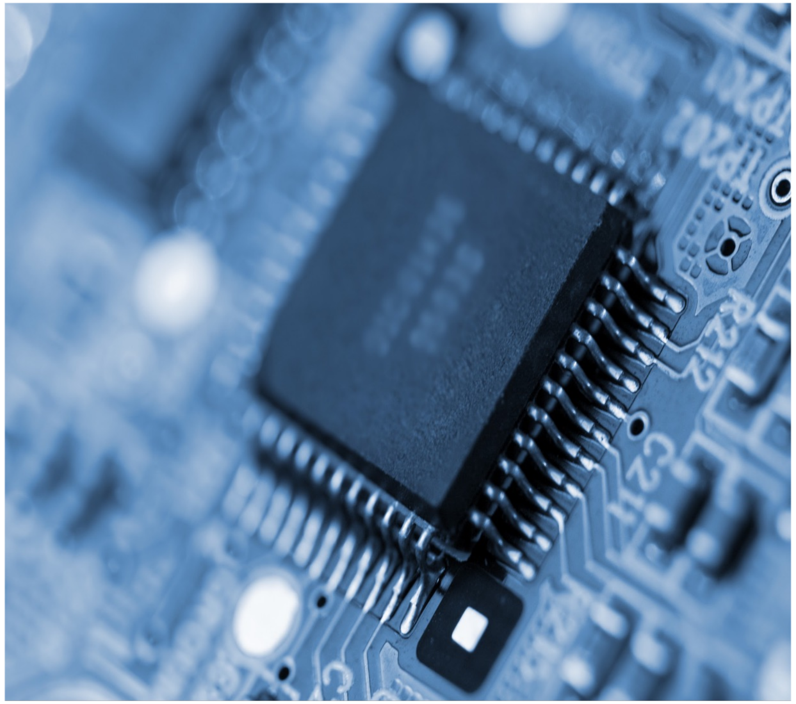
UNIVERSITY OF LEEDS

Computer Processors

Assembler



This lecture is based on the excellent course *Nand to Tetris* by Noam Nisam and Shimon Schocken, and we reuse here many of the slides provided at www.nand2tetris.org

The big picture

UNIVERSITY OF LEEDS

- In *(binary) machine language* instructions are written as series of 0's and 1's
- In assembly language (="symbolic" machine language) instructions are expressed using human-friendly mnemonics.

Assembly Language

```
  @i
  M=1  // i = 1


  @sum
  M=0  // sum = 0
(LOOP)
  @i   // if i>RAM[0]
  D=M  // GOTP WRITE
  @R0
  D=D-M
  @WRITE
  D;JGT
  ...  // Etc.
```

Machine Language

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

Both languages do exactly the same thing, and can be considered as equivalent. But, writing programs in assembly is far easier and safer then writing in binary.

The big picture

UNIVERSITY OF LEEDS

- In *(binary) machine language* instructions are written as series of 0's and 1's
- In assembly language (="symbolic" machine language) instructions are expressed using human-friendly mnemonics.

Assembly Language

```
  @i
  M=1  // i = 1


  @sum
  M=0  // sum = 0
(LOOP)
  @i   // if i>RAM[0]
  D=M  // GOTP WRITE
  @R0
  D=D-M
  @WRITE
  D;JGT
  ...  // Etc.
```

Machine Language

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

Both languages do exactly the same thing, and can be considered as equivalent. But, writing programs in assembly is far easier and safer then writing in binary.

However, the computer can only "understand" binary machine language!

The big picture

UNIVERSITY OF LEEDS

Assembly Language

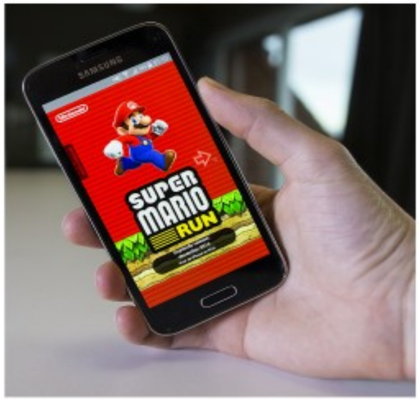
```
  @i
  M=1  // i = 1


  @sum
  M=0  // sum = 0
(LOOP)
  @i   // if i>RAM[0]
  D=M  // GOTP WRITE
  @R0
  D=D-M
  @WRITE
  D;JGT
  ...  // Etc.
```

assembler

Machine Language

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

run



The big picture

UNIVERSITY OF LEEDS

- We need some software that translates assembly language into (*binary) machine language* (could be written in binary ML → annoying and tedious!)

