We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

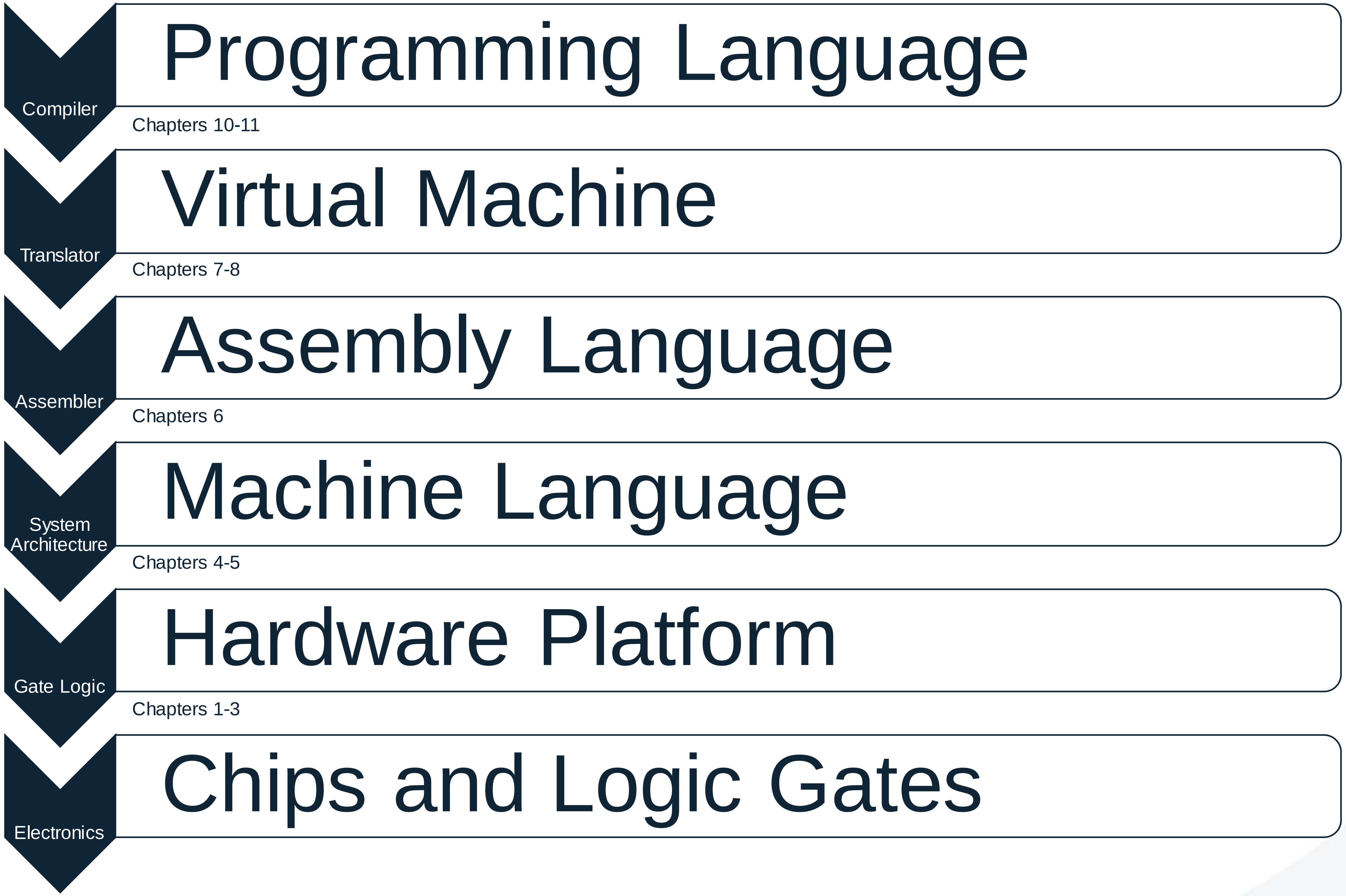
We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

