Review

- Floating Point numbers approximate values that we want to use.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers ($\dagger 1T$)
- New MIPS registers($f0$ - $f31$), instruct.:
  - Single Precision (32 bits, $2x10^{-38}...2x10^{38}$):
    add.s, sub.s, mul.s, div.s
  - Double Precision (64 bits, $2x10^{-308}...2x10^{308}$):
    add.d, sub.d, mul.d, div.d
- Type is not associated with data, bits have no meaning unless given in context

Overview

- Special Floating Point Numbers: NaN, Denorms
- IEEE Rounding modes
- Floating Point fallacies, hacks
- Catchup topics:
  - Representation of jump, jump and link
  - Reverse time travel:
    MIPS machine language
    MIPS assembly language
    C code
  - Logical, shift instructions (time permitting)

MIPS Floating Point Architecture (1/2)

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: $f0$, $f1$, ...
  - most registers specified in .s and .d instruction refer to this set
  - separate load and store: lwcl and swcl ("load word coprocessor 1", "store ...")
  - Double Precision: by convention, even/odd pair contain one DP FP number: $f0/f1$, $f2/f3$, ...

MIPS Floating Point Architecture (2/2)

- 1990 Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?... Yes, later
  - Today, cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - $mfc0$, $mtc0$, $mfcl$, $mtcl$, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

Special Numbers

- What have we defined so far? (Single Precision)
  - Exponent Significand Object
  - 0 0 nonzero ???
  - 1-254 anything +/- fl. pt. #
  - 255 0 +/- infinity
  - 255 nonzero ???
- Professor Kahan had clever ideas; "Waste not, want not"
Representation for Not a Number

° What do I get if I calculate sqrt(-4.0) or 0/0?
  - If infinity is not an error, it may be useful not to crash program for these too.
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
° Why is this useful?
  - Hope NaNs help with debugging
  - They contaminate: op(NaN,X) = NaN
  - OK if calculate but don’t use it
  - Ask math majors

Special Numbers (cont’d)

° What have we defined so far?
  (Single Precision)?
<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Representation for Denorms (1/2)

° Problem: There’s a gap among representable FP numbers around 0
  - Smallest representable pos num:
    - a = 1.0...1*2^-127 = 2^-127
  - Second smallest representable pos num:
    - b = 1.000...01*2^-127 = 2^-127 + 2^-150
    - a - 0 = 2^-127
    - b - a = 2^-150
° Gap! Gap!

Representation for Denorms (2/2)

° Solution:
  - We still haven’t used Exponent = 0, Significand nonzero
  - Denormalized number: no leading 1
  - Smallest representable pos num:
    - a = 2^-150
  - Second smallest representable pos num:
    - b = 2^-149

Rounding

° When we perform math on real numbers, we have to worry about rounding
° The actual math carries two extra bits of precision, and then round to get the proper value
° Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

4 IEEE Rounding Modes

° Round towards +infinity
  - ALWAYS round “up”: 2.001 -> 3
  -2.001 -> -2
° Round towards -infinity
  - ALWAYS round “down”: 1.999 -> 1,
  -1.999 -> -2
° Truncate: 2.001 -> 2, -2.001 -> -2
  - Just drop the last bits (round towards 0)
° Round to (nearest) even
  - Normal rounding, almost
Round to Even
- Round like you learned in grade school
- Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - 2.5 -> 2
  - 3.5 -> 4
- Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down
- Default C rounding mode; only Java mode

Floating Point Fallacy
- FP Add, subtract associative: FALSE!
  - \( x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, \) and \( z = 1.0 \)
  - \( x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + 1.5 \times 10^{38} = 0.0 \)
  - \( (x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = (0.0) + 1.0 = 1.0 \)
- Therefore, Floating Point add, subtract are not associative!
- Why? FP result approximates real result!
- This example: \( 1.5 \times 10^{38} \) is so much larger than 1.0 that \( 1.5 \times 10^{38} + 1.0 \) in floating point representation is still \( 1.5 \times 10^{38} \)

Casting floats to ints and vice versa

\( \text{int} \) -> \( \text{float} \) -> \( \text{int} \)
- Coerces and converts it to the nearest integer
- affected by rounding modes
  - \( yi = \text{(int)}(3.14159 \times f); \)

\( \text{float} \) -> \( \text{int} \) -> \( \text{float} \)
- Converts integer to nearest floating point
  - \( yf = f + \text{(float)}i; \)
- Will not always work
- Large values of integers don’t have exact floating point representations
- Similarly, we may round to the wrong value

Administrivia
- Need to catchup with Homework
- Reading assignment: Reading 4.8
For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.