Assembly Language

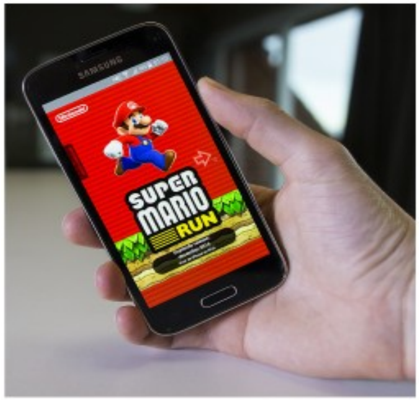
```
  @i
  M=1  // i = 1


  @sum
  M=0  // sum = 0
(LOOP)
  @i   // if i>RAM[0]
  D=M  // GOTP WRITE
  @R0
  D=D-M
  @WRITE
  D;JGT
  ...  // Etc.
```

assembler →

Machine Language

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

run →



The big picture

UNIVERSITY OF LEEDS

- We need some software that translates assembly language into (*binary) machine language* (could be written in binary ML → annoying and tedious!)
- Instead we assume we already built some computer that can run high-level languages

Assembly Language

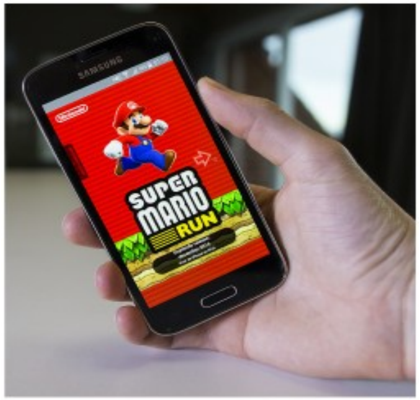
```
  @i
  M=1  // i = 1


  @sum
  M=0  // sum = 0
(LOOP)
  @i   // if i>RAM[0]
  D=M  // GOTP WRITE
  @R0
  D=D-M
  @WRITE
  D;JGT
  ...  // Etc.
```

