Review: Salient features of MIPS I

- 32-bit fixed format inst (3 formats)
- 32 32-bit GPR (R0 contains zero) and 32 FP registers (+ HI LO)
  – partitioned by software convention
- 3-address, reg-reg arithmetic instr.
- Single address mode for load/store: base+displacement
  – no indirection, scaled
- 16-bit immediate plus LUI
- Simple branch conditions
  – compare against zero or two registers for =, ≠
  – no integer condition codes
- Support for 8bit, 16bit, and 32bit integers
- Support for 32bit and 64bit floating point.

Review: MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register (direct)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td></td>
<td></td>
<td></td>
<td>immed</td>
</tr>
<tr>
<td>Base+index</td>
<td></td>
<td></td>
<td>rt</td>
<td>immed</td>
</tr>
<tr>
<td>PC-relative</td>
<td></td>
<td></td>
<td>rt</td>
<td>immed</td>
</tr>
</tbody>
</table>

Review: Details of the MIPS instruction set

- Register zero **always** has the value zero (even if you try to write it)
- Branch/jump and link put the return addr.
  PC+4 into the link register (R31), also called “ra”
- All instructions change **all 32 bits** of the destination register
  (including lui, lb, lh) and all read all 32 bits of sources (add, and, …)
- The difference between signed and unsigned versions:
  – For add and subtract: signed causes exception on overflow
    - No difference in sign-extension behavior!
  – For multiply and divide, distinguishes type of operation
- Thus, overflow can occur in these arithmetic and logical instructions:
  – add, sub, addi
  – it **cannot** occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu,
    div, divu
- Immediate arithmetic and logical instructions are extended as follows:
  – logical immediates ops are zero extended to 32 bits
  – arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  – [bu, lh are zero extended
  – lb, lh are sign extended
**Calls: Why Are Stacks So Great?**

Stacking of Subroutine Calls & Returns and Environments:

A:
- CALL B
- CALL C
- RET

B:
- CALL C
- RET

C:
- RET

Some machines provide a memory stack as part of the architecture (e.g., VAX)
Sometimes stacks are implemented via software convention (e.g., MIPS)
Some machines provide stack support in hardware (Embedded processor)

**Memory Stacks**

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:

Next Empty?
- Big → Little: Last Full
  - SP
  - Full?
  - Last

How is empty stack represented?
- Big → Little: Next Empty
  - SP
  - Full?
  - Last

**MIPS: Software conventions for Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp; function results</td>
</tr>
<tr>
<td>3</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>temporary: caller saves</td>
</tr>
<tr>
<td>15</td>
<td>ra Return Address (HW)</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>(callee can clobber)</td>
</tr>
<tr>
<td>18</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>temp (call must save)</td>
</tr>
<tr>
<td>20</td>
<td>...</td>
</tr>
<tr>
<td>21</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>22</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp Pointer to kernel</td>
</tr>
<tr>
<td>29</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra Return Address (HW)</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>...</td>
</tr>
<tr>
<td>18</td>
<td>temp (call can clobber)</td>
</tr>
<tr>
<td>19</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>...</td>
</tr>
<tr>
<td>21</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>22</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>...</td>
</tr>
<tr>
<td>24</td>
<td>...</td>
</tr>
<tr>
<td>25</td>
<td>...</td>
</tr>
<tr>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>27</td>
<td>...</td>
</tr>
<tr>
<td>28</td>
<td>...</td>
</tr>
<tr>
<td>29</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
</tbody>
</table>

**Call-Return Linkage: Stack Frames**

ARGs
- ARGs
- Callee Save Registers
- (old FP, RA)
- Local Variables
- FP
- SP

Reference args and local variables at fixed (positive) offset from FP

Grows and shrinks during expression evaluation

- Many variations on stacks possible (up/down, last pushed / next )
- Compilers normally keep scalar variables in registers, not memory!
MIPS / GCC Calling Conventions

**fact:**
- addiu $sp, $sp, -32
- sw $ra, 20($sp)
- sw $fp, 16($sp)
- addiu $fp, $sp, 32
- sw $a0, 0($fp)
- lw $ra, 20($sp)
- lw $fp, 16($sp)
- addiu $sp, $sp, 32
- jr $ra

First four arguments passed in registers
Result passed in $v0/$v1

---

Performance: Two notions of “performance”

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (pmph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>BAD/Sud Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

Which has higher performance?

