ELEC2665 Unit 2

Dr Craig A. Evans

2.1 – Assembly Language I

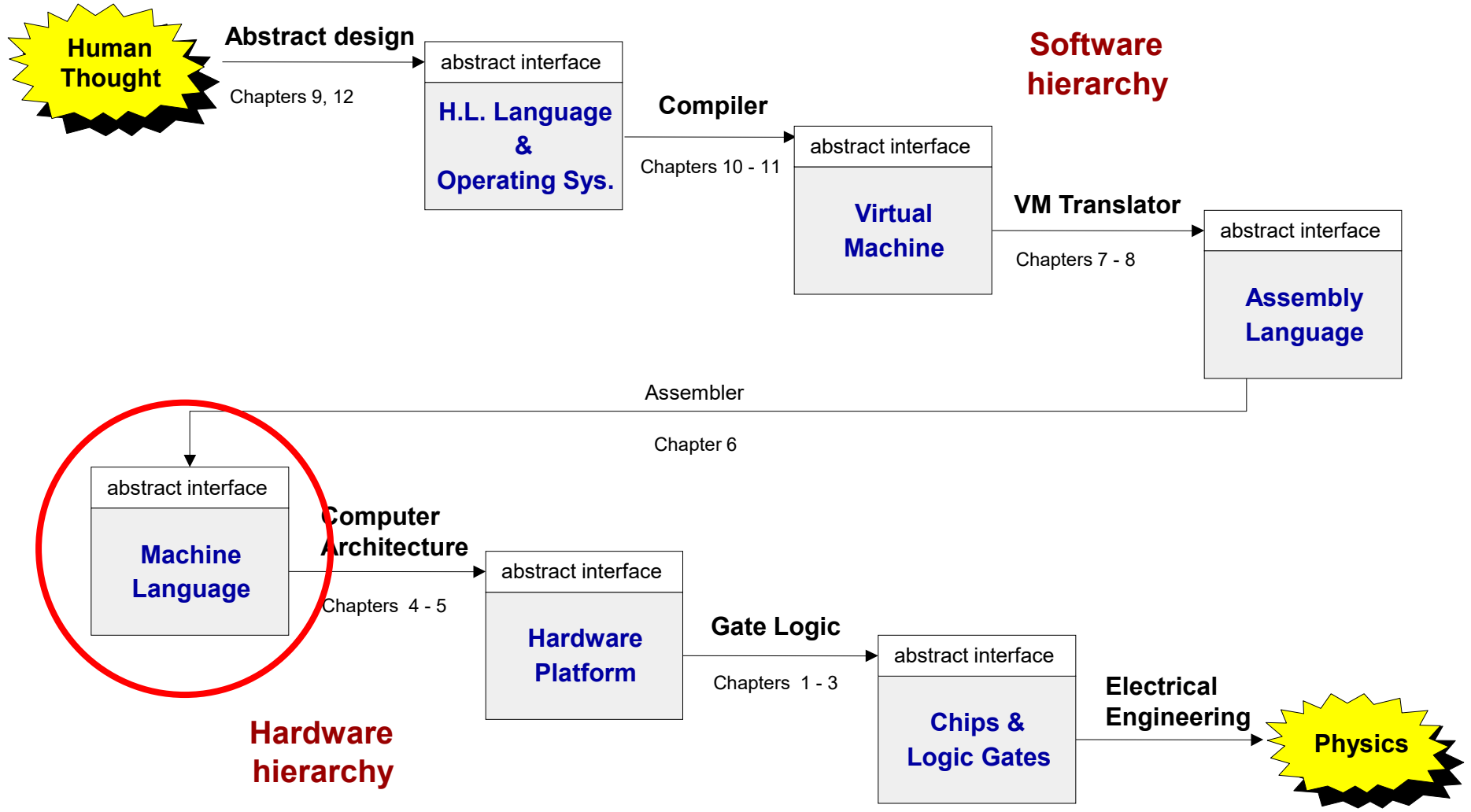
Dr Craig A. Evans

UNIVERSITY OF LEEDS

Outline

- Machine language
- Assembly language
- Hack computer architecture
- Hack machine language
- Examples
- CPU Emulator

Machine Language


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

- Machine language is also known as the **instruction set** of the processor
- Exists on the **hardware/software interface**
- Different ways of looking at it:
  - It is a programmer-oriented abstraction of the hardware platform
  - ...or you can view the hardware as being the actual implementation of the machine language abstraction
- The **lowest-level** coding possible!
- The instructions typically manipulate **memory** using a **processor** and a set of **registers**

Memory

- **Memory** loosely describes a device that stores data and instructions
- At the top-level each data element or instruction (**word**) is stored at a unique address in the memory
- We access a particular word by supplying its address
- Notation:
  - Memory[address]
  - RAM[address]
  - M[address]

Processor

- The processor is usually called the **central processing unit** (CPU)
- It is capable of performing a fixed set of elementary operations:
  - Arithmetic and logical operations (by the ALU)
  - Memory access operations
  - Control operations (e.g. **branching**)
- The operands for the operations are usually **loaded** from registers or selected memory locations
- The results of the operations can either be **stored** in registers or back into selected memory locations

Registers

- Accessing memory is a **relatively slow** operation – the memory is usually completely separate from the CPU
- Therefore CPUs usually have a small collection of **registers** inside the CPU that can be used to store single values
- Act as **high-speed** local memory
- Allows the CPU to process and manipulate data quickly and **minimise memory access**
- e.g. ARM® Cortex®-M microcontrollers have 13 (R0 to R12) 32-bit registers for general-purpose use

Load/Store Architecture

- In a **Load/Store** architecture, there are two types of instructions
- **Memory access** operations copy words from memory to registers in the CPU and vice versa
- **ALU operations** carry out arithmetic and logical operations on registers and store the result back in registers
- i.e. ALU operations can only be carried out on registers and **not memory locations**
- RISC machines (such as ARM) typically have a load/store architecture
- Different to **register memory architecture** (usually CISC – x86)

Assembly Language

- A machine language program is a series of coded instructions
- e.g. An instruction for a 16-bit machine may be 1010 0011 0010 1001
- We need to understand the **instruction set** of the underlying hardware platform in order to know what the instruction does
- For example, the instruction above may set the value of register 3 to be the sum of register 2 and register 9

- Writing programs in machine code is not feasible
- Therefore **symbolic** versions of machine code also exists
- **Mnemonics** are used as labels for operations and hardware elements
- The symbolic notation is called **assembly language**
- An **assembler** converts the assembly language to binary
- The previous instruction could be written as

```
ADD R3,R2,R9    // R3 = R2 + R9
```

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

.asm

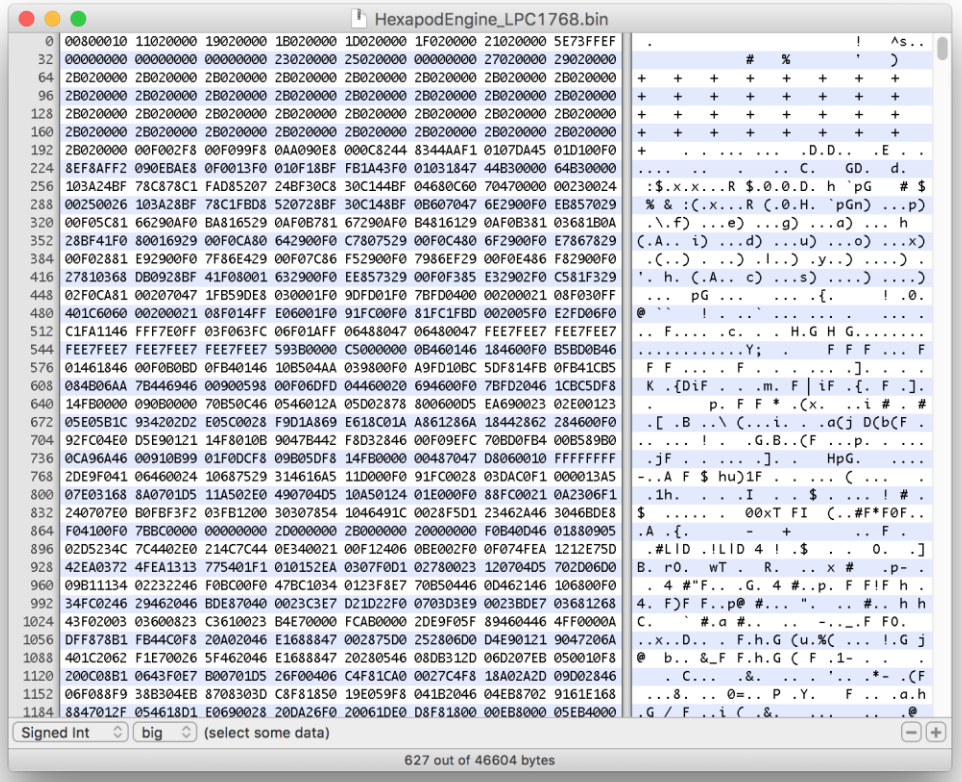
Assembler

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

.hack

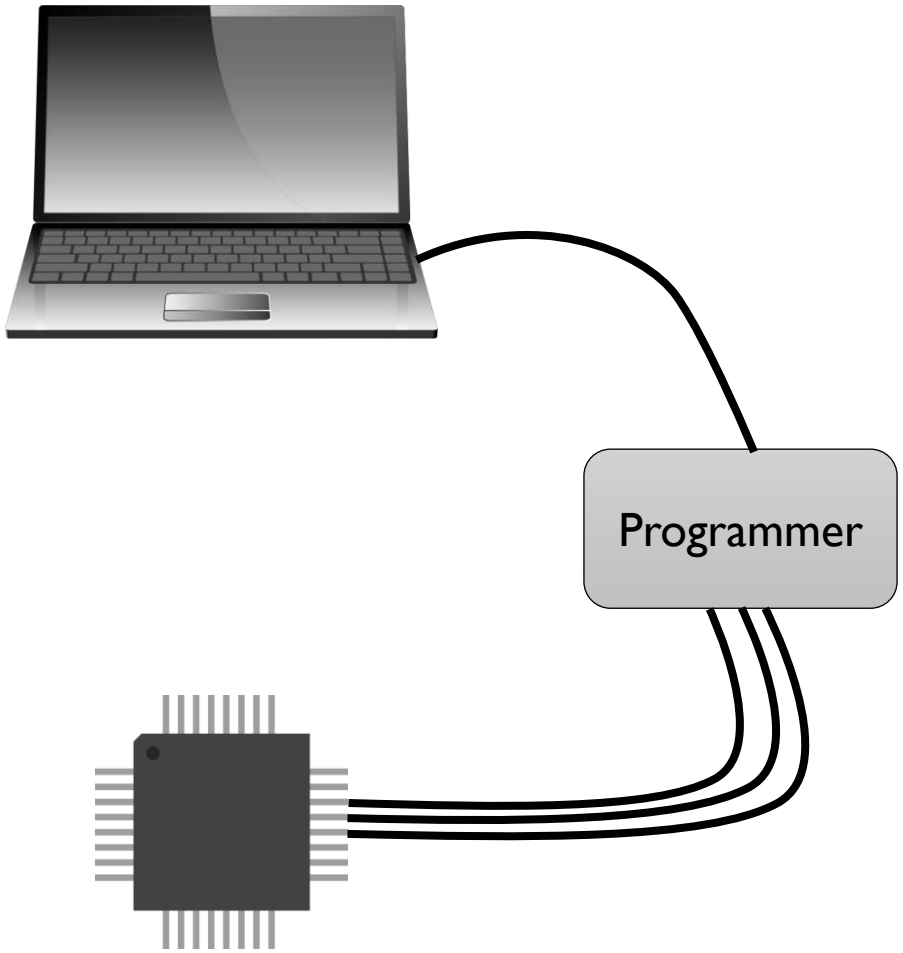
Hex Editor

- The assembler translates each assembly command into a binary machine instruction
- These are often called binary files (.bin)
- We can use **Hex Editors** to load and edit the raw binary files
- They usually show the hexadecimal equivalent of the binary instruction and ASCII interpretation



Programmers

- The binary files contain the 0's and 1's that are loaded into the Read-only Memory (ROM)
- In MCU's the binary instructions are loaded to the Flash memory
- This is usually done using an external Flash programmer
- MCU's often have **bootloaders** that allow us to copy the binary file over serial with a programmer



