School of Computer Science

COMP SCI 2000 Computer Systems

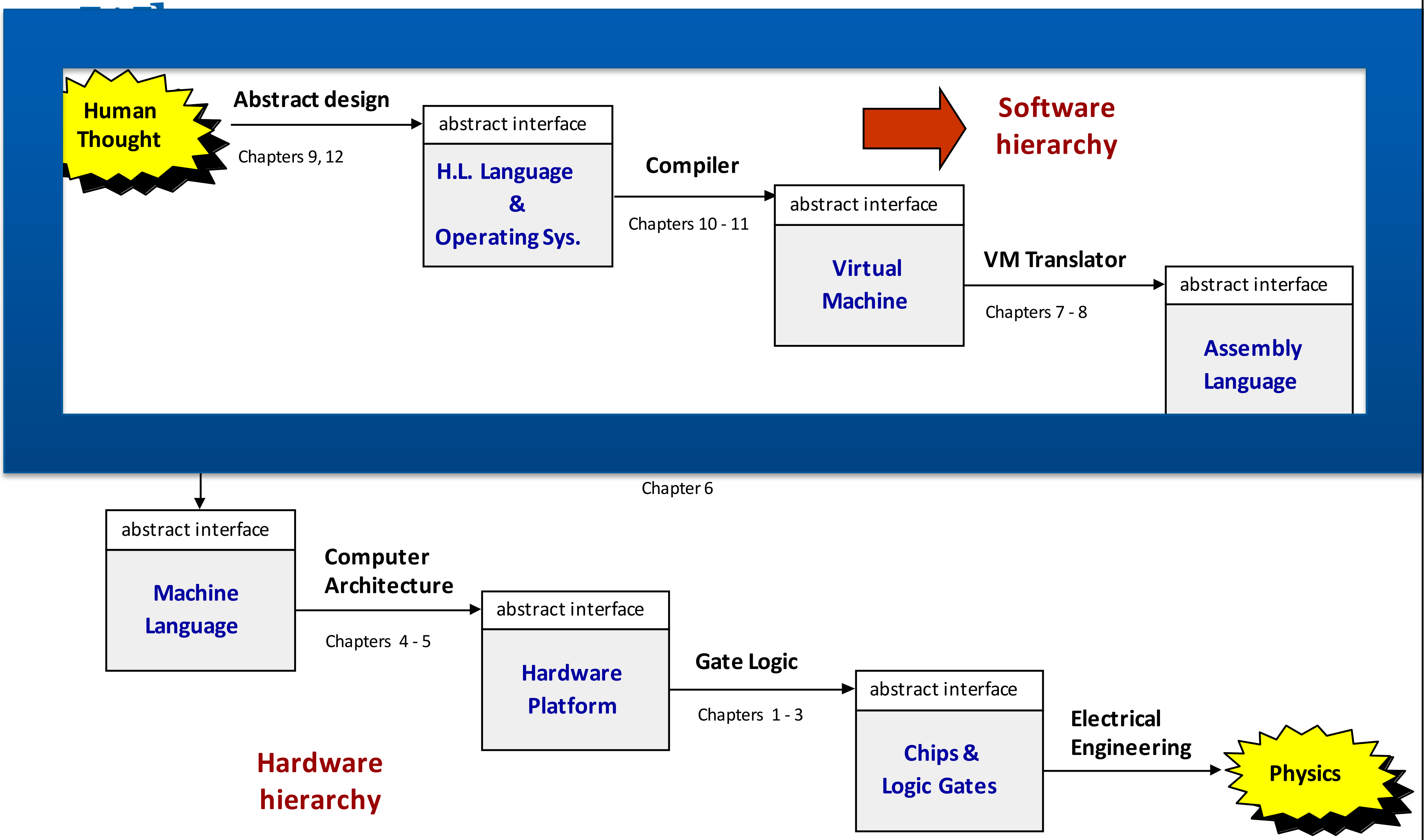
Lecture 10

Review – Last week

- We were talking hardware:
  - History of Architecture
  - Memory and I/O
  - The HACK machine
  - The CPU and basic computers
- That was the last week of talking hardware in *lectures*.