° Time to do the task (Execution Time)
  - execution time, response time, latency
° Tasks per day, hour, week, sec, .. (Performance)
  - throughput, bandwidth

Response time and throughput often are in opposition

---

Definitions

- Performance is in units of things-per-second
  - bigger is better
- If we are primarily concerned with response time
  - performance(x) = \( \frac{1}{\text{execution}_{\text{time}}(x)} \)

"X is n times faster than Y" means

\[ n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} \]

---

Example

° Time of Concorde vs. Boeing 747?
  - Concord is 1350 mph / 610 mph = 2.2 times faster
    = 6.5 hours / 3 hours

° Throughput of Concorde vs. Boeing 747?
  - Concord is 178,200 pmph / 286,700 pmph = 0.62 "times faster"
  - Boeing is 286,700 pmph / 178,200 pmph = 1.60 "times faster"

° Boeing is 1.6 times ("60%") faster in terms of throughput
° Concord is 2.2 times ("120%") faster in terms of flying time

We will focus primarily on execution time for a single job
Lots of instructions in a program => Instruction throughput important!
**Amdahl’s Law**

Speedup due to enhancement E:

\[
\text{Speedup}(E) = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}
\]

Suppose that enhancement E accelerates a fraction F of the task by a factor S and the remainder of the task is unaffected, then,

\[
\text{ExTime(with E)} = ((1-F) + \frac{F}{S}) \times \text{ExTime(without E)}
\]

\[
\text{Speedup(with E)} = 1 + \frac{F}{S}
\]

**Basis of Evaluation**

**Pros**
- representative
- portable
- widely used
- improvements useful in reality
- easy to run, early in design cycle
- identify peak capability and potential bottlenecks

**Cons**
- very specific
- non-portable
- difficult to run, or measure
- hard to identify cause
- less representative
- easy to “fool”
- “peak” may be a long way from application performance

**Metrics of performance**

- Application: Answers per month
- Programming Language: Useful Operations per second
- Compiler: (millions) of Instructions per second – MIPS
- Datapath: (millions) of (F,P.) operations per second – MFLOP/s
- Control: Megabytes per second
- Function Units: Cycles per second (clock rate)

Each metric has a place and a purpose, and each can be misused.

**Aspects of CPU Performance**

<table>
<thead>
<tr>
<th>CPU time</th>
<th>Seconds</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. Set</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
### CPI: Average Cycles per Instruction

\[
\text{CPI} = \frac{(\text{CPU Time} \times \text{Clock Rate})}{\text{Instruction Count}} = \frac{\text{Clock Cycles}}{\text{Instruction Count}}
\]

\[
\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i \quad \text{where} \quad F_i = \frac{l_i}{\text{Instruction Count}}
\]

\[
\text{CPI} = \text{ideal CPI} + \text{Memory Stalls/Inst} + \text{Other Stalls/Inst}
\]

\[
\text{Memory Stalls/Inst} = \text{Instruction Miss Rate} \times \text{Instruction Miss Penalty} + \text{Loads/Inst} \times \text{Load Miss Rate} \times \text{Load Miss Penalty} + \text{Stores/Inst} \times \text{Store Miss Rate} \times \text{Store Miss Penalty}
\]

### Example (RISC processor)

**Base Machine (Reg / Reg)**

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1.0</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5.0</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3.0</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2.0</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

Typical Mix: 2.2

How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

### Administrative Matters

- **Sections start tomorrow!**
  - 10:00—12:00 in 310 Hearst Mining and 4:00—6:00 in 3 Evans
  - TA Office hours now posted on information page
    - Office hours in 119 Cory

- **Want announcements directly via EMail?**
  - Look at information page to sign up for “cs152-announce” mailing list.