For general jumps (\( j \) and \( jal \)), we may jump to anywhere in memory.

Ideally, we could specify a 32-bit memory address to jump to.

Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

Define “fields” of the following number of bits each:

| 6 bits | 26 bits |

As usual, each field has a name:

- opcode
- target address

Key Concepts

- Keep opcode field identical to R-format and I-format for consistency.
- Combine all other fields to make room for target address.

For now, we can specify 26 bits of the 32-bit bit address.

Optimization:

- Note that, just like with branches, jumps will only jump to word aligned addresses (since all instructions are one word long), so last two bits are always 00 (in binary).
- So let’s just take this for granted and not even specify them.
- \( \Rightarrow \) 26 bits supplies a 28-bit byte address

For now, we can specify 28 bits of the 32-bit address.

Where do we get the other 4 bits?

- By definition, take the 4 highest order bits from the PC.
- Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999...% of the time, since programs rarely that long (> 2^28 or 256 MB)
- If we absolutely need to specify a 32-bit address, we can always put it in a register and use the \( jr \) instruction.

**Summary:**

\[
\text{New PC} = \text{PC}[31..28] \ || \text{target address (26 bits)} \ || \ 00
\]

Note: II means concatenation 4 bits || 26 bits || 2 bits = 32-bit address

Understand where each part came from!
Decoding Example (1/6)

° Here are six machine language instructions in hex:

00001025
0005402A
11000003
08410202
20A5FFFF
08100001

° Let the first instruction be at address $4,194,304_{10}$ (0x00400000).

° Next step: convert to binary

Decoding Example (2/6)

° Here are the six machine language instructions in binary:

00000000000000000001000000100101
000000000000010100000000101010
0001000100000000000000000000011
0000000001001000100000000100000
00100000010100011111111111111111
00010000000100000000000000000001

° Next step: separation of fields & convert each field to decimal

- For all instructions, first 6 bits is opcode, so can easily determine format/instruction

Decoding Example (3/6)

° Decimal representation, in fields:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>I</td>
<td>R</td>
<td>J</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0</td>
<td>+3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,048,577</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

° Next step: translate to MIPS instructions

Decoding Example (4/6)

° MIPS Assembly (Part 1):

```
0x00400000 or $2,$0,$0
0x00400004 slt $8,$0,$5
0x00400008 beq $8,$0,3
0x0040000c add $2,$2,$4
0x00400010 addi $5,$5,-1
0x00400014 j 0x100001
```

° Next step: translate to more meaningful instructions (fix the branch/jump and add labels)

- Remember: jump address add 00 to end

Decoding Example (5/6)

° MIPS Assembly (Part 2):

```
or $v0,$0,$0
Loop: slt $t0,$0,$a1
beq $t0,$0,Fin
add $v0,$v0,$a0
addi $a1,$a1,-1
j Loop
Fin:
```

° Next step: translate to C code (be creative!)

Decoding Example (6/6)

° C code:

```
or $v0,$0,$0
Loop: slt $t0,$0,$a1
beq $t0,$0,Fin
add $v0,$v0,$a0
addi $a1,$a1,-1
j Loop
Fin:
```

- Mapping: $v0: product
  $a0: mcand
  $a1: mplier

product = 0;
while (mplier > 0) {
  product += mcand;
  mplier -= 1;
}
Bitwise Operations (1/2)
- Up until now, we've done arithmetic (add, sub, addi) and memory access (lw and sw).
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer).
- New Perspective: View contents of register as 32 bits rather than as a single 32-bit number.

Bitwise Operations (2/2)
- Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions

Logical Operators (1/4)
- How many of you have taken Math 55?
- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- In general, can define them to accept >2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output.
  - Again, rigid syntax, simpler hardware.

