Systems Architecture I

Topics
Assemblers, Linkers, and Loaders*
Alternative Instruction Sets**

*This lecture was derived from material in the text (sec. 3.8-3.9).
**This lecture was derived from material in the text (sec. 3.12-3.15).


Notes Courtesy of Jeremy R. Johnson
Systems Architecture I

Topic 1: Assemblers, Linkers, and Loaders
Introduction

• Objective: To learn how the MIPS assembler, linker, and loader work. To review the MIPS instruction set and encoding.

• Topics
  – Immediate instructions
  – Addressing in branches and jumps
  – Review MIPS instruction set and addressing modes
  – Instruction formats and encoding
  – Assembler
  – Linker and object files
  – Loader and executable files
MIPS Instruction Set

- **Arithmetic/Logical**
  - add, sub, and, or
  - addi, andi, ori

- **Data Transfer**
  - lw, lb
  - sw, sb
  - lui

- **Control**
  - beq, bne
  - slt, slti
  - j, jal, jr
Immediate Addressing

- Many arithmetic operations involve small constants

- Example: (offset of 4 from stack pointer)
  - lw $t0, AddressofConstant4($zero)
  - add $sp, $sp, $t0

- Remove overhead of extra load by storing constant in the instruction itself

I-format instructions store a constant in the low order 16-bits

- Example:
  - addi $sp, $sp, 4

| 8 | 29 | 29 | 4 |
Loading a 32-Bit Constant

- Since MIPS only allows 16-bit constants (common case)
- Two instructions are required to create a 32-bit constant

- `li $s0, 0x003c0900`
  - `lui $s0, 0x003c;` load upper immediate instruction
  - `ori $s0, $s0, 0x0900`

- Book uses addi, but this doesn’t work in general due to negative numbers

- When constructing 32 bit offsets for memory addresses the MIPS assembler uses the $at register for temporary space
Addressing in Branches and Jumps

- J-Format instructions
- Example: `j 10000`

<table>
<thead>
<tr>
<th>2</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>26-bits</td>
</tr>
</tbody>
</table>

- Branches require 2 operand registers and an address ⇒ Use I-Format with 16-bit constant
- This implies that programs be restricted to $2^{16}$ bytes, which is unacceptable. However, branches are typically to addresses near the current address.
- PC-relative addressing
  - address computed relative to PC+4
Branch Examples

• Bne $s0, $s1, Exit ; goto Exit if $s0 ≠ $s1

  5  16  17           Exit

  16-bits

• Since instruction address are word aligned, branch address is treated as a word address (two extra bits)

• Branches to addresses more than ±2^15 words away must use two instructions

• Example: replace “beq $s0, $s1, FarAway” with
  – bne $s0, $s1, next
  – j FarAway
  – next:
MIPS Encoding

- All instructions are 32-bits long
- Opcode is always in the high-order 6 bits
- Only three instruction formats
- 32 registers implies 5 bit register addresses:
  - $zero R0 ; zero register always equal to 0
  - $at R1 ; temporary register (assembler temporary)
  - $v0 - $v1 R2-R3 ; return registers
  - $a0 - $a3 R4-R7 ; argument registers
  - $t0 - $t7 R8-R15 ; temporary - not preserved across calls
  - $s0 - $s7 R16-R23 ; saved registers - preserved across calls
  - $t8 - $t9 R24-R25 ; temporary not preserved across calls
  - $k0 - $k1 R26-R27 ; reserved by OS kernel
  - $gp R28 ; global pointer
  - $sp R29 ; stack pointer
  - $fp R30 ; frame pointer
  - $ra R31 ; return address
# MIPS Instruction Formats

- **R format** (register format - add, sub, ...) 

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

- **I format** (immediate format - lw, sw, ...) 

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
</table>

- **Example:** `lw $s1, 100($s2)` 

  | 35 | 18 | 17 | 100 |

- **Example:** `add $t0, $s0, $s1` 

  | 0 | 16 | 17 | 8 | 0 | 32 |
MIPS Addressing Modes

• Immediate Addressing
  – 16 bit constant from low order bits of instruction
  – addi $t0, $s0, 4

• Register Addressing
  – add $t0, $s0, $s1

• Base Addressing (displacement addressing)
  – 16-bit constant from low order bits of instruction plus base register
  – lw $t0, 16($sp)

• PC-Relative Addressing
  – (PC+4) + 16-bit address (word) from instruction
  – bne $s0, $s1, Target

• Pseudodirect Addressing
  – high order 4 bits of PC+4 concatenated with 26 bit word address - low order 26 bits from instruction shifted 2 bits to the left
  – j Address
Example

Loop: add $t1, $s3, $s3
     add $t1, $t1, $t1
     add $t1, $t1, $s6
     lw  $t0, 0($t1)
     bne $t0, $s5, Exit
     add $s3, $s3, $s4
     j   Loop

Exit:
Decoding Machine Code

- Example:

```
0000 0000 1010 1111 1000 0000 0010 0000
```
Design Principles

- Simplicity favors regularity
- Smaller is faster
- Good design demands good compromises
- Make common cases fast
Translation Hierarchy