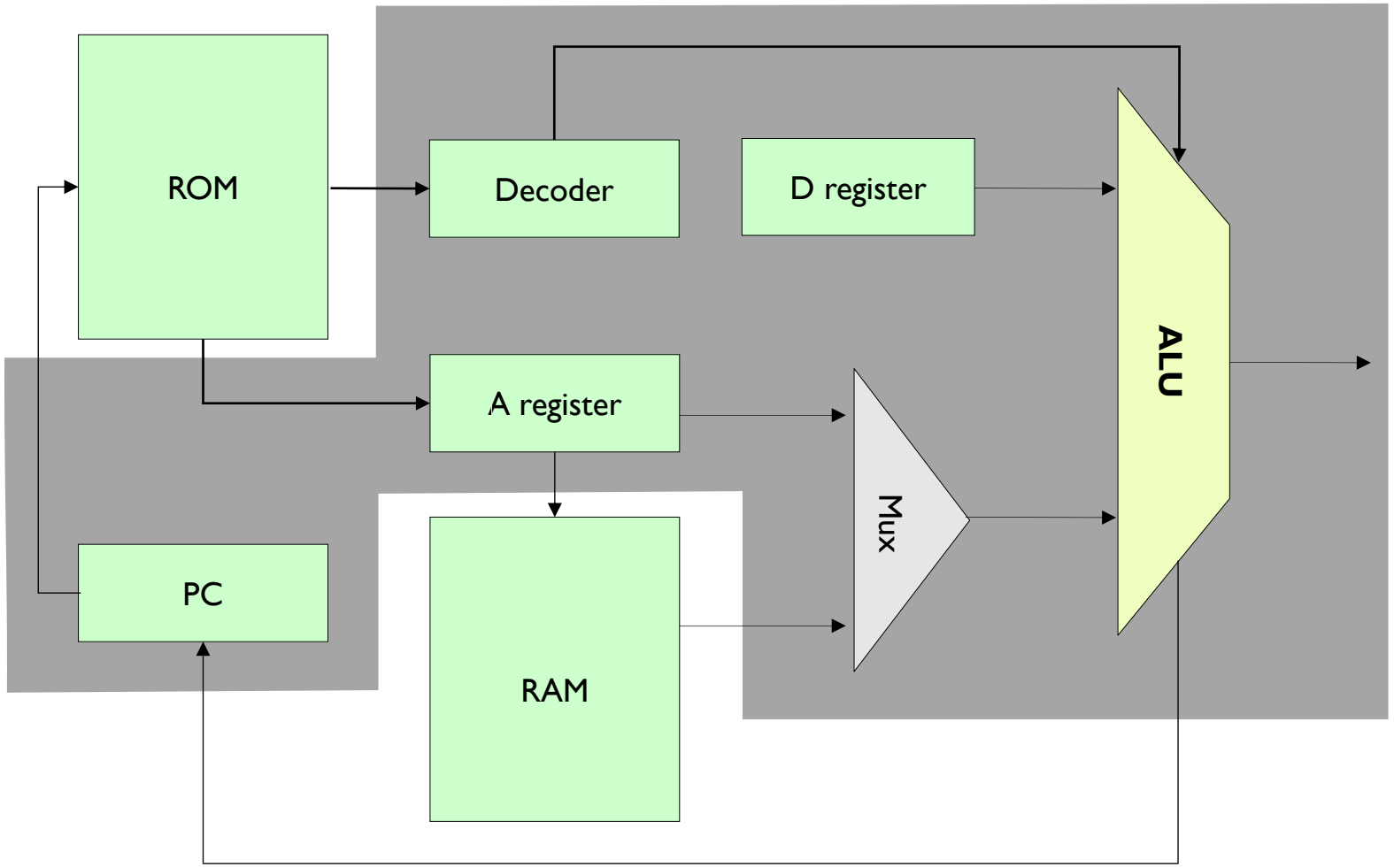
ROMs


NUS-01A-02
©1996Nintendo
U3
U2
C5
C2
U1

Hack Computer Architecture

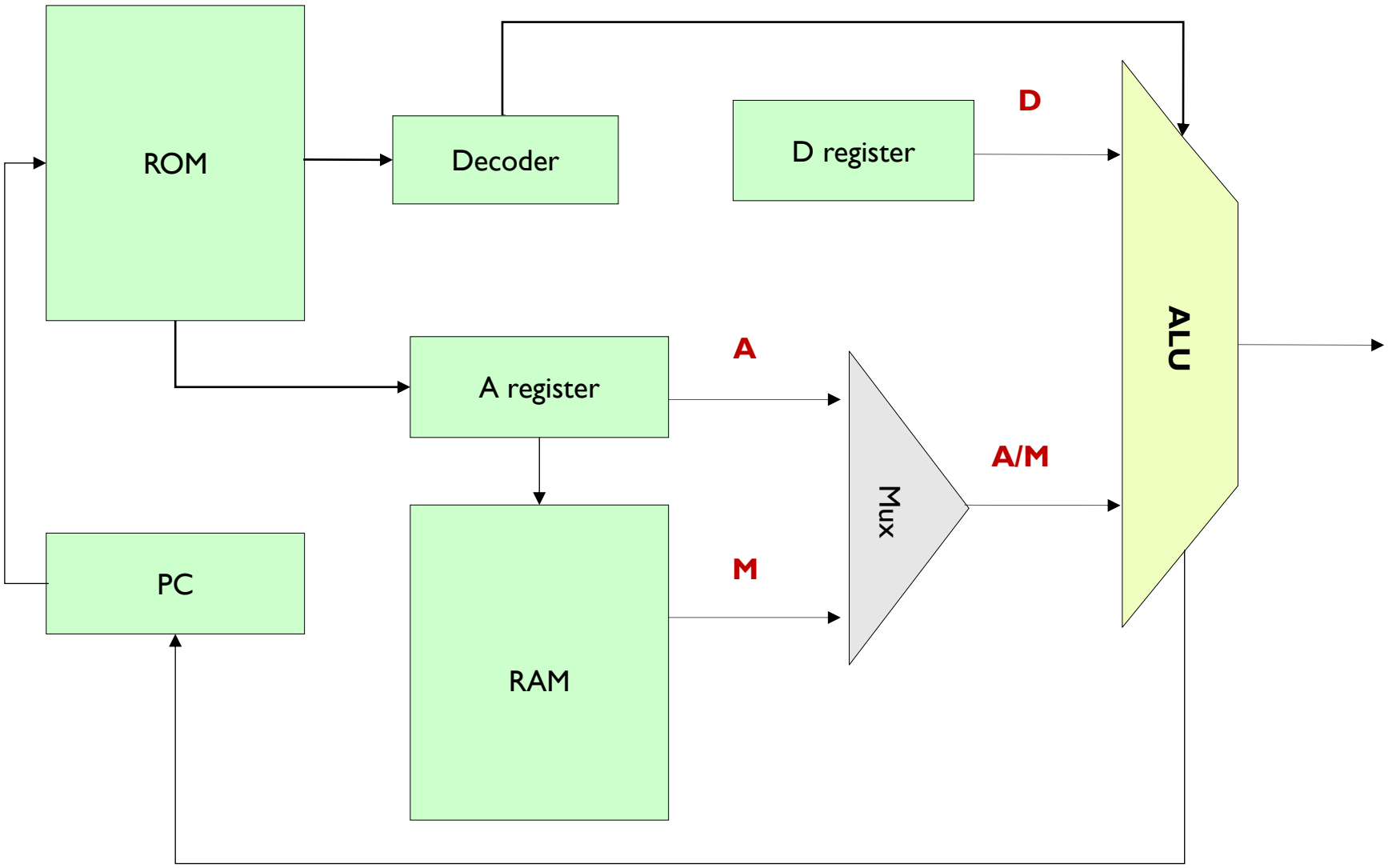
- The Hack computer is a 16-bit machine
- Consists of **CPU**, instruction memory (**ROM**) and data memory (**RAM**)
- **Harvard** architecture (different buses for data and instructions)
- Both memories are 16-bits wide with 15-bit address space
- Two memory-mapped I/O devices:
  - Screen
  - Keyboard
- Two 16-bit registers: **A** (address and data) and **D** (data only)

Simplified Architecture



Data sources

- Two registers in the Hack computer, **A** and **D**
- The computer can also access data stored in RAM
- The RAM address to access is always the current value of the **A** register
- i.e. Memory[A]
- or just **M** for short
- Takes two instructions to access **M**:
  1. First set **A** to the required address
  2. Then read/write from/to **M**


ROM
Decoder
D register
D
ALU
A register
A
Mux
A/M
PC
RAM
M

D-Register Uses

- **D** is purely used to store data values
- Can set the value directly to 0, 1 or -1 using ALU operations
- `D = 1`
- Cannot be set to other literal values directly
- `D = 67` is not valid
- Values are usually loaded from **M** or **A**
- `D = A`

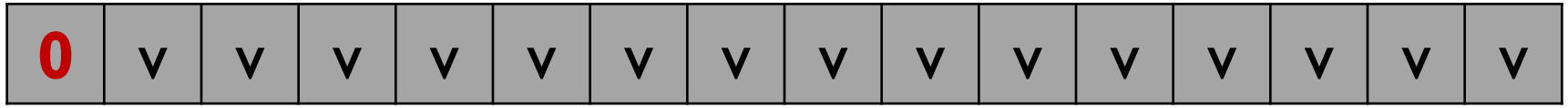
A-Register Uses

A has three uses:

1. Facilitate direct access to data memory (RAM) by storing the address **M** refers to
2. Store data. The only way to load a literal value into our programs is by loading **A** with that value
3. Facilitate direct access to the instruction memory (ROM). When a **jump** command is executed the instruction stored at the address stored in **A** is fetched in the next cycle.

A-Instruction

- The **A**-instruction is used to set the **A**-register to a 15-bit value
- Bit 15 is 0 for an **A**-instruction



- e.g. the following command will load 15 into the **A**-register

```
@15
```

- Note the `@` syntax and not assignment `=` e.g. `A = 15` is invalid

C-Instruction

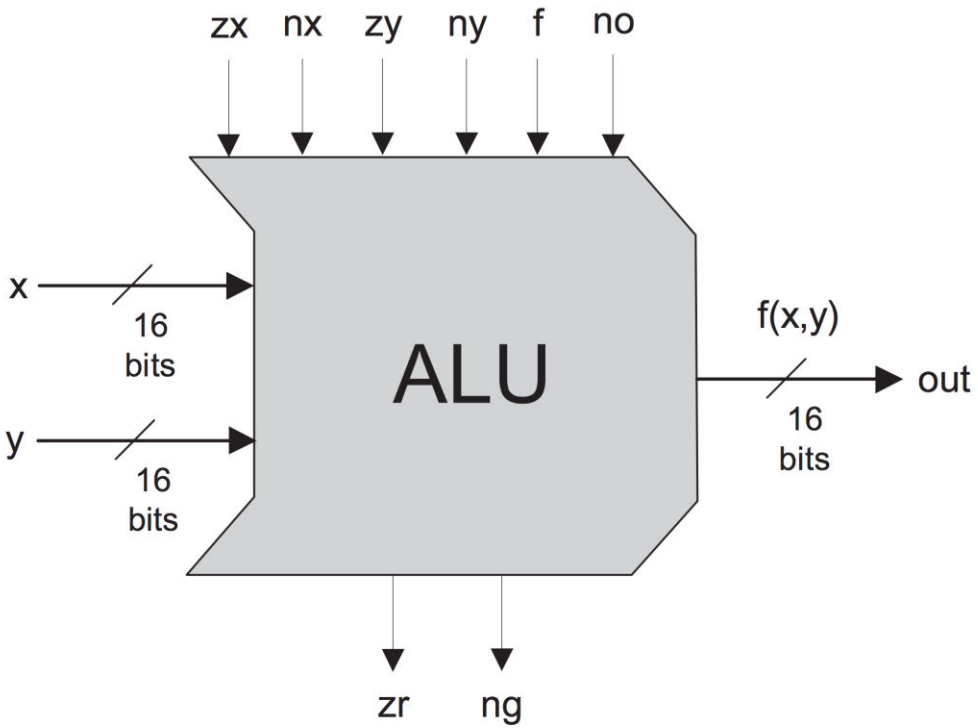
- The **C**-instruction is the workhorse of the Hack computer and handles the computations that are carried out
- It is made up of a **destination** to store the result in (*optional*), the **computation** to carry out and any **jump** condition that is required (*optional*)

```
dest = comp;jump

comp;jump

dest = comp
```

- Bit 15 is 1 for a **C**-instruction
- Bits 14 and 13 are not used
- The a-bit is used to select between **A** and **M**
- The six c-bits are used to select the computation to carry out
- The three d-bits are used to set the destination to store the result in (a combination of **A**, **D** and **M**)
- The three j-bits specify the jump condition

Computation

- The computation bits determine which operation to carry out
- The operation is carried out by the ALU
- The 6 c-bits are the same as the 6 instruction bits of the ALU
- 28 different instructions in the instruction set



zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

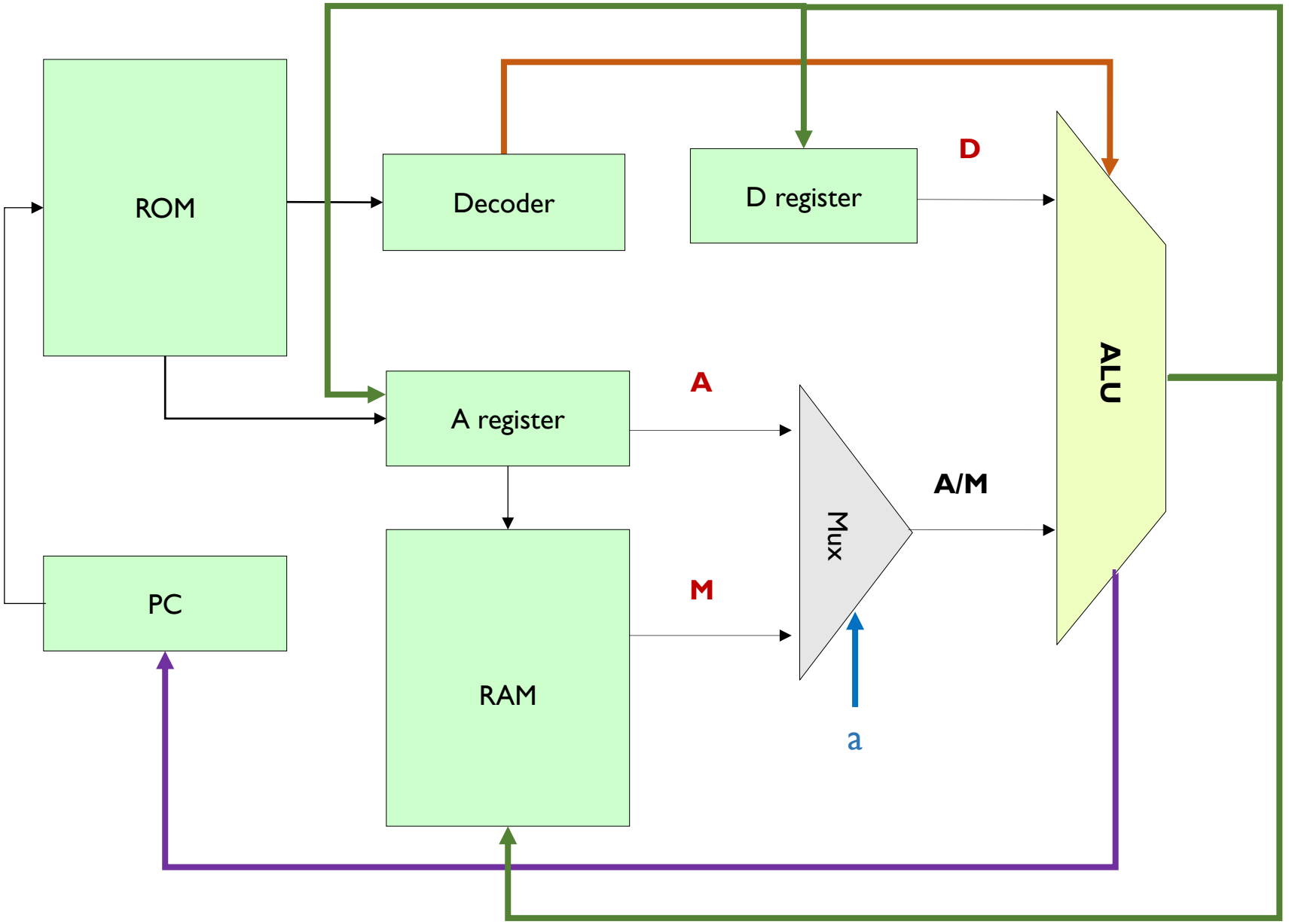
d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

- The result of the computation (ALU output) is fed to one of three locations depending on the d-bits

Jump

- The jump bits tell the CPU what to do next
- Typically it will fetch the next instruction in the ROM
- If a jump condition has been specified in the **C**-instruction then the PC will be loaded with a new address if certain conditions are met
- The address to jump to is the current value stored in **A**
- This must be loaded into **A** using the **A**-instruction before carrying out the **C**-instruction with a jump specified
- Whether the jump occurs or not depends on whether the output of the ALU meets the jump condition

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump


ROM
Decoder
D register
D
ALU
A register
A
A/M
Mux
PC
RAM
M
a

Examples

Loading values into registers

- **A** and **D** can both act as data registers and used to hold values
- They can easily be set to 0, 1 or -1 using ALU computations
- Simply set the **destination** of the instruction as the relevant register
- Use the required **computation**

```
dest = comp;jump
comp;jump
dest = comp
```

```
D = 1   // set D reg to 1
D = 0   // set D reg to 0
D = -1  // set D reg to -1
```

```
A = 1   // set A reg to 1
A = 0   // set A reg to 0
A = -1  // set A reg to -1
```

- **M** (RAM) can also be used to store data
- Before writing to RAM we must select the address using the **A**-instruction

```
@5      // set address to 5
M = 1   // RAM[5] = 1
```

- We may also wish to copy the contents of the D or A register to a particular RAM location

```
@12     // set address to 12
M = D   // RAM[12] = D
```

Storing literals in **D** or **A**

- To load literals values (other than 0, 1 or -1) into registers we must use an **A**-instruction
- The **A**-instruction allows us to load a 15-bit number into **A**
- Simple to store a literal value in **D** e.g.

```
@99      // store 99 in A
D = A    // D = 99
```