What we’re doing now

- This week we’re going to start talking about building an assembler.
- Assembly language is the first step on the ladder to high-level languages.
- A vital bridge between the hardware and the software.



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

Chapter 6

abstract interface

Machine Language

Computer Architecture

Chapters 4 - 5

abstract interface

Hardware Platform

Gate Logic

Chapters 1 - 3

abstract interface

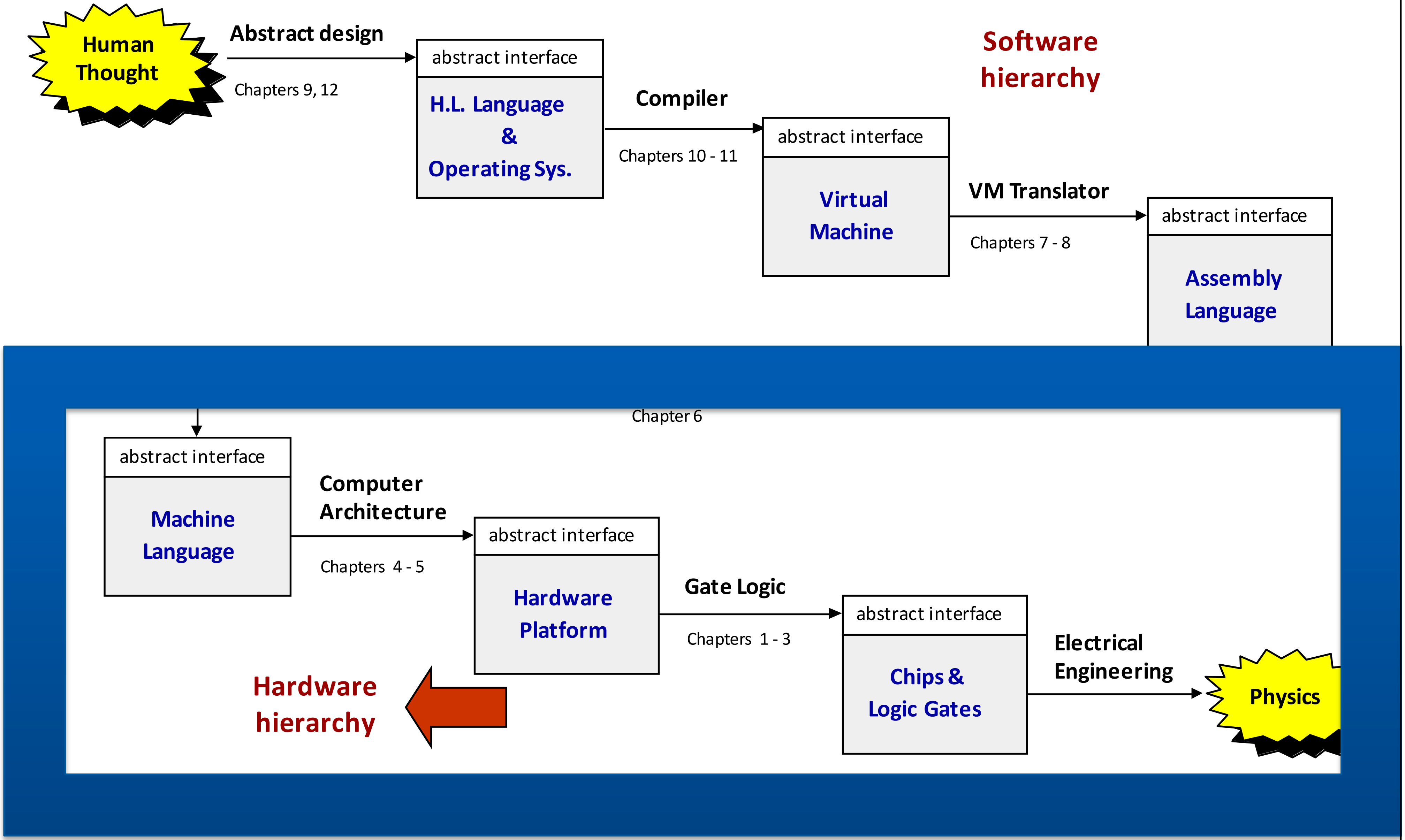
Chips & Logic Gates

Electrical Engineering

Physics

Hardware hierarchy

Where we are at:



