Agenda

Writing a Hack assembler

- Translating instructions to binary machine code
- Assigning memory locations to symbols (e.g. @i)
- Associating ROM locations with labels (e.g. (END))

NOTE: With slides from nand2tetris.org

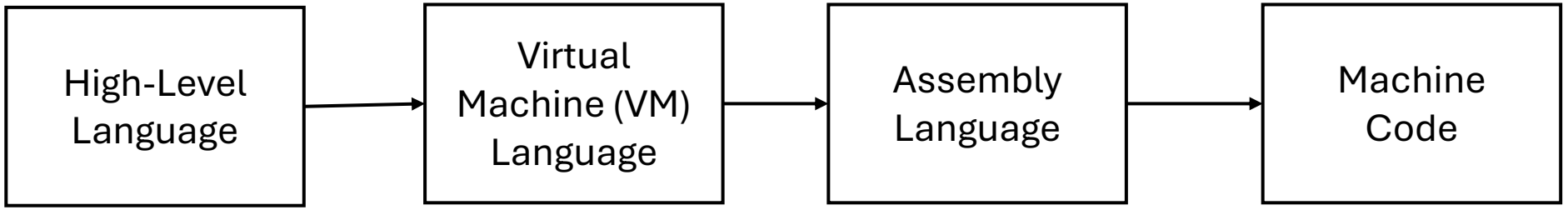
Compilation

How do we get from a high-level language to machine code?

Perform a series of translations that are increasingly low-level

Each level *abstracts* the details of the level below it

For example, assembly programs do not need to know the details of machine code

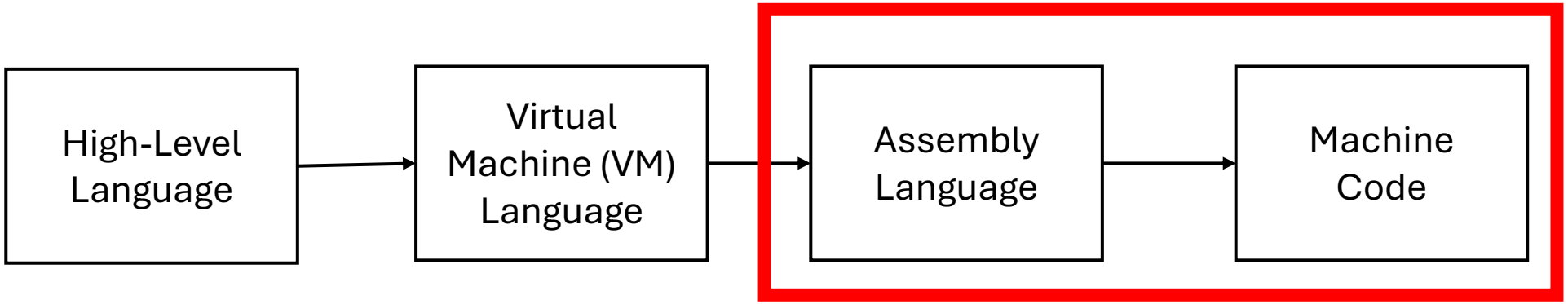


Writing an assembler for Hack

The assembler converts from assembly instructions to machine code instructions.

Features:

- Translating instructions to binary machine code
- Assigning memory locations to symbols (e.g. @i)
- Associating ROM locations with labels (e.g. (END))
- Ignores whitespace and comments



Writing an assembler for Hack: Example

Assembly program

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if (i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
   ...
```

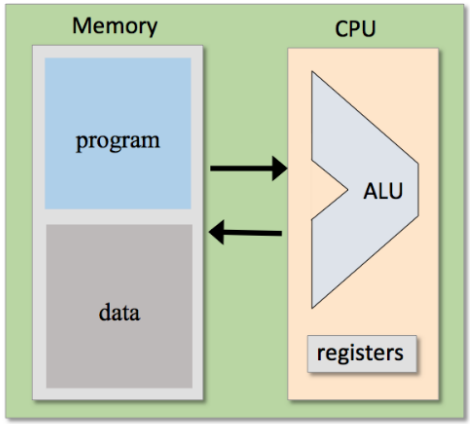
assembler

Binary code