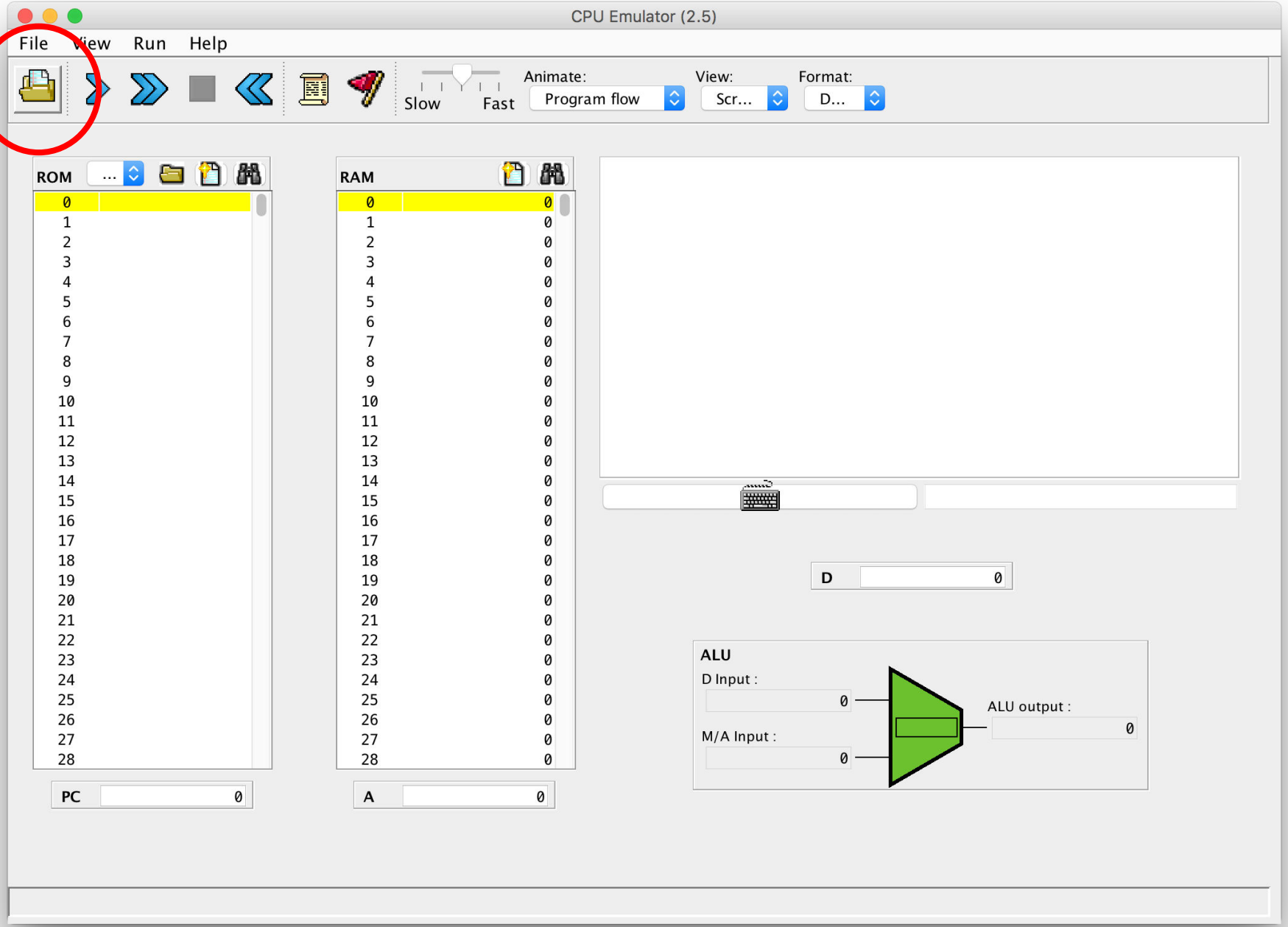
Storing literals in RAM

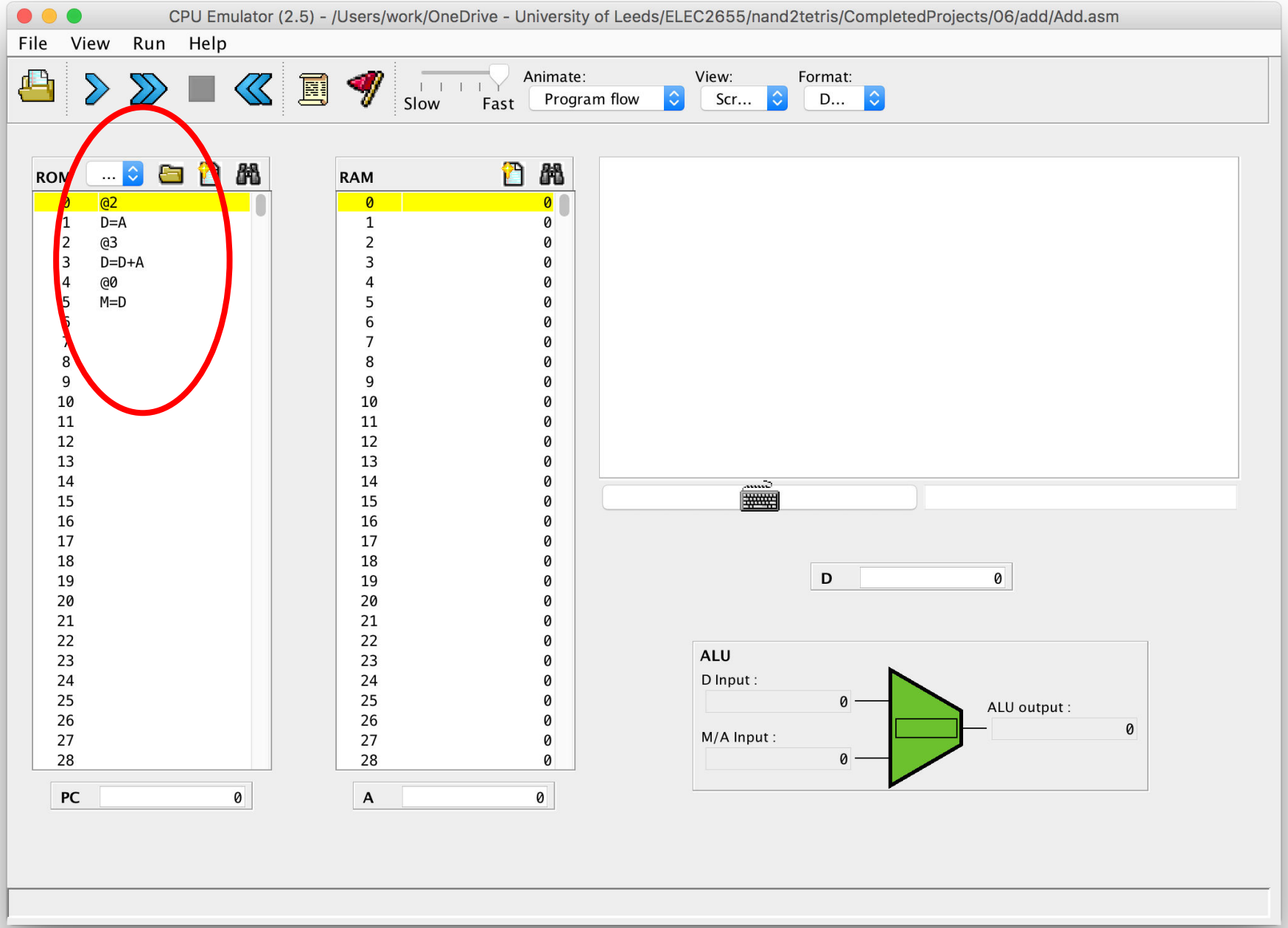
- To store a literal in **M** takes another **A**-instruction to set the RAM address e.g.

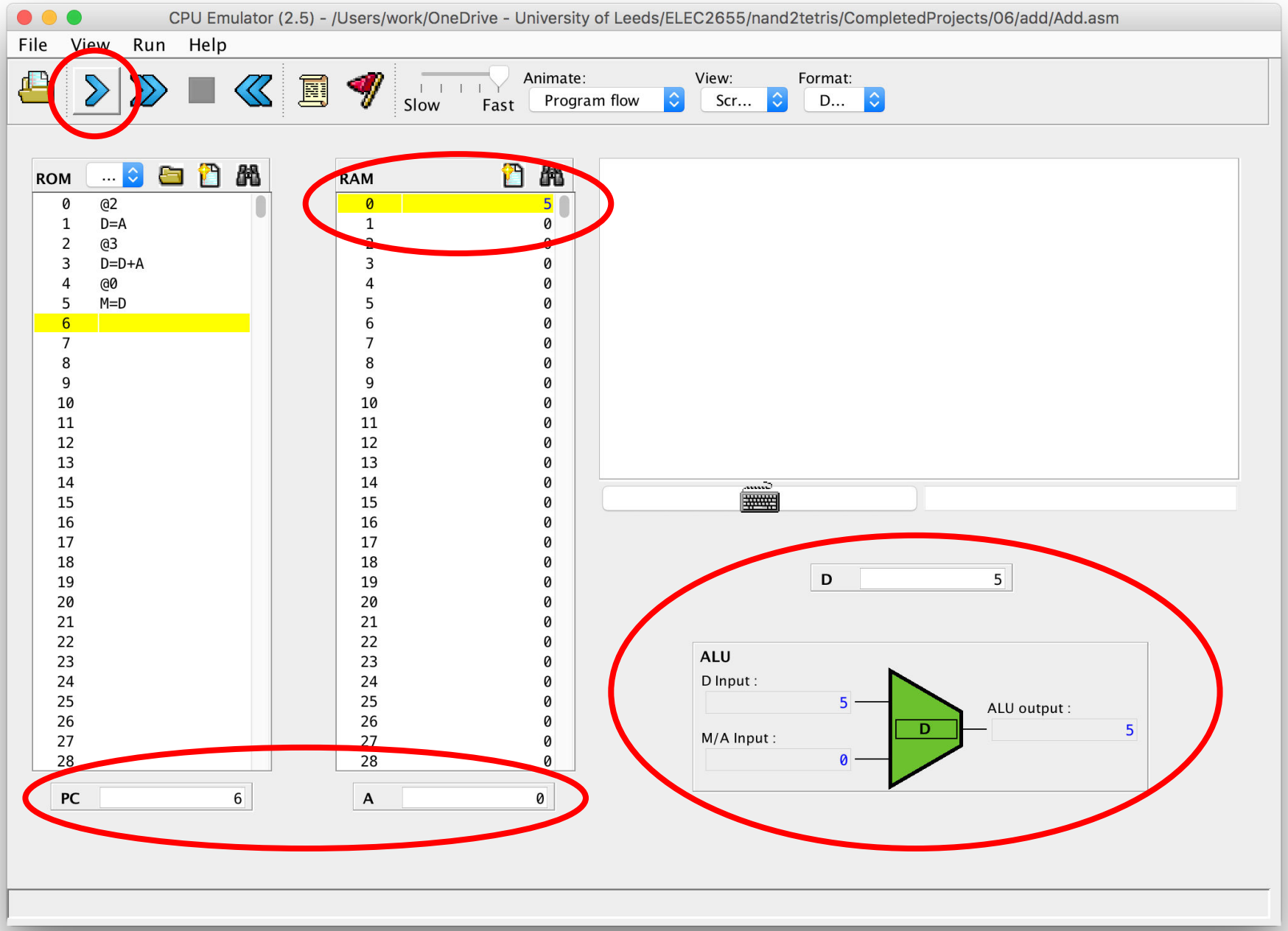
```
@57      // store 57 in A
D = A    // Use D as a temporary holding area
@10      // store 10 in A, this sets the memory address
M = D    // RAM[10] = 57
```

CPU Emulator

- We run our assembly code in the **CPU Emulator**
- This tool emulates the behaviour of the Hack computer
- Once you have finished building the Hack computer you can run machine code in the **Hardware Simulator**
- Using the CPU Emulator we can load assembly code (.asm) directly
- It automatically assembles it into the binary .hack files
- These are the raw 0's and 1's that are loaded in the ROM
- We can also load .hack files we've assembled using the supplied **Assembler**


CPU Emulator (2.5)
File
View
Run
Help
Slow
Fast
Animate:
Program flow
View:
Scr...
Format:
D...
ROM
RAM
PC
A
D
ALU
D Input :
M/A Input :
ALU output :


CPU Emulator (2.5) - /Users/work/OneDrive - University of Leeds/ELEC2655/nand2tetris/CompletedProjects/06/add/Add.asm
File
View
Run
Help
Slow
Fast
Animate:
Program flow
View:
Scr...
Format:
D...
ROM
0 @2
1 D=A
2 @3
3 D=D+A
4 @0
5 M=D
RAM
PC 0
A 0
D 0
ALU
D Input :
0
M/A Input :
0
ALU output :
0


CPU Emulator (2.5) - /Users/work/OneDrive - University of Leeds/ELEC2655/nand2tetris/CompletedProjects/06/add/Add.asm
File
View
Run
Help
Slow
Fast
Animate:
Program flow
View:
Scr...
Format:
D...
ROM
0 @2
1 D=A
2 @3
3 D=D+A
4 @0
5 M=D
6
RAM
0 5
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
10 0
11 0
12 0
13 0
14 0
15 0
16 0
17 0
18 0
19 0
20 0
21 0
22 0
23 0
24 0
25 0
26 0
27 0
28 0
PC 6
A 0
D 5
ALU
D Input :
5
M/A Input :
0
D
ALU output :
5

Hack Instruction Set Summary

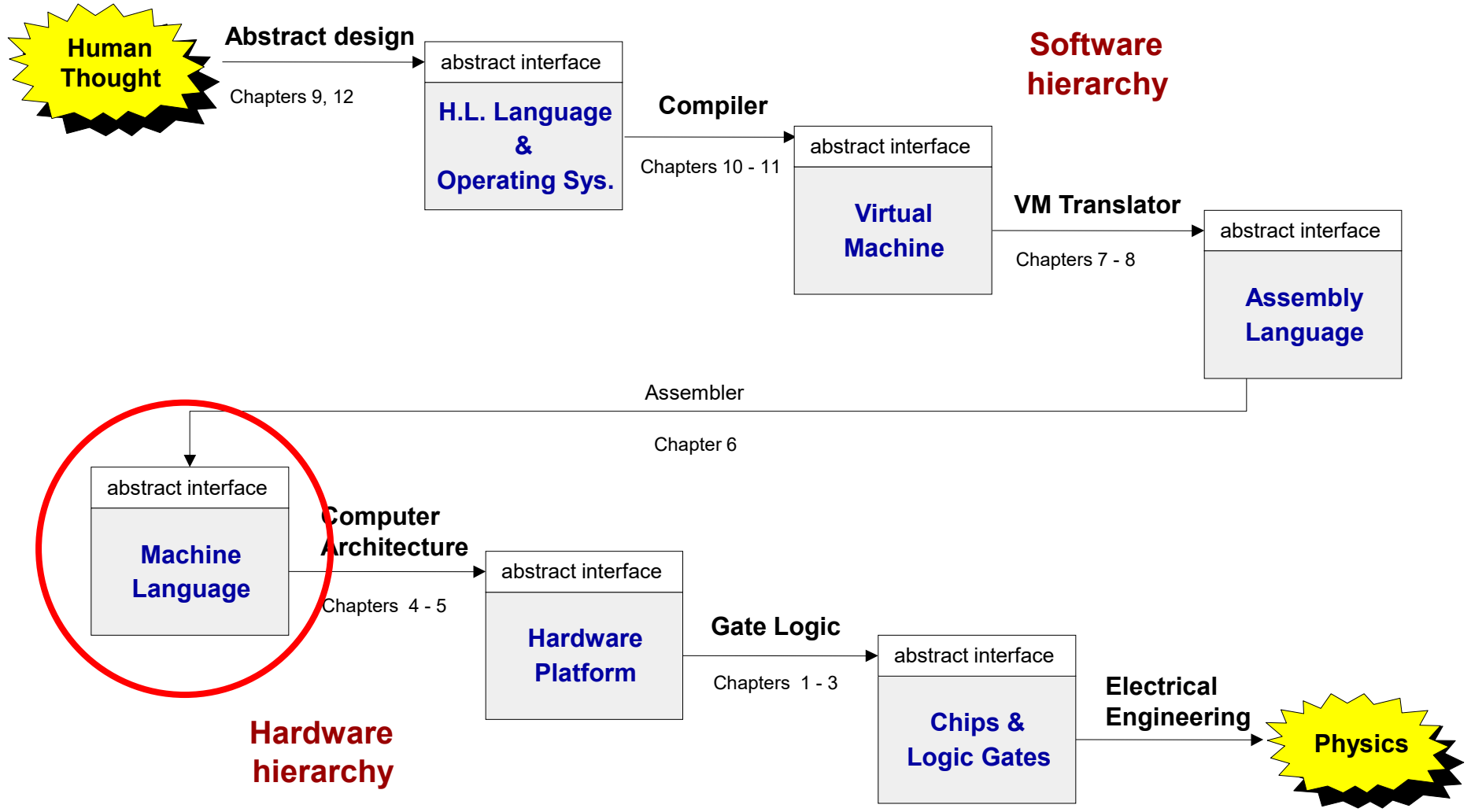
```
dest = comp;jump
comp;jump
dest = comp
```

