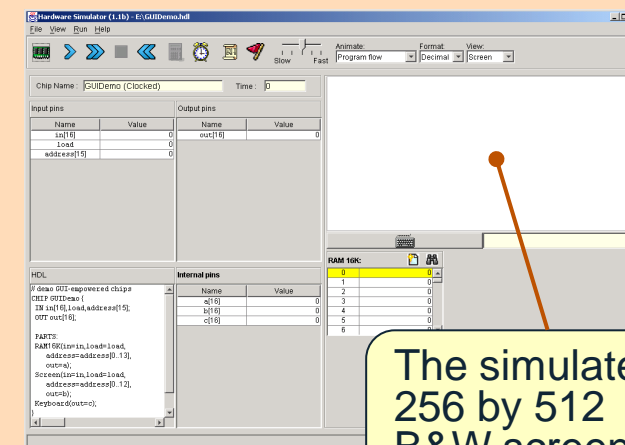
The Screen chip has a basic RAM chip functionality:

- read logic: $\text{out} = \text{Screen}[\text{address}]$
- write logic: if load then $\text{Screen}[\text{address}] = \text{in}$

Side effect:

Continuously refreshes a 256 by 512 black-and-white screen device

Screen in HardwareSimulator:

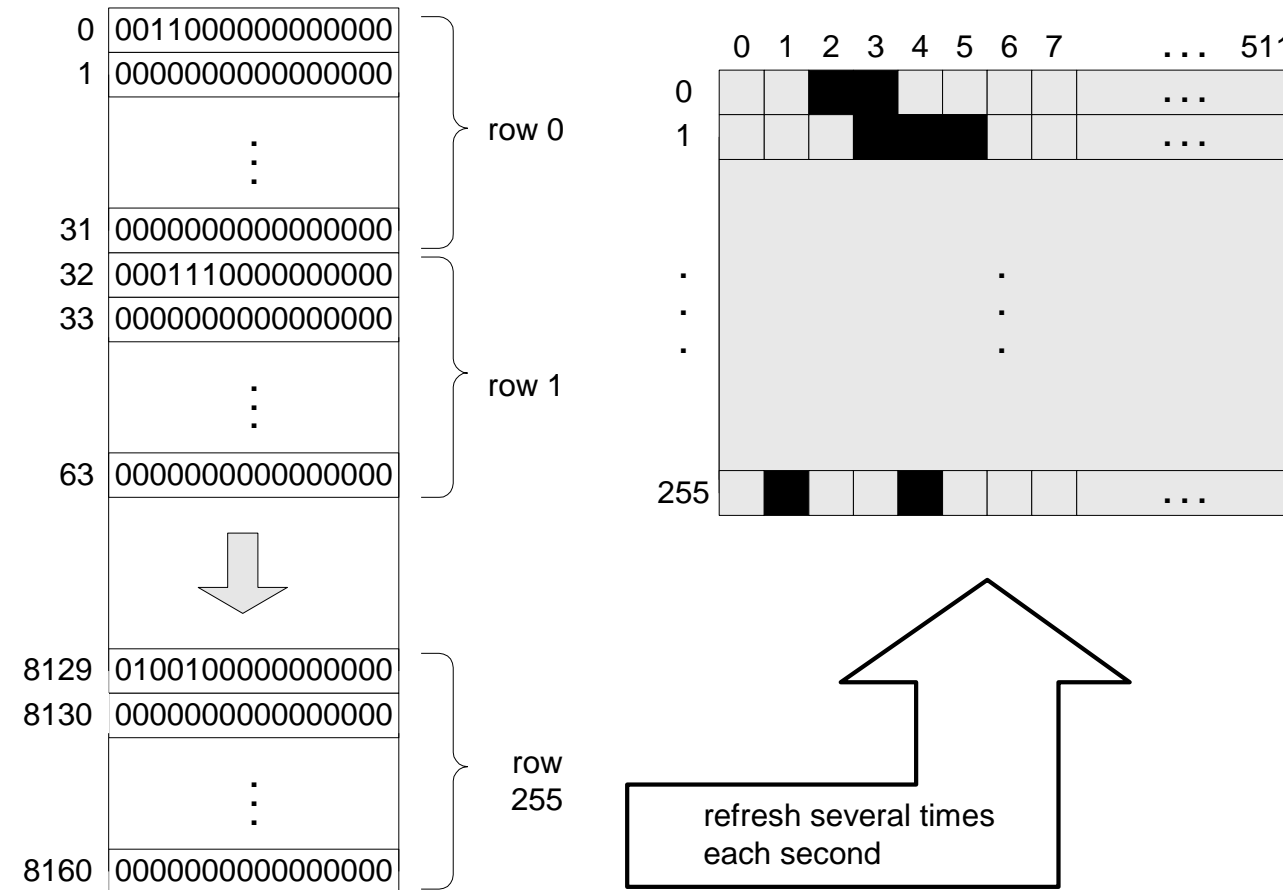
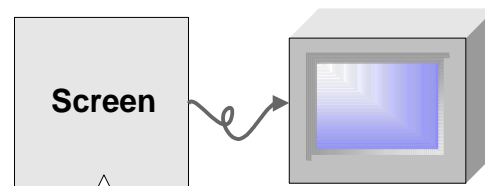


