Instruction Set Architectures
Part I: From C to MIPS

Readings: 2.1- 2.14
Goals for this Class

• Understand how CPUs run programs
  • How do we express the computation the CPU?
  • How does the CPU execute it?
  • How does the CPU support other system components (e.g., the OS)?
  • What techniques and technologies are involved and how do they work?

• Understand why CPU performance (and other metrics) varies
  • How does CPU design impact performance?
  • What trade-offs are involved in designing a CPU?
  • How can we meaningfully measure and compare computer systems?

• Understand why program performance varies
  • How do program characteristics affect performance?
  • How can we improve a programs performance by considering the CPU running it?
  • How do other system components impact program performance?
Goals

- Understand how we express programs to the computer.
  - The stored-program model
  - The instruction set architecture
- Learn to read and write MIPS assembly
- Prepare for your 141L Project and 141 homeworks
  - Your book (and my slides) use MIPS throughout
  - You will implement a subset of MIPS in 141L
- Learn to “see past your code” to the ISA
  - Be able to look at a piece of C code and know what kinds of instructions it will produce.
  - Begin to understand the compiler’s role
  - Be able to roughly estimate the performance of code based on this understanding (we will refine this skill throughout the quarter.)
**Have you had CSE30?**

<table>
<thead>
<tr>
<th>Select</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes, this year.</td>
</tr>
<tr>
<td>B</td>
<td>Yes, last year.</td>
</tr>
<tr>
<td>C</td>
<td>Yes, an equivalent course at another school.</td>
</tr>
<tr>
<td>D</td>
<td>No.</td>
</tr>
</tbody>
</table>
The Idea of the CPU
History Slides
The Stored Program Computer

• The program is *data*
  • It is a series of bits
  • It lives in memory
  • A series of discrete “instructions”

• The program counter (PC) control execution
  • It points to the current instruction
  • Advances through the program
The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute.
- All programs are combinations of these instructions.
- It is an abstraction that programmers (and compilers) use to express computations.
  - The ISA defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.
- The hardware can implement those rules IN ANY WAY IT CHOOSES!
  - Directly in hardware.
  - Via a software layer (i.e., a virtual machine).
  - Via a trained monkey with a pen and paper.
  - Via a software simulator (like SPIM).
- Also called “the big A architecture.”
Which of the following statement is generally true about ISAs?

<table>
<thead>
<tr>
<th>Select</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Many models of processors can support one ISA.</td>
</tr>
<tr>
<td>B</td>
<td>An ISA is unique to one model of processor.</td>
</tr>
<tr>
<td>C</td>
<td>Every processor supports multiple ISAs.</td>
</tr>
<tr>
<td>D</td>
<td>Each processor manufacturer has its own unique ISA.</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>

The MIPS ISA
We Will Study Two ISAs

• **MIPS**
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
  - The basis for your implementation project in 141L
  - Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)

• **x86**
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

You will learn to write MIPS code and implement a MIPS processor.

You will learn to read a common subset of x86.
MIPS Basics

• Instructions
  • 4 bytes (32 bits)
  • 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  • Instructions operate on memory and registers

• Memory Data types (also aligned)
  • Bytes -- 8 bits
  • Half words -- 16 bits
  • Words -- 32 bits
  • Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)

• Registers
  • 32 4-byte registers in the “register file”
  • Denoted “R” (e.g., R[2] is register 2)

• There’s a handy reference on the inside cover of your text book and a detailed reference in Appendix B.
## Bytes and Words

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x1513</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

### In modern ISAs (including MIPS) memory is “byte addressable”

### In MIPS, half words and words are aligned.
The MIPS Register File

• All registers are the same
  • Where a register is needed any register will work
• By convention, we use them for particular tasks
  • Argument passing
  • Temporaries, etc.
  • These rules ("the register discipline") are part of the ISA
• $zero is the "zero register"
  • It is always zero.
  • Writes to it have no effect.