Dest	Jump	Comp			
null	null	0	1	-1	D&A
M	JGT	D	A	M	D&M
D	JEQ	!D	!A	!M	D|A
MD	JGE	-D	-A	-M	D|M
A	JLT	D+1	A+1	M+1	
AM	JNE	D-1	A-1	M-1	
AD	JLE	D+A	D+M	A-D	
AMD	JMP	D-A	D-M	M-D	

2.2 – Assembly Language II

Dr Craig A. Evans

UNIVERSITY OF LEEDS


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Outline

- Symbols
- Control statements
  - Branching
  - Loops
- I/O
- Hack Instruction Set Summary

Symbols

- Previously we have seen that data values can be stored in specific RAM locations
- We have done this directly using **constants**
- e.g. the following code will store 99 in RAM[3]

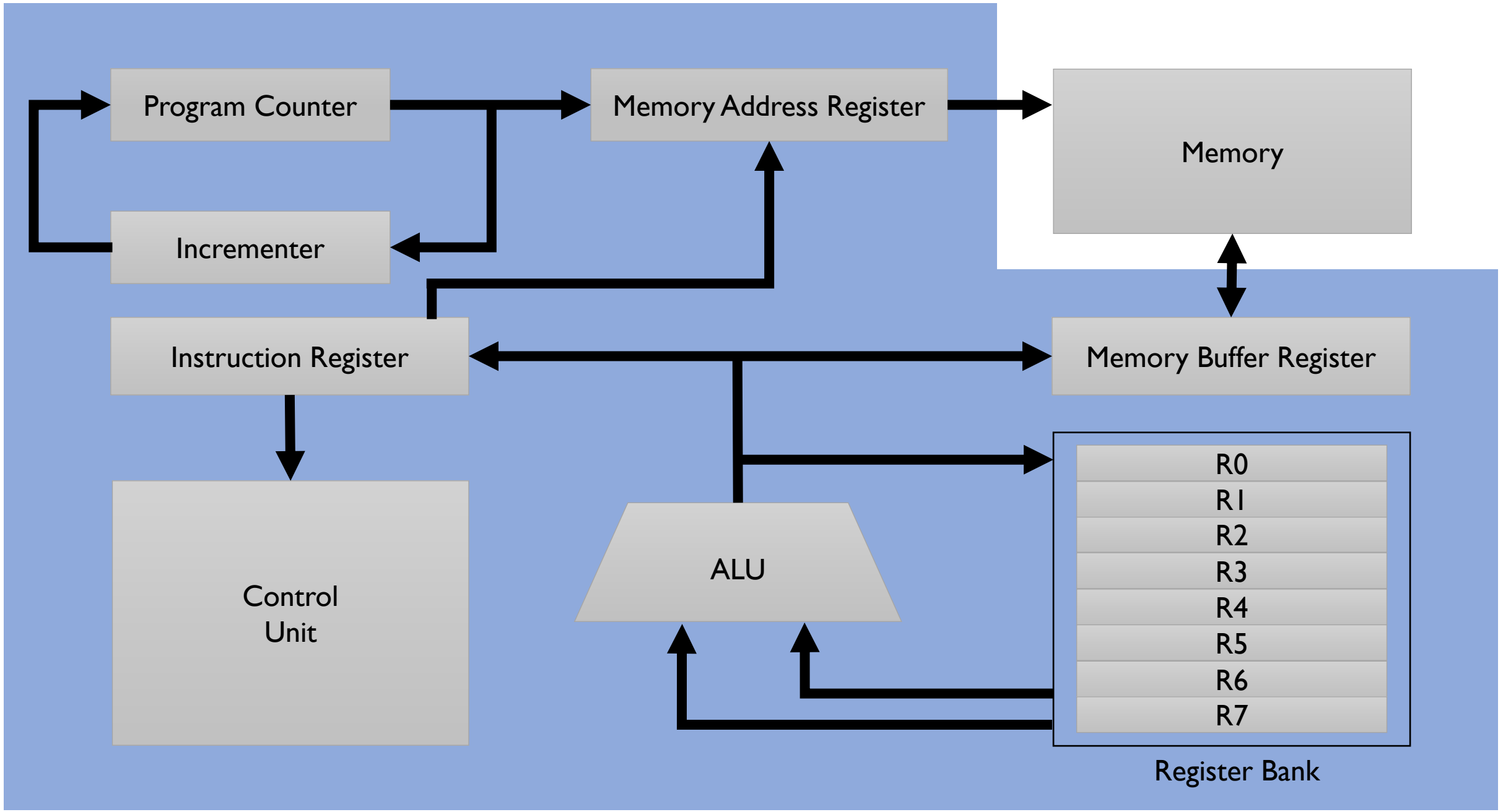
```
@99
D=A
@3
M=D
```

- Another way to refer to memory locations is by using **symbols**
- There are three ways of using symbols in the Hack assembly programs:

1. Pre-defined symbols
2. Label symbols
3. Variable symbols

Pre-defined Symbols

- **Virtual registers**
  - RAM addresses 0 to 15 can be referred to using the symbols `R0` to `R15`
  - This mimics the architecture of typical processors that have a small number of high-speed registers **inside** the CPU itself
- **Predefined pointers**
  - `SP`, `LCL`, `ARG`, `THIS`, `THAT` are pre-defined and refer to addresses 0 to 4 respectively
  - Can be used together with `R0` to `R4`
  - Used in the implementation of the Virtual Machine in Part II
- **I/O pointers**
  - `SCREEN` and `KBD` refer to 16384 (0x4000) and 24576 (0x6000) and are the base-addresses of the I/O memory maps


Program Counter
Memory Address Register
Memory
Incrementer
Instruction Register
Memory Buffer Register
Control
Unit
ALU
R0
R1
R2
R3
R4
R5
R6
R7
Register Bank

Label Symbols

- These are user-defined symbols
- They label the destination of jump commands
- Declared using the syntax

`(xxx)`

- `xxx` refers to the **instruction memory** (ROM) location holding the **next** command to be executed
- i.e. value to be loaded into the PC
- Can be defined only once in the code

Variable Symbols

- These are user-defined symbols
- Any symbol `xxx` that is not pre-defined or a label is treated as a variable
- Automatically assigned a RAM address starting at 16 (0x0010)
- e.g. Suppose `@sum` is the first variable used in the code
- It will be stored in RAM[16]
- `@total` is the next variable and will be stored in RAM[17]

Convention and Syntax

- Constants must be non-negative and written in decimal
- Symbol names can be any sequence of letters (upper- and lower-case), numbers, underscore, dot, $ and :
  - Cannot begin with a digit
- Convention:
  - Variables are lower-case
  - Labels are upper-case
  - Commands (mnemonics) are upper-case
- White space is ignored
- // denotes a comment that is ignored

Control Statements

- We'll now look how to implement common code constructs in Hack assembly
- High-level C provides two main ways of flow control, **branching** and **loops**
- Branching is usually decision-based and the program can flow down different branches (typically depending on whether a relational operation is true)
- Loops allow a portion of code to be repeated a specified number of times (typically depending on whether a relational operation is true)

If logic

```
if (RAM[0] > 100) {
      RAM[1] = 5;
}
```

```
0       @R0
1       D=M   // store R0 in D

2       @100
3       D=D-A // R0 - 100

4       @IF_TRUE  // @8
5       D;JGT // if D>0 then R0 > 100

6       @CONT    // @12
7       0;JMP
        // unconditional jump if false

(IF_TRUE)
8       @5
9       D=A
10      @R1
11      M=D

(CONT)
```

If/Else logic

```
if (RAM[0] > 100) {
        RAM[1] = 5;
}
else {
        RAM[1] = 7;
}
```

```
0       @R0
1       D=M   // store R0 in D
2       @100
3       D=D-A // R0 - 100
4       @IF_TRUE  // @12
5       D;JGT // if D>0 then R0 > 100
6       @7
7       D=A
8       @R1
9       M=D
10      @CONT     // @16
11      0;JMP
(IF_TRUE)
12      @5
13      D=A
14      @R1
15      M=D
(CONT)
```

While logic

```
// Adds 1+2+3+4+5

int i = 1;
int sum = 0;
while (i <= 5){
    sum += i;
    i++;
}
```

```
0       @i      // RAM[16]
1       M=1     // i=1
2       @sum    // RAM[17]
3       M=0     // sum=0
(LOOP)
4       @i       // @16
5       D=M      // D = i
6       @5
7       D=D-A    // D = i - 5
8       @CONT    // @18
9       D;JGT    // if(i-5)>0 goto CONT (18)
10      @i       // @16
11      D=M      // D = i
12      @sum     // @17
13      M=D+M    // sum += i
14      @i
15      M=M+1    // i++
16      @LOOP    // @4
17      0;JMP    // goto LOOP (line 4)
(CONT)
18
```

While logic

```
// Adds 1+2+3+4+5

int i = 1;
int sum = 0;
while (i <= 5){
    sum += i;
    i++;
}
```

```
0       @i      // RAM[16]
1       M=1     // i=1
2       @sum    // RAM[17]
3       M=0     // sum=0
(LOOP)
4       @i       // @16
5       D=M      // D = i
6       @5
7       D=D-A    // D = i - 5
8       @CONT    // @18
9       D;JGT    // if(i-5)>0 goto CONT (18)
10      @i       // @16
11      D=M      // D = i
12      @sum     // @17
13      M=D+M    // sum += i
14      @i
15      M=M+1    // i++
16      @LOOP    // @4
17      0;JMP    // goto LOOP (line 4)
(CONT)
18
```

Conflicting use of the **A**-register

- The **A**-register can be used to select a data memory (RAM) location for a subsequent C-instruction involving **M**

```
@15
M=D
```

- It can also be used to select an instruction memory (ROM) location for a subsequent C-instruction involving a jump

```
@CONT
D;JGT
```

- Therefore to avoid conflicting use, you should not use C-instructions with a jump that contain a reference to **M**

```
@67
D=M;JGT
```

- This will copy RAM[67] to **D-**register and then jump to instruction 67 in ROM

I/O

- The keyboard and screen are **memory-mapped** peripherals
- Means the memory associated with them appears to be a part of the main memory
- Each has a particular address (or range of addresses) associated with them
- Common in microcontrollers – Flash, RAM and peripherals all appear to be in the same memory space despite being physically separate devices

Keyboard

- Keyboard (`KBD`) is essentially a 16-bit register at memory location 24576 (0x6000)
- When read it contains the 16-bit ASCII value of the current key being pressed (0 when no key is pressed)
  - '0' to '9' (48 to 57)
  - 'A' to 'Z' (65 to 90)
  - 'a' to 'z' (97 to 122)

Key Pressed	Code	Key Pressed	Code
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

Screen

- The ‘screen’ has a resolution of 256 rows by 512 columns
- Represented by a 8K memory map starting at 16384 (0x4000)
- Each row is represented by 512/16 = 32 words (16-bit each)
- Each bit represents a pixel (1 = black , 0 = white)
- Pixel on row *r* and column *c* is the *c*%16 bit in the word at RAM[16834+32*r*+*c*/16]

Tip: -1 in binary is 1111 1111 1111 1111

Hack Instruction Set Summary

```
dest = comp;jump
comp;jump
dest = comp
```

Dest	Jump	Comp			
null	null	0	1	-1	D&A
M	JGT	D	A	M	D&M
D	JEQ	!D	!A	!M	D|A
MD	JGE	-D	-A	-M	D|M
A	JLT	D+1	A+1	M+1	
AM	JNE	D-1	A-1	M-1	
AD	JLE	D+A	D+M	A-D	
AMD	JMP	D-A	D-M	M-D	

