There is one handout today at the front and back of the room!

#9 bears win again!

⇒ Marshawn Lynch ran for 107 yds and a TD & Ayoob went 14-20. We won 28-0 and have outscored 'zona 66-0 the last 2 yrs!

We visit #16 UCLA next week.

I-Format Problems (0/3)

• Problem 0: Unsigned # sign-extended?
  • addiu, sltiu, sign-extends immediates to 32 bits. Thus, # is a “signed” integer.

• Rationale
  • addiu so that can add w/out overflow
    - See K&R pp. 230, 305
  • sltiu suffers so that we can have ez HW
    - Does this mean we’ll get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate \(2^{16} \leq n < 2^{17}\) (i.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large.

I-Format Problems (1/3)

• Problem 1:
  • Chances are that addi, lw, sw and slti will use immediates small enough to fit in the immediate field.
  • ...but what if it’s too big?
  • We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problems (2/3)

• Solution to Problem 1:
  • Handle it in software + new instruction
  • Don’t change the current instructions: instead, add a new instruction to help out

• New instruction:
  • lui register, immediate
  • stands for Load Upper Immediate
  • takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
  • sets lower half to 0s

I-Format Problems (3/3)

• Solution to Problem 1 (continued):
  • So how does lui help us?
  • Example:
    - \texttt{addi} $t0,$t0, 0xABABCDCD
    - \texttt{lui} $at, 0xABAB
    - \texttt{ori} $at, $at, 0xCDCD
    - \texttt{add} $t0,$t0,$at
  • Now each I-format instruction has only a 16-bit immediate.
  • Wouldn’t it be nice if the assembler would do this for us automatically? (later)

Review...

• Logical and Shift Instructions
  • Operate on individual bits (arithmetic operate on entire word)
  • Use to isolate fields, either by masking or by shifting back & forth
  • Use shift left logical, \texttt{sll}, for multiplication by powers of 2
  • Use shift right arithmetic, \texttt{sra}, for division by powers of 2

• Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use \texttt{lw} and \texttt{sw}).

• Computer actually stores programs as a series of these 32-bit numbers.

• MIPS Machine Language Instruction:
  • 32 bits representing a single instruction
    \begin{tabular}{lcccc}
    \texttt{opcode} & \texttt{rs} & \texttt{rt} & \texttt{rd} & \texttt{shamt} & \texttt{funct} \\
    \hline
    \texttt{R} & \texttt{opcode} & \texttt{rs} & \texttt{rt} & \texttt{shamt} & \texttt{funct} \\
    \texttt{I} & \texttt{opcode} & \texttt{rs} & \texttt{rt} & \texttt{immediate} & \texttt{target address} \\
    \end{tabular}
Branches: PC-Relative Addressing (1/5)

- Use I-Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

- opcode specifies beq v, bne
- rs and rt specify registers to compare
- What can immediate specify?
  - Immediate is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory
  - So immediate cannot specify entire address to branch to.

Branches: PC-Relative Addressing (2/5)

- How do we usually use branches?
  - Answer: if-else, while, for
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (j and jal), not the branches.
  - Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount

Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- Let the 16-bit immediate field be a signed two's complement integer to be added to the PC if we take the branch.
- Now we can branch ±2^15 bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the immediate in words.
- Now, we can branch ±2^15 words from the PC (or ±2^17 bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don't take the branch:
    \[ PC = PC + 4 \]
    \[ PC+4 = \text{byte address of next instruction} \]
  - If we do take the branch:
    \[ PC = (PC + 4) + \text{(immediate} \times 4) \]
- Observations
  - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
  - Immediate field can be positive or negative.
  - Due to hardware, add immediate to (PC+4), not to PC; will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

```assembly
Loop:   beq $9,$0, End
        add $8,$8,$10
        addi $9,$9,-1
        j Loop
End:
```

- beq branch is I-Format:
  - opcode = 4 (look up in table)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - immediate = ????
Branch Example (2/3)

• MIPS Code:
  Loop:
  beq $9, $0, End
  addi $8, $8, $10
  addi $9, $9, -1
  j Loop
  End:

• Immediate Field:
  • Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.
  • In beq case, immediate = 3

Questions on PC-addressing

• Does the value in branch field change if we move the code?
• What do we do if destination is > $2^{15}$ instructions away from branch?
• Since it’s limited to ± $2^{15}$ instructions, doesn’t this generate lots of extra MIPS instructions?
• Why do we need all these addressing modes? Why not just one?

Upcoming Calendar

<table>
<thead>
<tr>
<th>Week #</th>
<th>Mon</th>
<th>Wed</th>
<th>Thu Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>#6</td>
<td>This week</td>
<td>Floating Pt II (No Dan OH)</td>
<td>Floating Pt</td>
</tr>
<tr>
<td>#7</td>
<td>Next week</td>
<td>Running Program II</td>
<td>Running Program (Proj 2 really due)</td>
</tr>
<tr>
<td>#8</td>
<td>Midterm week</td>
<td>Midterm 5:30-8:30pm Here! (155 Dwin)</td>
<td>SDS I</td>
</tr>
<tr>
<td>Sun 2pm</td>
<td>Review 10 Evans</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

J-Format Instructions (1/5)

• 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.

