Agenda

- Review
- C Functions and Calling conventions
- Integers and Two’s Complement
- Administrivia
- Technology Break
- Real Numbers and Floating Point
- Summary

Review from Last Lecture

- C is function oriented; code reuse via functions
  - Jump and link (jal) invokes, jump register (jr $ra) returns
  - Registers $a0-$a3 for arguments, $v0-$v1 for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables
- Pointers/pointer arithmetic to reduce array overhead
  - No pointers to automatic data!

Allocating space on stack

- C has two storage classes: automatic and static
  - Automatic variables are local to function and discarded when function exits.
  - Static variables exist across exits from and entries to procedures
- Can use stack for automatic (local) variables that don’t fit in registers
  - procedure frame or activation record: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer ($fp) to point to first word of frame

Stack before, during, after call
Optimized Function Convention

• To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:
  1. Preserved across function call
     - Caller can rely on values being unchanged
     - $sra$, $sp$, $gp$, $fp$, “saved registers” $s0$ - $s7$
  2. Not preserved across function call
     - Caller cannot rely on values being unchanged
     - Return value registers $v0$, $v1$, Argument registers $a0$ - $a3$, “temporary registers” $t0$ - $t9$

Where is stack in memory?

• MIPS convention
  - Stack starts in high memory and grows down
    - Hexadecimal (base 16): $7ffe$ to $7ff0$
  - MIPS programs (text segment) in low end
    - 0000 to 0040
  - Static data segment (constants and other static variables) above text for static variables
  - MIPS convention global pointer ($gp$) points to static
  - Heap above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation

Number Representation

• Value of $i$-th digit is $d \times \text{Base}^i$ where $i$ starts at 0 and increases from right to left:
  - $123_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$
    = $100 + 20 + 3$
    = 123
  - Binary (base 2), Octal (base 8), Hexadecimal (base 16), Decimal (base 10) different ways to represent an integer
    - We use $1_{two}$, $8_{oct}$, $16_{hex}$, $10_{ten}$ to be clearer
      (vs. $1_{two}$, $4_{oct}$, $16_{hex}$, $10_{ten}$)
• Hexadecimal digits:
  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - $FFF_{hex} = 15_{ten} \times 16_{ten}^2 + 15_{ten} \times 16_{ten}^1 + 15_{ten} \times 16_{ten}^0$
    = $3840_{ten} + 240_{ten} + 15_{ten}$
    = 4095_{ten}
  - $1111\ 1111\ 1111_{two} = 7777_{oct} = FFF_{hex} = 4095_{ten}$
• May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal
Signed and Unsigned Integers

- C, C++, and Java have **signed integers**, e.g., 7, -255:
  ```
  int x, y, z;
  ```
- C++, also has **unsigned integers**, which are used for addresses
- 32-bit word can represent 2^32 binary numbers
- Signed integers in 32 bit word represent 0 to 2^32-1 (4,294,967,295)

Unsigned Integers

```plaintext
0000 0000 0000 0000 0000 0000 0000 0000 = 0
0000 0000 0000 0000 0000 0000 0000 0001 = 1
0000 0000 0000 0000 0000 0000 0000 0010 = 2
0111 1111 1111 1111 1111 1111 1111 1110 = 2,147,483,646
0111 1111 1111 1111 1111 1111 1111 1111 = 2,147,483,647
1000 0000 0000 0000 0000 0000 0000 0000 = -2,147,483,648
1000 0000 0000 0000 0000 0000 0000 0001 = -2,147,483,649
1000 0000 0000 0000 0000 0000 0000 0010 = -2,147,483,650
```

Signed Integers and Two’s Complement Representation

- Signed integers in C, want ½ numbers <0, want ½ numbers >0, and want one 0
- **Two’s complement** treats 0 as positive, so 32-bit word represents 2^32 integers from -2^31 to 2^31-1 (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which worse
  - Every computer uses two’s complement today
- **Most significant bit** (leftmost) called **sign bit**, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two’s Complement Integers

<table>
<thead>
<tr>
<th>Sign Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000 = 0</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0001 = 1</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0010 = 2</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1110 = 2,147,483,646</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111 = 2,147,483,647</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000 = -2,147,483,648</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0001 = -2,147,483,649</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0010 = -2,147,483,650</td>
</tr>
</tbody>
</table>

The Rules (delay dopamine squirt until break)

- To reduce time pressure, 3 hours for 1.5 hour midterm
- Midterm Exam Wednesday October 6, 6 – 9PM, Pimental 1
- Final Exam Monday December 13, 8 – 11AM, Location TBD

When is Midterm, Final?

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
Peer Instruction

• Increase real-time learning in lecture, test understanding of concepts vs. details
  natur-www.harvard.edu/education/pi.phtml
• As complete a “segment”
  ask multiple choice question
  – 1-2 minutes: decide yourself, vote
  – 2-3 minutes: discuss in pairs, then team vote; flash cards
  • Try to convince partner; learn by teaching

Question?

```c
static int *p;
int leaf (int g, int h, int i, int j) {
  int f; f = g + h - (i + j);
  return f;
}
int main(void) { int x;
    ...
    x = leaf(1,2,3,4);
    ...
    x = leaf(3,4,1,2);
    printf("%d\n",p);
}
```  

• What will a.out do?
  A. Print -4
  B. Print 4
  C. a.out will crash
  D. None of the above

Agenda

• Review
• C Functions and Calling conventions
• Integers and Two’s Complement
• Administrivia
• Technology Break
• Real Numbers and Floating Point
• Summary

MIPS Logical Instructions

• Useful to operate on fields of bits within a word
  e.g., characters within a word (8 bits)
• Operations to pack/unpack bits into words
• Called logical operations

