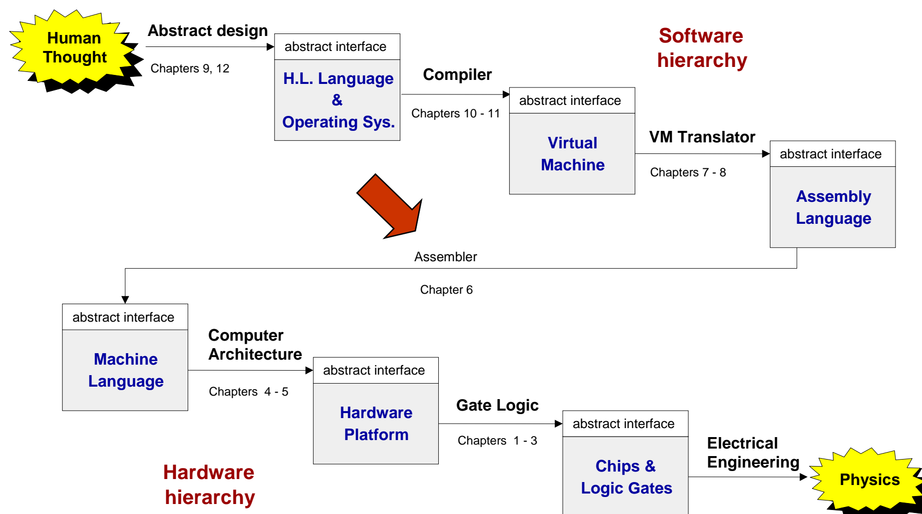
Assembler

Shimon Schocken

Spring 2005

Where we are at:



Copyright © Shimon Schocken

Why care about assemblers?

Because …

- Assemblers employ some nifty tricks
- Assemblers are the first rung up the software hierarchy ladder
- An assembler is a compiler for a simple language
- Writing an assembler is a good introduction for writing a compiler.

Program translation

Source code

```
// Computes sum=1+… +100
    i=1
    sum=0
loop:
    if i=101 goto end
    sum=sum+i
    i=i+1
    goto loop
end:
    goto end
```

Translator →

Target code

```
00  10001010110011001000111011001101
01  00011101101111001000111011001100
02  00100110110011001000111011001100
03  10001110110011001000111011001101
04  00011101100111001000111011000100
05  10001110110011001010111011001110
06  00011101110011001000111011001100
07  10001010110011001010111011001101
08  00011101100110010010111011000100
09  10001110110011001000111011001101
10  00111010110011001000111011001010
11  10001110110011001000111011001100
```

The program translation challenge

- Parse the source program, using the syntax rules of the source language
- Re-express the program’s semantics using the syntax rules of the target lang.

Assembler = simple translator

- Translates each assembly command into one or more machine instructions
- Handles symbols (`i`, `sum`, `loop`, `end`, …).

Copyright © Shimon Schocken

Symbol resolution

In low level languages, symbols are normally used to represent:

- Variables
- Destinations of goto commands
- Special memory locations

Code with Symbols

```
     // Computes sum=1+... +100
00       i=1
01       sum=0
     loop:
02       if i=101 goto end
03       sum=sum+i
04       i=i+1
05       goto loop
     end:
06       goto end
```

Symbol table

i	1024
sum	1025
loop	2
end	6

(assuming that variables are allocated to Memory[1024] onward)

Code with Symbols Resolved

```
00  M[1024]=1  //(M=memory)
01  M[1025]=0
02  if M[1024]=101 goto 6
03  M[1025]=M[1025]+M[1024]
04  M[1024]=M[1024]+1
05  goto 2
06  goto 6
```

(assuming that each symbolic command is translated into one word in memory)

The assembly process:

- First pass: construct a symbol table
- Second pass: translate the program, using the symbol table for symbols resolution.

Perspective

Code with Symbols

```
     // Computes sum=1+... +100
00       i=1
01       sum=0
     loop:
02       if i=101 goto end
03       sum=sum+i
04       i=i+1
05       goto loop
     end:
06       goto end
```

Symbol table

i	1024
sum	1025
loop	2
end	6

(assuming that variables are allocated to Memory[1024] onward)

Code with Symbols Resolved

```
00  M[1024]=1  //(M=memory)
01  M[1025]=0
02  if M[1024]=101 goto 6
03  M[1025]=M[1025]+M[1024]
04  M[1024]=M[1024]+1
05  goto 2
06  goto 6
```

(assuming that each symbolic command is translated into one word in memory)

- Simplifying assumptions:
  - Largest possible program is 1024 commands long
  - Each command fits into one memory location
  - Each variable fits into one memory location
- These assumptions can be relaxed rather easily, requiring more sophisticated assemblers.

