CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

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!

• 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: $gp, $sp, $fp
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
     - $ra, $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

Register Numbering

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Values for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-26</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Where is stack in memory?

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

- Value of i-th digit is \( d \times \text{Base}_i \) where i starts at 0 and increases from right to left:
  \[
  123_{10} = 1_{10} \times 10_{10}^2 + 2_{10} \times 10_{10}^1 + 3_{10} \times 10_{10}^0 \\
  = 1 \times 100_{10} + 2 \times 10_{10} + 3 \times 1_{10} \\
  = 100_{10} + 20_{10} + 3_{10} \\
  = 123_{10}
  \]

- Binary (Base 2), Octal (Base 8), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1\text{two}, 8\text{oct}, 5\text{ten}, 10\text{hex} to be clearer (vs. 1_2, 4_8, 5_{10}, 10_{16})

- Hexadecimal digits:
  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

  - \( \text{FFF}_{\text{hex}} = 15_{\text{ten}} \times 16_{\text{ten}}^2 + 15_{\text{ten}} \times 16_{\text{ten}}^1 + 15_{\text{ten}} \times 16_{\text{ten}}^0 = 3840_{\text{ten}} + 240_{\text{ten}} + 15_{\text{ten}} = 4095_{\text{ten}} \)

- 1111 1111 1111\text{two} = 7777\text{oct} = \text{FFF}_{\text{hex}} = 4095_{\text{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 has signed integers, e.g., 7, -255:
  ```
  int x, y, z;
  ```
- C, C++ also has unsigned integers, which are used for addresses
- 32-bit word can represent $2^{32}$ binary numbers
- Unsigned integers in 32 bit word represent 0 to $2^{32}-1$ (4,294,967,295)

Unsigned Integers

<table>
<thead>
<tr>
<th>Two</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0001</td>
<td>1</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0010</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>2,147,483,645</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1110</td>
<td>2,147,483,646</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000</td>
<td>2,147,483,648</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0001</td>
<td>2,147,483,649</td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0010</td>
<td>2,147,483,650</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>4,294,967,293</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1110</td>
<td>4,294,967,294</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>
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 $2^{31} - 1$ (−2,147,483,648) to 2,147,483,647
  – Note: one negative number with no positive version
  – Book lists some other options, all of which worse
  – Every computers 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>
<th>Two’s Complement Integer</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000000000000000000000000000_10</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>000000000000000000000000000010_10</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>0000000000000000000000000000010_10</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>111111111111111111111111110_10</td>
<td>2,147,483,645</td>
</tr>
<tr>
<td>0111</td>
<td>111111111111111111111111110_10</td>
<td>2,147,483,646</td>
</tr>
<tr>
<td>0111</td>
<td>111111111111111111111111110_10</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>1000</td>
<td>00000000000000000000000000000_10</td>
<td>−2,147,483,648</td>
</tr>
<tr>
<td>1000</td>
<td>00000000000000000000000000001_10</td>
<td>−2,147,483,647</td>
</tr>
<tr>
<td>1000</td>
<td>0000000000000000000000000000010_10</td>
<td>−2,147,483,646</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>111111111111111111111111110_10</td>
<td>−3</td>
</tr>
<tr>
<td>1111</td>
<td>111111111111111111111111110_10</td>
<td>−2</td>
</tr>
<tr>
<td>1111</td>
<td>111111111111111111111111110_10</td>
<td>−1</td>
</tr>
</tbody>
</table>
The Rules
(delay dopamine squirt until break)

When is Midterm, Final?

- 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
Peer Instruction

- Increase real-time learning in lecture, test understanding of concepts vs. details
  mazur-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

