Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We’ll be working with the MIPS instruction set architecture
  similar to other architectures developed since the 1980’s
  used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  to be read or written just like data
- Programs are stored in memory

- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register “control” the subsequent actions
  - Fetch the “next” instruction and continue

Processor

Memory

memory for data, programs, compilers, editors, etc.

Architecture Specification

- Data types:
  - bit, byte, bit field, signed/unsigned integers logical, floating point, character
- Operations:
  - data movement, arithmetic, logical, shift/rotate, conversion, input/output, control, and system calls
- # of operands:
  - 3, 2, 1, or 0 operands
- Registers:
  - integer, floating point, control
- Instruction representation as bit strings

Characteristics of Instruction Set

- Complete
  - Can be used for a variety of application
- Efficient
  - Useful in code generation
- Regular
  - Expected instruction should exist
- Compatible
  - Programs written for previous versions of machines need it
- Primitive
  - Basic operations
- Simple
  - Easy to implement
- Smaller
  - Implementation

Example of multiple operands

- Instructions may have 3, 2, 1, or 0 operands
- Number of operands may affect instruction length
- Operand order is fixed (destination first, but need not that way)

  add $s0, $s1, $s2 ; Add $s2 and $s1 and store result in $s0
  add $s0, $s1    ; Add $s1 and $s0 and store result in $s0
  add $s0       ; Add contents of a fixed location to $s0
  add            ; Add two fixed locations and store result

Where operands are stored

- Memory locations
  - Instruction include address of location
- Registers
  - Instruction include register number
- Stack location
  - Instruction opcode implies that the operand is in stack
- Fixed register
  - Like accumulator, or depends on inst
  - Hi and Lo register in MIPS
- Fixed location
  - Default operands like interrupt vectors
MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]
MIPS code: \[ \text{add} \ $s0, \ $s1, \ $s2 \]

(asociated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code: \[ A = B + C + D; \]
\[ E = F - A; \]
MIPS code: \[ \text{add} \ $t0, \ $s1, \ $s2 \]
\[ \text{add} \ $s0, \ $t0, \ $s3 \]
\[ \text{sub} \ $s4, \ $s5, \ $s0 \]

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?
  - More register will slow register file down.

Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables

<table>
<thead>
<tr>
<th>Control</th>
<th>Memory</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ...
|---|---|---|---|---|---|---|-----|
| Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | ...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ...
|---|---|---|---|---|---|---|-----|
| Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | ...

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ...
|---|---|---|---|---|---|---|---|---|---|------|---|---|-----|
| Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | ...

Addressing within a word

- Each word has four bytes
- Which byte is first and which is last
- Two Choices
  - Least significant byte is byte "0" -> Little Endian
  - Most significant byte is byte "0" -> Big Endian

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ...
|---|---|---|---|---|---|---|---|---|---|------|---|---|-----|
| Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | Bits of data | ...
Instructions

• Load and store instructions
  • Example:
    
    C code: \[ A[h] = h + A[h]; \]
    
    MIPS code: 
    \[
    \text{lw } \$t0, 32(\$s3) \\
    \text{add } \$t0, \$s2, \$t0 \\
    \text{sw } \$t0, 32(\$s3)
    \]
  
  • Store word has destination last
  • Remember arithmetic operands are registers, not memory!

Addressing

• Memory address for load and store has two parts
  • A register whose content are known
  • An offset stored in 16 bits
  • The offset can be positive or negative
    • It is written in terms of number of bytes
    • It is but in instruction in terms of number of words
    • 32 byte offset is written as 32 but stored as 8
  • Address is content of register + offset
  • All address has both these components
    • If no register needs to be used then use register 0
    • If no offset, then offset is 0

Our First Example

• Can we figure out the code?

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

```mips
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $15, 0($2)
sw $16, 4($2)
jr $31
```

So far we’ve learned:

• MIPS
  • loading words but addressing bytes
  • arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = \text{Memory} [$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>\text{Memory} [$s2+100] = $s1</td>
</tr>
</tbody>
</table>

Machine Language

• Instructions, like registers and words of data, are also 32 bits long
  • Example: add $t0, $s1, $s2
    • registers have numbers, $t0=9, $s1=17, $s2=18
  • Instruction Format:
    \[
    \begin{array}{cccccccc}
    000000 & 0001 & 10010 & 00000 & 00000 & 100000 \\
    \text{op} & \text{rs} & \text{rt} & \text{shamt} & \text{funct} \\
    \end{array}
    \]

• Can you guess what the field names stand for?

Machine Language

• Consider the load-word and store-word instructions.
  • What would the regularity principle have us do?
    • New principle: Good design demands a compromise
  • Introduce a new type of instruction format
    • I-type for data transfer instructions
    • other format was R-type for register

<table>
<thead>
<tr>
<th>Example</th>
<th>lw $t0, 32($s2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>35</td>
</tr>
<tr>
<td>rs</td>
<td>18</td>
</tr>
<tr>
<td>rt</td>
<td>9</td>
</tr>
<tr>
<td>16 bit number</td>
<td>32</td>
</tr>
</tbody>
</table>

• Where’s the compromise?
**Control**

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:
  
  - `bne $t0, $t1, Label`
  
  - `beq $t0, $t1, Label`
- Example:
  
  - `if (i==j) h = i + j;`
  
  - `bne $s0, $s1, Label`
  
  - `add $s3, $s0, $s1`
  
  - `Label: ....`

**Conditional Execution**

- A simple conditional execution
- Depending on imj or ihj, result is different

**Instruction Sequencing**

- MIPS unconditional branch instructions:
  
  - `j label`
- Example:
  
  - `f, g, and h are in registers $s3, $s4, and $s5`
  
  - `if (i<=j) beq $s4, $s5, Lab1`
  
  - `f=g-h; sub $s3, $s4, $s5`
  
  - `else j exit`
  
  - `f=g+h; Lab1: add $s3, $s4, $s5`
  
  - `exit: ....`

- Can you build a simple for loop?

**So far:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $s1, $s2, $s3</code></td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td><code>sub $s1, $s2, $s3</code></td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td><code>lw $s1, 100($s2)</code></td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td><code>sw $s1, 100($s2)</code></td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td><code>beq $s4, $s5, Lab1</code></td>
<td>Next instr. is at Label if $s4 = $s5</td>
</tr>
<tr>
<td><code>bne $s4, $s5, Lab2</code></td>
<td>Next instr. is at Label if $s4 ° $s5</td>
</tr>
<tr>
<td><code>j Lab1</code></td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

- Formats:
  
<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>op</td>
<td>26 bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Control Flow**

- We have: `beq`, `bne`, what about Branch-if-less-than?
- New instruction:
  
  - `slt $t0, $s1, $s2`  
  
  - `else $t0 = 0`
- Can use this instruction to build "bit $s1, $s2, Label" — can now build general control structures
  
  - Note that the assembler needs a register to do this,
  
  - there are policy of use conventions for registers

**Constants**

- Small constants are used quite frequently (50% of operands)
  
  - `A = A + 5;`
  
  - `B = 0 + 1;`
  
  - `C = C + 1;`
- Solutions? Why not?
  
  - put 'typical constants' in memory and load them.
  
  - create hard-wired registers (like `$zero`) for constants like one.
- MIPS Instructions:
  
  - `addi $29, $29, 4`
  
  - `slti $8, 0, 10`
  
  - `andi $29, $29, 6`
  
  - `ori $29, $29, 4`
  
  - `How do we make this work?`
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

\[\begin{array}{cccc}
R & 0 & 1 & 2 & 3 \\
I & 4 & 5 & 6 & 7 \\
J & 8 & 9 & 10 & 11 \\
& 12 & 13 & 14 & 15 \\
& 16 & 17 & 18 & 19 \\
& 20 & 21 & 22 & 23 \\
& 24 & 25 & 26 & 27 \\
& 28 & 29 & 30 & 31 \\
\end{array}\]

- rely on compiler to achieve performance
  - what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- Instructions:
  - bne $t4,$t5,Label Next instruction is at Label if $t4 \neq t5$
  - beq $t4,$t5,Label Next instruction is at Label if $t4 = t5$
  - j Label Next instruction is at Label

- Formats:
  - I op rs rt 16 bit address
  - J op 26 bit address

- Addresses are not 32 bits
  - How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:
  - bne $t4,$t5,Label Next instruction is at Label if $t4 \neq t5$
  - beq $t4,$t5,Label Next instruction is at Label if $t4 = t5$

- Formats:
  - I op rs rt 16 bit address

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
- most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

To summarize:

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0-s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t0-t7</td>
<td>8-15</td>
<td>arguments</td>
</tr>
<tr>
<td>$a0-a3</td>
<td>4-7</td>
<td>16-23</td>
</tr>
<tr>
<td>$v0-v1</td>
<td>2-3</td>
<td>24-25 temporary</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Other Issues

- support for procedures (Refer to section 3.6), stacks, frames, recursion
- manipulating strings and pointers
- linkers, loaders, memory layout
- Interrupts, exceptions, system calls and conventions
- Register use convention
Stack Manipulation

- Register $29 is used as stack pointer
- Stack grows from high address to low address
- Stack pointer should point to the last filled address
- Once entries are removed, stack pointer should be adjusted

Frame Pointer

- Stores the last address for the last frame
- When completing a subroutine, frame address can be used as the starting stack pointer value

Stack Manipulation

- Register $29 is used as stack pointer
- Stack grows from high address to low address
- Stack pointer should point to the last filled address
- Once entries are removed, stack pointer should be adjusted

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction
  \[
  \text{lui } \$t0, 1010101010101010
  \]
  \[
  \text{ori } \$t0, \$t0, 1010101010101010
  \]

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: \[ \text{lw } \$t1, \$a0+\$s3 \] \( \text{\$t1 = Memory[\$a0+\$s3]} \)
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
    - example: \[ \text{lwu } \$t0,4(\$s3) \]$t0=Memory[\$s3+4];\$s3=\$s3+4
    - What do we have to do in MIPS?
  - Others:
    - load multiple/store multiple
    - a special counter register “bc Loop”
      decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the “golden handcuffs” of compatibility"
"adding new features as someone might add clothing to a packed bag"
"an architecture that is difficult to explain and impossible to love"