assembler

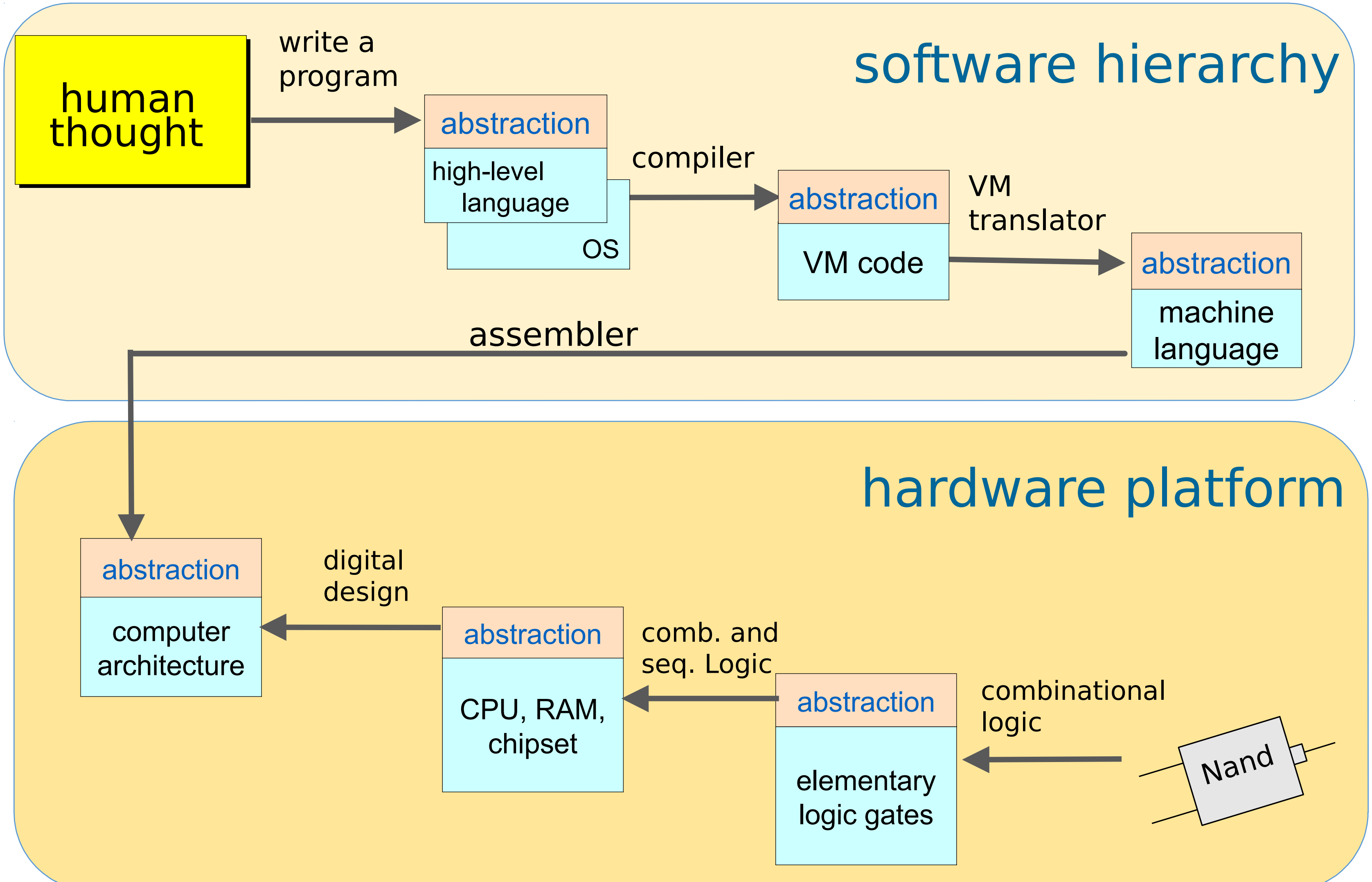
Machine Language

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

run



The big picture



Assembler is a software (= the first software layer above hardware)

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
// start processing the table ..
SomeCommand R1, 18

..// Etc.
```

Ignore white space (comments)

Read this into an array of characters

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
// start processing the table ..
SomeCommand R1, 18

..// Etc.
```

Ignore white space (comments)

Read this into an array of characters

|S|o|m|e|C|o|m|m|a|n|d| |R|1|,| |1|8|

Basic Assembler Logic

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
// start processing the table ..
SomeCommand R1, 18

..// Etc.
```

|S|o|m|e|C|o|m|m|a|n|d| |R|1|,| |1|8|

|S|o|m|e|C|o|m|m|a|n|d| |R|1| |1|8|

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
// start processing the table ..
SomeCommand R1, 18

..// Etc.
```

|S|o|m|e|C|o|m|m|a|n|d| |R|1|,| |1|8|

|S|o|m|e|C|o|m|m|a|n|d| |R|1| |1|8|

|S|o|m|e|C|o|m|m|a|n|d| → op-code eg. 10010
|R|1| → op-code eg. 01
|1|8| → op-code eg. 000010010

|1|0|0|1|0| |0|1| |0|0|0|0|1|0|0|1|

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
// start processing the table ..
SomeCommand R1, 18

..// Etc.
```

|S|o|m|e|C|o|m|m|a|n|d| |R|1|,| |1|8|

|S|o|m|e|C|o|m|m|a|n|d| |R|1| |1|8|

|1|0|0|1|0| |0|1| |0|0|0|0|1|0|0|1|

|1|0|0|1|0|0|1|0|0|0|0|1|0|0|1|

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

```
..// Etc.