THE UNIVERSITY of ADELAIDE

Computer Systems

Lecture 05: Machine Language and the Assembler

Review: The whole system



Programming Language

Compiler

Chapters 10-11

Virtual Machine

Translator

Chapters 7-8

Assembly Language

Assembler

Chapters 6

Machine Language

System Architecture

Chapters 4-5

Hardware Platform

Gate Logic

Chapters 1-3

Chips and Logic Gates

Electronics

Review: The A-instruction

```
@value        // A ← value
```

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

Used for:

Coding example:

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]
```

Review: 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 :

- ❑ 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.**

Review: 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:

j	3012
sum	4500
q	3812
arr	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 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

Complete program example

C language code:

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

Hack assembly convention:

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

Hack assembly code:

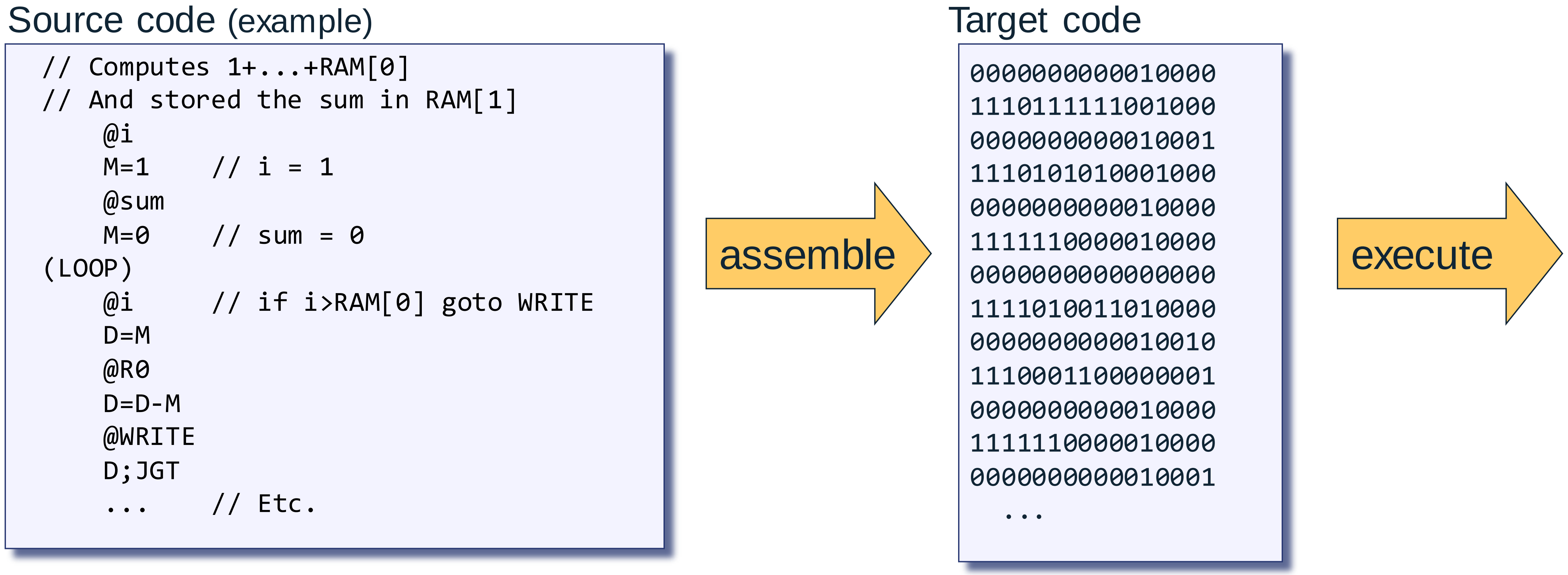
```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM location
      M=0     // sum=0
(LOOP)
      @i
      D=M      // D = i
      @100
      D=D-A    // D = i - 100
      @END
      D;JGT    // If (i-100) > 0 goto END
   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

The Assembler

THE UNIVERSITY
of ADELAIDE

What is an assembler?

- Translates simple, human-readable form of machine language to binary instructions.
- Contains most of the tricks and techniques required to make compilers work.