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

Chapter 6

abstract interface

Machine Language

Computer Architecture

Chapters 4 - 5

abstract interface

Hardware Platform

Gate Logic

Chapters 1 - 3

abstract interface

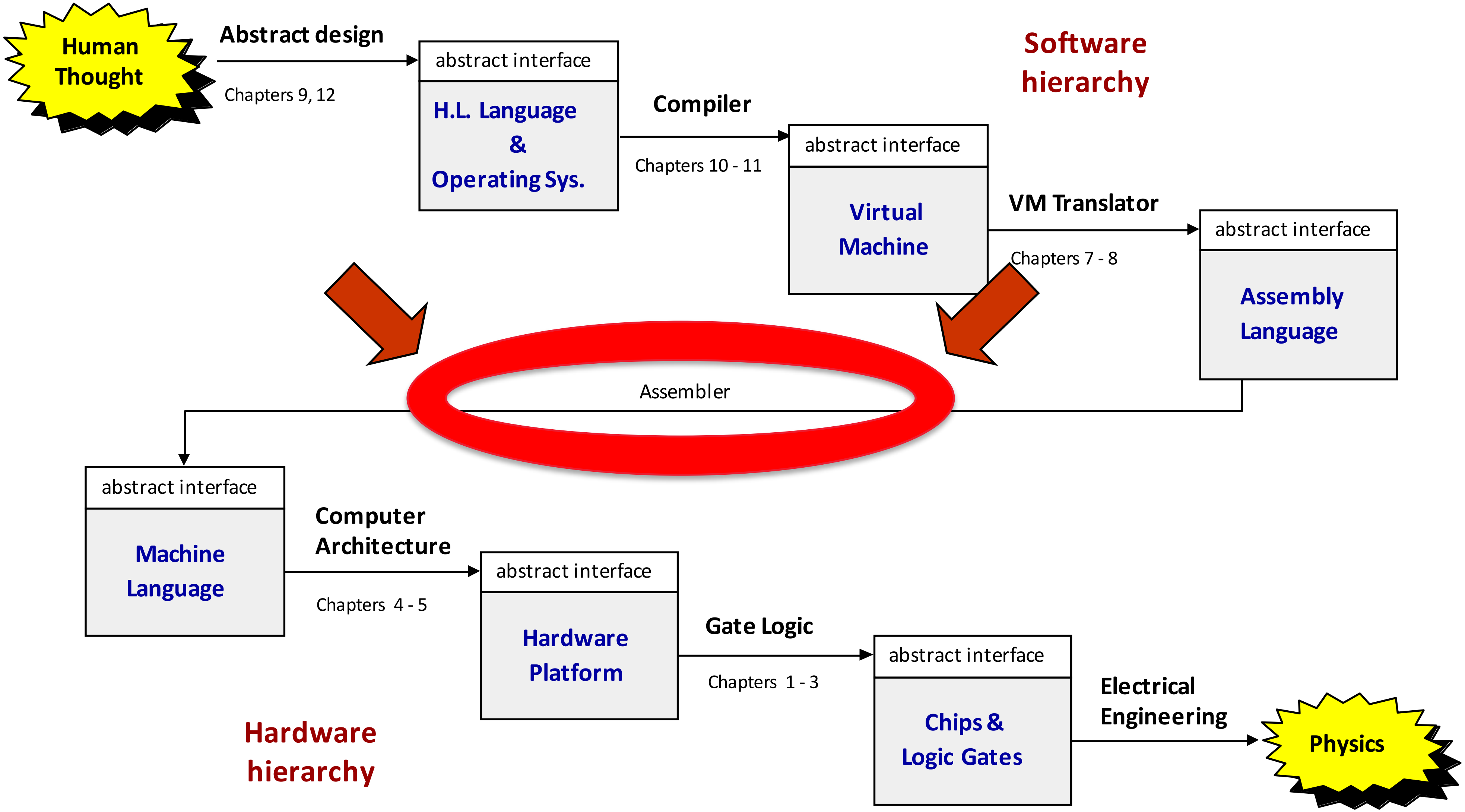
Chips & Logic Gates