SomeCommand R1, 18

..// Etc.
```

|S|o|m|e|C|o|m|m|a|n|d| |R|1|,| |1|8|

|S|o|m|e|C|o|m|m|a|n|d| |R|1| |1|8|

|1|0|0|1|0| |0|1| |0|0|0|0|1|0|0|1|0|

|1|0|0|1|0|0|1|0|0|0|0|1|0|0|1|0|

Machine Language

```
...
1001001000010010
```

Basic Assembler Logic

UNIVERSITY OF LEEDS

REPEAT:

- Read the next assembly language command
- Break it into the different „fields“ it is composed of
- Look up the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

UNTIL END-OF-FILE REACHED.

Handling (*user-defined, pre-defined) symbols* is a bit more challenging (later in detail)

Hack Assembler Language

UNIVERSITY OF LEEDS

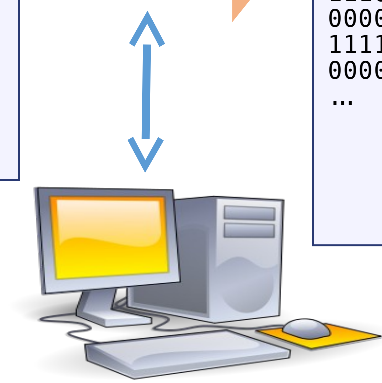
Assembly Language

```
   @i
   M=1  // i = 1


   @sum
   M=0  // sum = 0
 (LOOP)
   @i   // if i>RAM[0]
   D=M  // GOTP WRITE
   @R0
   D=D-M
   @WRITE
   D;JGT
   ...  // Etc.
```

assembler

Machine Language

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



Given: a source program written in symbolic Hack language and we have to translate it into an equivalent program written in binary ML.

→ Thus, the grammar of both languages must be known (the syntax rules).

What do we have to know about the Hack language?

→ We have A-instructions, C-instructions, and symbols.

Recap: A-instructions

UNIVERSITY OF LEEDS

Semantics: Sets the `A` register to *value*

Symbolic syntax:

`@` *value*

Example:

```
@21
```

sets `A` to 21

Where *value* is either:

- a non-negative decimal constant ≤ 65535 ($=2^{15}-1$) or
- a symbol referring to a constant

Binary syntax:

`0`*value*

Example:

```
0000000000010101
```

opcode signifying an `A`-instruction

sets `A` to 21

Where *value* is a 15-bit binary constant

Recap: C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

opcode signifying an C-instruction

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Recap: Symbols

UNIVERSITY OF LEEDS

The Hack assembly language features ***built-in symbols***:

symbol	value
`R0`	`0`
`R1`	`1`
`R2`	`2`
…	…
`R15`	`15`
`SCREEN`	`16384`
`KBD`	`24576`

symbol	value
`SP`	`0`
`LCL`	`1`
`ARG`	`2`
`THIS`	`3`
`THAT`	`4`

- `R0`, `R1` ,…, `R15` : "virtual registers", can be used as variables
- `SCREEN` and `KBD` : base addresses of I/O memory maps (later)
- Remaining symbols: used in the implementation of the Hack *virtual machine* (later)

Recap: Symbols

UNIVERSITY OF LEEDS

The Hack assembly language features ***symbols with label declaration***:

example:

```
   // Program: Signum.asm
   // Computes: if R0>0
   //                    R1=1
   //                 else
   //                    R1=0
   // Usage: put a value in RAM[0],
   //        run and inspect RAM[1].

0     @R0
1     D=M     // D = RAM[0]

2     @POSITIVE
3     D;JGT   // If R0>0 goto

4     @R1
5     M=0     // RAM[1]=0
6     @10
7     0;JMP   // goto end

   (POSITIVE)
8     @R1
9     M=1     // R1=1

    (END)
10    @END
11    0;JMP
```

referring to a label

declaring a label

Instead of @8 we use now @POSITIVE, and it is the job of the assembler to translate

Syntax: @LABEL referring to label „LABEL“
(LABEL) label declaration.

Recap: Symbols

The Hack assembly language features ***symbols as reference to variables***:

Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- If the reference @*symbol* occurs in the program for first time, *symbol* is allocated to address 16 onward (say *n*), and the generated code is @*n*
- All subsequent @*symbol* commands are translated into @*n*

Assembler Challenges

UNIVERSITY OF LEEDS

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler Challenges

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler

Hack machine code

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

Challenge: Handling these program elements!

Assembler Challenges

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

<u>Assembly program elements:</u>

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler

Hack machine code

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

Challenge: Handling these program elements!

Assembler Challenges

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @16
    M=1   // i = 1

    @17
    M=0   // sum = 0


    @16    // if i>RAM[0] goto STOP
    D=M
    @0
    D=D-M
    @18
    D;JGT
    @16    // sum += i
    D=M
    @17
    M=D+M
    @16    // i++
    M=M+1
    @4 // goto LOOP
    0;JMP

    @17
    D=M
    @1
    M=D  // RAM[1] = the sum

    @22
    0;JMP
```

Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler

Hack machine code

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

Challenge: Handling these program elements! – FOR simplicity first without symbols

Handling Programs without Symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @16
    M=1   // i = 1

    @17
    M=0   // sum = 0


    @16    // if i>RAM[0] goto STOP
    D=M
    @0
    D=D-M
    @18
    D;JGT
    @16    // sum += i
    D=M
    @17
    M=D+M
    @16    // i++
    M=M+1
    @4 // goto LOOP
    0;JMP

    @17
    D=M
    @1
    M=D  // RAM[1] = the sum

    @22
    0;JMP
```

<u>Challenges:</u>

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions

Assembler

Hack machine code

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

Handling Programs without Symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @16
    M=1   // i = 1

    @17
    M=0   // sum = 0

    @16    // if i>RAM[0] goto STOP
    D=M
    @0
    D=D-M
    @18
    D;JGT
    @16    // sum += i
    D=M
    @17
    M=D+M
    @16    // i++
    M=M+1
    @4 // goto LOOP
    0;JMP

    @17
    D=M
    @1
    M=D  // RAM[1] = the sum

    @22
    0;JMP
```

Challenges:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments

Assembler

Hack machine code

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

Handling Programs without Symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @16
    M=1   // i = 1

    @17
    M=0   // sum = 0


    @16    // if i>RAM[0] goto STOP
    D=M
    @0
    D=D-M
    @18
    D;JGT
    @16    // sum += i
    D=M
    @17
    M=D+M
    @16    // i++
    M=M+1
    @4 // goto LOOP
    0;JMP

    @17
    D=M
    @1
    M=D  // RAM[1] = the sum

    @22
    0;JMP
```

Challenges:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments

Handling: Ignore them!

Assembler

Hack machine code

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

Handling Programs without Symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @16
    M=1   // i = 1

    @17
    M=0   // sum = 0


    @16    // if i>RAM[0] goto STOP
    D=M
    @0
    D=D-M
    @18
    D;JGT
    @16    // sum += i
    D=M
    @17
    M=D+M
    @16    // i++
    M=M+1
    @4 // goto LOOP
    0;JMP

    @17
    D=M
    @1
    M=D  // RAM[1] = the sum

    @22
    0;JMP
```

Challenges:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions

Assembler

Hack machine code

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

Handling A-instructions

UNIVERSITY OF LEEDS

Symbolic syntax:

```
@value
```

Examples:

```
@ 21
```

Where *value* is either

- a non-negative decimal constant or
- a symbol referring to such a constant

Binary syntax:

```
0valueInBinary
```

Example:

```
0000000000010101
```

Translation to binary:

- If *value* is a decimal constant, generate the equivalent ***15-bit*** binary constant (you may have to add 0's)
- If *value* is a symbol, later.

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Symbolic:

Example: `MD=D+1`

Here: jump = `null`

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example: Symbolic: `MD=D+1` Binary: `111`

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic:	Binary:
MD=D+1	111

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example: Symbolic: MD=D+1 Binary: 111

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic: MD=D+1

Binary: 1110011111

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example: Symbolic: MD=D+1 Binary: 1110011111

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic: MD=D+1

Binary: 1110011111011

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic: MD=D+1

Binary: 1110011111011

Handling C-instructions

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic: `MD=D+1`

Binary: `1110011111011000`

Handling C-instructions

UNIVERSITY OF LEEDS

Symbolic syntax: *dest* = *comp* ; *jump* (both *dest* and *jump* are optional)

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

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

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example:

Symbolic: `MD=D+1`

Binary: `1110011111011000`

The overall assembly-logic

Assembly Program

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
@1
M=D
@22
0;JMP
```

For each instruction

- Parse the instruction:
  break it into its underlying fields
- A-instruction:
  translate the decimal value into a binary value
- 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.

The overall assembly-logic

Assembly Program

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
@1
M=D
@22
0;JMP
```

For each instruction

- Parse the instruction:
  break it into its underlying fields
- A-instruction:
  translate the decimal value into a binary value
- 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.

