School of Computer Science

COMP SCI 2000 Computer Systems
Lecture 11

adelaide.edu.au

seekLIGHT

Review – Last week

- We were talking assemblers:
  - Translation
  - Symbols
  - Symbol tables

What we're doing now

- This week we're going to keep talking about building an assembler.

What types of symbols are there?

What types of symbols are there?

- Label Symbols
- Variable Symbols
- (Virtual) Registers
- I/O Pointers
- VM Control Pointers in Virtual Machines
  - Predefined to point to special addresses in the VM
  - Overlap R0 to R4
  - More later!

How do we build a symbol table?

- What are the steps you think you need?
- Quick group discussion!

How do we build a symbol table?

- What are the steps you think you need?
- Two-pass approach
  - Initialisation: create an empty table and put any pre-defined symbols in there.
  - First pass: Go through the source code and add all of the user-defined labels to the table.
    - What goes in here?
  - Second pass: Go through the source code and use symbol table to translate the commands. This is where labels get turned back into actual addresses.

The assembly process (detailed)

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

- First pass: march 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

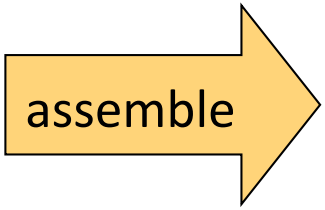
The assembly process (detailed)

- Second pass: march again 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
    - 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.

Example

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
    @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
```



Target 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
```

Note that comment lines and pseudo-commands (label declarations) generate no code.

Proposed assembler implementation

An assembler program can be written in any high-level language.

This is a *language-independent* design, as follows. Why is this important?

Software modules:

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

Proposed implementation stages

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

Parser

- What does a parser have to do?
  - "Unpacks each command into its underlying fields"
- What are the steps we need to take to turn a file full of text into some kind of internal representation of assembly commands?
- Group exercise!
  - Write down the steps to get text turned into commands that we could then convert into code.
  - You may assume you have a file of assembly code already.
  - Think about each line and element.
  - You may want to review the code specification.

Parser

- The parser will take the assembly code in the file and put it into a form that can then be directly converted to code.
- Parsers get more complicated as languages get more complicated – this one is actually very simple.
- Let's look at an example of *parsing* some lines:

```
@i
M=1    // i = 1
@sum
M=0    // sum = 0
```

Parser (a software module in the assembler program)

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.

Parser (a software module in the assembler program)

`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 (a software module in the assembler program)

- What do we need to do in order to turn parsed information into actual code?

Code (a software module in the assembler program)

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.

SymbolTable (a software module in the assembler program)

- What does our symbol table look like?
- What is it storing?
- How is it being stored?
- Which operations does it need to support?
- Group work!

SymbolTable (a software module in the assembler program)

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`.

Perspective

- Simple machine languages have a simple assembler
- Many assemblers are not stand-alone, but rather encapsulated in a translator of a higher order
- We can often improve upon the process.
  - C programmers that understand the code generated by a C compiler can improve their code considerably
- C programming (e.g. for real-time systems) may involve re-writing critical segments in assembly, for optimization
  - Why?
- Writing an assembler is an excellent practice for writing more challenging translators.

Next lecture

- New lecturer – Dr Fred Brown.
- We'll be looking at virtual machines.
  - What are they in this context?
  - Why use VMs?
- You have a tutorial this week.
- Keep working on your next assignment.
- Questions?