CIS 314
Assembly Languages and Compilers

Prof. Michel A. Kinsky
The Full View System

Applications Software

Hardware

Systems Software
The Full View System

Applications

Operating System

Compiler

Firmware

ISA

Processor

Memory organization

I/O system

Datapath & Control

Digital Design

Circuit Design

Layout
Application Side

• Higher-level languages
  ‣ Allow the programmer to think in a more natural language and for their intended use
  ‣ Improve programmer productivity
  ‣ Improve program maintainability
  ‣ Allow programs to be independent of the computer on which they are developed
    • Compilers and assemblers can translate high-level language programs to the binary instructions of any machine
• Higher-level languages
  ‣ Allow the programmer to think in a more natural language and for their intended use
  ‣ Improve programmer productivity Improve program maintainability
  ‣ Allow programs to be independent of the computer on which they are developed
  ‣ Emergence of optimizing compilers that produce very efficient assembly code
  ‣ As a result, very little programming is done today at the assembler level
System Software Side

• System software
  ‣ Operating system – supervising program that interfaces the user’s program with the hardware (e.g., Linux, MacOS, Windows)
    • Handles basic input and output operations
    • Allocates storage and memory
    • Provides for protected sharing among multiple applications
  ‣ Compiler – translate programs written in a high-level language (e.g., C, Java) into instructions that the hardware can execute
Application Compiling Process

- **C Language**

    [Diagram]

    - C program
    - compiler
    - assembly code
    - assembler
    - object code
    - library routines
    - linker
    - executable
    - loader
    - memory

Human Readable

Machine Code
Assembly Code

• Three types of statements in assembly language
  ‣ Typically, one statement per a line

1. Executable assembly instructions
   • Operations to be performed by the processor

2. Pseudo-Instructions and Macros
   • Translated by the assembler into real assembly instructions
   • Simplify the programmer task

3. Assembler Directives
   • Provide information to the assembler while translating a program
   • Used to define segments, allocate memory variables, etc.
There are 3 main types of assembly instructions:

- **Arithmetic** - add, sub, mul, shifts, and, or, etc.
- **Load/store**
- **Conditional** - branches

Assembly language instructions have the format:

```
[label:] mnemonic [operands] [#comment]
```

- **Label**: (optional)
  - Marks the address of a memory location
  - Typically appear in data and text segments
Assembly Code

• Mnemonic
  ‣ Identifies the operation (e.g. add, sub, etc.)

• Operands
  ‣ Specify the data required by the operation
  ‣ Operands can be registers, memory variables, or constants
  ‣ In MIPS most instructions have three operands
    • L1: addiu $t0, $t0, 1  #increment $t0
Assemblers:

- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Build the machine instructions in the proper format
- Convert the data constants to internal machine representations
- Write the object program and the assembly listing
Why is assembly level view?

- To become familiar with the process of compiling a program/application (e.g., C) onto a computer system
- To know what assemblers are and what compilers do
- To understand the computer hardware view of the program/application
Why is assembly level view?

• To become familiar with the process of compiling a program/application (e.g., C) onto a computer system

• To, then, fully realize why computers are built the way they are
  ‣ In turn, you will gain new insights into how to write better and more efficient code
  ‣ And explore new opportunities in the field of embedded system programming
Illustration Through MIPS

- **.DATA directive**
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive

- **.TEXT directive**
  - Defines the code segment of a program containing instructions

- **.GLOBL directive**
  - Declares a symbol as global
Assembly Through MIPS

• Sets aside storage in memory for a variable
• May optionally assign a name (label) to the data
• Syntax:
  
  \[ \text{name[:]} \hspace{1em} \text{directive} \hspace{1em} \text{initializer} \ [, \text{initializer}] \ldots \]

  var1: .WORD 5

• .WORD Directive
  
  ‣ Stores the list as 32-bit values aligned on a word boundary

• All initializers become binary data in memory
Assembly Through MIPS

- **.BYTE Directive**
  - Stores the list of values as 8-bit bytes

- **.HALF Directive**
  - Stores the list as 16-bit values aligned on half-word boundary

- **.WORD w:n Directive**
  - Stores the 32-bit value w into n consecutive words aligned on a word boundary

- **.FLOAT Directive**

- **.DOUBLE Directive**
Assembly Through MIPS

• .ASCII Directive
  ‣ Allocates a sequence of bytes for an ASCII string

• .ASCIIZ Directive
  ‣ Same as .ASCII directive, but adds a NULL char at end of string
  ‣ Strings are null-terminated, as in the C programming language

