Topic 3 – Overview of an Assembler

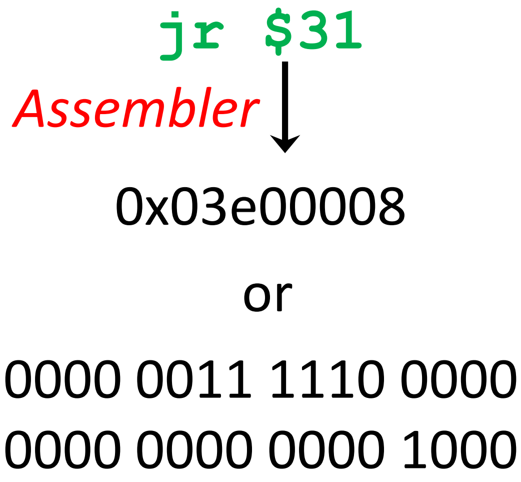
Key Ideas

- the purpose of an assembler
- a binary file vs. an ASCII representation of a binary file
- The two passes an Assembler takes:
  - Pass 1 Analysis
  - Pass 2 Synthesis
- syntactic and semantic errors
- scanning and tokens
- intermediate representation
- the symbol table

The Assembler

Overview

- *An assembler converts an assembly language program* (i.e. what you created in Assignment 2) *into its corresponding machine code* (i.e. what you created Assignment 1).



```
jr $31
```

Assembler

0x03e00008

or

0000 0011 1110 0000 0000 0000 0000 1000

- In Assignment 1: *you were* the assembler.
- In Assignment 2: *you used* the assembler cs241.binasm.
- In Assignments 3 and 4: *you will create* (most of) a small assembler.

The Assembler

Overview

- The input to an assembler is a text file containing a sequence of assembly language instructions, e.g. `jr $31`
- The *input file is represented in ASCII text,* i.e. something that can be edited with a text editor.
- The *output is a binary file,* i.e. something which cannot be edited with a text editor, which encode MIPS instructions.
- For a single MIPS instruction the file would be 4 bytes long.
- You can view with xxd.
- *This is not the same as an ASCII text file* that contains the a sequence of 1's and 0's that represent the `jr $31` instruction, which would be 32 bytes long (since each 0 or 1 is an ASCII character).

The Assembler

Steps in the Process

- We take two passes through the code: *Analysis* and *Synthesis*
- *Pass 1:*

  Read in the text file containing MIPS assembly language instructions and

  - *Analysis 1:* Scan each line, breaking it into components.
  - *Analysis 2:* Parse components, checking for errors.
- *Pass 2:*
  - *Synthesis:* Construct equivalent binary MIPS machine code.
  - Output the binary MIPS machine code.

The Assembler: Analysis

Pass 1 Analysis

- *The input* is a text file containing a sequence of assembly language instructions, e.g.

```
total: beq $1, $2, end    ; $1 total cost
```

- *Purpose: to recognize components in the instructions*
- break down each line of assembly language into *tokens*
  - LABEL: declaration of a label, e.g. total:, end:, main:, …
  - ID: an opcode (e.g. add, sub, jr, bne, …) or the use of a label without a colon (e.g. `end` in the `beq` instruction above)
  - REGISTER: e.g. $0, $1, $2, …
  - INT, HEXINT
  - DOTWORD: e.g. the .word directive
  - LPAREN, RPAREN, COMMA, WHITESPACE, ERR

The Assembler: Analysis

Pass 1 Analysis

- We will provide code (in C++, Racket and Scala), called a *scanner,* that will handle reading in the file and breaking it *down each line into a series of tokens* for you, e.g.

```
main:  lis $1
       .word 1
```

Token: LABEL {main:}
Token: ID {lis}
Token: REGISTER {$1} 1
Token: DOTWORD {.word}
Token: INT {1} 1

This means, of course, you can only do the rest of the assignments in one of these 3 languages.

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- This pass checks for *syntax errors*, i.e. *improper form or structure*.
- e.g. in English: Look at the barking brown big two dogs.
- e.g. in MIPS assembly language
  - error: lw $1
  - error: lw $3 0($4)
  - error: lw $3, 0($4
  - error: lw lw $3, 0($4)
  - error: lw $3, $4, $5
  - error: lw $3, 9999999999($4)

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- This pass checks for *semantic errors,* i.e. *what does it mean*?
- In English the sentence “Colorless green ideas sleep furiously.” (N. Chomsky) is grammatically correct but meaningless.
- In MIPS assembly language a semantic error would be defining the same label twice. If you encountered that label in a beq instruction you would not know which of the two locations to branch to. I.e. semantic analysis answers the question: What does this label mean here?
- The version the we use is officially documented here: https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsasm.html
- In Assignments 5 and 6 you learn how to formally describe a language.

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- For CS241: just recognize proper form and call everything else an error, no need to identify the type of error

The output is

1. an *intermediation representation*
   which a form of the input that is easy to work with and
2. the *Symbol Table*
   which maps labels, such as `total:`, to addresses (such as 0x0000 001c)

The Symbol Table

Pass 1 Analysis: Input

```
main:   lis $2
        .word main
        add $3,$0,$0
top:    add $3,$3,$2
        lis $1
        .word 1
        sub $2,$2,$1
        bne $2,$0,next
        bne $0,$0,top
next:   mult $3,$4
        mflo $4
        slt $6,$5,$4
```

Output: Symbol Table

- maps labels to addresses e.g.

Label	Address
main	0x0000
top	0x000C
next	0x0024

Intermediate Representation

Pass 1 Analysis: Intermediate Representation

At the very least, intermediate representation

- removes comments
- creates tokens
- keeps your program as ASCII / Unicode characters

More elaborate versions of intermediate representation

- take a bigger step towards representing elements of the program as machine code rather than ASCII

The Assembler: Synthesis

Pass 2 Synthesis

- *The input* is the intermediation representation and the symbol table (i.e. the output from the analysis pass).
- *The purpose* is to translate
  - the intermediate representation into machine code,
  - the labels into addresses.
- *The output is* machine code for a particular processor.

The Assembler

Why Two Passes?

- A label can be used before it is defined (especially in the equivalent of an if … then… else statement)

```
        bne $1, $0, next        ; if r1==0
        add $2, $2, $4          ; then r2 += r4
next:   sw $2 0($3)
```

- Two labels can refer to each other

```
prev:   bne $1, $0, next
        …
next:   beq $1, $0, prev
```

- So in the first past if you may encounter a label before it is defined.