Electrical Engineering

Physics

Hardware hierarchy

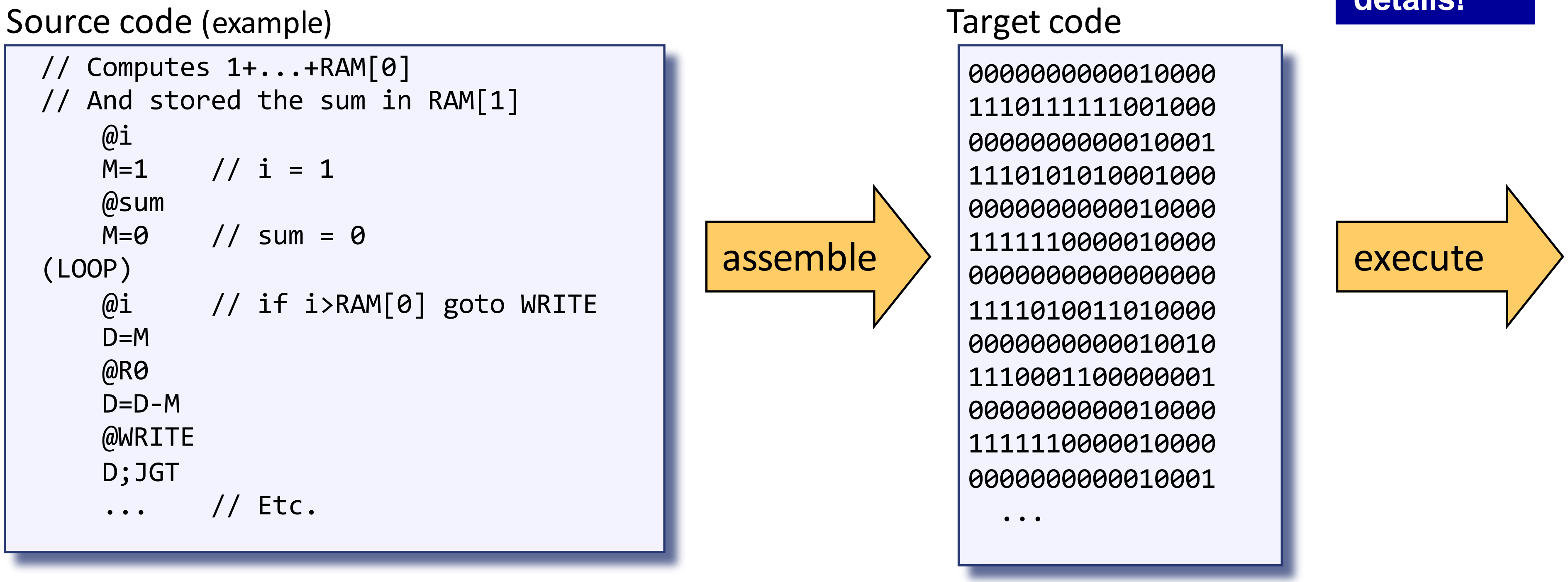
Where we are at:



What is an assembler?

- Translator for a simple language.
- Contains most of the tricks and techniques required to make compilers work.

For now, ignore all details!