Logical Operators (2/4)
- Truth Table: standard table listing all possible combinations of inputs and resultant output for each.
- Truth Table for AND and OR:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logical Operators (3/4)
- Logical Instruction Syntax:
  - 1 2,3,4
  - where
    - 1) operation name
    - 2) register that will receive value
    - 3) first operand (register)
    - 4) second operand (register) or immediate (numerical constant)

Logical Operators (4/4)
- Instruction Names:
  - \( \& \text{and}, \| \text{or} \): Both of these expect the third argument to be a register.
  - \( \& \text{andi}, \| \text{ori} \): Both of these expect the third argument to be an immediate.
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
Shift Instructions (1/4)
° Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
  • Example: shift right by 8 bits
    0001 0010 0011 0100 0101 0110 0111 1000
    0000 0000 0001 0010 0011 0100 0101 0110
  • Example: shift left by 8 bits
    0011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (2/4)
° Shift Instruction Syntax:
  1. operation name
  2. register that will receive value
  3. first operand (register)
  4. second operand (register)

Shift Instructions (3/4)
° MIPS has three shift instructions:
  1. sll (shift left logical): shifts left and fills emptied bits with 0s
  2. srl (shift right logical): shifts right and fills emptied bits with 0s
  3. sra (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (4/4)
° Example: shift right arith by 8 bits
  0001 0010 0011 0100 0101 0110 0111 1000
  1111 1111 1001 0010 0011 0100 0101 0110

Uses for Logical Operators (1/3)
° Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
° This can be used to create a mask.
  • Example:
    1011 0110 1010 0100 0011 1101 1001 1010
    Mask: 0000 0000 0000 0000 0000 1111 1111 1111
  • The result of anding these two is:
    0000 0000 0000 0000 0000 1101 1001 1010

Uses for Logical Operators (2/3)
° The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).
° Thus, the and operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  • In particular, if the first bitstring in the above example were in $t0, then the following instruction would mask it:
    andi $t0,$t0,0xFFFF
Uses for Logical Operators (3/3)

- Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, if $t0$ contains 0x12345678, then after this instruction:
    ```
    or   $t0,$t0, 0xFFFF
    ```
  - ... $t0$ contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Uses for Shift Instructions (1/5)

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0$. Simply use:
  ```
  andi $t0,$t0,0xFF
  ```
- Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0$. We can use:
  ```
  andi $t0,$t0,0xFF00
  ```
  but then we still need to shift to the right by 8 bits...

Uses for Shift Instructions (2/5)

- Instead, use:
  ```
  sll  $t0,$t0,16  
srl  $t0,$t0,24
  ```

Uses for Shift Instructions (3/5)

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - 714_{10} x 10_{10} = 7140_{10}
    - 56_{10} x 10_{10} = 560_{10}
  - Multiplying by 100 is same as shifting left by 2:
    - 714_{10} x 100_{10} = 71400_{10}
    - 56_{10} x 100_{10} = 5600_{10}
  - Multiplying by 10^n is same as shifting left by n

Uses for Shift Instructions (4/5)

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - 1112 x 10_2 = 110_2
    - 1010_2 x 10_2 = 10100_2
  - Multiplying by 4 is same as shifting left by 2:
    - 1112 x 100_2 = 1100_2
    - 1010_2 x 100_2 = 101000_2
  - Multiplying by 2^n is same as shifting left by n

Uses for Shift Instructions (5/5)

- Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  ```
  a *= 8;  // (in C)
  ```
  would compile to:
  ```
  sll  $s0,$s0,3  // (in MIPS)
  ```
Things to Remember (1/3)

- **IEEE 754 Floating Point Standard**: Kahan pack as much in as could get away with
  - \(\pm\)infinity, Not-a-Number (Nan), Denorms
  - 4 rounding modes
- **Stored Program Concept**: Both data and actual code (instructions) are stored in the same memory.
- **Type is not associated with data**, bits have no meaning unless given in context

Things to Remember (2/3)

- **Machine Language Instruction**: 32 bits representing a single MIPS instruction

<table>
<thead>
<tr>
<th>R</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rd</td>
</tr>
<tr>
<td>opcode</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>opcode</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>

- Instructions formats are kept as similar as possible.
- Branches and Jumps were optimized for greater branch distance and hence strange, so clear these up in your mind now.

Things to Remember (3/3)

- **New Instructions**:
  - and, andi, or, ori
  - sll, srl, sra