Peer Instruction

- Suppose we had a 5 bit word. What integers can be represented in two’s complement?
  A. -32 to +31
  B. -31 to +32
  C. 0 to +31
  D. -16 to +15
  E. -15 to +15
  F. -15 to +16
**Question?**

```
static int *p;
int leaf (int g, int h, int i, int j)
{
    int f; p = &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 operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<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>1</td>
</tr>
</tbody>
</table>
Examples

• If register $t2$ contains and $0000\ 0000\ 0000\ 0000\ 0000\ 11011100\ 0000_{\text{two}}$
• Register $t1$ contains $0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{\text{two}}$
• What is value of $t0$ after:
  1. and $t0,$t1,$t2$ # reg $t0$ = reg $t1$ & reg $t2$
  2. or $t0,$t1,$t2$ # reg $t0$ = reg $t1$ | reg $t2$
  3. nor $t0,$t1,$\text{zero}$ # reg $t0$ = $\sim$ (reg $t1$ | 0)

Shifting

• Shift left logical moves $n$ bits to the left (insert 0s into empty bits)
  – Same as multiplying by $2^n$ for two’s complement num.
• For example, if register $s0$ contained $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{\text{two}} = 9_{\text{ten}}$
• If executed sll $s0,$ $s0,$ 4, result is: $0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{\text{two}} = 144_{\text{ten}}$
• And $9_{\text{ten}} \times 2_{\text{ten}}^4 = 9_{\text{ten}} \times 16_{\text{ten}} = 144_{\text{ten}}$
• Shift right logical moves $n$ bits to the right (insert 0s into empty bits)
  – NOT same as dividing by $2^n$ (negative numbers fail)
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 slti do signed comparisons, while sltu and sltiu do unsigned comparisons
  – Nothing to do with overflow
  – When would get different answer for slt vs. sltu?
What about Real Numbers?

- Normalized scientific notation (aka standard form or exponential notation):
  - \( r \times E^i \), \( E \) is where exponent (usually 10), \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0 \), \( < 10 \)
  - 61 is \( 6.10 \times 10^2 \), 0.000061 is \( 6.10 \times 10^{-5} \)
- Computers version of normalized scientific notation called *Floating Point* notation
- \( r \times E^i \), \( E \) where is exponent (2), \( i \) is a positive or negative integer, \( r \) is a real number \( \geq 1.0 \), \( < 2 \)

Floating Point Numbers

- 32-bit word has \( 2^{32} \) 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)
  \((-1)^s \times (1 + F) \times 2^E\)
- Can represent from \( 2.0 \times 10^{-38} \) to \( 2.0 \times 10^{38} \)
Floating Point Numbers

• What about bigger or smaller numbers?
• IEEE 754 Floating Point Standard:
  
  **Double Precision** (64 bits)
  
  – 1 bit for *sign (s)* of floating point number
  – 11 bits for *exponent (E)*
  – 52 bits for *fraction (F)*
    (get 1 extra bit of precision if leading 1 is implicit)

\[-1^s \times (1 + F) \times 2^E\]

• Can represent from $2.0 \times 10^{-308}$ to $2.0 \times 10^{308}$
• 32 bit format called **Single Precision**

More Floating Point

• What about 0?
  – Bit pattern all 0s means 0, so no 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? ($\infty - \infty$, $0 \div 0$)
  – 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? ($2 \times 10^{308} \times 2 \times 10^2$)
  – Get *overflow* in exponent, alert programmer!
• What if result is too small? ($2 \times 10^{-308} \div 2 \times 10^2$)
  – 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:

• Since rarely mix integers and Fl. Pt., MIPS has separate registers for floating-point operations: $f0, f1, ..., f31
  – Double precision uses adjacent even-odd pairs of registers:
    – $f0 and $f1, $f2 and $f3, $f4 and $f5, ..., $f30 and $f31

• Need data transfer instructions for these new registers
  – lw1 (load word), swc1 (store word)
  – Double precision uses two lw1 instructions, two swc1 instructions
Unsigned Integers

0000 0000 0000 0000 0000 0000 0000 0000\two = 0_{\text{ten}}
0000 0000 0000 0000 0000 0000 0000 0001\two = 1_{\text{ten}}
0000 0000 0000 0000 0000 0000 0000 0010\two = 2_{\text{ten}}

... ... ...
0111 1111 1111 1111 1111 1111 1110\two = 2,147,483,645_{\text{ten}}
0111 1111 1111 1111 1111 1111 1110\two = 2,147,483,645_{\text{ten}}
0111 1111 1111 1111 1111 1111 1111\two = 2,147,483,647_{\text{ten}}
1000 0000 0000 0000 0000 0000 0000 0000\two = 2,147,483,650_{\text{ten}}
1000 0000 0000 0000 0000 0000 0000 0001\two = 2,147,483,651_{\text{ten}}
1000 0000 0000 0000 0000 0000 0000 0010\two = 2,147,483,652_{\text{ten}}

... ... ...
1111 1111 1111 1111 1111 1111 1111\two = 4,294,967,293_{\text{ten}}
1111 1111 1111 1111 1111 1111 1111\two = 4,294,967,293_{\text{ten}}
1111 1111 1111 1111 1111 1111 1111\two = 4,294,967,295_{\text{ten}}

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: 0000 0000 0000 0000 0000 0000 0000 0000\two

What does it mean if its treated as
1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters
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:
  \[ \text{1111 1111 1111 1111 1111 1111 1111} \text{two} \]
  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: $gp$, $sp$, $fp$
- 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. wierdness 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