When loaded into the hardware simulator, the built-in Screen.hdl chip opens up a screen window; the simulator then refreshes this window from the screen memory map several times each second.

Screen memory map

0→white
1→black

In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip.

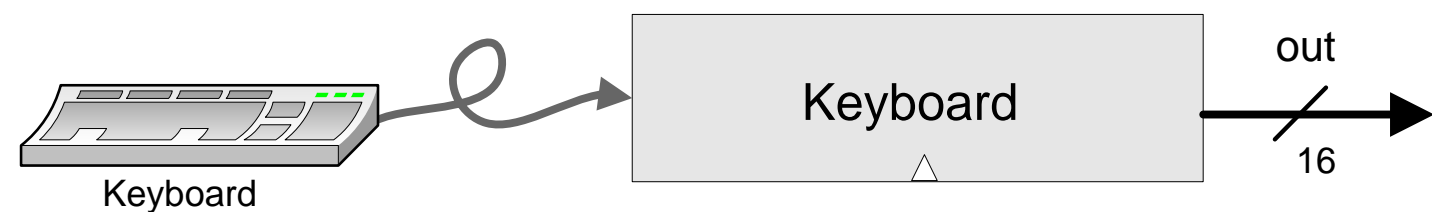


How to set the (row,col) pixel of the screen to black or to white:

- ❑ **Low-level (machine language):** Set the $col\%16$ bit of the word found at $Screen[row*32+col/16]$ to 1 or to 0
($col/16$ is integer division)
- ❑ **High-level:** Use the OS command `drawPixel(row,col)`
(effects the same operation, discussed later in the course, when we'll write the OS).



Keyboard



Keyboard chip: a single 16-bit register

Input: physical keyboard keystroke

Output: 16-bit ASCII-like code of the currently pressed

<u>Special keys:</u>				
Key pressed	Keyboard output		Key pressed	Keyboard output
newline	128		end	135
backspace	129		page up	136
left arrow	130		page down	137
up arrow	131		insert	138
right arrow	132		delete	139
down arrow	133		esc	140
home	134		f1-f12	141-152

How to read the keyboard:

- Low-level (hardware): probe the contents of the Keyboard chip
- High-level: use the OS function keyPressed() (effects the same operation)

Simulated keyboard:

The screenshot shows a hardware simulator window titled 'Hardware Simulator (1.1.1) - E1-GUIDemo.hdl'. It displays various components like 'Input pins', 'Output pins', and 'Internal pins'. A callout box with an orange arrow points to a button labeled 'The simulated keyboard enabler button'.

The keyboard is implemented as a built-in **Keyboard.hdl** chip. When this java chip is loaded into the simulator, it connects to the regular keyboard and pipes the scan-code of the currently pressed key to the keyboard memory map.