```
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
...
```

load and execute

Computer



The assembler is…

- The first step in a typical hierarchy of translators (assembler, VM translator, compiler)
- A program that introduces basic software engineeri techniques used by every translator:
  - Files handling
  - Parsing
  - Code generation
  - Symbol tables

Writing an assembler for Hack: Goal

Goal: Develop a program that translates symbolic Hack programs into binary Hack instructions

The source assembly program (input) is read from a text file named Prog.asm

The generated binary code (output) is written to a text file named Prog.hack

Assumption: Prog.asm is error-free.

Usage: ./hack-assembler prog.asm

Translating the A-instruction

Symbolic syntax:

`@`*xxx*

Where *xxx* is a non-negative decimal value, or a symbol bound to such a value

Binary syntax:

`0`*vvvvvvvvvvvvvvv*

Where:

`0` is the A-instruction op-code, and

v v v ... v is a binary value

Example:

`@7`

`0000000000000111`

Implementation

If *xxx* is a decimal value: Translate the value into its 16-bit representation;

If *xxx* is a symbol: Later.

Translating the C-instruction

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: `1 1 1 a c c c c c c d d d j j j`

comp		*c*	*c*	*c*	*c*	*c*	*c*
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	*d*	*d*	*d*	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
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

jump	*j*	*j*	*j*	effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	Unconditional jump

Exercise: Translate these instructions

@7

D=D+1; JLE

A = -1

Exercise: Write pseudocode that computes the first 4 bits of machine code from an instruction

Handling symbols

Three types in Hack assembly

- Built-in
- Variable symbols
- Label symbols

Built-in Symbols

In the Hack language:

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

Handling symbols: overview

- Replace built-in symbol names with their values
  - Example: @R12 → 0000 0000 0000 1100
- Maintain a table for variable symbols
  - New symbols are assigned a place in memory, starting at memory address 16
  - When a symbol is encountered, replace it with its memory address
- Maintain a table for label symbols
  - New labels are assignment the address of the next instruction
  - When a label is encountered, replace it with its memory address

Handling symbols: Example

Line Number	Instruction
0	@i
1	M=1
2	@sum
3	M=0
4	(LOOP)
5	@i
6	D=M
7	@R0
8	D=D-M

Original Code

Line Number	Instruction
0	
1	
2	
3	
4	
5	
6	
7	
8	

Symbols Replaced

Symbol Table
(Built-in symbols are added first)

Name	Value
R0	0
R1	1
….	…
SCREEN	16384
KBD	24576
…	…

Assembler Example

Line Number	Instruction
0	@i
1	M=1
2	@sum
3	M=0
4	(LOOP)
5	@i
6	D=M
7	@R0
8	D=D-M

Original Code

Line Number	Instruction
0	@16
1	M=1
2	@17
3	M=0
4	@16
5	D=M
6	@0
7	D=D-M
8	

Symbols Replaced

Line Number	Instruction
0	0000 0000 0001 0000
1	1110 1111 1100 1000
2	0000 0000 0001 0001
3	1110 1010 1000 1000
4	0000 0000 0001 0000
5	1111 1100 0001 0000
6	0000 0000 0000 0000
7	1111 0100 1101 0000
8	

Converted to binary

Assembler Algorithm

Initialize the symbol table with built-in symbol names
Open the file (.asm file)

// First pass
For each line in the file
  Ignore comments and white space
  Add symbols to the symbol table

//Second pass
For each instruction
  Replace symbols with values
  Convert to machine code and write to file (.hack file)

C++ Helpers: map (hash table)

```
#include <unordered_map> // hash table
...
 std::unordered_map<std::string, std::string> colors =
 {
  {"RED", "#FF0000"},
  {"GREEN", "#00FF00"},
  {"BLUE", "#0000FF"}
 };

 std::cout << "Number of items: " << colors.size() << std::endl;
 for (auto it = colors.begin(); it != colors.end(); ++it) {
  std::cout << it->first << ", " << it->second << std::endl; // C++ style print
  printf("%s, %s\n", it->first.c_str(), it->second.c_str()); // C-style print
 }

 for (auto [key, value] : colors) {
  std::cout << key << ", " << value << std::endl;
 }
```

C++ Helpers: map (tree)