Source code (example)

```
// 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
    ...    // Etc.
```

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
  ...
```

The program translation challenge

- Extract the program's semantics from the source program, using the syntax rules of the source language
- Re-express the program's semantics in the target language, using the syntax rules of the target language

Assembler = simple translator

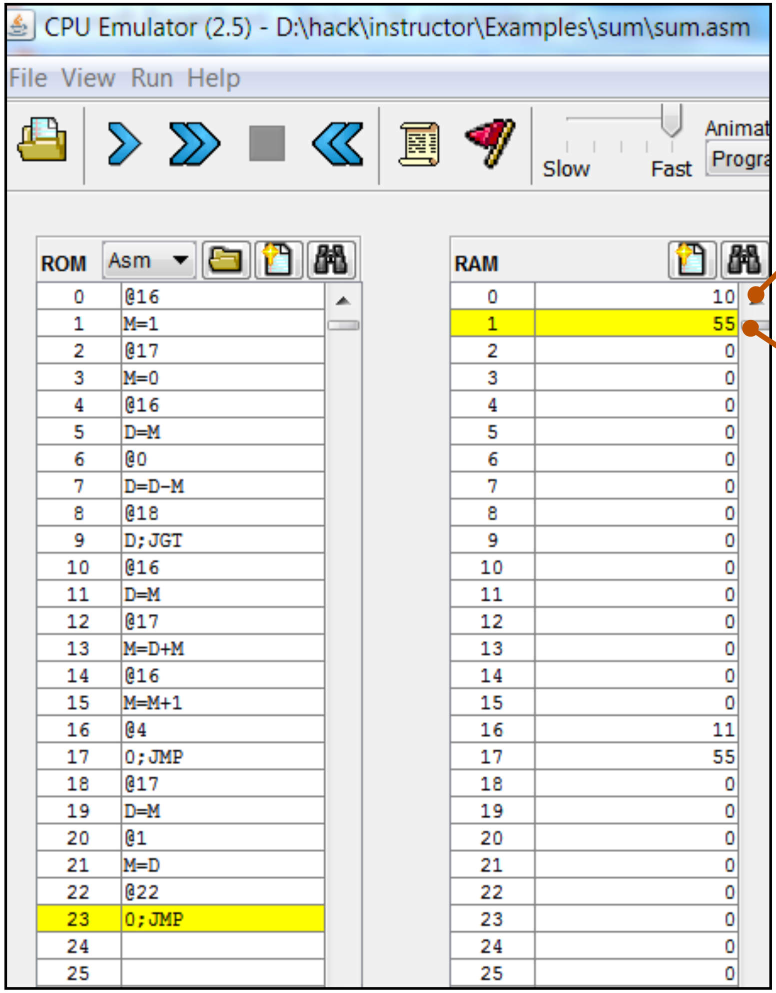
- Translates each assembly command into one or more binary machine instructions
- Handles symbols (e.g. i, sum, LOOP, …).

Revisiting Hack low-level programming

Assembly program (sum.asm)

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

CPU emulator screen shot after running this program



CPU Emulator (2.5) - D:\hack\instructor\Examples\sum\sum.asm

ROM

0	@16
1	M=1
2	@17
3	M=0
4	@16
5	D=M
6	@0
7	D=D-M
8	@18
9	D;JGT
10	@16
11	D=M
12	@17
13	M=D+M
14	@16
15	M=M+1
16	@4
17	0;JMP
18	@17
19	D=M
20	@1
21	M=D
22	@22
23	0;JMP
24	
25	

RAM

0	10
1	55
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	11
17	55
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0

user supplied input

program generated output

The CPU emulator allows loading and executing symbolic Hack code. It resolves all the symbolic symbols to memory locations, and executes the code.

Assembler's view of an assembly program

Assembly program