Resulting code:

Hack ML

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

The overall assembly-logic

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

<u>Assembly program elements:</u>

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler

Hack machine code

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

Handling Symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Type of Symbols:

- Pre-defined Symbols:
  - Represent special memory locations
- Label Symbols:
  - Represent destinations of goto instructions
- Variable Symbols
  - Represent memory locations where the programmer wants to maintain values

Handling pre-defined symbols

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Pre-defined Symbols:

- Represent special memory locations

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576

symbol	value
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Translating @ *preDefinedSymbol* :

Replace *preDefinedSymbol* with binary code of its value.

Handling label symbols

UNIVERSITY OF LEEDS

Assembly Program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0      @i
1      M=1   // i = 1

2      @sum
3      M=0   // sum = 0

   (LOOP)
4      @i    // if i>RAM[0] goto STOP
5      D=M
6      @R0
7      D=D-M
8      @STOP
9      D;JGT
10     @i    // sum += i
11     D=M
12     @sum
13     M=D+M
14     @i    // i++
15     M=M+1
16     @LOOP // goto LOOP
17     0;JMP
   (STOP)
18     @sum
19     D=M
20     @R1
21     M=D  // RAM[1] = the sum
   (END)
22     @END
23     0;JMP
```

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 memory location holding the next instruction in the program

Handling label symbols

Assembly Program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0      @i
1      M=1   // i = 1

2      @sum
3      M=0   // sum = 0

   (LOOP)
4      @i    // if i>RAM[0] goto STOP
5      D=M
6      @R0
7      D=D-M
8      @STOP
9      D;JGT
10     @i    // sum += i
11     D=M
12     @sum
13     M=D+M
14     @i    // i++
15     M=M+1
16     @LOOP // goto LOOP
17     0;JMP
   (STOP)
18     @sum
19     D=M
20     @R1
21     M=D  // RAM[1] = the sum
   (END)
22     @END
23     0;JMP
```

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 memory location holding the next instruction in the program

symbol	value
LOOP	4
STOP	18
END	22

Handling label symbols

UNIVERSITY OF LEEDS

Assembly Program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0      @i
1      M=1   // i = 1

2      @sum
3      M=0   // sum = 0

   (LOOP)
4      @i    // if i>RAM[0] goto STOP
5      D=M
6      @R0
7      D=D-M
8      @STOP
9      D;JGT
10     @i    // sum += i
11     D=M
12     @sum
13     M=D+M
14     @i    // i++
15     M=M+1
16     @LOOP // goto LOOP
17     0;JMP
   (STOP)
18     @sum
19     D=M
20     @R1
21     M=D  // RAM[1] = the sum
   (END)
22     @END
23     0;JMP
```

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 memory location holding the next instruction in the program

symbol	value
LOOP	4
STOP	18
END	22

Translating `@` *labelSymbol* :

Replace *labelSymbol* with binary code of its value.

Handling symbols that denote variables

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Variable Symbol

- Any symbol XXX appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a *variable*
- Each variable is assigned a unique memory address, starting at 16 (onwards)

Handling symbols that denote variables

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Variable Symbol

- Any symbol XXX appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a *variable*
- Each variable is assigned a unique memory address, starting at 16 (onwards)

symbol	value
i	16
sum	17

Handling symbols that denote variables

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Variable Symbol

- Any symbol XXX appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a *variable*
- Each variable is assigned a unique memory address, starting at 16 (onwards)

symbol	value
i	16
sum	17

Translating @*variableSymbol* :

- If seen for the first time, assign a unique memory address
- Replace variable with binary code of its value

Symbol Table

Assembly Program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0      @i
1      M=1   // i = 1

2      @sum
3      M=0   // sum = 0

   (LOOP)
4      @i    // if i>RAM[0] goto STOP
5      D=M
6      @R0
7      D=D-M
8      @STOP
9      D;JGT
10     @i    // sum += i
11     D=M
12     @sum
13     M=D+M
14     @i    // i++
15     M=M+1
16     @LOOP // goto LOOP
17     0;JMP
   (STOP)
18     @sum
19     D=M
20     @R1
21     M=D  // RAM[1] = the sum
   (END)
22     @END
23     0;JMP
```