• .SPACE n Directive
  ‣ Allocates space of n uninitialized bytes in the data segment
# Description: The program reads and prints a string

### Data segment

```
data
str: .space 10 # array of 10 bytes
```

### Code segment

```
.text
.globl main
main:
    # main program entry
    la $a0, str # $a0 = address of str
    li $a1, 10 # $a1 = max string length
    li $v0, 8 # read string
    syscall
    li $v0, 4 # Print string str
    syscall
    li $v0, 10 # Exit program
    syscall
```
If ($t5 < 0) then
{
    $s0 = 0 - $t5;
    $t1 = $t1 + 1;
}
else
{
    $s0 = $t5;
    $t2 = $t2 + 1;
}

bgez  $t5, else
    # if ($t5 is > or = zero) branch to else
sub  $s0, $zero, $t5
    # $s0 gets the negative of $t5
addi  $t1, $t1, 1
    # increment $t1 by 1
b  next
    # branch around the else code
else:
    ori $s0, $t5, 0
        # $s0 gets a copy of $t5
addi  $t2, $t2, 1
        # increment $t2 by 1
next:
While Do Assembly

$v0 = 1$

While ($a1 < $a2) do
{
    $t1 = \text{mem}[$a1];
    $t2 = \text{mem}[$a2];
    If ($t1 \neq $t2)
        go to break;
    $a1 = $a1 + 1;
    $a2 = $a2 - 1;
}
break: $v0 = 0$

li $v0, 1  # Load $v0 with the value 1
loop:
    bgeu $a1, $a2, done
    # If ($a1 >= $a2) Branch to done
    lb $t1, 0($a1)
    # Load a Byte: $t1 = \text{mem}[$a1 + 0]
    lb $t2, 0($a2)
    # Load a Byte: $t2 = \text{mem}[$a2 + 0]
    bne $t1, $t2, break
    # If ($t1 \neq $t2) Branch to break
    addi $a1, $a1, 1  # $a1 = $a1 + 1
    addi $a2, $a2, -1  # $a2 = $a2 - 1
    b loop  # Branch to loop
break:
    li $v0, 0
    # Load $v0 with the value 0
done:
$a0 = 0;
For ( $t0 = 10;
    $t0 > 0;
    $t0 = $t0 - 1)
do { $a0 = $a0 + $t0}

li $a0, 0      # $a0 = 0
li $t0, 10
# Initialize loop counter to 10
loop:
add $a0, $a0, $t0
addi $t0, $t0, -1
# Decrement loop counter
bgtz $t0, loop
# If ($t0 > 0) Branch to loop
System Calls

- Programs do input/output through system calls
- To obtain services from the operating system
- Using the syscall system services
- In MIPS:
  - Load the service number in register $v0
  - Load argument values, if any, in registers $a0, $a1, etc.
- Issue the syscall instruction
- Retrieve return values, if any, from result registers
Big Endian – Little Endian

- Processors can order bytes within a word in two ways
  - **Little Endian**
    - Least significant byte stored at lowest byte address
    - Intel IA-32, Alpha, AMD
  - **Big Endian**
    - Most significant byte stored at lowest byte address
    - SPARC, PA-RISC, IBM
Application Compiling Process

- High-level language program (in C)

```c
swap (int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Assembly language program (for MIPS)

```
swap:  sll   $2, $5, 2
      add   $2, $4, $2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```
Application Compiling Process

- A compiler is a software program that translates a human-oriented high-level programming language code into computer-oriented machine language.
Application Compiling Process

- Detailed compilation process

- More on this later when you take a course on compilers
• Symbol Table

› Identifiers are names of variables, constants, functions, data types, etc.

› Store information associated with identifiers

› Information associated with different types of identifiers can be different

› Information associated with variables are name, type, address, size (for array), etc.
Program memory management

Lower addresses → Top of the stack

Stack Segment
[Stack frames consisting of parameters, return addresses and local variables]

Variable Size

Free space

Heap Segment
[Dynamic variables managed by malloc(), free(), etc.]

Variable Size

BSS Segment
[Initialized global and static variables]

Fixed Size

Data Segment
[Initialized global and static variables]

Fixed Size

Text Segment
[Program code]

Fixed Size

Free space

Top of the stack → Bottom of the stack

Lower addresses
Application Compiling Process

- Assembly language program (for MIPS)
  ```assembly
  swap: sll $2, $5, 2
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
  ```

- Machine (object, binary) code (for MIPS)
  ```text
  000000 00000 00101 0001000010000000
  000000 00100 00010 0001000000100000
  . . .
  ```
Next Class

- Instruction Set Architecture