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Assembly program =

a stream of text lines, each being one of the following:

- ☐ A-instruction
- ☐ C-instruction
- ☐ Symbol declaration: (SYMBOL)
- ☐ Comment or white space:
  // comment

The challenge:

Translate the program into a sequence of 16-bit instructions that can be executed by the target hardware platform.

Worksheet 10 question 1.

Translating / assembling A-instructions

Symbolic: `@`*value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

value (v = 0 or 1)

Binary: 0 v v v v v v v v v v v v v v v

Translation to binary:

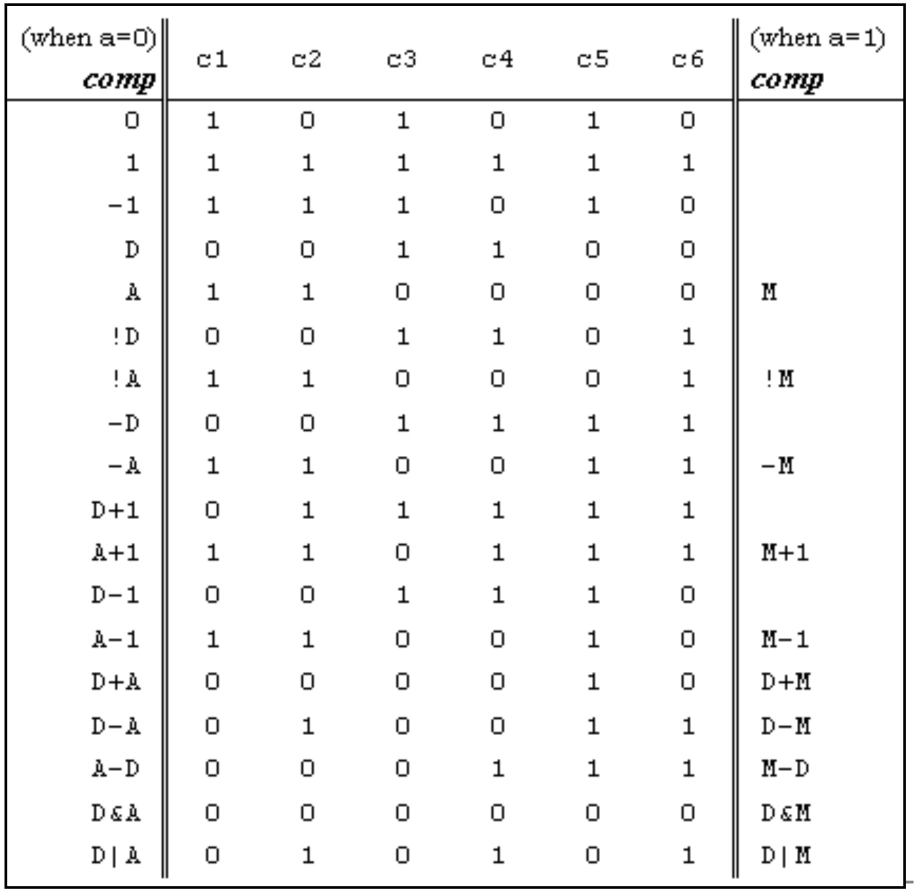
- If *value* is a non-negative decimal number, simple
- If *value* is a symbol...

Translating / assembling C-instructions

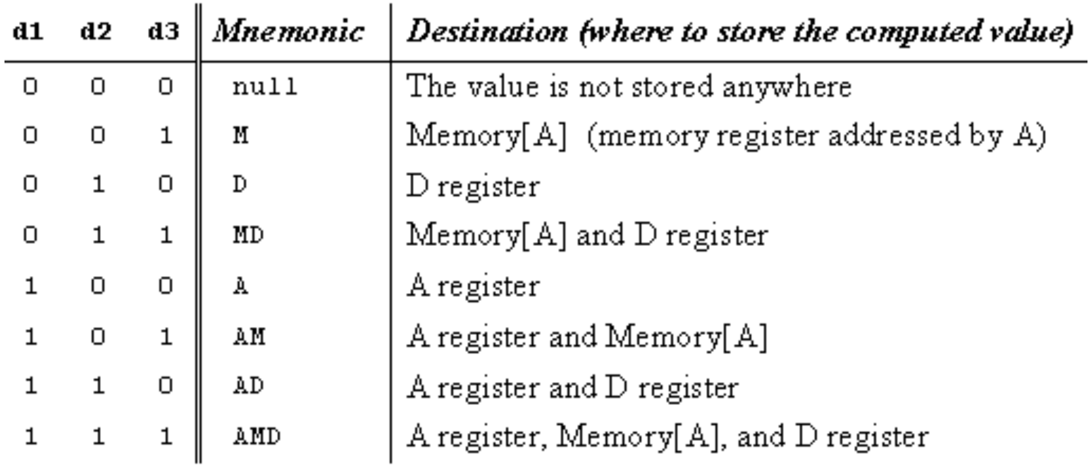
Symbolic: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is ommitted;
// If *jump* is empty, the ";" is omitted.

comp *dest* *jump*

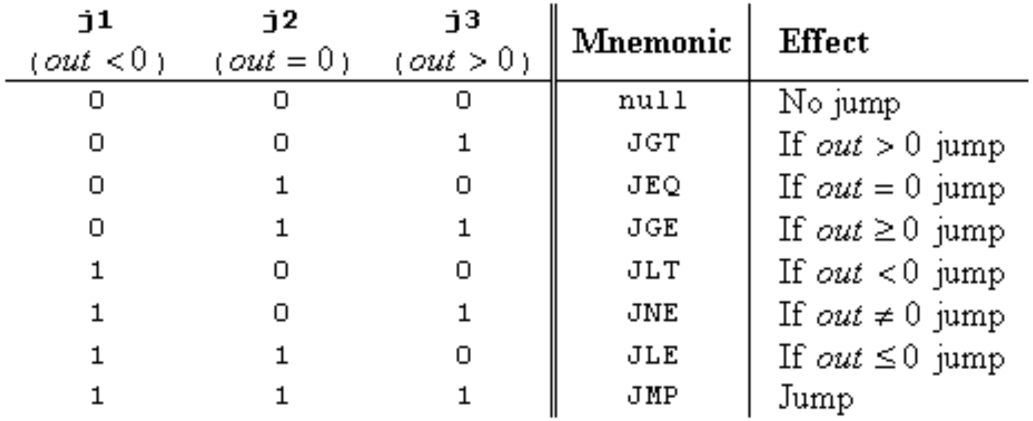
Binary: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3



(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M



d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
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



j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

Translation to binary: relatively simple!

The overall assembly logic

Assembly program

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

<u>For each (real) command</u>

