Mid-Term QuickFire Review

Fall 2014
CS 465
Huzefa Rangwala, PhD

Chapters 1, 2 and 3 and 4.1-4.3
Disclaimer:

These slides are not a substitute for reading the book.

To do well: Go over the detailed readings, solve exercises from HW, Quizzes, Classroom Exercises and back of the book*. 
Detailed Readings

Performance

- Performance = 1 / Execution time
- Speedup = $T_{old} / T_{new}$
- Amdahl's Law: Make the common case fast!
- CPI (Cycles Per Instruction)
- CPU Time = $x \times y/z$
  - What are $x$, $y$, and $z$?
- Don’t forget units.
- CPI Instruction Mix
Million Instructions Per Second

\[
\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\text{Instruction count} \times \text{CPI} \times 10^6} \times \frac{\text{Clock rate}}{\text{Clock rate}} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}
\]

Ahmdal’s Law

\[
T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}
\]
Benchmarks

- Define
- How do you design effective benchmarks?

- SPEC Benchmarks
- What would a work load of AI game search test?
- What would a work load of Word Processing Test?
Exercises (WPS)

- Suppose that we can improve the floating point instruction performance of machine by a factor of 15 (the same floating point instructions run 15 times faster on this new machine). What percent of the instructions must be floating point to achieve a Speedup of at least 4?
Multicores

- State challenges
Instruction Set Architecture/ MIPS
MIPS-32 ISA

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

- Registers
  - R0 - R31

- PC - Program Counter
  - HI
  - LO

- HI,LO for Mul/Div

3 Instruction Formats: all 32 bits wide

- R format
  - op – 6b  rs-5b  rt-5b  rd-5b  sa-5b  funct-6b

- I format
  - op  rs  rt  Immediate – 16 bits

- J format
  - op  jump target – 26 bits
MIPS (RISC) Design Principles

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

- 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

- Good design demands good compromises
  - three instruction formats
Exercise (In-Class)

- C code:
  

  - h in $s2, base address of A in $s3

- Compiled MIPS code: ???
Number Conversion

- Given a number in base X, can you convert it to base Y?
- Binary (see 12c lab manual)
  - Unsigned: cannot represent negative numbers
  - Sign-Magnitude: MSb is sign bit
  - One's Complement
  - Two's Complement
  - Bias (or Excess): to get in, add bias to number; to get out, subtract bias from number
# Logical Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td><code>&lt;&lt;</code></td>
<td><code>&lt;&lt;</code></td>
<td><code>sll</code></td>
</tr>
<tr>
<td>Shift right</td>
<td><code>&gt;&gt;</code></td>
<td><code>&gt;&gt;&gt;</code></td>
<td><code>srl</code></td>
</tr>
<tr>
<td>Bitwise AND</td>
<td><code>&amp;</code></td>
<td><code>&amp;</code></td>
<td><code>and, andi</code></td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>`</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td><code>~</code></td>
<td><code>~</code></td>
<td><code>nor</code></td>
</tr>
</tbody>
</table>
Exercise (What MIPS operations do we need?)

- \( i = 5, j = 22 \)
Compiling If Statements

- **C code:**

  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

  - f, g, ... in $s0, $s1, ...

- **Compiled MIPS code:**

  ```assembly
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j   Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

  Assembler calculates addresses
Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call
Another Procedure Example

- C code:
  ```c
  int fact (int n)
  {
    if (n < 1) return f;
    else return n * fact(n - 1);
  }
  ```
  - Argument n in $a0
  - Result in $v0
Non-Leaf Procedure Example

- **MIPS code:**

```mips
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw $ra, 4($sp)       # save return address
    sw $a0, 0($sp)       # save argument
    slti $t0, $a0, 1     # test for n < 1
    beq $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr $ra                # and return
L1: addi $a0, $a0, -1  # else decrement n
    jal fact             # recursive call
    lw $a0, 0($sp)       # restore original n
    lw $ra, 4($sp)       # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul $v0, $a0, $v0    # multiply to get result
    jr $ra               # and return
```
Stacks in MIPS

- **Spill** registers into memory
  - STACK: LIFO
  - A stack is a data structure, at least two operations:
    - push put a value on the top of the stack
    - pop remove an item from the top of the stack.
  - Register #29 is $sp which is the stack pointer

- Historically, stacks grow from higher to lower address (push)
Memory Layout

$sp \rightarrow 7fff \text{ fff}_{16}

$gp \rightarrow 1000 \text{ 8000}_{16}
1000 \text{ 0000}_{16}

pc \rightarrow 0040 \text{ 0000}_{16}
0

Stack

↓

Dynamic data

Static data

Text

Reserved
MIPS Addressing Modes

- Five Different Types
  - What?
  - How?
  - Examples?
Conditional Branches

- Branch not very far.
- SPEC – most branches are less than 16 instructions away.

Strategy

- PC-relative addressing
  - Target address = PC + offset × 4 (+/- 2^{15})
  - PC already incremented by 4 by this time.

- But not the same for jal
Jump Addressing

Jump (j and jal) targets could be anywhere in text segment

- Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

(Pseudo) Direct jump addressing

- Target address = PC_{31...28} : (address \times 4)
- 28 bits from the address
- Assembler will fix things!
Many compilers produce object modules directly.
Arithmetic
## Overflow Conditions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result indicating overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
</tr>
</tbody>
</table>
MIPS Multiply

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

1a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Done
MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits

- Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - mul rd, rs, rt
    - Least-significant 32 bits of product -> rd
Div Hardware

Initially divisor in left half

Initially dividend

Divisor
Shift right

64 bits

64-bit ALU

Quotient
Shift left

32 bits

Remainder
Write

Control test

64 bits
### IEEE Floating-Point Format

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign bit</td>
<td>8 bits</td>
<td>11 bits</td>
</tr>
<tr>
<td>Exponent</td>
<td>23 bits</td>
<td>52 bits</td>
</tr>
<tr>
<td>Fraction</td>
<td>23 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

*\[ x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} \]*

- **S**: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: \(1.0 \leq |\text{significand}| < 2.0\)
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- **Exponent**: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203
Add/Multiply/FP

- Steps to FP Add/FP Multiply
- Why we do not need HI/LO in FP operations?
  - We don’t have any remainders for division.
FP Adder Hardware (Understand)

Step 1
- Compare exponents

Step 2
- Shift smaller number right
- Add

Step 3
- Normalize

Step 4
- Round
- Rounding hardware
- Increment or decrement
- Shift left or right
- Shift right
- Exponent difference
- Small ALU
Exercise

```c
int i
void set_array(int num) {
    int array[10];
    for (i=0; i < 10; i++) {
        array[i]=compare(num,i);
    }
}

int compare(int a, int b) {
    if (sub(a,b) >= 0 )
        return 1;
    else
        return 0;
}

int sub(int a, int b) {
    return a-b;
}
```