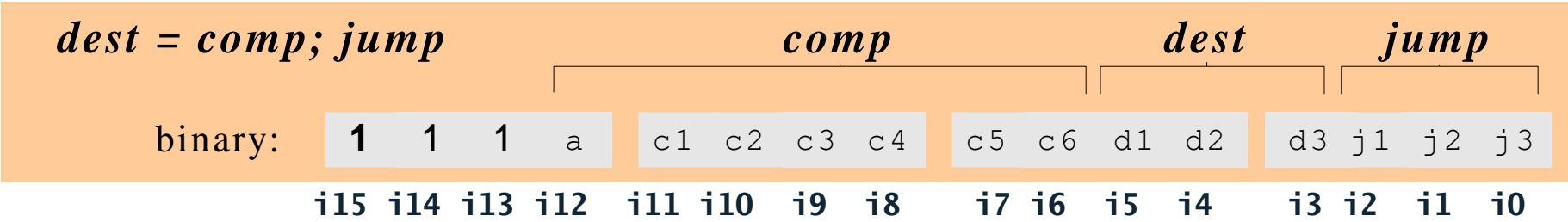
Hack CPU instructions

- Two instructions A and C
 - A-instruction load the A register with a 15-bit value.
 - The op-code (the MSB or i15) is 0.
 - The rest 15 bits (i14 to i0) are 15-bit unsigned binary.
 - For example: 00000000000000101, stores the binary representation of 5 into A register
 - A-instruction is the only way to enter constant into hack computer.
 - A-instruction sets address for the subsequent C-instruction which accesses RAM.
 - In hack assembly RAM[A] is referred as M.
 - C-instruction coordinates 3 operations: what to compute (*comp*), where to store the result (*dest*) and what to do next (*jump*).
 - The op-code (the MSB or i15) is 1.
 - See C-instruction in detail in the following slides.



The C-instruction

In C-instruction, we have *dest* field, why we still have A-instruction?