- Parse the command,
  i.e. break it into its underlying fields
- A-instruction: replace the symbolic reference (if any) with the corresponding memory address, which is a number

  (how to do it, later)
- C-instruction: for each field in the instruction, generate the corresponding binary code
- Assemble the translated binary codes into a complete 16-bit machine instruction
- Write the 16-bit instruction to the output file.

Worksheet 10 question 2.

Handling Symbols

- Also called *symbol resolution*
- Assembly programs typically have many symbols:
    - Labels that mark destinations of goto commands
    - Labels that mark special memory locations
    - Variables

- These symbols fall into two categories:
    - User–defined symbols (created by programmers)
    - Pre-defined symbols (used by the Hack platform).

Handling Symbols

- 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.
- 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.
- Conventions
  - Hack programmers use lower-case and upper-case to represent variable and label names, respectively.

Typical symbolic Hack assembly code:

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @x
    M=D
(LOOP)
    @x
    A=M
    M=-1
    @x
    D=M
    @32
    D=D+A
    @x
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Handling Symbols

- Virtual registers
    - The symbols R0,…, R15 are automatically predefined to refer to RAM addresses 0,…,15.
- I/O pointers
    - The symbols SCREEN and KBD are automatically predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the *screen* and *keyboard* memory maps).
- VM control pointers
    - The symbols SP, LCL, ARG, THIS, and THAT (that don’t appear in the code example on the right) are automatically predefined to refer to RAM addresses 0 to 4, respectively.

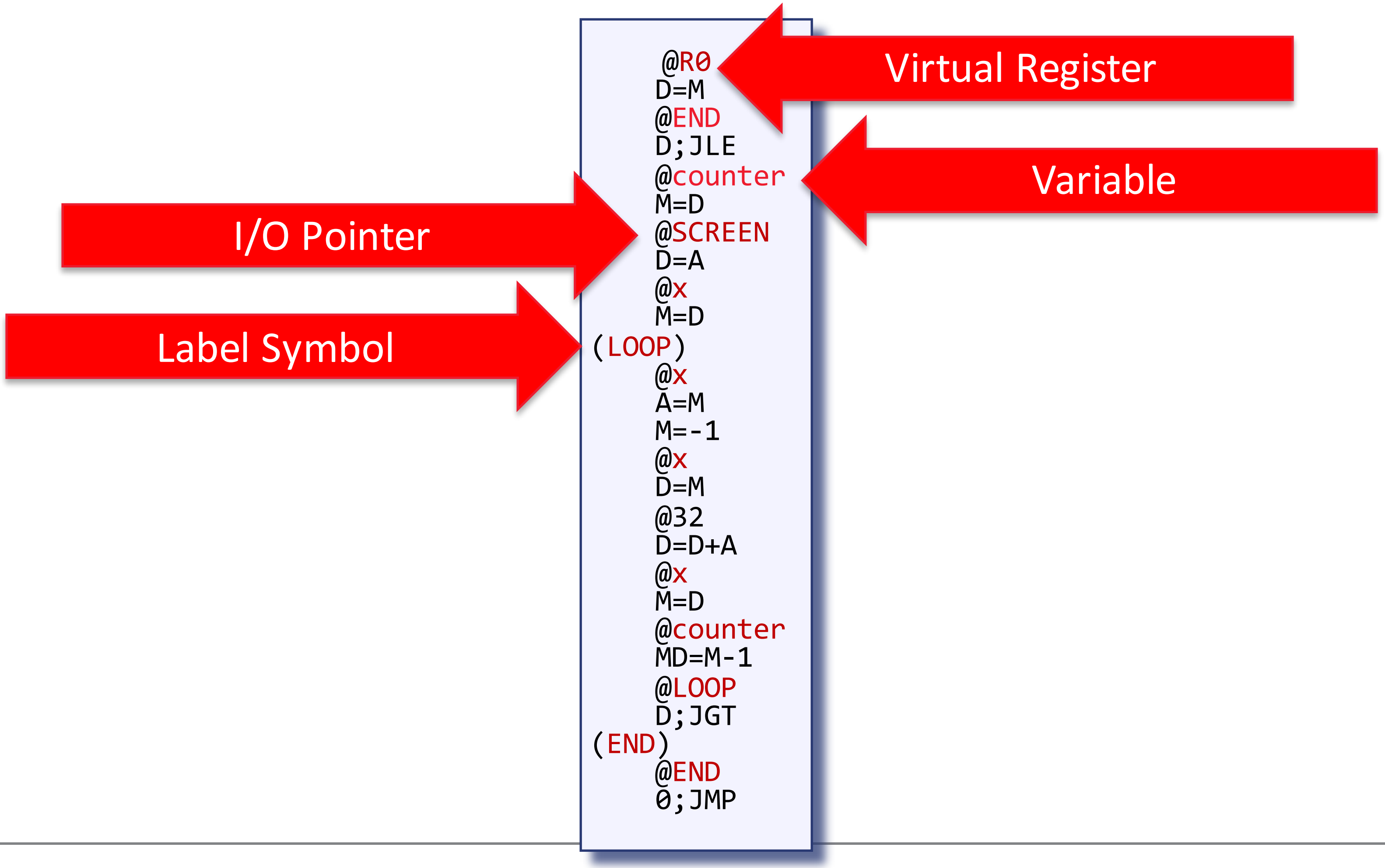
Typical symbolic Hack assembly code:

```
   @R0
   D=M
   @END
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @x
   M=D
(LOOP)
   @x
   A=M
   M=-1
   @x
   D=M
   @32
   D=D+A
   @x
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(END)
   @END
   0;JMP
```

What types of symbols are there?

- Look at some HACK code – how are we using symbols?

Some symbolic Hack assembly code:



```
   @R0
   D=M
   @END
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @x
   M=D
(LOOP)
   @x
   A=M
   M=-1
   @x
   D=M
   @32
   D=D+A
   @x
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(END)
   @END
   0;JMP
```

Virtual Register

Variable

I/O Pointer

Label Symbol

Your turn!

```
// 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
```

- When this program has finished assembling, what does the symbol table look like for:
  - The registers?
  - The variable and label symbols?
- Use the format:

```
R0          Value
R1          Value
…
LOOP        Value
….
```

Worksheet 10 question 3.

Next

- Completing the assembler
- Questions?