Copyright © Shimon Schocken

The Hack assembly language

Assembly program (`Prog.asm`)

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      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=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
 (END)
      @END
      0;JMP  // infinite loop
```

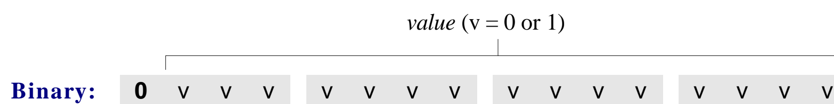
Assembly program =
a stream of text lines, each being:

- An instruction:
  `A-instruction` or
  `C-instruction`
- A symbol declaration:
  `(symbol)`
- A comment / white space:
  `// comment.`

A-instruction

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

value (v = 0 or 1)



Translation to binary:

- If *value* is a number: simple
- If *value* is a symbol: later.

Copyright © Shimon Schocken

C-instruction

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

(comp = a c1 c2 c3 c4 c5 c6; dest = d1 d2 d3; jump = 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	[illegible]	
A	1	1	0	0	0	[illegible]	
!D	0	0	1	1	0	[illegible]	
!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
[illegible]	[illegible]	[illegible]	[illegible]	[A] and D register
[illegible]	[illegible]	[illegible]	[illegible]	er
[illegible]	[illegible]	[illegible]	[illegible]	er 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

Translation to binary: simple!

The overall assembly logic

Assembly program (`Prog.asm`)

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      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=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
 (END)
      @END
      0;JMP  // infinite loop
```

For each symbolic command

- Parse into the underlying symbolic fields
- Replace each symbolic reference (if any) with the correspondng memory address (a binary number)
- For each field, generate the corresponding binary code
- Assemble the binary codes into a complete machine instruction.

Copyright © Shimon Schocken

Symbols handling (in the Hack language)

Program example

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      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=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
 (END)
      @END
      0;JMP  // infinite loop
```

- **Predefined symbols:** (don't appear in this example)

Label	*RAM address*
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
R0-R15	0-15
SCREEN	16384
KBD	24576

- **Label symbols:** The pseudo-command "`(Xxx)`" declares that the user-defined symbol `Xxx` should refer to the memory location holding the next command in the program

- **Variable symbols:** Any symbol `Xxx` appearing in an assembly program that is not predefined and is not defined elsewhere using the "`(Xxx)`" pseudo command is treated as a variable. Variables are mapped to consecutive memory locations starting at RAM address 16.

Example

Assembly code (`Prog.asm`)

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      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=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
 (END)
      @END
      0;JMP  // infinite loop
```

Assembler →

Binary code (`Prog.hack`)

```
(this line should be erased)
0000 0000 0001 0000
1110 1111 1100 1000
0000 0000 0001 0001
1110 1010 1000 1000
(this line should be erased)
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100
1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001
1111 0000 1000 1000
0000 0000 0001 0000
1111 1101 1100 1000
0000 0000 0000 0100
1110 1010 1000 0111
(this line should be erased)
0000 0000 0001 0010
1110 1010 1000 0111
```


Copyright © Shimon Schocken

Proposed implementation

An assembler program can be implemented (in any language) based on the following software modules:

- Parser: Unpacks each command into its underlying fields
- Code: Translates each field into its corresponding binary value
- SymbolTable: Manages the symbol table
- Main: Initializes files and drives the show.

Parser module

Parser: Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

Routine	Arguments	Returns	Function
Constructor / initializer	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	--	Boolean	Are there more commands in the input?
`advance`	--	--	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.
`commandType`	--	`A_COMMAND`, `C_COMMAND`, `L_COMMAND`	Returns the type of the current command: • `A_COMMAND` for `@Xxx` where `Xxx` is either a symbol or a decimal number • `C_COMMAND` for `dest=comp;jump` • `L_COMMAND` (actually, pseudo-command) for `(Xxx)` where `Xxx` is a symbol.

Copyright © Shimon Schocken

Parser module (cont.)

`symbol`	--	string	Returns the symbol or decimal `Xxx` of the current command `@Xxx` or `(Xxx)`. Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`.
`dest`	--	string	Returns the dest mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.
`comp`	--	string	Returns the comp mnemonic in the current *C*-command (28 possibilities). Should be called only when `commandType()` is `C_COMMAND`.
`jump`	--	string	Returns the jump mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.

Code module

Code: Translates Hack assembly language mnemonics into binary codes.

Routine	Arguments	Returns	Function
`dest`	mnemonic (string)	3 bits	Returns the binary code of the `dest` mnemonic.
`comp`	mnemonic (string)	7 bits	Returns the binary code of the `comp` mnemonic.
`jump`	mnemonic (string)	3 bits	Returns the binary code of the `jump` mnemonic.