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

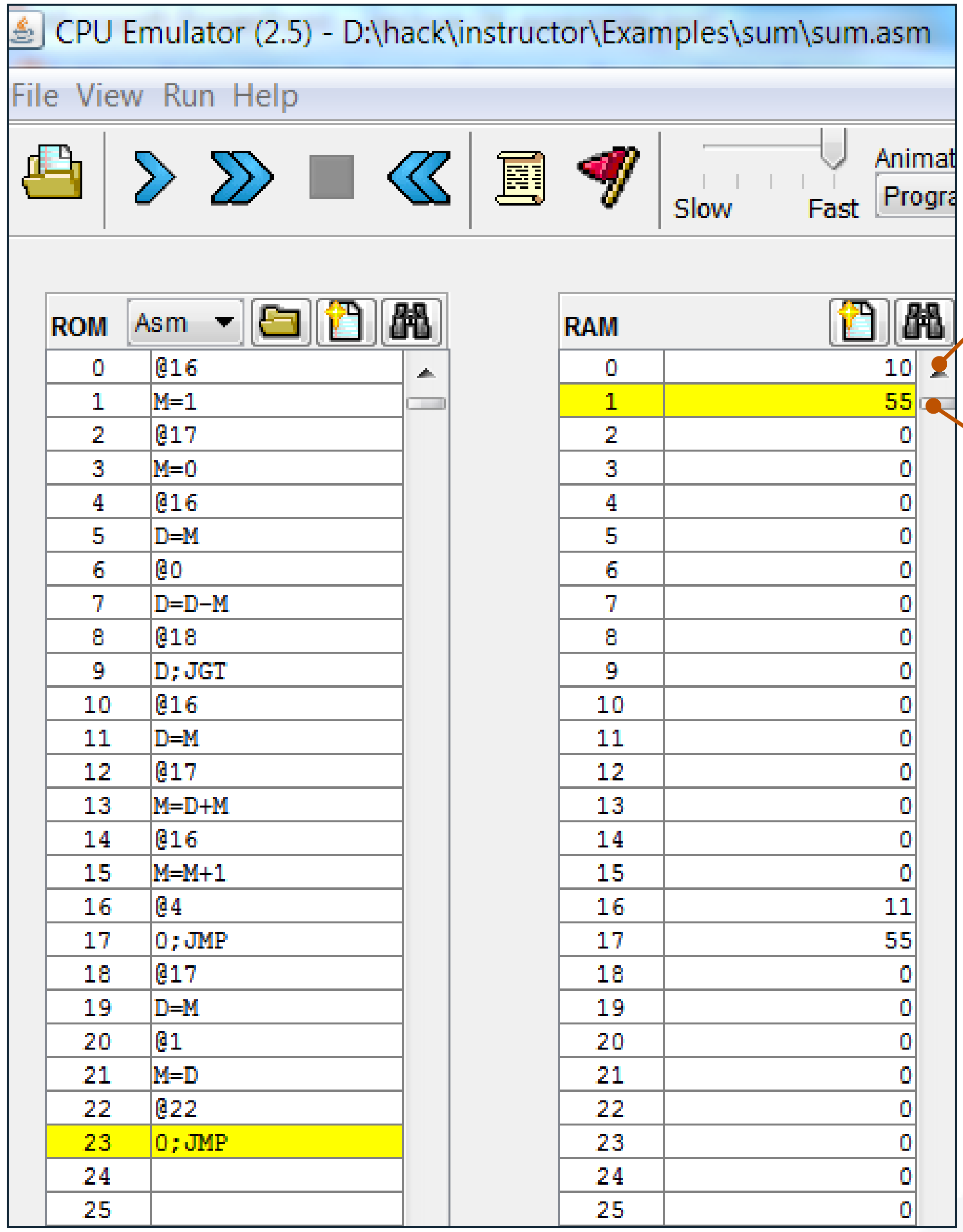
- Translates each assembly command into one or more binary machine instructions
- Handles symbols (e.g. i, sum, LOOP, …).

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	Asm
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.