```
#include <map> // usually a red-black tree
...
std::map<std::string,int> names2age;
names2age["giles"] = 54;
names2age["buffy"] = 18;
names2age["joyce"] = 38;

std::cout << "Number of items: " << names2age.size() << std::endl;
for (auto it = names2age.begin(); it != names2age.end(); ++it) {
  std::cout << it->first << ", " << it->second << std::endl;
}

names2age.clear();
names2age = { {"giles", 54}, {"buffy", 18}, {"drusilla", 176} };
for (auto [key, value] : names2age) {
  std::cout << key << ", " << value << std::endl;
}
```

C++ Helpers: std::vector

```
vector<float> values = {1.0, -2.0, 3.0};
values.push_back(-4.0);

for (unsigned int i = 0; i < values.size(); i++) {
   cout << values[i] << endl;
}

values.clear();
values = vector<float>(10);
for (unsigned int i = 0; i < 10; i++) {
  values[i] = i;
}

values[1] *= 10.0;
for (float v : values) {
  cout << v << endl;
}
```

C++ Helpers: std::string

```
string phrase = "the quick, brown dog";

if (phrase.find("quick") != string::npos) {
  cout << "Found quick in phrase!\n";
}

cout << "The string length is " << phrase.size() << std::endl;

string newphrase = "";
for (unsigned int i = 0; i < phrase.size(); i++) {
   if (phrase[i] == 'i') newphrase += "1";
   else if (phrase[i] == 'o') newphrase += "0";
   else if (phrase[i] == 'e') newphrase += "3";
   else newphrase += phrase[i];
}

cout << "newphrase: " << newphrase << endl;

string filename = "hello.txt";
string newfilename = filename.substr(0, filename.size()-4) + ".bvh";
cout << "newfilename: " << newfilename << endl;
printf("5s\n", newfilename.c_str());
```

Assembler Implementation (C with C++ data structures)

```
struct Instruction {
   InstructionType type;
   int instruction_num;
   unsigned char mcode[16];
   char cmd[128];
};

main(int argc, char* argv[]) {
   // parse command line arguments
   std::vector<Instruction> instructions; → list of instructions
   std::map<std::string,int> symtable; → maps names to values
  ...
```

Assembler Implementation (C with C++ data structures)

```
// Reading a file and removing whitespace and comments
FILE* fp = fopen(argv[1], “r”);
char line[1024];
while (fgets(line, 1024, fp)) {
   char* comment = strstr(line, “//”);
   if (comment) *comment = ‘\0’;
   remove_whitespace(line, 1024);
   //Initialize and add instructions to vector
}
fclose(fp);
```

Assembler Implementation (C with C++ data structures)

```
// Process instructions

for (int i = 0; i < instructions.size(); i++) {
  // do pass 1
}

for (int i = 0; i < instructions.size(); i++) {
  // do pass 2
}

// write machine code to stdout
```

Assembler Implementation – First pass (C with C++ data structures)

```
for (int i = 0; i < instructions.size(); i++)  {
  Instruction instruction = instructions[i];
  if (instruction.type == A_INSTRUCTION) {
      get_symbol(instruction.cmd, sym);
      symtable[sym] = memory_location;
  }
  else if (instruction.type == L_INSTRUCTION) {
      get_symbol(instruction.cmd, sym);
      symtable[sym] = instructions[i+1].instruction_num;
  }
 // etc
}
```

Recall: Assembly Language Syntax

- Two assembly commands: A-instructions and C-instructions
- Symbols can be any sequence of letters, digits, underscore, dot, dollar sign, and colon. Symbols cannot begin with a digit
- Constants must be non-negative and written in decimal notation
- Comments begin with //
- White space is ignored
- Case conventions: Assembly mnemonics (KBD, SCREEN, R0, etc) must be all uppercase. Symbols and labels are case sensitive
- Assembly programs have the .asm extension

Tools

Use the assembler from nand2tetris.org to test, e.g.

java HackAssembler Prog.asm

Use the emulator to visualize the asm and machine code

Testing on the CPU emulator: