ISA is a Contract

- Between the programmer and the hardware:
  - Defines visible state of the system
  - Defines how the state changes in response to instructions
- Programmer obtains a model of how programs will execute
- Hardware designer obtains a formal definition of the correct way to execute instructions
- ISA Specification:
  - Instruction set
  - How instructions modify the state of the machine
  - Binary representation
- Today: MIPS ISA
ISA is a Contract

- Between the programmer and the hardware:
  - Defines visible state of the system
  - Defines how the state changes in response to instructions
- Programmer obtains a model of how programs will execute
- Hardware designer obtains a formal definition of the correct way to execute instructions
- ISA Specification:
  - Instruction set
  - How instructions modify the state of the machine
  - Binary representation
- Today: MIPS ISA
ISA Specification

Instruction formats
Instruction types
Addressing modes

Instruction Representation

Data types
Operations
Interrupts/Events

Machine state
Memory organization
Register organization

Before State

After State

Mem
Regs

Mem
Regs

Op Mode Ra Rb
MIPS ISA

- 32 GP registers (R0-R31) – 32 bits each
- 32 FP registers (F0-F31) – 32 bits each
  - 16 double-precision (using adjacent 32-bit registers)
- 8-, 16-, and 32-bit integer & 32- and 64-bit floating point data types
- Load/Store architecture (no memory access in ALU ops)
- A few, simple addressing modes:
  - Immediate: R1 ← 0x21
  - Displacement: R1 ← 0x100(R2)
- Simple fixed instruction format
  - Three types
  - <100 instructions
- Fused compare and branch
- Pseudo instructions
- Designed for pipelining and ease of compilation
### MIPS ISA: A Visual

#### Instruction Categories
- Computational
- Load/Store
- Jump and Branch
- Floating Point
- Memory Management
- Special

#### Registers
- R0 - R31
- PC
- HI
- LO

#### Memory
- 32 bits wide

#### 3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R format**
- **I format**
- **J format**

---

University of Texas at Austin   CS352H - Computer Systems Architecture   Fall 2009   Don Fussell   6
### MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$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 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>reserved for interrupts/traps</td>
<td>n.a.</td>
</tr>
<tr>
<td>$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 addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
### MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $8, $9, $10</td>
<td>$8 = $9 + $10</td>
<td>3 opnds; Exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $8, $9, $10</td>
<td>$8 = $9 - $10</td>
<td>3 opnds; Exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $8, $9, 100</td>
<td>$8 = $9 + 100</td>
<td>+ const; Exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $8, $9, $10</td>
<td>$8 = $9 + $10</td>
<td>3 opnds; No exception</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $8, $9, $10</td>
<td>$8 = $9 - $10</td>
<td>3 opnds; No exception</td>
</tr>
<tr>
<td>add imm. Unsиг.</td>
<td>addiu $8, $9, 100</td>
<td>$8 = $9 + 100</td>
<td>+ const; No exception</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $8, $9</td>
<td>Hi,Lo = $8 * $9</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $8, $9</td>
<td>Hi,Lo = $8 * $9</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $8, $9</td>
<td>Lo = $8 ÷ $9 , Hi = $8 mod $9</td>
<td>Lo = quotient , Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $8, $9</td>
<td>Lo = $8 ÷ $9 , Hi = $8 mod $9</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td>move from Hi</td>
<td>mfhi $8</td>
<td>$8 = Hi</td>
<td></td>
</tr>
<tr>
<td>move from Lo</td>
<td>mflo $8</td>
<td>$8 = Lo</td>
<td></td>
</tr>
</tbody>
</table>

Which add for address arithmetic? Which for integers?
Multiply & Divide

- **Start multiply/Divide**
  - `mult rs, rt`
  - `multu rs, rt`
  - `div rs, rt`
  - `divu rs, rt`

- **Move result from Hi or Lo**
  - `mfhi rd`
  - `mflo rd`

- **Move to Hi or Lo (Why?)**
  - `mthi rd`
  - `mtlo rd`
## MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $8, $9, $10</td>
<td>$8 = $9 &amp; $10</td>
<td>3 opnds; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $8, $9, $10</td>
<td>$8 = $9</td>
<td>$10</td>
</tr>
<tr>
<td>xor</td>
<td>xor $8, $9, $10</td>
<td>$8 = $9 ⊕ $10</td>
<td>+ const; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $8, $9, $10</td>
<td>$8 = ~( $9</td>
<td>$10)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $8, $9, 10</td>
<td>$8 = $9 &amp; 10</td>
<td>Logical AND; Reg, Const</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $8, $9, 10</td>
<td>$8 = $9</td>
<td>10</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $8, $9, 10</td>
<td>$8 = $9 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $8, $9, 10</td>
<td>$8 = $9 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>sra $8, $9, 10</td>
<td>$8 = $9 &gt;&gt; 10</td>
<td>Const srl with sign extension</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slv $8, $9, $10</td>
<td>$8 = $9 &lt;&lt; $10</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>sr $8, $9, $10</td>
<td>$8 = $9 &gt;&gt; $10</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>sra $8, $9, $10</td>
<td>$8 = $9 &gt;&gt; $10</td>
<td>Var. srl with sign extension</td>
</tr>
</tbody>
</table>
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)

- **Big Endian:** leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian:** rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

- Nowadays endian-ness is configurable

```
big endian byte 0
msb 3 2 1 0
lsb 0 1 2 3

little endian byte 0
```
## MIPS Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>load word</td>
<td>lw $8, 100($9)</td>
<td>$8 = Mem[$9 + 100]</td>
</tr>
<tr>
<td>load half word</td>
<td>lh $8, 102($9)</td>
<td>$8 = Mem[$9 + 102]</td>
</tr>
<tr>
<td>load half word unsigned</td>
<td>lhu $8, 102($9)</td>
<td>$8 = Mem[$9 + 102]</td>
</tr>
<tr>
<td>load byte</td>
<td>lb $8, 103($9)</td>
<td>$8 = Mem[$9 + 103]</td>
</tr>
<tr>
<td>load byte unsigned</td>
<td>lbu $8, 103($9)</td>
<td>$8 = Mem[$9 + 103]</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui $8, 47</td>
<td>Upper 16 bits of $8 = 47</td>
</tr>
<tr>
<td>store word</td>
<td>sw $8, 100($9)</td>
<td>Mem[$9 + 100] = $8</td>
</tr>
<tr>
<td>store half word</td>
<td>sh $8, 102($9)</td>
<td>Mem[$9 + 102] = $8</td>
</tr>
<tr>
<td>store byte</td>
<td>sb $8, 103($9)</td>
<td>Mem[$9 + 103] = $8</td>
</tr>
</tbody>
</table>

- Where do half words & bytes get placed on lh & lb?
- What happens to the rest of the word on sb?
MIPS Compare and Branch

- **Compare and branch**
  - `beq rs, rt, offset` if \( R[rs] = R[rt] \) then PC-relative branch
  - `bne rs, rt, offset` \(!=\)

- **Compare to zero and branch**
  - `blez rs, offset` if \( R[rs] \leq 0 \) then PC-relative branch
  - `bgtz rs, offset` \(>\)
  - `bltz` \(<\)
  - `bgez` \(\geq\)
  - `bltzal rs, offset` if \( R[rs] < 0 \) then branch and link (into R31)
  - `bgezal` \(\geq\)

- Remaining compare and branches take two instructions
- Almost all comparisons are against zero
  - Hence \( 0 \) is always \( 0! \)
## MIPS Jump, Branch & Compare Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $8, $9, 100</td>
<td>If $(8 == 9) goto PC+4+100</td>
</tr>
<tr>
<td>branch on not equal</td>
<td>bne $8, $9, 100</td>
<td>If $(8 != 9) goto PC+4+100</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $8, $9, $10</td>
<td>If $(9 &lt; 10) then $8 = 1 else $8 = 0;</td>
</tr>
<tr>
<td>set less than immed.</td>
<td>slti $8, $9, 100</td>
<td>If $(9 &lt; 100) then $8 = 1 else $8 = 0;</td>
</tr>
<tr>
<td>set less than unsig.</td>
<td>sltu $8, $9, $10</td>
<td>If $(9 &lt; 10) then $8 = 1 else $8 = 0;</td>
</tr>
<tr>
<td>set less than immed. unsig.</td>
<td>sltiu $8, $9, 100</td>
<td>If $(9 &lt; 100) then $8 = 1 else $8 = 0;</td>
</tr>
<tr>
<td>jump</td>
<td>j 10004</td>
<td>goto 10004</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>goto $31</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10004</td>
<td>$31 = PC + 4; goto 10004</td>
</tr>
</tbody>
</table>
**Procedure Support**