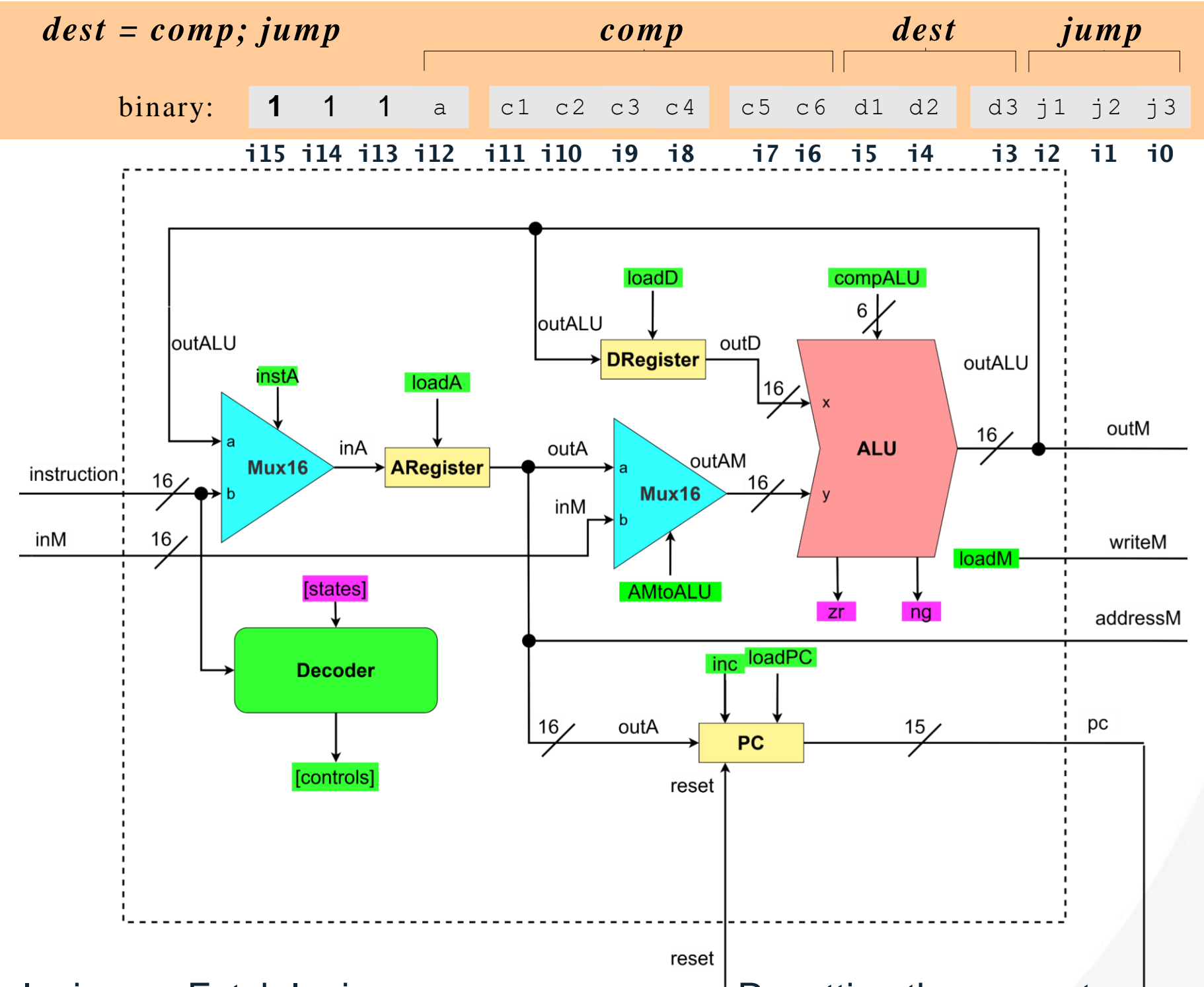


(when a=0) <i>comp</i>	c1	c2	c3	c4	c5	c6	(when a=1) <i>comp</i>	d1	d2	d3	Mnemonic	Destination (where to store the computed value)
0	1	0	1	0	1	0		0	0	0	null	The value is not stored anywhere
1	1	1	1	1	1	1		0	0	1	M	Memory[A] (memory register addressed by A)
-1	1	1	1	0	1	0		0	1	0	D	D register
D	0	0	1	1	0	0		0	1	1	MD	Memory[A] and D register
A	1	1	0	0	0	0	M	1	0	0	A	A register
!D	0	0	1	1	0	1		1	0	1	AM	A register and Memory[A]
!A	1	1	0	0	0	1	!M	1	1	0	AD	A register and D register
-D	0	0	1	1	1	1		1	1	1	AMD	A register, Memory[A], and D register
-A	1	1	0	0	1	1	-M					
D+1	0	1	1	1	1	1			j1	j2	j3	
A+1	1	1	0	1	1	1	M+1	(out < 0)	(out = 0)	(out > 0)	Mnemonic	Effect
D-1	0	0	1	1	1	0		0	0	0	null	No jump
A-1	1	1	0	0	1	0	M-1	0	0	1	JGT	If out > 0 jump
D+A	0	0	0	0	1	0	D+M	0	1	0	JEQ	If out = 0 jump
D-A	0	1	0	0	1	1	D-M	0	1	1	JGE	If out ≥ 0 jump
A-D	0	0	0	1	1	1	M-D	1	0	0	JLT	If out < 0 jump
D&A	0	0	0	0	0	0	D&M	1	0	1	JNE	If out ≠ 0 jump
D A	0	1	0	1	0	1	D M	1	1	0	JLE	If out ≤ 0 jump
								1	1	1	JMP	Jump

CPU implementation

Chip diagram:

- Includes most of the CPU's execution logic
- The CPU's control logic is hinted: each signal colored in green represents one or more control bits, decoded from the instruction
- The 'Decoder' block does not represent a chip, but rather indicates that the instruction bits are decoded somehow.



Cycle:

- Execute
- Fetch

Execute logic:

- Decode
- Execute

Fetch logic:

If there should be a jump,
set PC to A
else set PC to PC+1

Resetting the computer:

Set reset to 1,
then set it to 0.

Observations

We can use individual bits from the C-instruction to control:

The ALU (multiple wires).

The Choice of Source (A,D,M).

The Choice of Destination (A,D,M).

Some Input signals to the PC to let it know if it should be updated with the value of A.

- Note that this decision also depends on some output status bits from the ALU.



Assembly Language



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)

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

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

Coding example:

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

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

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



The C-instruction

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}

A/M/D/1 are
interchangeable in
x or *y*

Must follow the order
of **A/M/D** in *dest*

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

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:

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

Symbol table:

<code>j</code>	3012
<code>sum</code>	4500
<code>q</code>	3812
<code>arr</code>	20561

(All symbols and values are arbitrary examples)



THE UNIVERSITY
of ADELAIDE

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 materializes 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 = 11111111 11111111
- False is represented by 0 = 00000000 00000000



WHILE logic – Hack style

High level:

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

Hack:

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

(END)
    code block 2
```

Hack convention:

- True is represented by -1 = 11111111 11111111
- False is represented by 0 = 00000000 00000000



Complete program example

C language code:

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

Hack assembly convention:

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

Standard way to terminate
a program execution in
hack assembly.



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
(LLOOP)
    @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
```



This Week

- Review Chapters 4 & 5 of the Text Book (if you haven't already).
- Assignment 3 due next Friday (26 Aug 2022).
- Practical Exam; details in announcement.
 - The mobile coverage is poor in IW B23 where our practical exam take place, you may not be able to receive SMS to login Myuni.
 - Make sure you have Okta or Google Authenticator MFA Setup.
- Read Chapter 6 of the Text Book before next week.