J-Format Instructions (2/5)

• 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 large target address.
J-Format Instructions (3/5)

• 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, so last two bits are always 00 (in binary).
  • So let’s just take this for granted and not even specify them.

J-Format Instructions (4/5)

• Now specify 28 bits of a 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 99.999% of the time, since programs aren’t that long
    - only if straddle a 256 MB boundary
  • If we absolutely need to specify a 32-bit address, we can always put it in a register and use the jz instruction.

J-Format Instructions (5/5)

• Summary:
  • New PC = (PC[31..28], target address, 00 )
• Understand where each part came from!
• Note: { , , } means concatenation
  • 4 bits , 26 bits , 2 bits } = 32 bit address
  • { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100
• Note: Book uses ||

In semi-conclusion...

• MIPS Machine Language Instruction: 32 bits representing a single 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>rt</td>
</tr>
<tr>
<td>opcode</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>
• Branches use PC-relative addressing.
  • Jumps use absolute addressing.
• Disassembly is simple and starts by decoding opcode field. (more in a week)

Quote of the day

“95% of the folks out there are completely clueless about floating-point.”

James Gosling
Sun Fellow
Java Inventor
1998-02-28

Review of Numbers

• Computers are made to deal with numbers
• What can we represent in N bits?
  • Unsigned integers: 0 to 2^N - 1
  • Signed Integers (Two’s Complement)
    -2^(N-1) to 2^(N-1) - 1
Other Numbers

• What about other numbers?
  • Very large numbers? (seconds/century)
    3,155,760,000,000 (3.15576 x 10^9)
  • Very small numbers? (atomic diameter)
    0.00000001,0 (1.0 x 10^-9)
  • Rationals (repeating pattern)
    2/3 (0.666666666. . .)
  • Irrationals
    2^(1/2) (1.414213562373. . .)
  • Transcendentals
    e (2.718...), π (3.141...)

All represented in scientific notation

Scientific Notation (in Decimal)

Scientific Notation (in Binary)

Floating Point Representation (1/2)

Floating Point Representation (2/2)

Double Precision Fl. Pt. Representation

• Next Multiple of Word Size (64 bits)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>20</th>
<th>19</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exponent</td>
<td>Significand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 bit</td>
<td>11 bits</td>
<td>20 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Significand (cont'd)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Double Precision (vs. Single Precision)
  • C variable declared as double
  • Represent numbers almost as small as 2.0 x 10^-308 to as large as 2.0 x 10^308
  • But primary advantage is greater accuracy due to larger significand
QUAD Precision Fl. Pt. Representation

- Next Multiple of Word Size (128 bits)
- Unbelievable range of numbers
- Unbelievable precision (accuracy)
- This is currently being worked on
- The current version has 15 bits for the exponent and 112 bits for the significand
- Oct-Precision? That’s just silly! It’s been implemented before...

IEEE 754 Floating Point Standard (1/4)

- Single Precision, DP similar
- Sign bit: 1 means negative 0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: Significand < 1 (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/4)

- Kahan wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
- Could break FP number into 3 parts: compare signs, then compare exponents, then compare significands
- Wanted it to be faster, single compare if possible, especially if positive numbers
- Then want order:
  - Highest order bit is sign (negative < positive)
  - Exponent next, so big exponent => bigger #
  - Significand last: exponents same => bigger #

IEEE 754 Floating Point Standard (3/4)

- Negative Exponent?
  - 2’s comp? 1.0 x 2^{-1} v. 1.0 x 2^{+1} (1/2 v. 2)
  - 1/2
    - | 0111 1111 | 000 0000 0000 0000 0000 0000
  - 2
    - | 0000 0001 | 000 0000 0000 0000 0000 0000
  - This notation using integer compare of 1/2 v. 2 makes 1/2 > 2!
  - Instead, pick notation 000 0001 is most negative, and 1111 1111 is most positive
  - 1.0 x 2^{-1} v. 1.0 x 2^{+1} (1/2 v. 2)
  - 1/2
    - | 0111 1111 | 000 0000 0000 0000 0000 0000
  - 2
    - | 0000 0001 | 000 0000 0000 0000 0000 0000

IEEE 754 Floating Point Standard (4/4)

- Called Biased Notation, where bias is number subtract to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision
- Summary (single precision):
  - 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    | Exponent | Significand |
    1 bit 8 bits 23 bits
    • (-1)^S x (1 + Significand) x 2^(Exponent-127)
    • Double precision identical, except with exponent bias of 1023

“And in conclusion…”

- 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
  - Every desktop or server computer sold since ~1997 follows these conventions
- Summary (single precision):
  - 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    | Exponent | Significand |
    1 bit 8 bits 23 bits
    • (-1)^S x (1 + Significand) x 2^(Exponent-127)
    • Double precision identical, bias of 1023