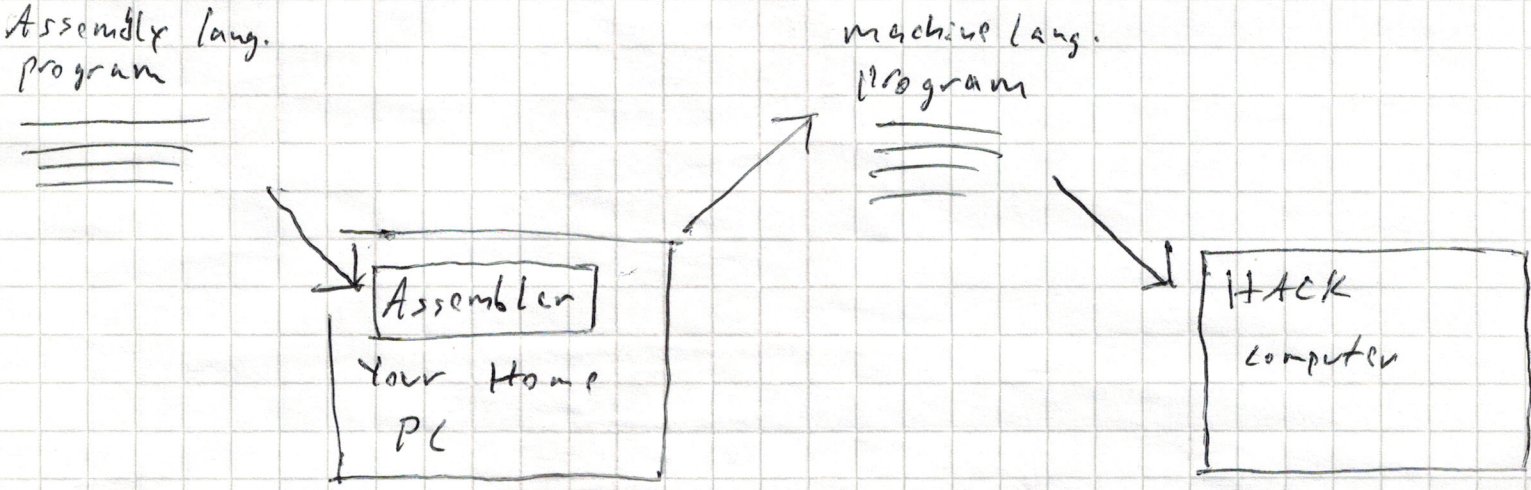
June 22, 202

Unit 6.1 - Assembly Languages and Assemblers

- The assembler is software
- First layer above the hardware

Where does the Assembler Program run?



- so far we built a computer that can execute binary machine lang. code
- writing code in binary is tedious, but can think of HACK compter as "second" computer and personal PC as "first" computer made
- Assembler ... is written in high level language & runs on personal PC
- It will produce code for the HACK computer "cross-compiler"
- Eliminates bootstrap problem of writing Assembler in machine language

Basic Assembler Logic

Repeat:
- Read the next Assembly language command
- Break it into the different fields it is composed of
- Lookup the binary code for each field
- Combine these codes into a single machine language command
- Output this machine language command

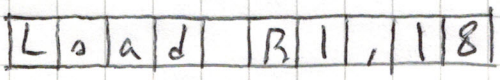
Until end-of-file reached

June 22, 2020

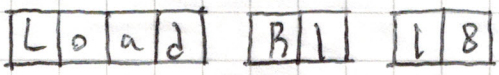
Unit 6.1 - Basic Assembler Logic (continued)

Read the next Assembly language command, e.g.:

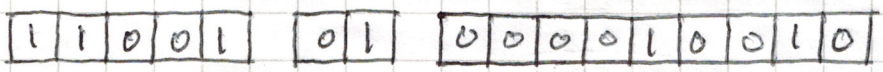
```
// start processing the table   ← ignore white space / comments
Load R1, 18                     ← read code into an array of characters
```



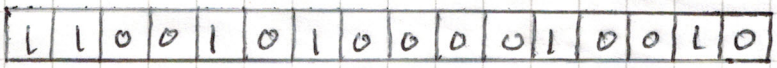
Break it into the different fields it is composed of

3 sub-strings: 

Lookup binary code for each field (based on machine lang. spec)



Combine these codes into a single machine language command



(might need extra bits or padding)