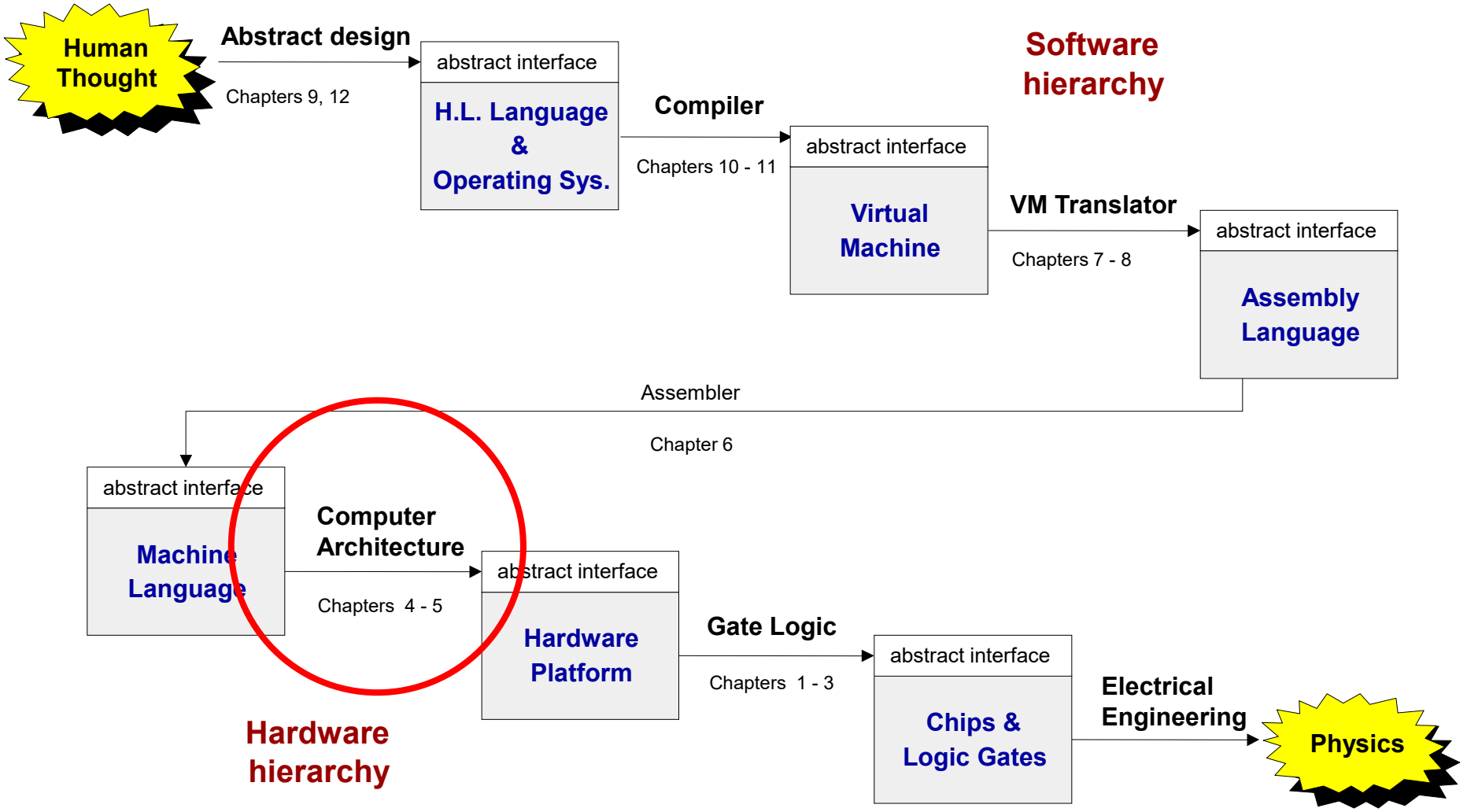
Symbols

Symbol	Value
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

2.3 – Computer Architecture

Dr Craig A. Evans

UNIVERSITY OF LEEDS


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Outline

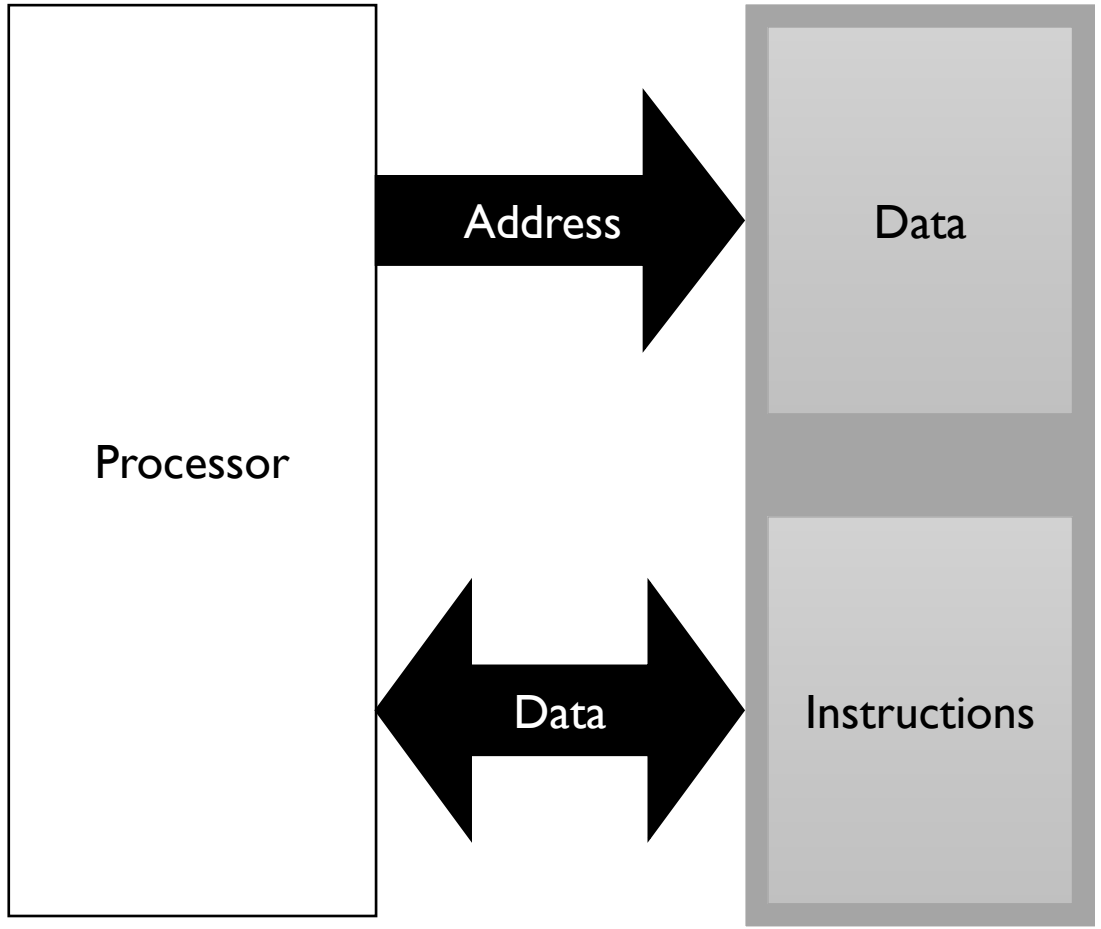
- Stored-program Computer
- Hack Computer
- Memory
- CPU
- Computer

Stored Program Computer

- A computer is a fairly unique machine in that it consists of a fixed **hardware** platform yet it can carry out an almost infinite number of tasks
- It does this by running different **software** – games, web browsers, media players, word processors, custom embedded firmware etc.
- The software consists of **instructions** that are stored within the computer memory and can be changed by loading new instructions into the memory
- This is called the **stored-program** concept
- The computer then **fetches** these instructions from memory and **executes** them

Components

- **Processor** – the thing that actually executes the instructions
- **Memory -** stores the instructions themselves and any data required
- **Buses** – used to move the data and instructions back and forth between the processor and memory



Fetch-Execute Cycle

- In essence a computer simply executes a series of instructions
- Executing an instruction involves the following steps:
  - The ALU computing some operation
  - Writing the result to a register and/or memory location
  - Deduce which instruction to execute next

Architecture

- The term be used in different ways:
- The **instruction set architecture** (ISA) describes the instructions that the computer can execute and is the view that is most important to a computer programmer
- The term **microarchitecture** refers to the way in which the ISA is implemented in hardware i.e. an ADD instruction may be implemented using a binary adder
- The **computer architecture** describes the complete system including the processor, memory, buses and peripherals

Hack Computer

- At this point in the course we have built the entire Hack chipset:
  - *Nand, Not, And, Or, Xor, Mux, Dmux, Not16, And16, Or16, Mux16, Or8Way, Mux4Way16, Mux8Way16, DMux4Way, DMux8Way*
  - *HalfAdder, FullAdder, Add16, Inc16, ALU*
  - *DFF, Bit, Register, RAM8, RAM64, RAM512, RAM4K, RAM16K, PC*
- This project is the culmination of the hardware design and implements the final few pieces of the Hack computer:
  - *Memory, CPU, Computer*

Specification

- 16-bit machine
- Physically separate instruction and data memory
  - 32K instruction memory (ROM)
  - 16K data memory (RAM)
- 512 x 256 screen (black and white)
- Standard keyboard
- Executes programs written in the Hack machine language

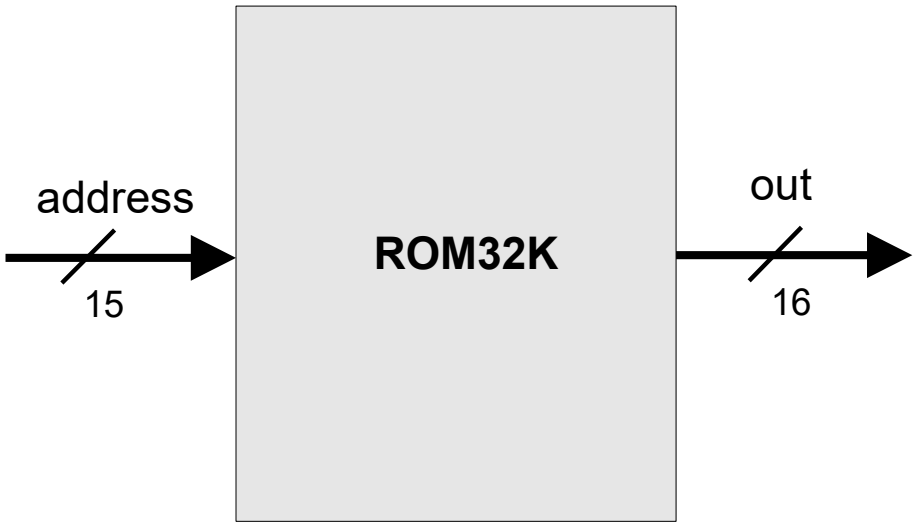
Memory

Instruction memory

- The ROM is pre-loaded with a program written in the Hack machine language
- Consists of 32K 16-bit registers
- Each register can contain an instruction

```
out = ROM32K[address]
```

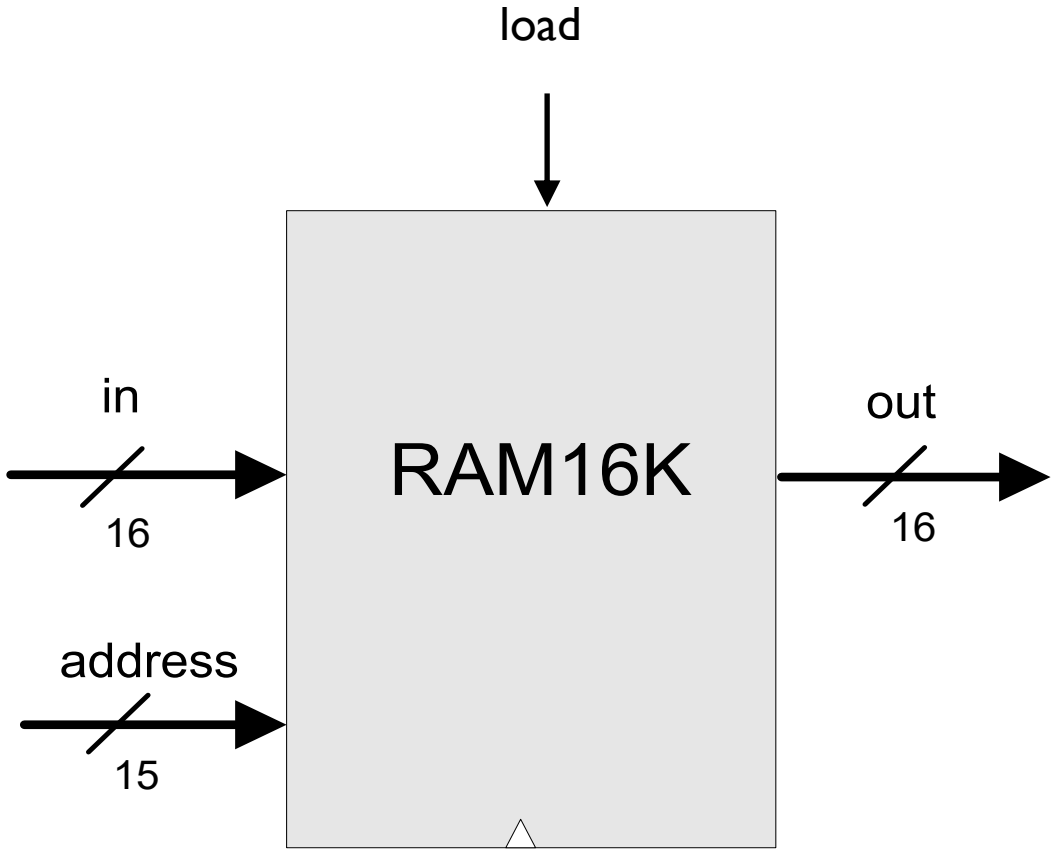
- Built-in – do not need to implement



- We won't worry about how the program is loaded to the ROM
- We'll just use the simulation tools to load in a assembled .hack file
- In reality a **programmer** is used to write the binary instructions to the Flash memory (for a microcontroller)
- For an actual ROM device, the binary image is **burned** in the factory and cannot be changed
- Some microcontrollers have **electrically-erasable programmable read-only memory** (EEPROM)
- Typically just a few kilobytes

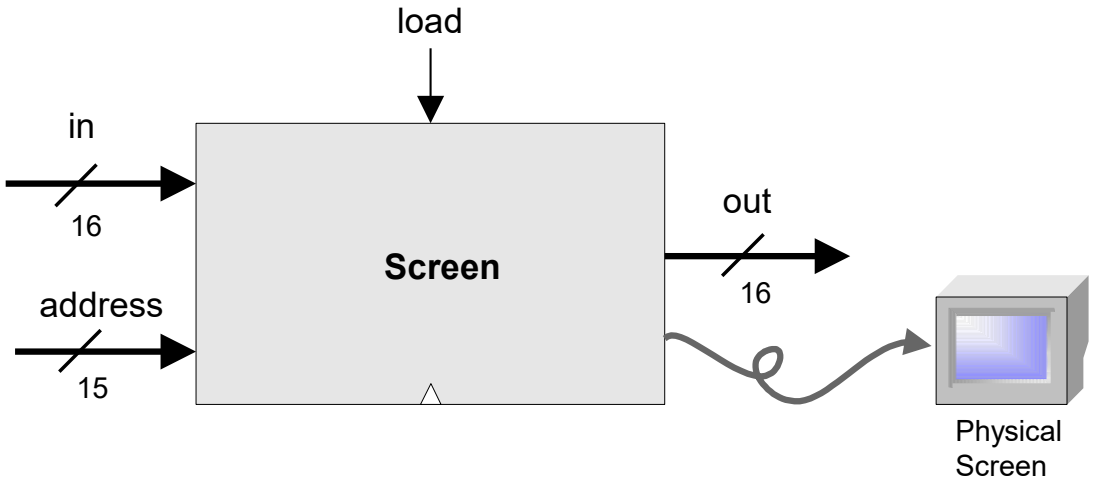
Data memory

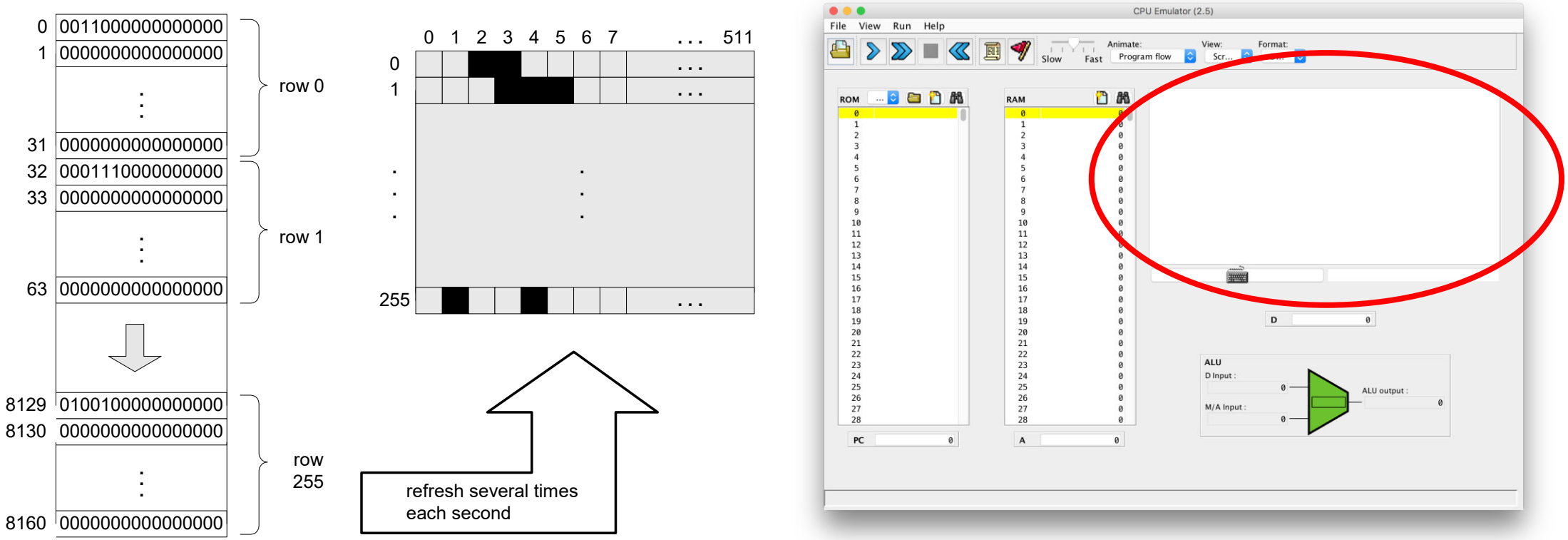
- To read from memory (RAM[*k*]) we set the address to *k* and then probe *out*
- To write to memory (RAM[*k*] = *x*) we set the address to *k*, set *in* to *x*, set *load* to 1 and then clock it in
- We've already built this



Screen

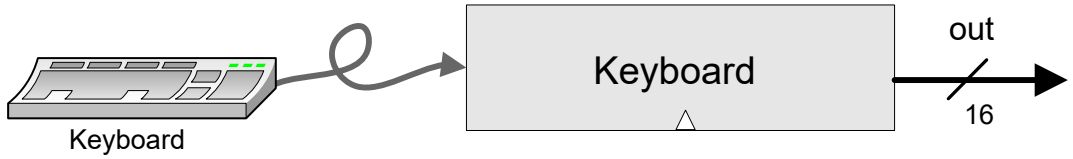
- Has basic RAM chip functionality (8K)
- Base address 0x4000
- The bit contents of the screen chip is known as the screen **memory map**
- Built-in, we don't have to implement
- Essentially just RAM but with a GUI element in the CPU Emulator