Symbol table

symbol	value

Symbol Table

UNIVERSITY OF LEEDS

Assembly Program

```
     // Computes RAM[1] = 1 + ... + RAM[0]
 0        @i
 1        M=1   // i = 1

 2        @sum
 3        M=0   // sum = 0

     (LOOP)
 4        @i    // if i>RAM[0] goto STOP
 5        D=M
 6        @R0
 7        D=D-M
 8        @STOP
 9        D;JGT
10        @i    // sum += i
11        D=M
12        @sum
13        M=D+M
14        @i    // i++
15        M=M+1
16        @LOOP // goto LOOP
17        0;JMP
     (STOP)
18        @sum
19        D=M
20        @R1
21        M=D  // RAM[1] = the sum
     (END)
22        @END
23        0;JMP
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Initialization:
Add the pre-defined symbols

Symbol Table

UNIVERSITY OF LEEDS

Assembly Program

```
     // Computes RAM[1] = 1 + ... + RAM[0]
0        @i
1        M=1   // i = 1

2        @sum
3        M=0   // sum = 0

     (LOOP)
4        @i    // if i>RAM[0] goto STOP
5        D=M
6        @R0
7        D=D-M
8        @STOP
9        D;JGT
10       @i    // sum += i
11       D=M
12       @sum
13       M=D+M
14       @i    // i++
15       M=M+1
16       @LOOP // goto LOOP
17       0;JMP
     (STOP)
18       @sum
19       D=M
20       @R1
21       M=D  // RAM[1] = the sum
     (END)
22       @END
23       0;JMP
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
END	22

Initialization:
Add the pre-defined symbols

First pass:
Add the label symbols

Symbol Table

Assembly Program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0      @i
1      M=1   // i = 1

2      @sum
3      M=0   // sum = 0

   (LOOP)
4      @i    // if i>RAM[0] goto STOP
5      D=M
6      @R0
7      D=D-M
8      @STOP
9      D;JGT
10     @i    // sum += i
11     D=M
12     @sum
13     M=D+M
14     @i    // i++
15     M=M+1
16     @LOOP // goto LOOP
17     0;JMP
   (STOP)
18     @sum
19     D=M
20     @R1
21     M=D  // RAM[1] = the sum
   (END)
22     @END
23     0;JMP
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
…	…
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
END	22
i	16
sum	17

Initialization:
Add the pre-defined symbols

First pass:
Add the label symbols

Second pass:
Add the var. symbols

Symbol Table

UNIVERSITY OF LEEDS

Assembly Program

```
     // Computes RAM[1] = 1 + ... + RAM[0]
 0       @i
 1       M=1   // i = 1

 2       @sum
 3       M=0   // sum = 0

     (LOOP)
 4       @i    // if i>RAM[0] goto STOP
 5       D=M
 6       @R0
 7       D=D-M
 8       @STOP
 9       D;JGT
10       @i    // sum += i
11       D=M
12       @sum
13       M=D+M
14       @i    // i++
15       M=M+1
16       @LOOP // goto LOOP
17       0;JMP
     (STOP)
18       @sum
19       D=M
20       @R1
21       M=D  // RAM[1] = the sum
     (END)
22       @END
23       0;JMP
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
END	22
i	16
sum	17

Usage:
To resolve a symbol, look up its value in the symbol table

Initialization:
Add the pre-defined symbols

First pass:
Add the label symbols

Second pass:
Add the var. symbols

The assembly process

UNIVERSITY OF LEEDS

<u>Initialization:</u>

- Construct an empty symbol table
- Add the pre-defined symbols to the symbol table

<u>First pass:</u>

Scan the entire program;

For each "instruction" of the form `(xxx)`:

- Add the pair (`xxx`, *address*) to the symbol table, where *address* is the number of the instruction following `(xxx)`

<u>Second pass:</u>

Set *n* to 16

Scan the entire program again; for each instruction:

- If the instruction is @*symbol*, look up *symbol* in the symbol table;
  - If (*symbol, value*) is found, use *value* to complete the instruction's translation;
  - If not found:
    - Add (*symbol*, *n*) to the symbol table,
    - Use *n* to complete the instruction's translation,
    - *n*++
- If the instruction is a C-instruction, complete the instruction's translation
- Write the translated instruction to the output file.

Hack Assembler

UNIVERSITY OF LEEDS

Assembly Program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0

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

Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

Assembler

Hack machine code

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