A Processor

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 2.16-20, 4.1-3
Let’s build a **MIPS CPU**

• ...but using Harvard architecture
High Level Language
- C, Java, Python, Ruby, ...
- Loops, control flow, variables

Assembly Language
- No symbols (except labels)
- One operation per statement

Machine Language
- Binary-encoded assembly
- Labels become addresses
Arithmetic
• add, subtract, shift left, shift right, multiply, divide

Memory
• load value from memory to a register
• store value to memory from a register

Control flow
• unconditional jumps
• conditional jumps (branches)
• jump and link (subroutine call)

Many other instructions are possible
• vector add/sub/mul/div, string operations
• manipulate coprocessor
• I/O
MIPS = Reduced Instruction Set Computer (RISC)

- ≈ 200 instructions, 32 bits each, 3 formats
  - mostly orthogonal
- all operands in registers
  - almost all are 32 bits each, can be used interchangeably
- ≈ 1 addressing mode: Mem[reg + imm]

x86 = Complex Instruction Set Computer (CISC)

- > 1000 instructions, 1 to 15 bytes each
- operands in special registers, general purpose registers, memory, on stack, ...
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  - e.g. Mem[segment + reg + reg*scale + offset]
MIPS register file

- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via RW
  - Writes occur on falling edge but only if WE is high
- Read ports indexed via RA, RB
MIPS Memory

- Up to 32-bit address
- 32-bit data
  (but byte addressed)
- Enable + 2 bit memory control
  00: read word (4 byte aligned)
  01: write byte
  10: write halfword (2 byte aligned)
  11: write word (4 byte aligned)
Basic CPU execution loop

1. fetch one instruction
2. increment PC
3. decode
4. execute
Instruction Fetch Circuit

- Fetch instruction from memory
- Calculate address of next instruction
- Repeat

Program Memory

PC

\[ \text{inst} \]

\[ 32 \]

\[ +4 \]

\[ 00 \]

\[ 2 \]
### Arithmetic Instructions

**Op** | **Func** | **Mnemonic** | **Description**
---|---|---|---
0x0 | 0x21 | ADDU rd, rs, rt | \( R[rd] = R[rs] + R[rt] \)
0x0 | 0x23 | SUBU rd, rs, rt | \( R[rd] = R[rs] - R[rt] \)
0x0 | 0x25 | OR rd, rs, rt | \( R[rd] = R[rs] \lor R[rt] \)
0x0 | 0x26 | XOR rd, rs, rt | \( R[rd] = R[rs] \oplus R[rt] \)
0x0 | 0x27 | NOR rd, rs rt | \( R[rd] = \neg ( R[rs] | R[rt] ) \)
Example Programs

\[ r_4 = (r_1 + r_2) \mid r_3 \]

\[ r_8 = 4 \times r_3 + r_4 - 1 \]

\[ r_9 = 9 \]
Instruction fetch + decode + ALU
= Babbage’s engine + speed + reliability – hand crank
### Arithmetic Instructions: Shift

<table>
<thead>
<tr>
<th>op</th>
<th>func</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>SLL rd, rs, shamt</td>
<td>( R[rd] = R[rt] \ll \text{shamt} )</td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>SRL rd, rs, shamt</td>
<td>( R[rd] = R[rt] \ggg \text{shamt} ) (zero ext.)</td>
</tr>
<tr>
<td>0x0</td>
<td>0x3</td>
<td>SRA rd, rs, shamt</td>
<td>( R[rd] = R[rs] \gg \text{shamt} ) (sign ext.)</td>
</tr>
</tbody>
</table>

**ex:** \( r5 = r3 \times 8 \)
### Arithmetic Instructions: Immediates

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9</td>
<td>ADDIU rd, rs, imm</td>
<td>$R[rd] = R[rs] + \text{sign	extunderscore extend}(imm)$</td>
</tr>
<tr>
<td>0xc</td>
<td>ANDI rd, rs, imm</td>
<td>$R[rd] = R[rs] &amp; \text{zero	extunderscore extend}(imm)$</td>
</tr>
<tr>
<td>0xd</td>
<td>ORI rd, rs, imm</td>
<td>$R[rd] = R[rs]</td>
</tr>
</tbody>
</table>

**Example:**
- `ex: r5 += 5`
- `ex: r9 = -1`
- `ex: r9 = 65535`
<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF</td>
<td>LUI rd, imm</td>
<td>$R[rd] = imm \ll 16$</td>
</tr>
</tbody>
</table>

ex: $r5 = 0xdeadbeef$
19

Immediates

Program Memory (Prog. Mem) provides instructions (inst) to the register file (Reg. File). The immediate value (imm) is extended (extend) and combined with the shamt (shift amount) to control the ALU. The control signals (5 5 5) guide the processing. The program counter (PC) increments by 4. The output of the ALU is 16 bits long.
**Arithmetic/Logical**

- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

**Memory Access**

- load/store between registers and memory
- word, half-word and byte operations

**Control flow**

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute
<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>LB rd, offset(rs)</td>
<td>( R[rd] = \text{sign_ext}(\text{Mem}[\text{offset}+R[rs]]) )</td>
</tr>
<tr>
<td>0x24</td>
<td>LBU rd, offset(rs)</td>
<td>( R[rd] = \text{zero_ext}(\text{Mem}[\text{offset}+R[rs]]) )</td>
</tr>
<tr>
<td>0x21</td>
<td>LH rd, offset(rs)</td>
<td>( R[rd] = \text{sign_ext}(\text{Mem}[\text{offset}+R[rs]]) )</td>
</tr>
<tr>
<td>0x25</td>
<td>LHU rd, offset(rs)</td>
<td>( R[rd] = \text{zero_ext}(\text{Mem}[\text{offset}+R[rs]]) )</td>
</tr>
<tr>
<td>0x23</td>
<td>LW rd, offset(rs)</td>
<td>( R[rd] = \text{Mem}[\text{offset}+R[rs]] )</td>
</tr>
<tr>
<td>0x28</td>
<td>SB rd, offset(rs)</td>
<td>( \text{Mem}[\text{offset}+R[rs]] = R[rd] )</td>
</tr>
<tr>
<td>0x29</td>
<td>SH rd, offset(rs)</td>
<td>( \text{Mem}[\text{offset}+R[rs]] = R[rd] )</td>
</tr>
<tr>
<td>0x2b</td>
<td>SW rd, offset(rs)</td>
<td>( \text{Mem}[\text{offset}+R[rs]] = R[rd] )</td>
</tr>
</tbody>
</table>
int h, A[];
Examples:

# r5 contains 0xDEADBEEF
sb r5, 2(r0)
lb r6, 2(r0)
sw r5, 8(r0)
lb r7, 8(r0)
lb r8, 11(r0)

0x00000000
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007
0x00000008
0x00000009
0x0000000a
0x0000000b
...
Endianness: Ordering of bytes within a memory word

**Little Endian** = least significant part first (MIPS, x86)

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>as 4 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>as 2 halfwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>as 1 word</td>
<td></td>
<td></td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

**Big Endian** = most significant part first (MIPS, networks)

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>as 4 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>as 2 halfwords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>as 1 word</td>
<td></td>
<td></td>
<td>0x12345678</td>
</tr>
</tbody>
</table>
Absolute addressing for jumps

- Jump from 0x30000000 to 0x20000000? **NO**  Reverse? **NO**
  - But: Jumps from 0x2FFFFFFF to 0x3xxxxxxx *are* possible, but not reverse
- Trade-off: out-of-region jumps vs. 32-bit instruction encoding

MIPS Quirk:
- jump targets computed using *already incremented* PC
<table>
<thead>
<tr>
<th>op</th>
<th>func</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x08</td>
<td>JR rs</td>
<td>PC = R[rs]</td>
</tr>
</tbody>
</table>
jump to 0xabcd1234
jump to 0xabcd1234

# assume 0 <= r3 <= 1
if (r3 == 0) jump to 0xdecafe0
else jump to 0xabcd1234
jump to 0xabcd1234

# assume 0 <= r3 <= 1
if (r3 == 0) jump to 0xdedcafe0
else jump to 0xabcd1234
### Control Flow: Branches

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4</td>
<td>BEQ rs, rd, offset</td>
<td>if R[rs] == R[rd] then PC = PC + 4 + (offset &lt;&lt; 2)</td>
</tr>
<tr>
<td>0x5</td>
<td>BNE rs, rd, offset</td>
<td>if R[rs] != R[rd] then PC = PC + 4 + (offset &lt;&lt; 2)</td>
</tr>
</tbody>
</table>

#### I-Type
- **op**: 6 bits
- **rs**: 5 bits
- **rd**: 5 bits
- **offset**: 16 bits

Signed offsets:
if (i == j) { i = i * 4; }
else { j = i - j; }
Absolute Jump

Could have used ALU for branch add

Could have used ALU for branch cmp
Could have used ALU for branch add

Could have used ALU for branch cmp
Control Flow: More Branches

### Instruction Format

<table>
<thead>
<tr>
<th>op</th>
<th>subop</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>0x0</td>
<td>BLTZ rs, offset</td>
<td>if R[rs] &lt; 0 then PC = PC+4+ (offset&lt;&lt;2)</td>
</tr>
<tr>
<td>0x1</td>
<td>0x1</td>
<td>BGEZ rs, offset</td>
<td>if R[rs] ≥ 0 then PC = PC+4+ (offset&lt;&lt;2)</td>
</tr>
<tr>
<td>0x6</td>
<td>0x0</td>
<td>BLEZ rs, offset</td>
<td>if R[rs] ≤ 0 then PC = PC+4+ (offset&lt;&lt;2)</td>
</tr>
<tr>
<td>0x7</td>
<td>0x0</td>
<td>BGTZ rs, offset</td>
<td>if R[rs] &gt; 0 then PC = PC+4+ (offset&lt;&lt;2)</td>
</tr>
</tbody>
</table>

**操码 (op)**: 6 bits

**寄存器索引 (rs)**: 5 bits

**子操作码 (subop)**: 5 bits

**偏移量 (offset)**: 16 bits

*almost I-Type*

*signed offsets*
Could have used ALU for branch cmp
### J-Type

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3</td>
<td>JAL target</td>
<td>r31 = PC+8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = (PC+4)$_{32..29}$</td>
</tr>
</tbody>
</table>
Could have used ALU for link add