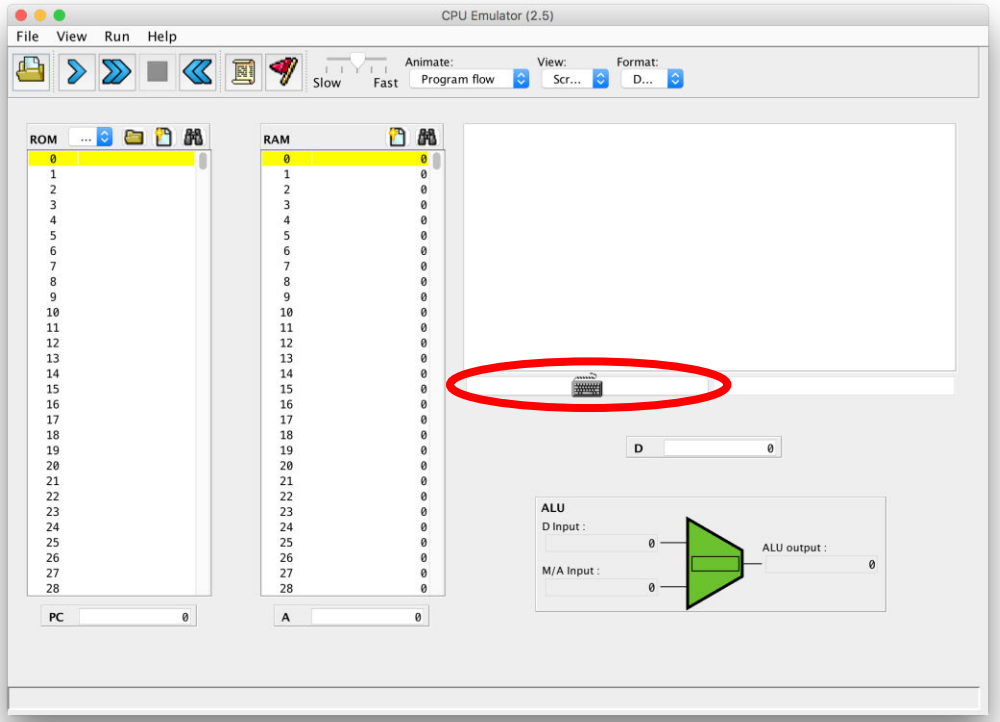



0 0011000000000000
1 0000000000000000
31 0000000000000000
32 0001110000000000
33 0000000000000000
63 0000000000000000
8129 0100100000000000
8130 0000000000000000
8160 0000000000000000
row 0
row 1
row 255
0 1 2 3 4 5 6 7 . . . 511
0
1
255
refresh several times each second
CPU Emulator (2.5)
File View Run Help
Slow
Fast
Animate:
Program flow
View:
Format:
ROM
RAM
PC
A
D
ALU
D Input :
M/A Input :
ALU output :

Keyboard

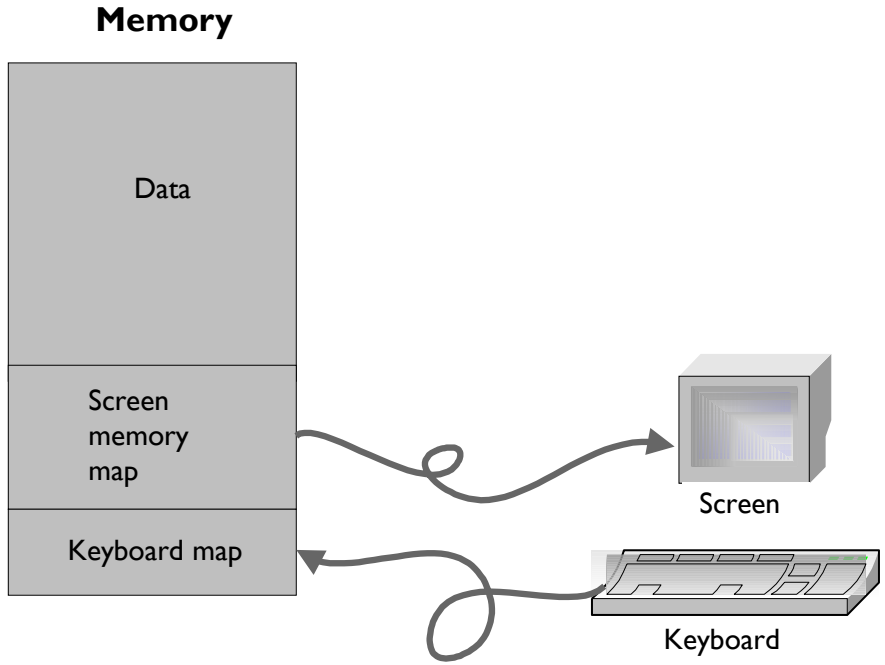
- 16-bit register that contains the ASCII code of the key currently being pressed
- Base address 0x6000
- Built-in, we don't have to implement
- Essentially just a read-only 16-bit register with a GUI element in the CPU Emulator





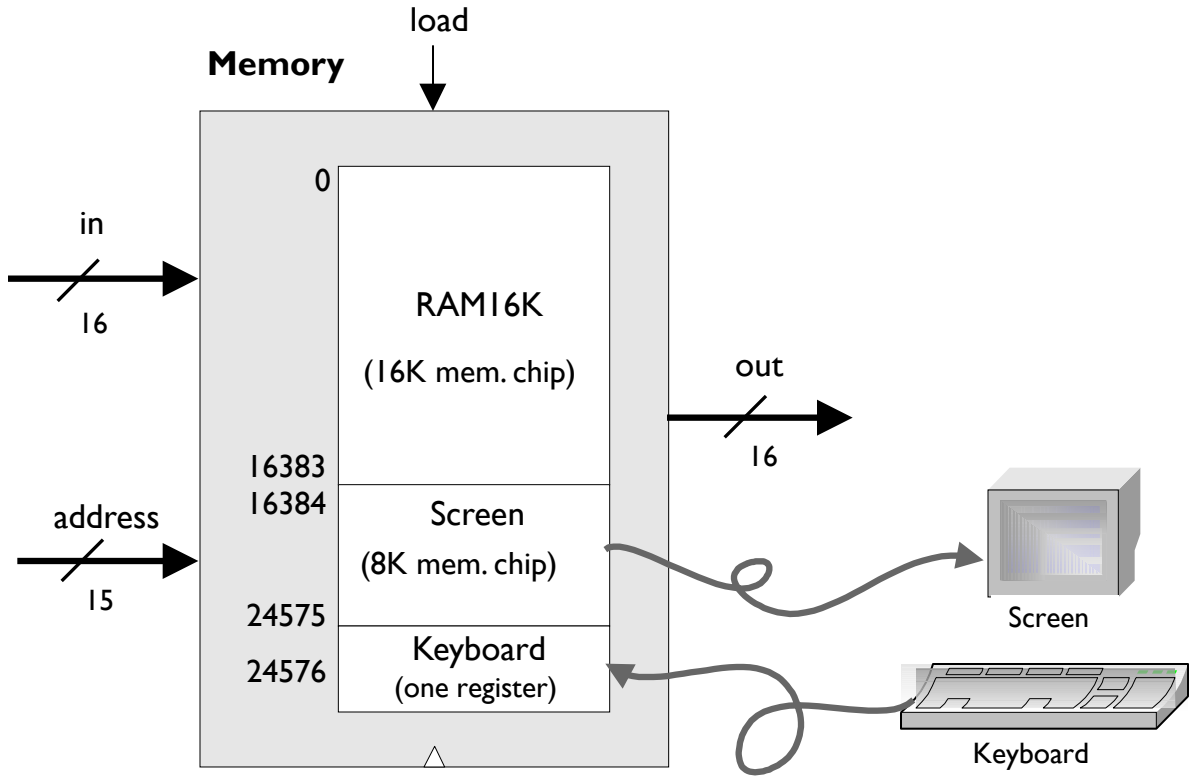
Memory Map

- The RAM, Screen and Keyboard appear together in the same **memory map**
- To the programmer they appear as a single memory device
- Each has a different address range



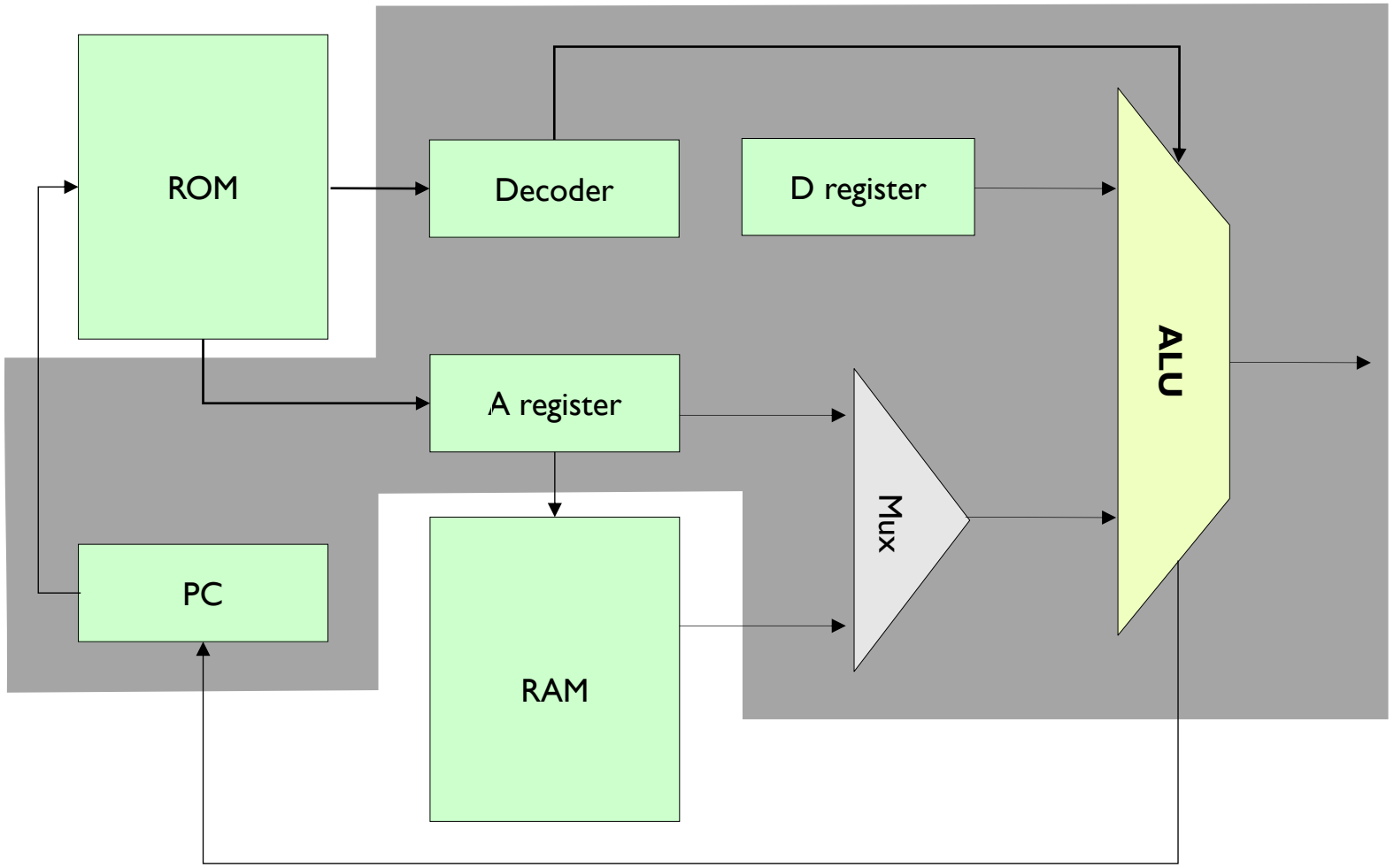
Physical Implementation

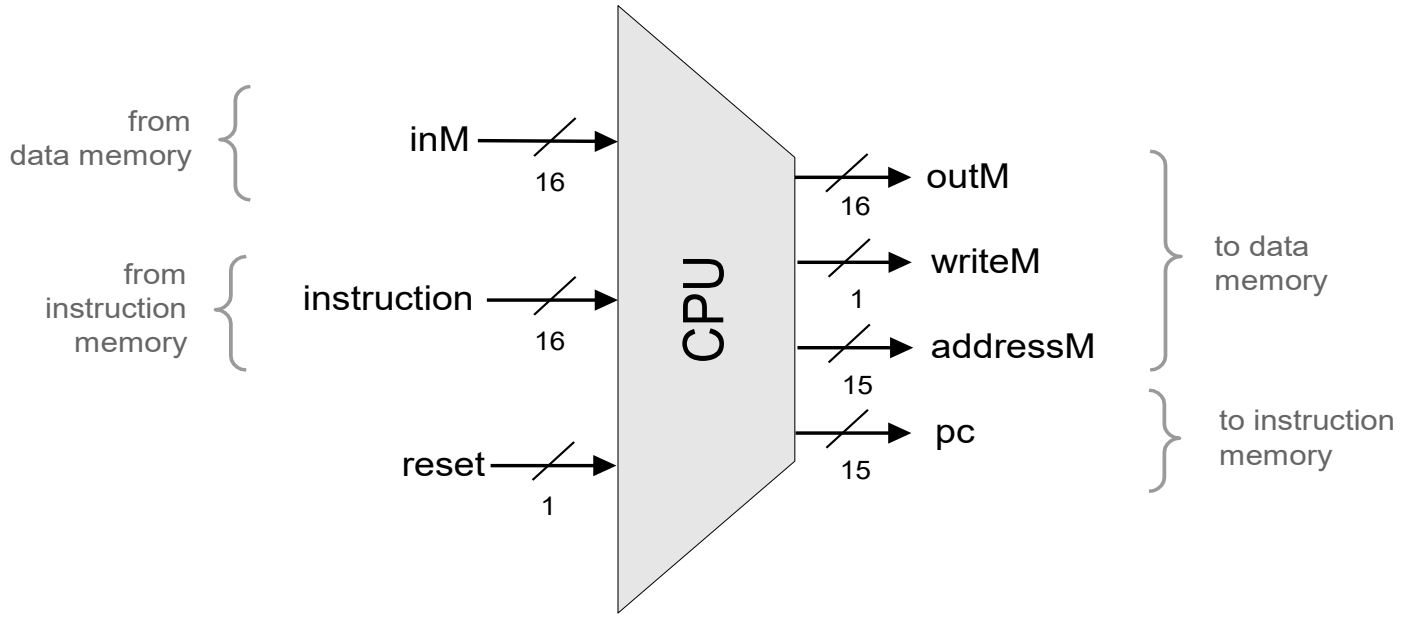
- The **Memory.hdl** chip is essentially a package that integrates the RAM, Screen and Keyboard into a single contiguous address space
- RAM - 0 to 16383 (0x0000 to 0x3FFF)
- Screen – 16384 to 24575 (0x4000 to 0x5FFF)
- Keyboard – 24576 (0x6000)