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory
Assembler

- Translates assembly code to machine code
- creates object file
- Symbolic labels to addresses
- Pseudoinstructions (move, la, li, blt, bgt, ...)
- Assembly directives (.text, .globl, .space, .byte, .asciiz,...)
- Loading a 32-bit constant (lui and ori)
- Constructing 32-bit addresses (use $at)
- Branching far away (beq $s0, $s1, L1 => bne and j)
Format of Object File

• Object file header (size and position)
• Text segment (instructions - machine code)
• Data segment (data that comes with the program, both static and dynamic)
• Relocation information (instructions and data words that depend on absolute addresses when program is loaded into memory)
• Symbol table (external references)
• Debugging information
Linker

- Place code and data modules symbolically in memory
- Determine the addresses of data and instruction labels
- Patch both the internal and external references
MIPS Memory Convention

- $sp \rightarrow 7fffff \text{ hex}
- $gp \rightarrow 1000 \text{ hex}
- 10000000 \text{ hex}
- pc \rightarrow 00400000 \text{ hex}
- 0

Areas:
- Stack
- Dynamic data
- Static data
- Text
- Reserved
Loader

- Read executable file header to determine size of text and data segments
- Creates an address space large enough for the text and data
- Copies instructions and data from executable file into memory
- Copies parameters (if any) to the main program onto the stack
- Initializes the machine registers and sets stack pointer to first free location
- Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.
## Linking Example

**Object File Header**

<table>
<thead>
<tr>
<th></th>
<th>Procedure A</th>
<th></th>
<th>Procedure B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text size</td>
<td>0x100</td>
<td></td>
<td>0x200</td>
<td></td>
</tr>
<tr>
<td>Data size</td>
<td>0x20</td>
<td></td>
<td>0x30</td>
<td></td>
</tr>
</tbody>
</table>

**Text Segment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw $a0, 0($gp)</td>
<td>lw</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>jal 0</td>
<td>jal</td>
<td>B</td>
</tr>
</tbody>
</table>

**Data Segment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(X)</td>
</tr>
</tbody>
</table>

**Relocation Information**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw</td>
<td>lw</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>jal</td>
<td>jal</td>
<td>B</td>
</tr>
</tbody>
</table>

**Symbol Table**

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
</tr>
</tbody>
</table>

**Object File Header**

<table>
<thead>
<tr>
<th></th>
<th>Procedure B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text size</td>
<td>0x200</td>
<td></td>
</tr>
<tr>
<td>Data size</td>
<td>0x30</td>
<td></td>
</tr>
</tbody>
</table>

**Text Segment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sw $a1, 0($gp)</td>
<td>sw</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>jal 0</td>
<td>jal</td>
<td>A</td>
</tr>
</tbody>
</table>

**Data Segment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Y)</td>
</tr>
</tbody>
</table>

**Relocation Information**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sw</td>
<td>sw</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>jal</td>
<td>jal</td>
<td>A</td>
</tr>
</tbody>
</table>
# Resulting Executable File

<table>
<thead>
<tr>
<th>Executable File Header</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Text size</td>
<td>0x300</td>
</tr>
<tr>
<td>Data size</td>
<td>0x50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text Segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x00400000</td>
<td>lw $a0, 0x8000($gp)</td>
</tr>
<tr>
<td></td>
<td>0x00400004</td>
<td>jal 400100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00400100</td>
<td>lw $a0, 0x8020($gp)</td>
</tr>
<tr>
<td></td>
<td>0x00400104</td>
<td>jal 400000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Segment</th>
<th>Address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x10000000</td>
<td>(X)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x10000020</td>
<td>(Y)</td>
</tr>
</tbody>
</table>

...
Systems Architecture I

Topic 2: Alternative Instruction Sets
Introduction

- **Objective**: To compare MIPS to several alternative instruction set architectures and to better understand the design decisions made in MIPS.
- **MIPS** is an example of a RISC (Reduced Instruction Set Computer) architecture as compared to a CISC (Complex Instruction Set Computer) architecture.
- **MIPS** trades complexity of instructions and hence greater number of instructions, for a simpler implementation and smaller clock cycle or reduced number of clocks per instruction.

- **Topics**
  - RISC vs. CISC
  - PowerPC
  - Intel 80x86
  - Historical survey
Characteristics of MIPS

- Load/Store architecture
- General purpose register machine (32 registers)
- ALU operations have 3 register operands (2 source + 1 dest)
- 16 bit constants for immediate mode
- Simple instruction set
  - Simple branch operations (beq, bne)
  - Use register to set condition (e.g. slt)
  - Operations such as move built, li, blt from existing operations
- Uniform encoding
  - All instructions are 32-bits long
  - Opcode is always in the high-order 6 bits
  - 3 types of instruction formats
  - Register fields in the same place for all formats
Design Principles

• Simplicity favors regularity
  – uniform instruction length
  – all ALU operations have 3 register operands
  – register addresses in the same location for all instruction formats

• Smaller is faster
  – register architecture
  – small number of registers

• Good design demands good compromises
  – fixed length instructions and only 16 bit constants
  – several instruction formats but consistent length

