Orange Coast College
Business Division
Computer Science Department

CS 116- Computer Architecture

The Instructions
Topics:

• Assembly language, assemblers
  – MIPS R2000 Assembly language
  – Instruction set
  – MIPS design goals
  – Memory & registers
  – Instruction formats
  – Some MIPS instructions

• Advanced topics
  – Macros
  – Procedure calls
  – I/O
Introduction

• Machine instructions:
  – Language of the Machine
  – Primitive (w.r.t. HLL)
    • No sophisticated control flow (Loops, If-statements)
  – Restrictive
    • Arithmetic instructions operate on register operands only, two operands at a time (MIPS)
  – Different m/c language dialects for different machines
  – Once you learn one dialect, it’s easy to learn the others
Introduction

• Instruction set:
  – The complete vocabulary used by a machine

• Instruction Set Architecture (ISA):
  – An abstract interface between the hardware and the lowest-level software of a machine
  – Includes:
    • Necessary information to write correct machine-language programs
    • Specification of instructions, registers, memory size, ...etc.

• We will concentrate on MIPS- ISA
  – Used by NEC, Nintendo, Silicon Graphics, Sony
HLL Translation

- Compilers generate either machine language or assembly language object files
Some Definitions

• Compiler
  – Translates a high level language into object files or an assembly language files

• Assembler:
  – Translates assembly language into machine language (object files)

• Linker:
  – Combines object files into an executable file
    • Resolves all external & forward references

• Executable file:
  – Object files combined by the linker with library files
Some more definitions

- **Program library:** Pre-written routines
  - Collection of subroutines and functions
  - Stored in one or more files, usually in compiled form,
  - One of the earliest forms of organized code reuse.
  - Libraries are linked with the user's program to form a complete executable.

- The linking may be static linking or, in some systems, dynamic linking (DLL)
Definitions

• Global label
  – Visible outside the file

• External reference
  – Referencing a label in another file

• Forward reference
  – Referencing a label that appears later in the same file
Definitions

- **Macro:**
  - Extends assembly instructions through defining new (SW) instructions
  - **Properties**
    - A name (and possible arguments)
    - Equated with some text that is expanded at compile time
  - Information hiding device
Assembly Language Translation

- How are ‘.exe’ files generated from Assembly source file?

- Source file → Assembler → Object file
- Source file → Assembler → Object file
- Source file → Assembler → Object file

- Assembly language instructions
- Non-executable machine language instructions

- Linker → Program library → Executable file
- Linker → Program library → Executable file
- Linker → Program library → Executable file

- Executable file
- Executable machine language instructions
Object File Format

- **Object file header**
  - Program/procedure name
  - Text Size
  - Data size

- **Text segment**
  - Machine language instructions code (non-executable)
  - Identifies data & instructions that depend on absolute addresses
  - References should change if stored in a different place in memory

- **Data segment**
  - Binary data representation
  - Contains unresolved references

- **Relocation information**
  - Associates labels with addresses
  - Lists unresolved references

- **Symbol table**
  - Correspondence of source program lines with instruction addresses
  - Used by the debugger

- **Debugging information**
  - Lists unresolved references

Assembly Language

• A symbolic representation of the machine language of a specific processor.

• Advantages
  – High execution speed
  – Smaller code size

• Disadvantages:
  – Machine specific
  – Long programs
  – Difficult to read, understand, & debug
  – Lacks structure
But how do we use it in real life?

• Hybrid approach:
  – Most of the program written in HLL
  – Critical sections written in Assembly language

• Expansion factor:
  – Ratio of length of assembly to HLL programs
What do assemblers do?

- Most assembly instructions map directly to machine instruction
  - Translation from mnemonic to machine instruction
- Two phases:
  - Get locations of labels (build symbol table)
  - Translate statements into equivalent binary code
- Symbol Table
  - Maintains a table of variable names and properties
  - Used to help resolve forward & external referencing to create the object file
Translation

• Example: C-Program

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
    int i;
    int sum = 0;
    for (i=0; i<= 100; i=i+1)
        sum = sum +i*i;
    printf("The sum from 0 .. 100 is %d\n", sum);
}
```
Translation

- Example: Equivalent Assembly Program (No labels)