CPU

Simplified Architecture

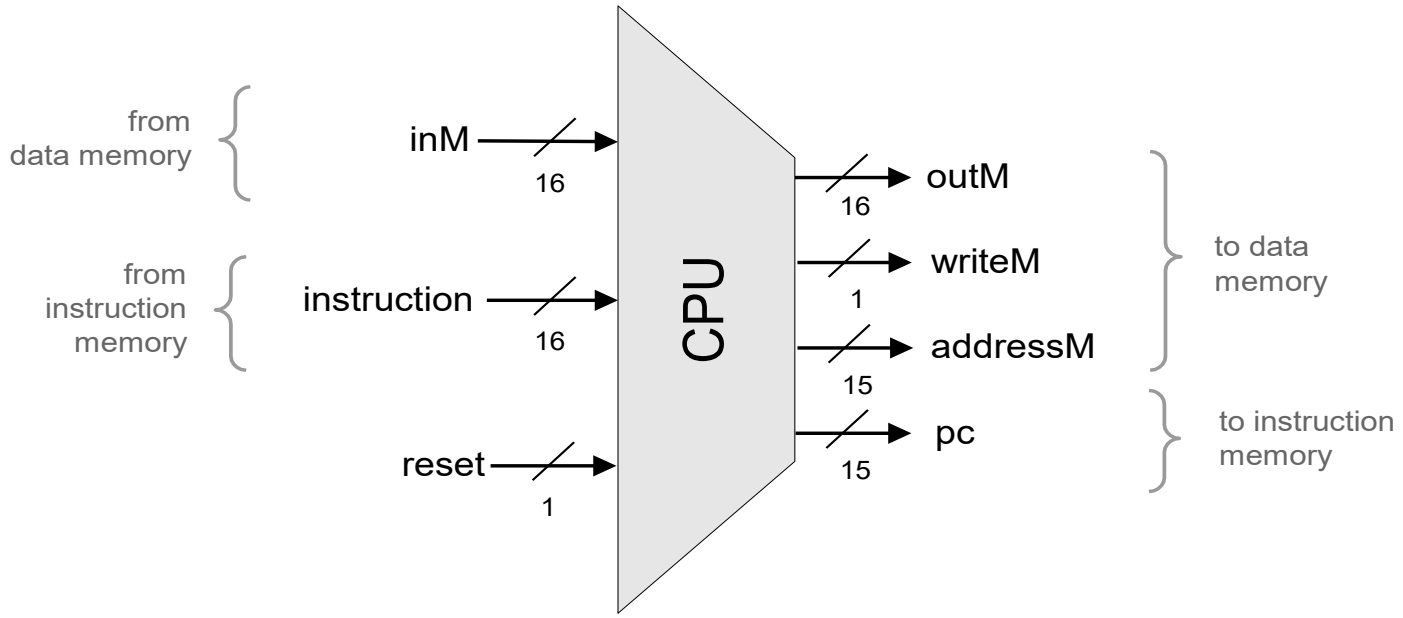




- Internally the CPU is composed of the ALU, **A**-register, **D**-register and PC along with various combinational chips

Execute Logic

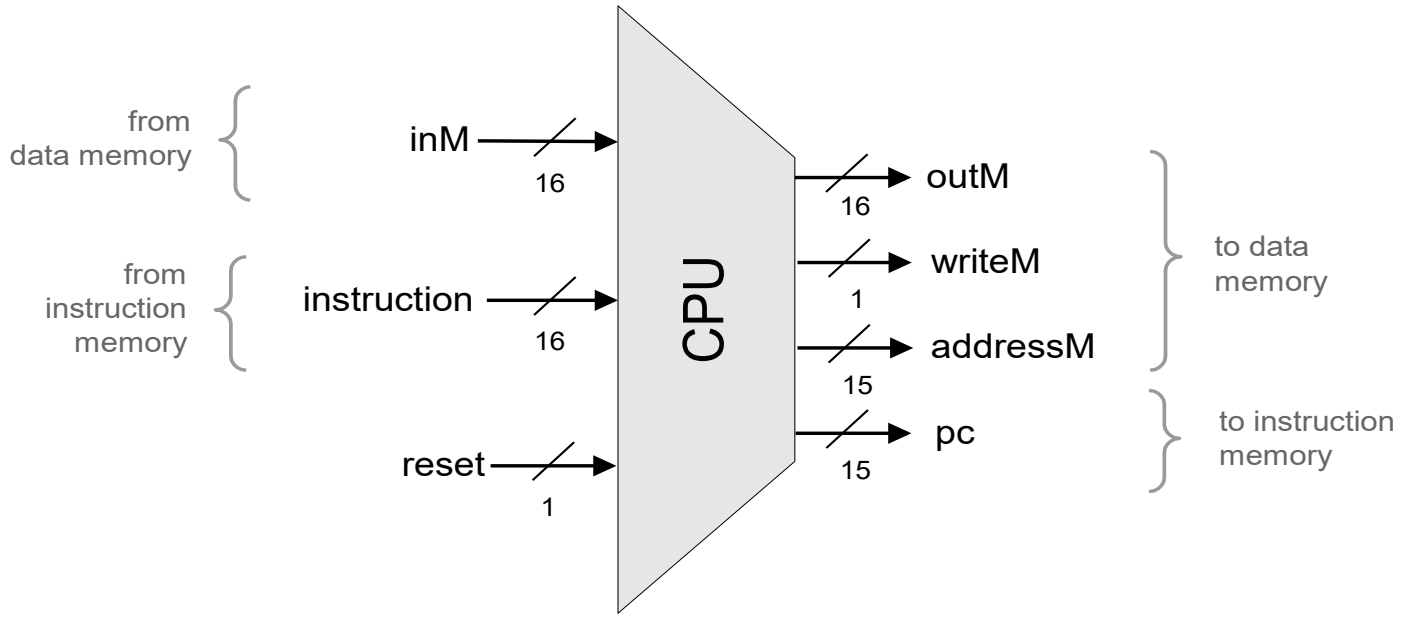
```
dest = comp;jump
```



- If **comp** contains **M** then it is read from **inM**
- If **dest** contains **M** then the ALU output is piped to **outM**, the address (**A-**register) appears on **addressM** and **writeM** is asserted

Fetch Logic

```
dest = comp;jump
```



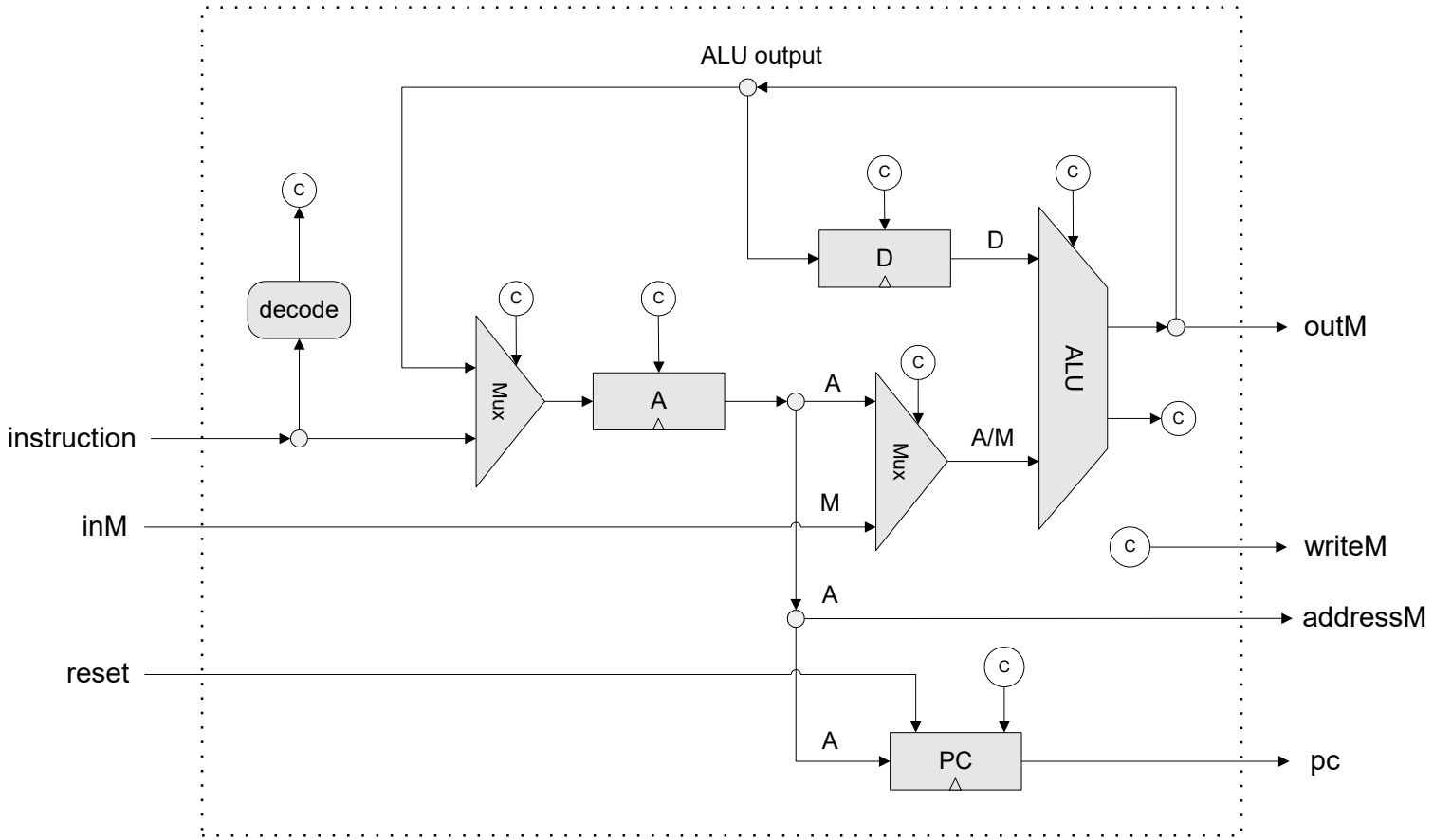
Elements of Computing Systems, Nisan & Schocken, MIT Press, www.nand2tetris.org

- The instruction may contain a **jump** directive as signified by the jump bits in the instruction
- If **reset** is 0 and the ALU flags indicate a jump should occur then the value of **A** is loaded into the **PC**. Otherwise the value in PC is incremented
- If reset is 1 then the value in PC is set to 0.

C-instruction

- Bit 15 is 1 for a **C**-instruction
- Bits 14 and 13 are not used
- The a-bit is used to select between **A** and **M**
- The six c-bits are used to select the computation to carry out
- The three d-bits are used to set the destination to store the result in (a combination of **A**, **D** and **M**)
- The three j-bits specify the jump condition

CPU Implementation

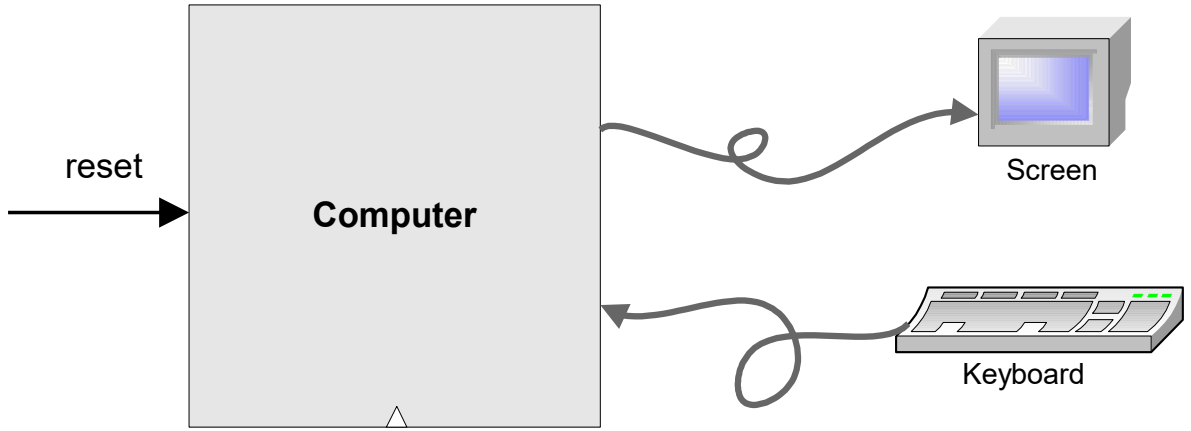


- © indicates a control bit
- The **decode** part is not a chip part, but represents that the control bits be decoded from the instruction somehow

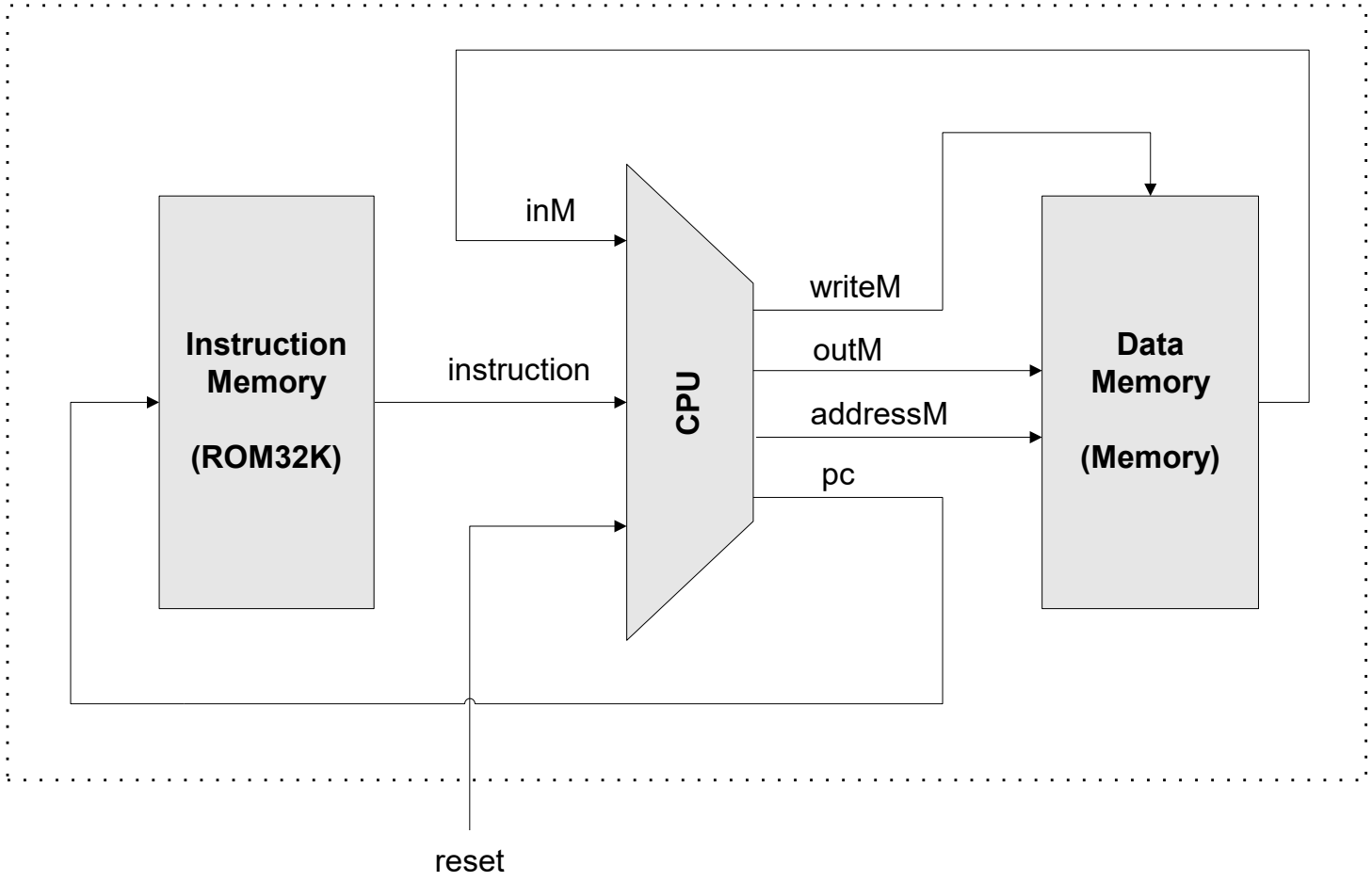
Computer

Interface

- The **Computer** is the top-most chip in the Hack platform
- The pinnacle of your journey from NAND gates to a 'real' computer
- ELEC2655 **Gates to PC**