• Make common cases fast
  – immediate addressing
  – 16 bit constants
  – only beq and bne
MIPS Addressing Modes

• Immediate Addressing
  – 16 bit constant from low order bits of instruction
  – addi $t0, $s0, 4

• Register Addressing
  – add $t0, $s0, $s1

• Base Addressing (displacement addressing)
  – 16-bit constant from low order bits of instruction plus base register
  – lw $t0, 16($sp)

• PC-Relative Addressing
  – (PC+4) + 16-bit address (word) from instruction
  – bne $s0, $s1, Target

• Pseudodirect Addressing
  – high order 4 bits of PC+4 concatenated with 26 bit word address - low order 26 bits from instruction shifted 2 bits to the right
  – j Address
PowerPC

- Similar to MIPS (RISC)
- Two additional addressing modes
  - Indexed addressing - base register + index register
    - PowerPC: `lw $t1, $a0+$s3`
    - MIPS: `add $t0, $a0,$s3`
      `lw $t1, 0($t0)`
  - Update addressing - displacement addressing + increment
    - PowerPC: `lwu $t0, 4($s3)`
    - MIPS: `lw $t0, 4($s3)`
      `addi $s3, $s3, 4`
- Additional instructions
  - Separate counter register used for loops
  - PowerPC: `bc Loop, ctr!=0`
  - MIPS: `Loop:
    addi $t0, $t0, -1
    bne $t0, $zero, Loop`
Characteristics of 80x86

- Evolved from 8086 (backward compatible)
- Register-Memory architecture
- 8 General purpose registers (evolved)
- 7 data addressing modes
  - 8 or 32 bit displacement
- Complex instruction set
  - Many variants of an instruction
  - special instructions (move, push, pop, string, decimal)
  - Use condition codes
  - Instructions can operate on 8, 16, or 32 bits (mode) changed with prefix
- Variable length encoding
  - 1-17 bytes, postbyte used to indicate addressing mode when not in opcode
Intel 80x86 History and Backward Compatibility

- 1978
  - Intel 8086 (16-bit architecture with dedicated registers)
- 1980
  - Intel 8087 floating point coprocessor (with stack)
- 1982
  - 80286 (24-bit address space using elaborate memory mapping and protection model)
- 1985
  - 80386 (extends 80286 to 32-bit architecture, new instructions, general purpose register usage)
- 1989-95
  - 80486, Pentium Pro (a few extra instructions, but mainly performance enhancements)
- 1997-present
  - MMX enhancements, Pentium II, Pentium III, SSE, IA-64
80x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
Addressing Modes

- **Register indirect**
  - address in register
  - can’t use ESP or EBP

- **Based mode with 8 or 32-bit displacement**
  - address is contents of base register plus displacement
  - can’t use ESP or EBP

- **Base plus scaled index (not in MIPS)**
  - Base + \(2^{\text{scale}} \times \text{index}\)
  - index not ESP

- **Base plus scaled index 8 or 32-bit plus displacement (not in MIPS)**
  - Base + \(2^{\text{scale}} \times \text{index}\) + displacement
  - index not ESP
Example Instructions

- **JE name**
  - if equal(condition code) EIP = name, EIP - 128 < name < EIP + 128
- **JMP name**
  - EIP = name
- **CALL name**
  - SP = SP - 4; M[SP] = EIP + 5; EIP = name
- **MOVW EBX,[EDI+45]**
  - EBX = M[EDI+45]
- **PUSH ESI**
  - SP = SP - 4; M[SP] = ESI
- **POP EDI**
  - EDI = M[SP]; SP = SP + 4
Example Instructions

- **ADD EAX,#6765**
  - \( \text{EAX} = \text{EAX} + 6765 \)

- **TEST EDX,#42**
  - set condition code(flags) with EDX and 42 (logical and)

- **MOVSL**
  - \( \text{M}[\text{EDI}] = \text{M}[\text{ESI}]; \ \text{EDI} = \text{EDI} + 4; \ \text{ESI} = \text{ESI} + 4 \)
80x86 Instruction Encoding

a. JE EIP + displacement

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>Condition</td>
<td>Displacement</td>
</tr>
</tbody>
</table>

b. CALL

<table>
<thead>
<tr>
<th>8</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Offset</td>
</tr>
</tbody>
</table>

c. MOV EBX, [EDI + 45]

<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
<th>1</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>d</td>
<td>w</td>
<td>r-m</td>
<td>postbyte</td>
</tr>
</tbody>
</table>

d. PUSH ESI

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>Reg</td>
</tr>
</tbody>
</table>

e. ADD EAX, #6765

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>1</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Reg</td>
<td>w</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

f. TEST EDX, #42

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>8</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>w</td>
<td>Postbyte</td>
<td>Immediate</td>
</tr>
</tbody>
</table>
Architecture Evolution

• Accumulator
  – EDSAC

• Extended Accumulator (special purpose register)
  – Intel 8086

• General Purpose Register
  – register-register (CDC 6600, MIPS, SPARC, PowerPC)
  – register-memory (Intel 80386, IBM 360)
  – memory-memory (VAX)

• Alternative
  – stack
  – high-level language