```
addiu $29, $29, -32  # add immediate unsigned
sw   $31, 20($29)  # $29 = $sp stack pointer
sw   $4, 32($29)  # $31 = $ra return address
sw   $5, 36($29)  #
sw   $0, 24($29)  # Store relevant values
sw   $0, 28($29)  # onto stack
lw   $14, 28($29) #
lw   $24, 24($29) #
multi $14,$14      # multiply unsigned
addiu $8,$14,1    #
slti  $1,$8,101    # set $1 if < immediate
sw   $8, 28($29)  #
mflo$15           # move from lo of register
addu $25,$24,$15
bne  $1,$0,-9
sw   $25, 24($29)
lui  $4,4096      # load upper immediate
lw   $5, 24($29)
jal  1048812      # jump & link
addiu $4,$4,1072
lw   $31, 20($29)
addiu $29,$29,32
jr    $31        # jump register
move  $2, $0
```
Translation

• Example: Equivalent Assembly Program (Labeled)

```assembly
.text
.align 2
.globl main
main: subu $sp,$sp, 32  # incr stack by a stack frame
    sw $ra,20($sp)  # Save return address
    sd $a0,32($sp)  # pseudo-instr. save doubleword
    sw $0,24($sp)
    sw $0,28($sp)
loop: lw $t6,28($sp)
    mul $t7,$t6,$t6
    lw $t8,24($sp)
    addu $t9,$%8,$t7
    sw $t9,24($sp)
    addu $t0,$t6,1
    sw $t0,28($sp)
    ble $t0,100,loop
    la $a0,str  # Pseudo-instr. Load address
    lw $a1,24($sp)
    jal printf  # Jump & link
move $v0,$0
lw $ra,20($sp)
addu $sp,$sp,32
jr $ra
.data
.align0  # Turn off automatic alignment
str: .asciiz “The sum from 0..100 is %d\n”
```
Pseudo-Instructions

• Instructions provided by an assembler, but not implemented in HW

• Make assembly programming easier without complicating HW
  – Macros can replace pseudo-instructions
Assembly vs. Machine Language

- **Assembly Language:**
  - Provides convenient symbolic representation
    - Easier than writing down numbers
    - Operands can be written in a different order than in the instruction (e.g., destination first)
  - Provide 'pseudo-instructions'
    - Example:
      
      \[
      \textit{move} \; \$t0, \; \$t1
      \]
      
      - exists only in Assembly
    - would be implemented using
      
      \[
      \textit{add} \; \$t0,\$t1,\$zero
      \]

- **Machine language:**
  - The underlying reality
    - e.g., destination is no longer first
  - When considering performance count real machine instructions, not pseudo-instructions
Linker

- Combines object files from separate modules
- Resolves references among files
- Searches for library routines needed by the program
- Determines memory locations needed by each module
- Adjusts absolute references accordingly
- Often done both at compile time and load time
The Loader

- Loader
  - Loads program into memory and provides actual memory references
  - Usually an operating system component
MIPS Memory Map

- Memory is typically in 3 parts
  - Text segment:
    - Holds program instructions
  - Data segment
    - Static data:
      - Size is known to the compiler
      - Lifetime = all program execution period
    - Dynamic data (Heap)
      - Size is known at runtime
      - Allocated at runtime
  - Stack segment
    - Local variables allocated
MIPS Processors

• 3 processors:
  – Main processor:
    • For integer arithmetic
  – Coprocessor 0:
    • For exceptions, interrupts, & virtual memory system
  – Coprocessor 1:
    • For floating-point arithmetic
MIPS Processors

CPU
  Registers
    $0
    ...
    $31
  Arithmetic Unit
  Multiply Divide
    Lo
    Hi

Coprocessor 0 (traps & memory) registers
  BadVAddr
  Status
  Cause
  EPC

Coprocessor 1 (FPU)
  Registers
    $0
    ...
    $31
  Arithmetic Unit
MIPS Processors

- Addressing modes:
  - Describe the manner in which addresses for memory accesses are constructed

- MIPS is a “Load-Store” architecture
  - Only load/store instructions access memory

- Data should be aligned
  - (usually multiple of 4 bytes)

- More details:
  - [http://www.ece.gatech.edu/academic/courses/ece2030/readings/instructions/spim.pdf](http://www.ece.gatech.edu/academic/courses/ece2030/readings/instructions/spim.pdf)
MIPS Addressing Terminology

• (register)
  – Contents of register

• imm
  – immediate

• imm (register)
  – immediate + contents of register

• label
  – address of label

• label ± imm
  – address of label ± immediate

• label ± imm [register]
  – address of label ±( immediate+contents of register)
Summary of Addressing Modes

1. Immediate addressing

```
| op | rs | rt | Immediate |
```

2. Register addressing

```
| op | rs | rt | rd | funct |
```

3. Base addressing

```
| op | rs | rt | Address |
```

4. PC-Relative addressing

```
| op | rs | rt | Address |
```

5. Pseudo addressing

```
| op | Address |
```

PC

```
| Address |
```

Memory

```
Byte  Halfword  Word
```

Registers

```
Register
```

Memory

```
Word
```

Memory

```
Word
```

Bye

```
Halword
```
Memory Organization

- Essentially a large, single-dimensional array
  - Each cell has an address
- Address is supplied by instruction
- Memory address
  - An index into the array, starting at 0