Implementation



API

Chip Name:	Memory
Inputs:	in[16], load, address[15]
Outputs:	out[16]
Function:	The complete address space of the Hack computer's memory, including RAM and memory-mapped I/O. The chip facilitates read and write operations, as follows: Read: out(t) = Memory[address(t)](t) Write: if load(t-1) then Memory[address(t-1)](t) = in(t-1) In words: the chip always outputs the value stored at the memory location specified by address. If load==1, the in value is loaded into the memory location specified by address. This value becomes available through the out output from the next time step onward. Address space rules: Only the upper 16K+8K+1 words of the Memory chip are used. Access to address>0x6000 is invalid. Access to any address in the range 0x4000-0x5FFF results in accessing the screen memory map. Access to address 0x6000 results in accessing the keyboard memory map. The behavior in these addresses is described in the Screen and Keyboard chip specifications given in the book.
Comment:	

Chip Name:	CPU
Inputs:	inM[16], instruction[16], reset
Outputs:	outM[16], writeM, addressM[16], pc[15]
Function:	The Hack CPU, consisting of an ALU, two registers named A and D, and a program counter named PC. The CPU is designed to fetch and execute instructions written in the Hack machine language. In particular, functions as follows: Executes the inputted instruction according to the Hack machine language specification. The D and A in the language specification refer to CPU-resident registers, while M refers to the external memory location addressed by A, i.e. to Memory[A]. The inM input holds the value of this location. If the current instruction needs to write a value to M, the value is placed in outM, the address of the target location is placed in the addressM output, and the writeM control bit is asserted. (When writeM==0, any value may appear in outM). The outM and writeM outputs are combinational: they are affected instantaneously by the execution of the current instruction. The addressM and pc outputs are clocked: although they are affected by the execution of the current instruction, they commit to their new values only in the next time step. If reset==1 then the CPU jumps to address 0 (i.e. pc is set to 0 in next time step) rather than to the address resulting from executing the current instruction.
Comment:	

Chip Name:	Computer
Inputs:	reset
Outputs:	
Function:	The HACK computer, including CPU, ROM and RAM. When reset is 0, the program stored in the computer's ROM executes. When reset is 1, the execution of the program restarts. Thus, to start a program's execution, reset must be pushed "up" (1) and "down" (0). From this point onward the user is at the mercy of the software. In particular, depending on the program's code, the screen may show some output and the user may be able to interact with the computer via the keyboard.
Comment:	

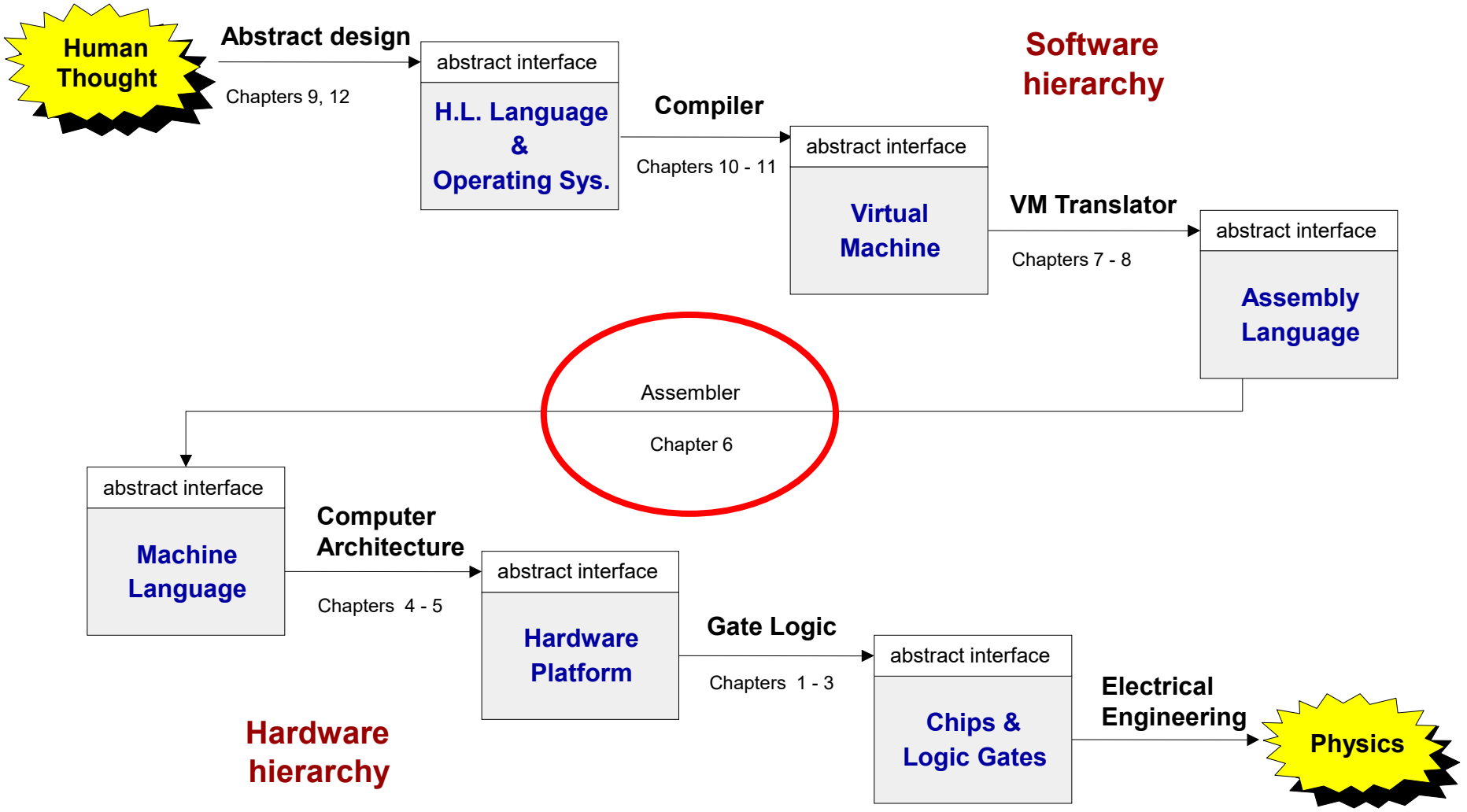
Verified?

Memory ☐ CPU ☐ Computer ☐

2.4 – Assembler

Dr Craig A. Evans

UNIVERSITY OF LEEDS


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Software hierarchy
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Outline

- Machine Language (recap)
- Instruction Format
- Symbols
- Assembly Procedure
- Example

Machine Language

- A machine language program is a series of coded instructions
- e.g. An instruction for a 16-bit machine may be 1010 0011 0010 1001
- We need to understand the **instruction set** of the underlying hardware platform in order to know what the instruction does
- For example, the instruction above may set the value of register 3 to be the sum of register 2 and register 9

- Writing programs in machine code is not feasible
- Therefore **symbolic** versions of machine code also exists
- **Mnemonics** are used as labels for operations and hardware elements
- The symbolic notation is called **assembly language**
- An **assembler** converts the assembly language to binary
- The previous instruction could be written as

```
ADD R3,R2,R9    // R3 = R2 + R9
```

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

.asm

Assembler

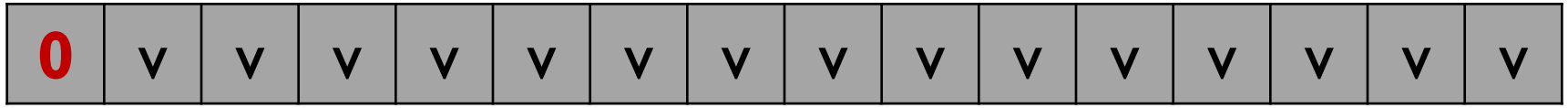
```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

.hack

Instruction Format

A-Instruction

- The **A**-instruction is used to set the **A**-register to a 15-bit value
- Bit 15 is 0 for an **A**-instruction



- e.g. the following command will load 15 into the **A**-register

```
@15
```

- Note the `@` syntax and not assignment `=` e.g. `A = 15` is invalid

C-Instruction

- The **C**-instruction is the workhorse of the Hack computer and handles the computations that are carried out
- It is made up of a **destination** to store the result in (*optional*), the **computation** to carry out and any **jump** condition that is required (*optional*)

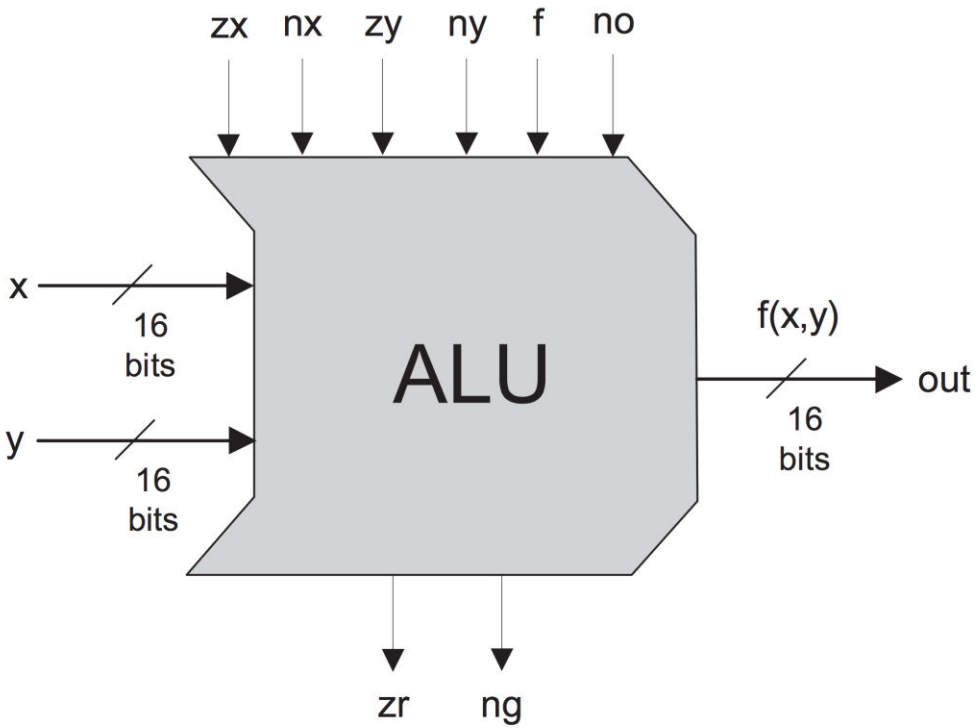
```
dest = comp;jump

comp;jump

dest = comp
```

C-instruction

- Bit 15 is 1 for a **C**-instruction
- Bits 14 and 13 are not used
- The a-bit is used to select between **A** and **M**
- The six c-bits are used to select the computation to carry out
- The three d-bits are used to set the destination to store the result in (a combination of **A**, **D** and **M**)
- The three j-bits specify the jump condition

Computation

- The computation bits determine which operation to carry out
- The operation is carried out by the ALU
- The 6 c-bits are the same are the 6 instruction bits of the ALU
- 28 different instructions in the instruction set



Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

- The result of the computation (ALU output) is fed to one of three locations depending on the d-bits

Jump

- The jump bits tell the CPU what to do next
- Typically it will fetch the next instruction in the ROM
- If a jump condition has been specified in the **C**-instruction then the PC will be loaded with a new address if certain conditions are met
- The address to jump to is the current value stored in **A**
- This must be loaded into **A** using the **A**-instruction before carrying out the **C**-instruction with a jump specified
- Whether the jump occurs or not depends on whether the output of the ALU meets the jump condition

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Symbols

- Without symbols the assembly process is trivial
- It is simply decoding the A- and C-instructions according to the Hack machine language specification
- When the assembly code has symbols the process becomes slightly more difficult and requires **two passes** through the code
- A **look-up** table is created that contains all the **pre-defined** symbols and their numeric values

Pre-defined Symbols

Symbol	Value
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbols

- User-defined labels **(XXX)** are added to the look-up table on the first-pass of the code
- User-defined symbols **@xxx** are added on the second pass and are automatically allocated an address in RAM (starting at 16)

Assembly Procedure

- Initialisation
  - Create a look-up table and populate it with the pre-defined symbols
  - Strip out comments and white-space
- First-pass
  - Go through the source code line-by-line (without generating code)
  - For each label declaration **(LABEL)** that appears add the pair **<LABEL,n>** to the look-up table
- Second-pass
  - Go through the source code line-by-line
  - Translate **C**- and **A**-instructions with literals to binary instructions
  - If the line is a @symbol then look in the look-up table
    - If it is found, replace with numeric value and then translate to binary as normal
    - If not found, it must be a variable, so add to look-up table at the next available RAM address and translate to binary
    - RAM addresses for variables start at 16

Example (no symbols)

```
0    @0
1    D=M
2    @1
3    D=D+M
4    @9
5    D=D+A
6    @2
7    M=D
8
9
10
11
```

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Example (with symbols)

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

```
0    @i
1    M=1
2    @sum
3    M=0
(LOOP)
4    @i
5    D=M
6    @5
7    D=D-A
8    @CONT
9    D;JGT
10   @i
```

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

```
11  D=M
12  @sum
13  M=D+M
14  @i
15  M=M+1
16  @LOOP
17  0;JMP
(CONT)
18
19
20
21
```

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

First-pass

- Go through code and look for **(LABELS)**
- Add the **<LABEL,n>** pair to the look-up table
- **n** is the **next** instruction (line) number

Second-pass

- Go through line-by-line
- Translate **C**- and **A**-instructions with literals to binary instructions
- If a line is a @symbol then look in look-up table
- Replace with numeric value if found
- If not found then add to next RAM slot (starting at 16)
- Replace with numeric value
- Translate to binary instruction

Testing

- The assembled code should be typed into a text file with **.hack** extension
- Can then be loaded into the CPU Emulator
- Can also use the supplied **Assembler** tool to assemble the .asm source file and **compare** to your manually assembled .hack file
  - Load the .asm source file (File/Load Source File)
  - Load your manually assembled .hack file (File/Load Comparison File)
  - Step through the assembly procedure and it will compare the files line by line

Assembly Tables

Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Comp

zx	nx	zy	ny	f	no	comp mnemonic		alu
c_1	c_2	c_3	c_4	c_5	c_6	a=0	a=1	
1	0	1	0	1	0	0		**0**
1	1	1	1	1	1	1		**1**
1	1	1	0	1	0	-1		**-1**
0	0	1	1	0	0	D		**x**
1	1	0	0	0	0	A	M	**y**
0	0	1	1	0	1	!D		**!x**
1	1	0	0	0	1	!A	!M	**!y**
0	0	1	1	1	1	-D		**-x**
1	1	0	0	1	1	-A	-M	**-y**
0	1	1	1	1	1	D+1		**x+1**
1	1	0	1	1	1	A+1	M+1	**y+1**
0	0	1	1	1	0	D-1		**x-1**
1	1	0	0	1	0	A-1	M-1	**y-1**
0	0	0	0	1	0	D+A	D+M	**x+y**
0	1	0	0	1	1	D-A	D-M	**x-y**
0	0	0	1	1	1	A-D	M-D	**y-x**
0	0	0	0	0	0	D&A	D&M	**x&y**
0	1	0	1	0	1	D|A	D|M	**x|y**

Destination

d_1	d_2	d_3	Mnemonic	Destination
0	0	0	`null`	Not stored anywhere
0	0	1	`M`	Memory[A]
0	1	0	`D`	D register
0	1	1	`MD`	Memory[A] and D
1	0	0	`A`	A register
1	0	1	`AM`	A register and Memory[A]
1	1	0	`AD`	A register and D register
1	1	1	`AMD`	A register, Memory[A] and D register

Jump

j_1 out < 0	j_2 out = 0	j_3 out > 0	**Mnemonic**	**Destination**
0	0	0	`null`	No jump
0	0	1	`JGT`	If out > 0 then jump
0	1	0	`JEQ`	If out = 0 then jump
0	1	1	`JGE`	If out ≥ 0 then jump
1	0	0	`JLT`	If out < 0 then jump
1	0	1	`JNE`	If out ≠ 0 then jump
1	1	0	`JLE`	If out ≤ 0 then jump
1	1	1	`JMP`	Jump

Lookup Table

Pre-defined Symbol	Number
SCREEN	16384
KBD	24576
R0 to R15	0 to 15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

User-defined Symbol	Number

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3