<table>
<thead>
<tr>
<th>Name</th>
<th>number</th>
<th>use</th>
<th>Callee saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assemble Temp</td>
<td>no</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2 - 3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4 - 7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8 - 15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16 - 23</td>
<td>saved temporaries</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24 - 25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26 - 27</td>
<td>Res. for OS</td>
<td>yes</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
In each column we have which - reg or mem - is better. Which row is correct?

<table>
<thead>
<tr>
<th></th>
<th>Faster access</th>
<th>Fewer bits to specify</th>
<th>More locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Mem</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>B</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>C</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>D</td>
<td>Reg</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS R-Type Arithmetic Instructions

- R-Type instructions encode operations of the form “a = b OP c” where ‘OP’ is +, -, <<, &, etc.

- Bit fields
  - “opcode” encodes the operation type.
  - “funct” specifies the particular operation.
  - “rs” are “rt” source registers; “rd” is the destination register
    - 5 bits can specify one of 32 registers.
  - “shamt” is the “shift amount” for shift operations
    - Since registers are 32 bits, 5 bits are sufficient

Examples
- \texttt{add \$t0, \$t1, \$t2}
  - opcode = 0, funct = 0x20
- \texttt{nor \$a0, \$s0, \$t4}
  - \( R[4] = \sim (R[16] \mid R[12]) \)
  - opcode = 0, funct = 0x27
- \texttt{sll \$t0, \$t1, 4}
  - \( R[4] = R[16] \ll 4 \)
  - opcode = 0, funct = 0x0, shamt = 4
MIPS R-Type Control Instructions

- R-Type encodes “register-indirect” jumps
- Jump register
  - jr rs: PC = R[rs]
- Jump and link register
  - jalr rs, rd: R[rd] = PC + 8; PC = R[rs]
  - rd default to $ra (i.e., the assembler will fill it in if you leave it out)

Examples
- jr $t2
- PC = r[10]
- opcode = 0, funct = 0x8
- jalr $t0
- PC = R[8]
- R[31] = PC + 8
- opcode = 0, funct = 0x9
- jalr $t0, $t1
- PC = R[8]
- R[9] = PC + 8
- opcode = 0, funct = 0x9
MIPS I-Type Arithmetic Instructions

- I-Type arithmetic instructions encode operations of the form “a = b OP #”
  - ‘OP’ is +, -, <<, &, etc and # is an integer constant
    - More formally, e.g.: R[rt] = R[rs] + 42
- Components
  - “opcode” encodes the operation type.
  - “rs” is the source register
  - “rd” is the destination register
- “immediate” is a 16 bit constant used as an argument for the operation

Examples
- addi $t0, $t1, -42
- opcode = 0x8
- ori $t0, $zero, 42
  - R[4] = R[0] | 42
  - opcode = 0xd
  - Loads a constant into $t0
### MIPS I-Type Branch Instructions

<table>
<thead>
<tr>
<th>I-Type</th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>31-26</td>
<td>rs</td>
<td>rt</td>
<td>Immediate</td>
</tr>
<tr>
<td></td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **I-Type also encode branches**
  - if \((R[rd] \text{ OP } R[rs])\)
    \[ \text{PC} = \text{PC} + 4 + 4 \times \text{Immediate} \]
  - else
    \[ \text{PC} = \text{PC} + 4 \]

- **Components**
  - “rs” and “rt” are the two registers to be compared
  - “rt” is sometimes used to specify branch type.

- **“immediate” is a 16 bit branch offset**
  - It is the signed offset to the target of the branch
  - Limits branch distance to 32K instructions
  - Usually specified as a label, and the assembler fills it in for you.

### Examples

- \(\text{beq } $t0, $t1, -42\)
  - if \(R[8] == R[9]\)
    \[ \text{PC} = \text{PC} + 4 + 4 \times -42 \]
  - opcode = 0x4

- \(\text{bgez } $t0, -42\)
  - if \(R[8] \geq 0\)
    \[ \text{PC} = \text{PC} + 4 + 4 \times -42 \]
  - opcode = 0x1
  - rt = 1