Translating / assembling A-instructions

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

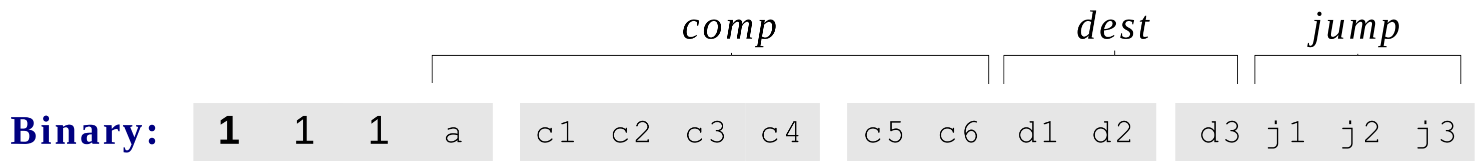
$$\mathbf{Binary:}\quad 0\;\overbrace{v\;v\;v\quad v\;v\;v\;v\quad v\;v\;v\;v\quad v\;v\;v\;v}^{\textit{value}\ (v\,=\,0\ \text{or}\ 1)}$$

Translation to binary:

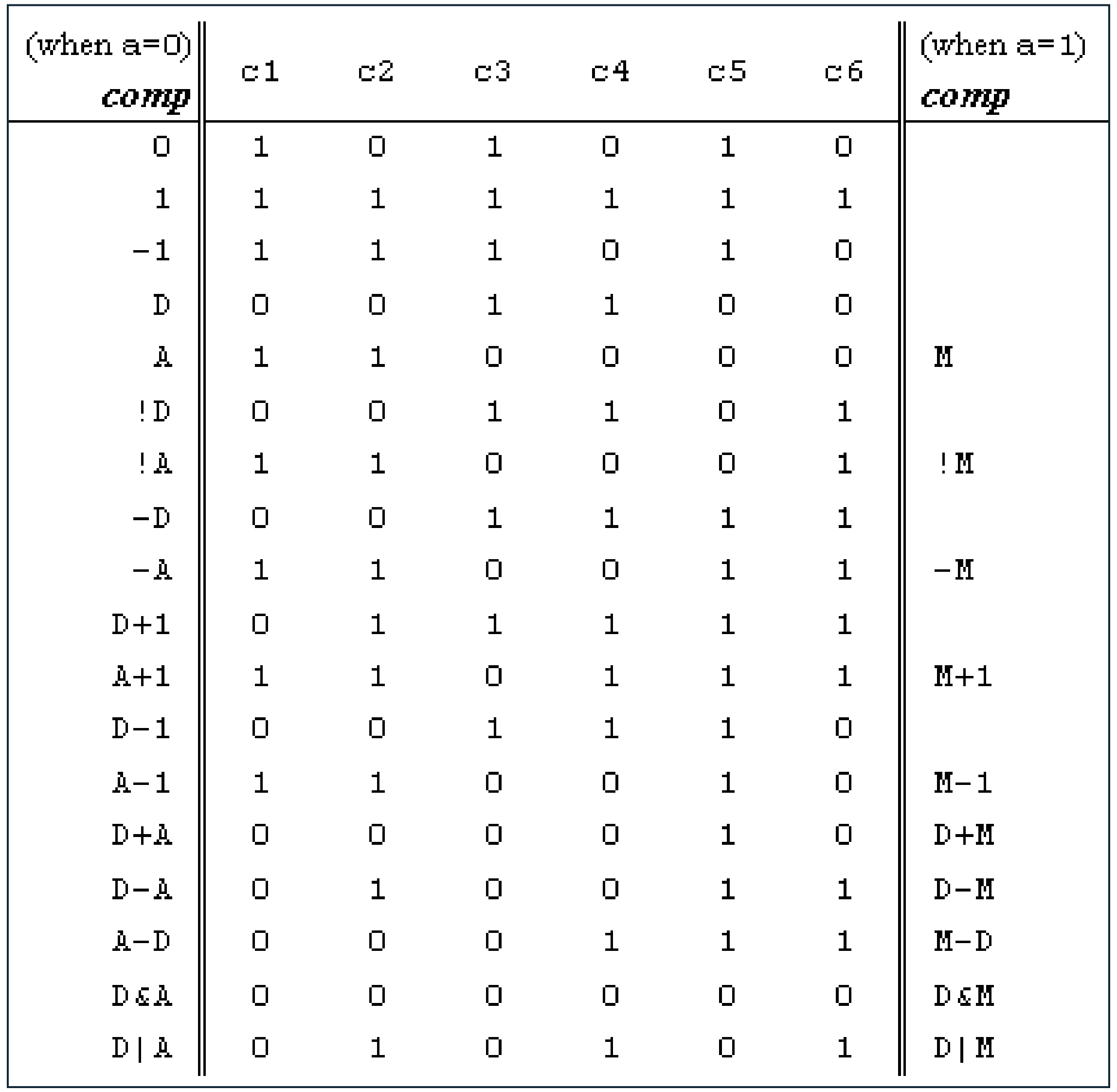
- ❑ If *value* is a non-negative decimal number, simple
- ❑ If *value* is a symbol…