- **Caller**
  - Save $t0 - $t9, if needed
  - Save $fp
  - Push arguments on stack (and leave in $a0-$a3)
  - Set $fp to point to arg’s
  - Adjust $sp
  - jal

- **Callee**
  - Save $s0-$s7, $ra as needed
  - Get arg’s
  - Do its thing
  - Put return values in $v0, $v1
  - jr $ra

- **Preserved state**
  - $s0 - $s7
  - $sp
  - $ra

- **Not preserved**
  - $t0 - $t9
  - $a0 - $a3
  - $v0, $v1

```small
jal foo
Return here
```
### Machine Language: R-format

- **R format instructions: three operands**

  \[
  \text{add } \$t0, \$s1, \$s2
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>5-bits</td>
<td>5-bits</td>
<td>5-bits</td>
<td>5-bits</td>
<td>6-bits</td>
</tr>
</tbody>
</table>

- **op** 6-bits opcode that specifies the operation
- **rs** 5-bits register file address of the first source operand
- **rt** 5-bits register file address of the second source operand
- **rd** 5-bits register file address of the result’s destination
- **shamt** 5-bits shift amount (for constant shift instructions)
- **funct** 6-bits function code augmenting the opcode
Machine Language: I-format

- **I-format Instructions:**
  - Load/store
  - Branches
  - Arithmetic with an immediate operand

```
lw $t0, 24($s2)
```

- **Fields in I-format instructions:**
  - **op** 6-bits: opcode that specifies the operation
  - **rs** 5-bits: register file address of the first source operand
  - **rt** 5-bits: register file address of the second source operand
  - **offset** 16-bits: a constant value
Machine Language: J-format

- J-format instructions:
  - Jump
  - Jump and Link

<table>
<thead>
<tr>
<th>op</th>
<th>26-bit address</th>
</tr>
</thead>
</table>

- What about Jump Register?
MIPS Operand Addressing Modes

- **Register addressing** – operand is in a register
  
  op  rs  rt  rd  funct

  Register

  word operand

- **Base (displacement) addressing** – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction
  
  op  rs  rt  offset

  Memory

  word or byte operand

  base register

- **Register relative (indirect) with** 0($a0)
- **Pseudo-direct with** addr($zero)

- **Immediate addressing** – operand is a 16-bit constant contained within the instruction
  
  op  rs  rt  operand
MIPS Instruction Addressing Modes

- **PC-relative addressing** – instruction address is the sum of the PC and a 16-bit constant contained within the instruction.

  - op  
  - rs  
  - rt  
  - offset

- **Pseudo-direct addressing** – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC.

  - op  
  - jump address

  - Program Counter (PC)
MIPS Organization So Far

**Processor**

- **Register File**
  - 32 registers ($zero - $ra)
  - 32 bits
  - src1 addr
  - src2 addr
  - dst addr
  - write data

- **PC**
  - 32 bits
  - branch offset

- **ALU**
  - 32 bits
  - 32

- **Fetch**
  - PC = PC + 4

- **Decode**
  - 32

- **Add**
  - 32

- **Exec**
  - 32

**Memory**

- 2^{30} words
- 1...1100
- 0...0000
- 0...1100
- 0...1000
- Read/Write addr
- Read data
- Write data
- 32 bits
- Word address (binary)

- Byte address (big Endian)

- 0 1 2 3

- 4 5 6 7

- 0 1 2 3

- 0 1 2 3
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits

- Good design demands good compromises
  - three instruction formats

- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes

- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
Next Lecture

- ISA Principles
- ISA Evolution
- Make sure you’re comfortable with the contents of Ch. 2
- You will need to read Appendix B to do the programming part of the assignment