```
0  8 bits of data
1  8 bits of data
2  8 bits of data
3  8 bits of data
4  8 bits of data
5  8 bits of data
6  8 bits of data
```
Different ways to address memory

• **Byte addressing:**
  - Index points to a byte of memory
  - $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$

• **Words are aligned at word boundaries**
  - Most data items use "words"

• **MIPS- Word is 32 bits (4 bytes)**
  - $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{30}-4$

Registers hold 32 bits of data

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

...
What about the instructions?

• When designing an ISA, need to consider
  – Instruction length:
    • Variable length instructions:
      – Assembler needs to keep track of all instruction sizes to
determine the position of the next instruction
    • Fixed length instructions:
      – Require less housekeeping
  – Number of operands
    • Depend on type of instruction
MIPS Instruction Formats

• Instructions are fixed length
  – 32 bits long
  – Example:  \texttt{add \$t0, \$s1, \$s2}

• Registers have numbers,
  – \$t0=8, \$s1=17, \$s2=18

• MIPS has 3 instruction formats
  – R-type (Register) format
  – J-type (Jump) format
  – I-type (Immediate) format
MIPS Instruction Formats

• R-Format (Register):
  – \( op \): Operation code (6-bits)
  – \( rs \): 1\(^{\text{st}}\) source register (5-bits)
  – \( rt \): 2\(^{\text{nd}}\) source register (5-bits)
  – \( rd \): Destination register (5-bits)
  – \( shamt \): Shift amount (5-bits)
  – \( funct \): Function code (6-bits)

  • The first (\( op \)) & last fields (\( funct \)), combined, indicate the type of instruction
  • Second (\( rs \)) & third (\( rt \)) fields are the source operands
  • Fourth field (\( rd \)) is the destination operand

\[
\begin{array}{ccccccc}
0 & 17 & 18 & 8 & 0 & 32 \\
000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
op & rs & rt & rd & shamt & funct
\end{array}
\]
# Registers Names & Numbers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register#</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results &amp; expr. evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
SPIM

• Software simulator for running MIPS R-Series processors’ programs

• Why use a simulator?
  – MIPS workstations
    • Not always available
    • Difficult to understand & program
  – Simulator
    • Better programming environment
    • Provide more features
    • Easily modified
Some Features of SPIM

- SPIM Simulator simulates most of the functions of three MIPS processors
- More about SPIM next week
MIPS Peek

• Assembly Syntax
  – Comments
    • Starts with the sharp sign ‘#’ until end of line
  – Identifiers
    • Alphanumeric, ‘_’, and ‘.’
    • Can’t start with a digit
  – Labels
    • At beginning of line, followed by ‘:’
MIPS Assembly Language

• Numbers
  – Decimal (default)
  – Hexadecimal

• Strings
  – Enclosed in double quotes ‘ ” ’

• Special characters
  \n  new line
  \t  tab
  \”  quote
MIPS Assembler Directives

• Used to give information to the assembler
  – Describe data or references

  .align n  # Align data on $2^n$ byte boundary
  .ascii str  # Store strings without null termination
  .asciiiz str  # Store null-terminated string
  byte b1, ..., bn  # Store in successive bytes
  .data <addr>  # Store in data segment
  .double dl1, ..., dn  # Store floating-pt. double-precision
  .extern sym size  # External symbol with ‘size’ bytes
  .float fl1, ..., fn  # Store floating-pt. single precision
  .globl sym  # sym can be referenced from other files
  .half h1, ..., hn  # Half-words in successive memory
  .space n  # Allocate n-bytes of space in data seg.
  .text <addr>  # Put text items in text segment
  .word w1, ..., wn  # Full word in successive memory
MIPS Assembler Directives

• Example:

\[ \text{\texttt{\textasciiz \text{\textit{\textup{The sum from 0..10 is } \%d\n}}} \]

Is equivalent to:

\[ \begin{align*}
\text{\texttt{\text{.byte 84, 104, 101, 32, 115, 117, 109, 32}}} \\
\text{\texttt{\text{.byte 102, 114, 111, 109, 32, 48, 32, 46}}} \\
\text{\texttt{\text{.byte 46, 32, 49, 48, 32, 105, 115, 32}}} \\
\text{\texttt{\text{.byte 37, 100, 10, 0}}}
\end{align*} \]

• Note:
  – 10 is ASCII code for line feed
  – 0 is ASCII code for the NULL character

• For more Assembler directives see the course web page