Translating / assembling C-instructions

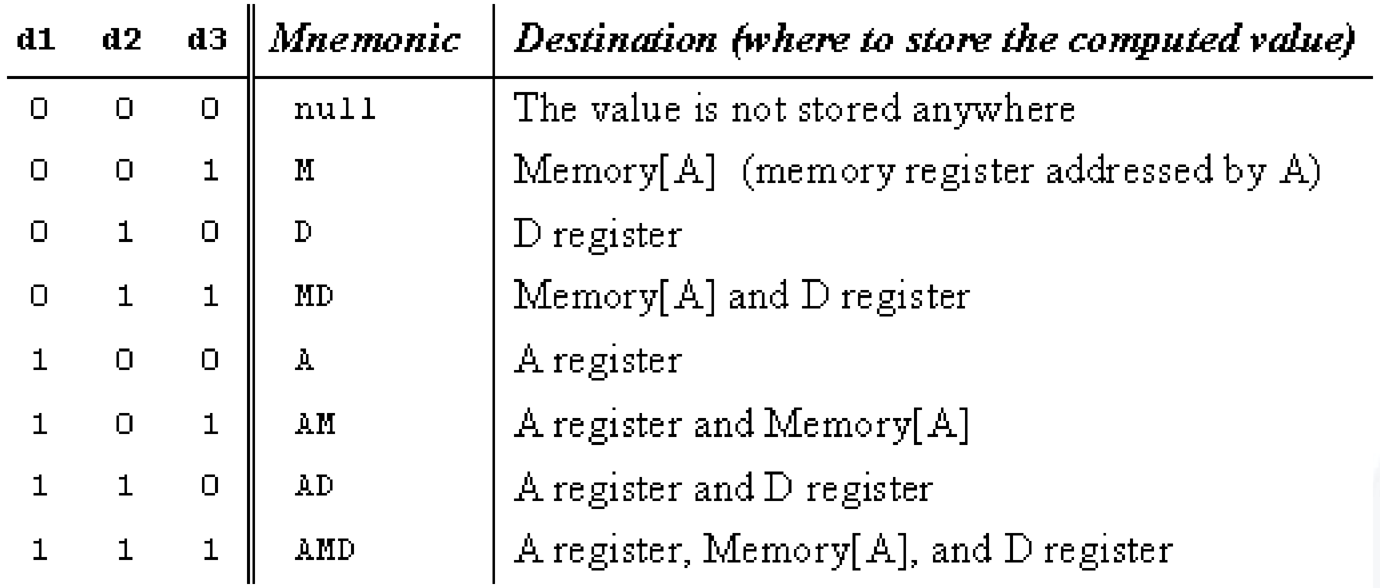
Symbolic: *dest*=*comp*;*jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is ommitted;
// If *jump* is empty, the ";" is omitted.