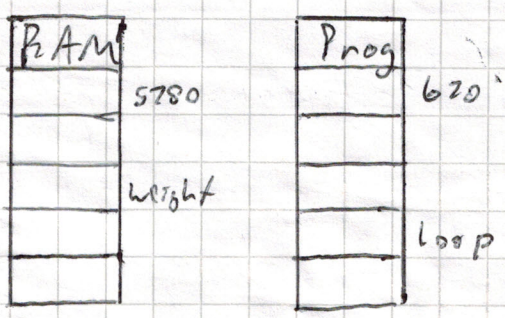
Output this machine language command

```
. . .
1 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0
. . .
```

Symbol Table, e.g.:

```
Load R1, weight  ⟶  Load R1, 5782
JMP loop         ⟶  JMP 673
```

Symbol	Address
weight	5782
loop	673



Forward References e.g. jumping into label before it is defined

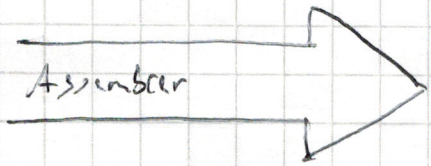
```
. . .
JGT cont
. . .
Label cont:
. . .
```

Possible solutions:
- leave blank until label appears, then fix
- in first pass, just figure out addresses
  - ↳ easier

June 22, 20

Unit 6.2 - The Hack Assembly Language: A Translator's Perspective

Assembly Program



Hack machine code

Issues:

- White space — ignore
  - empty lines
  - comments
  - In-line comments
- Instructions — look up machine lang. spec
  - A-instructions
  - C-instructions
- Symbols
  - References
  - label declaration

June 22, 2021

Unit 6.3 - The Assembly Process - Handling Instructions

Translating A-instructions

Symbolic syntax: @ value Example: @21

- where value is either
  - non-negative decimal constant
  - symbol referring to a constant

Binary syntax: 0 valueInBinary Example: 0 0000000000010101

Translation to binary:

- if value is a decimal constant, generate equivalent 15-bit binary constant
- append 0
- If value is symbol, see next unit

Translating C-Instructions

Symbolic syntax: dest=comp;jump 111 a cccccc ddd jjj → Binary syntax

- Parse instructions into the color coded fields
- Use machine language specification tables to build out instruction

June 22, 2020

Unit 6.4 - The Assembly Process - Handling Symbols

3 types of Symbols

- **Variable symbols**: represent memory location where the programmer wants to maintain values
- **label symbols**: represent destinations of goto instructions
- **pre-defined symbols**: represent special memory locations

Handling pre-defined symbols

- The Hack language specification describes 23 pre-defined symbols
- To translate @preDefinedSymbol:
  - replace preDefined symbol with the value from spec

Handling label symbols

- Declared by pseudo-command (XXX)
- Directive defines the symbol XXX to refer to the memory location holding the next instruction in program

e.g.

Symbol	value
LOOP	4
STOP	18
END	22

```
        .
        .
        .
3       M=0
(LOOP)
4       @i
        .
        .
        .
```

- so to translate @label symbol, lookup table value

Handling variable symbols

- Any symbol XXX which is not (a) pre-defined, and (b) not defined elsewhere using (XXX) directive is a variable
- Each variable assigned a unique address starting at 16

e.g.

Symbol	value
i	16
sum	17

Translating @variableSymbol:

- If you see it for first time, assign a unique memory address
- Replace variable symbol with its value

June 24, 2024

Unit 6.4 - Symbol Table

Symbol	Value	
R0	0	Initialization
R1	1	• Add pre-defined symbols
R2	2	
...	...	
R15	15	
SCREEN	16384	
KBD	24576	
SP	0	
LCL	1	
ARG	2	
THIS	3	
THAT	4	
LOOP	4	First Pass
STOP	18	• Add the label symbols
END	22	
i	16	Second Pass
sum	17	• Add the variable symbols

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

once translation process is done, can throw away table

Jan 22, 2020

Unit 6.4 - The Assembly Process

Initialization

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

First Pass

- Scan entire program
- For each "instruction" in the form (XXX):
  - Add pair (XXX, address) to table, where address is number of instruction following (XXX)

Second Pass

- set n to 16
- Scan entire program; for each instruction:
  - If ~~(symbol, value)~~ instruction is @symbol, look up symbol in table:
    - if (symbol, value) is found, use value to complete translation
    - If not found:
      - Add (symbol, value) to table
      - Use n to complete instruction's translation
      - n++
  - ~~else~~ If instruction is C-instruction, use spec. to translate
  - write translated line to output file