- **Prerequisite quiz will be Monday 2/3 during class**
  - Review Sunday (2/2), 7:00 – 9:00 pm here (306 Soda)
  - Turn in survey form (with picture!)

- **Homework #1 also due Monday 2/3 at beginning of lecture!**
  - No homework quiz this time (Prereq quiz may contain homework material, since this is supposed to be review)
Example: Impact of Caches on Performance

- Basic Cache View: Cache CPI = 1 + stalls
- Suppose a processor executes at
  - Base CPI = 1.1 (no cache misses)
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- CPI = Base CPI + average stalls per instruction
  \[
  1.1 \text{(cycles/ins)} + \left[ 0.30 \left( \text{DataMops/ins} \right) \times 0.10 \left( \text{miss/DataMop} \right) \times 50 \left( \text{cycle/miss} \right) \right] + \left[ 1 \left( \text{InstMop/ins} \right) \times 0.01 \left( \text{miss/InstMop} \right) \times 50 \left( \text{cycle/miss} \right) \right]
  \]
  \[
  = (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1
  \]
- 58% of the time the proc is stalled waiting for memory!

Example: 1 KB Direct Mapped Cache with 32 B Blocks

- For a $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Tag</td>
<td>Ex: 0x50</td>
<td>Cache Index</td>
<td>0x01</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Valid Bit | Cache Tag | Cache Data
---|---|---
| Stored as part of the cache “state”

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Data</td>
<td>Byte 31</td>
<td>Byte 1</td>
<td>Byte 0</td>
<td>Byte 63</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>**</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>**</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluating Instruction Sets?

Design-time metrics:
- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:
- How many bytes does the program occupy in memory?

Dynamic Metrics:
- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

Best Metric: Time to execute the program!

NOTE: this depends on instructions set, processor organization, and compilation techniques.

Finite State Machines:

- System state is explicit in representation
- Transitions between states represented as arrows with inputs on arcs.
- Output may be either part of state or on arcs

"Mod 3 Machine"
Implementation as Combinational logic + Latch

Performance and Technology Trends

Range of Design Styles

General C/L Cell Delay Model

- Technology Power: \(1.2 \times 1.2 \times 1.2 = 1.7 \times\) / year
  - Feature Size: shrinks 10% / yr. => Switching speed improves 1.2 / yr.
  - Density: improves 1.2x / yr.
  - Die Area: 1.2x / yr.
- The lesson of RISC is to keep the ISA as simple as possible:
  - Shorter design cycle => fully exploit the advancing technology (~3yr)
  - Advanced branch prediction and pipeline techniques
  - Bigger and more sophisticated on-chip caches

- Combinational Cell (symbol) is fully specified by:
  - functional (input -> output) behavior
    » truth-table, logic equation, VHDL
  - Input load factor of each input
  - Propagation delay from each input to each output for each transition
    » \( T_{HL}(A, o) = \) Fixed Internal Delay + Load-dependent-delay \( \times \) load
- Linear model composes
Basic Technology: CMOS

- CMOS: Complementary Metal Oxide Semiconductor
  - NMOS (N-Type Metal Oxide Semiconductor) transistors
  - PMOS (P-Type Metal Oxide Semiconductor) transistors

- NMOS Transistor
  - Apply a HIGH (Vdd) to its gate turns the transistor into a “conductor”
  - Apply a LOW (GND) to its gate shuts off the conduction path

- PMOS Transistor
  - Apply a HIGH (Vdd) to its gate shuts off the conduction path
  - Apply a LOW (GND) to its gate turns the transistor into a “conductor”

Summary

- Total execution time is the most reliable measure of performance
- MIPS register conventions: used by compiler and assembler
  - Callee-Saves: Must be preserved
  - Caller-Saves: Can be used arbitrarily
- Amdall’s law: Law of Diminishing Returns
- State Machines: Technology for sequential execution
- Performance and Technology Trends
  - Keep the design simple (KISS rule) to take advantage of the latest technology
  - CMOS inverter and CMOS logic gates