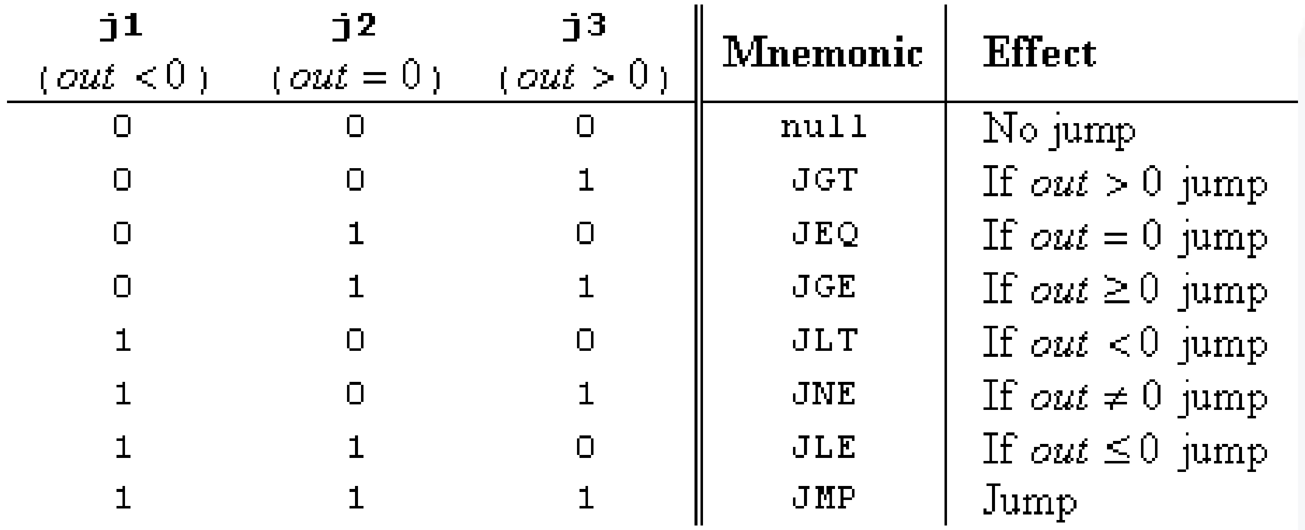
Binary: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3



(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* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

THE UNIVERSITY of ADELAIDE

Translating basic instructions 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 number

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

Example Assembly

- Assume BOB has value 31

@7	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
D=0	1	1	1	0	1	0	1	0	1	0	0	1	0	0	0	0
0;JMP	1	1	1	0	1	0	1	0	1	0	0	0	0	1	1	1
A=D&M;JLT	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	0
@BOB	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
AD=!M	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0

Handling Symbols

(Also called *symbol resolution*)

Assembly programs typically have many symbols:

- Labels that mark destinations of jump commands
- Labels that mark special memory locations
- Variables

In Hack assembler there are three categories:

- Pre-defined symbols (used by the Hack platform)
- Labels (User–defined symbols)
- Variables (User-defined symbols)

Predefined Symbols

- Are initialised before the assembler starts and have no special significance on their own.
- Key predefined symbols include:
  - **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 (why these numbers?).
  - **VM control pointers**
    The symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT` are predefined to refer to RAM addresses `0` to `4`.

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
```

THE UNIVERSITY of ADELAIDE

User Defined Symbols

Label symbols

- Used to label destinations of jump 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.
- The assembler automatically assigns each variable a unique RAM address, starting at RAM address 16.

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
```

User Defined Symbols

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
```

Conventions

- Hack programmers use lower-case for variables and upper-case for labels and predefined symbols.
- ** **Communicating with humans!** **

Can you identify each of these in this assembly code ==>

Example

When this program has finished assembling, what does the symbol table look like?

```
// 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
```

How do we build a symbol table?

Initialisation

- Create an empty table and put any pre-defined symbols in there.

First Pass

- Go through the source code and add all the user-defined labels to the table.
- The label's value is the location of the first instruction after the label.

Second Pass

- Go through the source code and use the symbol table to translate the commands. This is where names get turned into actual numbers.

The assembly process (detailed)

Initialization: Create the symbol table and initialize it with the pre-defined symbols

First pass: go through the source code without generating any code.

- For each label declaration `(LABEL)` that appears in the source code,
- add the pair <LABEL , n > to the symbol table where n is the location of the next instruction in ROM

The assembly process (detailed)

Second pass: go through the source code, and process each line:

If the line is a C-instruction, simple

If the line is @xxx where xxx is a number, simple

If the line is @xxx and xxx is a symbol, look it up in the symbol table and proceed as follows:

- If the symbol is found, replace it with its numeric value and complete the command's translation

Example

When this program has finished assembling, what is the resulting machine code?

```
// 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
```

This Week

- Review Chapters 5 & 6 of the Text Book (if you haven’t already)
- Assignment 3 Due
- Start Assignment 4
- Review Chapter 7 of the Text Book before next week.