<table>
<thead>
<tr>
<th>Logical</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>operators</td>
<td>operators</td>
<td>instructions</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
</tbody>
</table>

Bit-by-bit Definition

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Examples

- If register $t2$ contains and
  
  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
  \end{array}
  \]
  
  \(0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000\ 0000\_{\text{two}}\)
  
  - Register $t1$ contains
  
  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
  \end{array}
  \]
  
  \(0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000\_{\text{two}}\)
  
- What is value of $t0$ after:
  
  1. and $t0, t1, t2$ # reg $t0 = \text{reg } t1 \& \text{reg } t2$
  2. $t0, t1, t2$ # reg $t0 = \text{reg } t1 \mid \text{reg } t2$
  3. nor $t0, t1, t2$ # reg $t0 = \neg (\text{reg } t1 \mid \text{0})$

### Impact of Signed and Unsigned Integers on Instruction Sets

- What (if any) instructions affected?
  
  - Load word, store word?
  - branch equal, branch not equal?
  - and, or, sll, srl?
  - add, sub, mult, div?
  - set less than?

### What if result of operation doesn’t fit in 32 bits?

- Called overflow: calculate too big a number to represent within a word
  
  - Unsigned numbers: \(1 + 4,294,967,295\ (2^{32} - 1)\)
  - Signed numbers: \(1 + 2,147,483,647\ (2^{31} - 1)\)

### Answer Depends on Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo \(2^{32}\))
  
  \(1 + 4,294,967,295 = \)
  
- C signed number arithmetic also ignores overflow
  
  \(1 + 2,147,483,647\ (2^{31} - 1) = \)
  
- Other languages want overflow signal on signed numbers (e.g., Fortran)
  
- What’s a computer architect to do?

### MIPS Solution: offer both

- Instructions that overflow:
  
  - add, sub, mult, div, addi, multi, divi
  
- Instructions that don’t overflow called “unsigned” (but really means no overflow):
  
  - addu, subu, multu, divu, addiu, multiu, diviu
  
- Given semantics of C, always use unsigned versions
  
- Note: slt and slli do signed comparisons, while sllt and slti do unsigned comparisons
  
  - Nothing to do with overflow
  
  - When would get different answer for slt vs. slli?
What about Real Numbers?

- Normalized scientific notation (aka standard form or exponential notation):
  - \( r \times 10^E \), where \( E \) is the exponent (usually 10), \( r \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 10 \)
  - \( 61 \times 10^{-3}, 0.000061 \) is \( 6.10 \times 10^{-4} \)
- Computers version of normalized scientific notation called Floating Point notation
- \( r \times 10^E \) where \( E \) is the exponent (2), \( r \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0, < 2 \)

Floating Point Numbers

- 32-bit word has 2^{23} patterns, so must be approximation of real numbers \( \geq 1.0, < 2 \)
- IEEE 754 Floating Point Standard:
  - 1 bit for sign (s) of floating point number
  - 8 bits for exponent (E)
  - 23 bits for fraction (F)
    - (get 1 extra bit of precision if leading 1 is implicit)
  - \((-1)^s (1 + F) \times 2^E\)
- Can represent from \( 2.0 \times 10^{-38} \) to \( 2.0 \times 10^{38} \)

More Floating Point

- What about 0?
  - Bit pattern all 0s means 0, no so implicit leading 1
- What if divide 1 by 0?
  - Can get infinity symbols \( +\infty, -\infty \)
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? (\( 1 \times \infty, 0 + 0 \))
  - Can get special symbols NaN for Not-a-Number
  - Sign bit 0 or 1, largest exponent, not zero in fraction
- What if result is too big? (\( 2 \times 10^{308} \times 2 \times 10^3 \))
  - Get overflow in exponent, alert programmer!
- What if result is too small? (\( 2 \times 10^{-308} \times 2 \times 10^3 \))
  - Get underflow in exponent, alert programmer!

MIPS Floating Point Instructions

- C, Java has single precision (float) and double precision (double) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision:
    - Fl. Pt. Addition double precision:
  - Fl. Pt. Subtraction single precision:
  - Fl. Pt. Subtraction double precision:
  - Fl. Pt. Multiplication single precision:
  - Fl. Pt. Multiplication double precision:
  - Fl. Pt. Divide single precision:
  - Fl. Pt. Divide double precision:
Everything in a computer is just a Binary Number

- Up to program to decide what data means
- Example 32-bit data shown as binary number:
  
<table>
<thead>
<tr>
<th>ASCII characters</th>
<th>Unicode characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000, \text{two}</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000, \text{two}</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111, \text{two}</td>
<td>0111 1111 1111 1111 1111 1111 1111 1111, \text{two}</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111, \text{two}</td>
<td>1111 1111 1111 1111 1111 1111 1111 1111, \text{two}</td>
</tr>
</tbody>
</table>

What does it mean if its treated as

1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters

Peer Instruction

- Why does C provide two sets of operators for AND (\& and \&\&) and two sets of operators for OR (| and ||) while MIPS doesn’t?
  
  A. Logical operations AND and OR implement \& and | while conditional branches implement \&\& and ||.
  B. The previous statement has it backwards: \&\& and || correspond to logical operations while | and || map to conditional branches.
  C. They are redundant and mean the same thing: \&\& and || are simply inherited from the programming language B, the predecessor of C.

Summary

- Registers selectively saved/restored on call
  - Saved registers $s0-$s7; temporary regs $t0-$t9 not saved
- C splits memory into text, static, heap, stack, with registers dedicated to support: $sp, $sp, $fp
- Program can interpret binary number as unsigned integer, two’s complement signed integer, floating point number, ASCII characters, Unicode characters, ...
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two’s comp. weirdness is one extra negative num
- Floating point is an approximation of reals
- Integer and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Everything is a (binary) number in a computer