Copyright © Shimon Schocken

Proposed implementation plan

- Stage I: Build a basic assembler for programs with no symbols
- Stage II: Extend the basic assembler with symbol handling capabilities

Symbol table

SymbolTable: A symbol table that keeps a correspondence between symbolic labels and numeric addresses.

Routine	Arguments	Returns	Function
`Constructor`	--	--	Creates a new empty symbol table.
`addEntry`	`symbol` (string), `address` (int)	--	Adds the pair `(symbol,address)` to the table.
`contains`	`symbol` (string)	Boolean	Does the symbol table contain the given `symbol`?
`GetAddress`	`symbol` (string)	int	Returns the address associated with the `symbol`.

Copyright © Shimon Schocken

Building the final assembler

- Initialization: create the symbol table and initialize it with the pre-defined symbols
- First pass: march through the program and build the symbol table, without generating any code
- Second pass: march again through the program, and translate 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`" where `Xxx` is a symbol, look it up in the symbol table
    - If the symbol is found, replace it with its numeric meaning and complete the command's translation
    - If the symbol is not found, then it must represent a new variable:
      add the pair (`Xxx`,*n*) to the symbol table, where *n* is the next available RAM address, and complete the command's translation.

      (The allocated RAM addresses are running, starting at address 16).

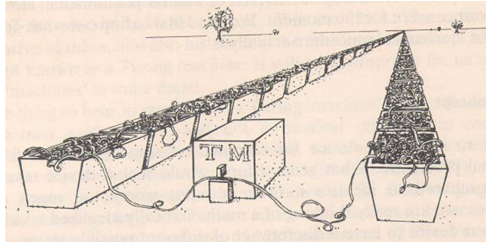
Perspective

- Simple machine language, simple assembler
- Most assemblers are not stand-alone, but rather encapsulated in a translator of a higher order
- Typically, low-level C programming (e.g. for real-time systems) involves some assembly programming (e.g. for optimization)
- Macro assemblers:

```
// R1 = 1+2+...+100
  sum=0
  n=1
loop:
  if n=101 goto end
  sum=sum+n
  n=n+1
  goto loop
end:
  R1=sum
```


Copyright © Shimon Schocken

Endnote I: Turing machine (1935)

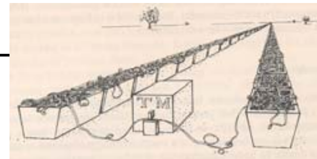


Alan Turing
1912 - 1954

Informal description:

- A *tape*, divided into *cells*, each containing a symbol
- A *head* that can move over the tape left and right and read and write symbols
- A *state register* that stores the machine's state
- An action table (transition function):
  If the current state is *S*, and the current symbol is *s*, then move the tape *n* positions right/left, write a symbol *s'*, and enter state *S'*.
- Important conjecture: for any program running on any computer there is an equivalent TM that does the same thing.

The Halting Problem



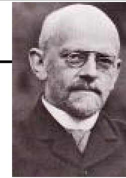
- Program = data: a TM program can be written on the tape on another TM, becoming its input
- The halting problem: a program H that, for any given program p, prints 1 if p halts on any input, and 0 otherwise
- The halting theorem: H does not exist
- Theoretical significance: If H existed, it would imply that we can prove theorems automatically. Example:

```
// Goldbach conjecture: every even number greater than 2 is the sum of two primes.
Function goldbach()
    i = 4
    while true {
       if i = sum of two primes {
          i = i + 2
       else {
          print("the conjecture is false. Counter example: ",i)
          return
       }
}
```

- If H existed, we could apply it to the goldbach() function, thus proving or disprove the Goldbach conjecture.

Copyright © Shimon Schocken

Historical perspective


David Hilbert
1862 - 1943

- Hilbert's challenge (1928): Can we devise a *mechanical procedure (algorithm) which could, in principle, prove or disprove any given mathematical proposition?*

Alan Turing
1912 - 1954

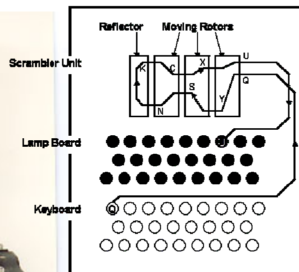
- Alan Turing (1935): NO.
  Proof: uncomputability of the halting problem

Kurt Godel
1906 - 1978

- Kurt Godel (1931): NO.
  Proof: Incompleteness theorem (any system containing the arithmetic of natural numbers is either incomplete or inconsistent)

- Philosophical implications.

Endnote II: The Enigma



- *Great book:*
  Alan Turing: The Enigma,
  by Andrew Hodges

Copyright © Shimon Schocken