New-School Machine Structures
(It’s a bit more complicated!)

- **Parallel Requests**
  Assigned to computer
e.g., Search “Katz”

- **Parallel Threads**
  Assigned to core
e.g., Lookup, Ads

- **Parallel Instructions**
  >1 instruction @ one time
e.g., 5 pipelined instructions

- **Parallel Data**
  >1 data item @ one time
e.g., Add of 4 pairs of words

- **Hardware descriptions**
  All gates @ one time

- **Programming Languages**
Big Idea #1: Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

Anything can be represented as a number, i.e., data or instructions

Review

- 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
Agenda

• Memory Heap
• Everything is a Number
• Administrivia
• Overflow and Real Numbers
• Technology Break
• Instructions as Numbers
• Assembly Language to Machine Language
• And in Conclusion, ...
What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in $a0 to $a1 and $ra
- What is the solution?

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
- 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
- (29 of 32, 3 left!)
Stack Before, During, After Call

Recursive Function Factorial

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```
Recursive Function Factorial

Fact:
- # adjust stack for 2 items
  addi $sp,$sp,-8
- # save return address
  sw $ra, 4($sp)
- # save argument n
  sw $a0, 0($sp)
- # test for n < 1
  slti $t0,$a0,1
- # if n >= 1, go to L1
  beq $t0,$zero,L1
- # Then part (n>=1) return 1
  addi $v0,$zero,1
- # pop 2 items off stack
  addi $sp,$sp,8
- # return to caller
  jr $ra

L1:
- # Else part (n >= 1)
  addi $sp,$sp,8
- # arg. gets (n - 1)
  sw $a0,$a0,-1
- # call fact with (n - 1)
  jal fact
- # return from jal: restore n
  lw $a0, 0($sp)
- # restore return address
  lw $ra, 4($sp)
- # adjust sp to pop 2 items
  addi $sp, $sp,8
- # return n * fact (n - 1)
  mul $v0,$a0,$v0
- # return to the caller
  jr $ra

mul is a pseudo instruction

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
Where is the Stack in Memory?

- **MIPS convention**
- Stack starts in high memory and grows down
  - Hexadecimal (base 16): 7fff ffff\textsubscript{hex}
- MIPS programs (**text segment**) in low end
  - 0040 0000\textsubscript{hex}
- **Static data segment** (constants and other static variables) above text for static variables
  - MIPS convention **global pointer** ($gp$) points to static
  - (30 of 32, 2 left! – will see when talk about OS)
- **Heap** above static for data structures that grow and shrink; grows up to high addresses

MIPS Memory Allocation
Register Allocation and 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>24-25</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>

Which statement is FALSE?

- MIPS uses jal to invoke a function and jr to return from a function
- jal saves PC+1 in $ra

- The callee can use temporary registers ($t/i) without saving and restoring them
- The caller can rely on save registers ($s/f) without fear of callee changing them
Which statement is FALSE?

- MIPS uses jal to invoke a function and jr to return from a function
- jal saves PC+1 in $ra
- The callee can use temporary registers ($t/i) without saving and restoring them
- The caller can rely on save registers ($s/i) without fear of callee changing them

Agenda

- Memory Heap
- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Technology Break
- Instructions as Numbers
- Assembly Language to Machine Language
- Summary
Key Concepts

- Inside computers, everything is a number
- But everything is of a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, ...
- Integer and floating point operations can lead to results too big to store within their representations: overflow/underflow

Number Representation

- Value of i-th digit is \( d \times \text{Base}^i \) where i starts at 0 and increases from right to left:
  \[
  123_{10} = 1_{10} \times 10_{10}^2 + 2_{10} \times 10_{10}^1 + 3_{10} \times 10_{10}^0
  = 1 \times 100_{10} + 2 \times 10_{10} + 3_{10}
  = 100_{10} + 20_{10} + 3_{10}
  = 123_{10}
  \]
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use \( 1_{\text{two}}, 5_{\text{ten}}, 10_{\text{hex}} \) to be clearer
    (vs. \( 1_{2}, 4_{8}, 5_{10}, 10_{16} \))
Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- \( \text{FFF}_{\text{hex}} = 15_{\text{ten}} \times 16_{\text{ten}}^2 + 15_{\text{ten}} \times 16_{\text{ten}}^1 + 15_{\text{ten}} \times 16_{\text{ten}}^0 \)
  \[= 3840_{\text{ten}} + 240_{\text{ten}} + 15_{\text{ten}} \]
  \[= 4095_{\text{ten}} \]
- \( 111111111111_{\text{two}} = \text{FFF}_{\text{hex}} = 4095_{\text{ten}} \)
- 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 have \textit{signed integers}, e.g., 7, -255:
  \[
  \text{int } x, \ y, \ z;
  \]
- C, C++ also have \textit{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

0000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = 0\textsubscript{ten}
0000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = 1\textsubscript{ten}
0000 0000 0000 0000 0000 0000 0000 0010\textsubscript{two} = 2\textsubscript{ten}
...
0111 1111 1111 1111 1111 1111 1111 1101\textsubscript{two} = 2,147,483,645\textsubscript{ten}
0111 1111 1111 1111 1111 1111 1111 1110\textsubscript{two} = 2,147,483,646\textsubscript{ten}
0111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = 2,147,483,647\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0000\textsubscript{two} = 2,147,483,648\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0001\textsubscript{two} = 2,147,483,649\textsubscript{ten}
1000 0000 0000 0000 0000 0000 0000 0010\textsubscript{two} = 2,147,483,650\textsubscript{ten}
...
1111 1111 1111 1111 1111 1111 1111 1101\textsubscript{two} = 4,294,967,293\textsubscript{ten}
1111 1111 1111 1111 1111 1111 1111 1110\textsubscript{two} = 4,294,967,294\textsubscript{ten}
1111 1111 1111 1111 1111 1111 1111 1111\textsubscript{two} = 4,294,967,295\textsubscript{ten}

Signed Integers and Two’s Complement Representation

- Signed integers in C; want \(\frac{1}{2}\) numbers <0, want \(\frac{1}{2}\) numbers >0, and want one 0
- Two’s complement treats 0 as positive, so 32-bit word represents \(2^{32}\) integers from \(-2^{31} \text{ (–2,147,483,648)}\) to \(2^{31}-1\) (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computers uses two’s complement today
- Most significant bit (leftmost) is the 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>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000</td>
<td>0&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>00000000000000000000000000000001</td>
<td>1&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01111111111111111111111111111101</td>
<td>2,147,483,645&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>01111111111111111111111111111100</td>
<td>2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>01111111111111111111111111111101</td>
<td>2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>10000000000000000000000000000000</td>
<td>-2,147,483,648&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>10000000000000000000000000000001</td>
<td>-2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>10000000000000000000000000000010</td>
<td>-2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111111111111111111111111111</td>
<td>-3&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>11111111111111111111111111111111</td>
<td>-2&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>11111111111111111111111111111111</td>
<td>-1&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

### Two's Complement Examples

- **Assume for simplicity 4 bit width, -8 to +7 represented**

<table>
<thead>
<tr>
<th>3</th>
<th>0011</th>
<th>+2</th>
<th>0010</th>
<th>5</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Overflow!**

<table>
<thead>
<tr>
<th>7</th>
<th>0111</th>
<th>+1</th>
<th>0001</th>
<th>-8</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

- **Underflow!**

<table>
<thead>
<tr>
<th>-8</th>
<th>1000</th>
<th>+7</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

**Carry into MSB =** Carry Out MSB

**Overflow!**

**Underflow!**
Suppose we had a 5 bit word. What integers can be represented in two’s complement?

-32 to +31

0 to +31

-16 to +15

-15 to +16
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>1</td>
</tr>
<tr>
<td>NOR</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Examples

• If register $t2$ contains
  0000 0000 0000 0000 0000 1101 1100 0000\text{two}
• And register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000\text{two}
• What is value of $t0$ after:
  and $t0,t1,t2$ # reg $t0 = \text{reg } t1 \& \text{reg } t2$
  0000 0000 0000 0000 0000 1100 0000 0000\text{two}
Examples

• If register $t2$ contains
  0000 0000 0000 0000 0000 1101 1100 0000\textsubscript{two}
• And register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000\textsubscript{two}
• What is value of $t0$ after:
  or $t0,t1,t2$ # reg $t0$ = reg $t1$ | reg $t2$

0000 0000 0000 0000 00\textsubscript{11} 11\textsubscript{01} 11\textsubscript{00} 0000\textsubscript{two}
Examples

• If register $t2$ contains
  0000 0000 0000 0000 0000 0000 1101 1100 0000_{two}

• And register $t1$ contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}

• What is value of $t0$ after:
  nor $t0$,$t1$,$zero$ # reg $t0 = \sim (\text{reg } t1 | 0)$
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 number
• For example, if register $s0$ contained
  0000 0000 0000 0000 0000 0000 0000 1001<sub>two</sub> = 9<sub>ten</sub>
• If executed sll $s0$, $s0$, 4, result is:
  0000 0000 0000 0000 0000 0000 1001 0000<sub>two</sub> = 144<sub>ten</sub>
• And 9<sub>ten</sub> × 2<sub>ten</sub><sup>4</sup> = 9<sub>ten</sub> × 16<sub>ten</sub> = 144<sub>ten</sub>
• Shift right logical moves n bits to the right (insert 0s into empty bits)
  — NOT same as dividing by 2<sub>n</sub> (negative numbers fail)

Shifting

• Shift right arithmetic moves n bits to the right
  (insert high order sign bit into empty bits)
• For example, if register $s0$ contained
  0000 0000 0000 0000 0000 0000 0001 1001<sub>two</sub> = 25<sub>ten</sub>
• If executed sra $s0$, $s0$, 4, result is:
Shifting

• Shift right arithmetic moves \( n \) bits to the right (insert high order sign bit into empty bits)

• For example, if register \( $s0 \) contained

\[
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \ \text{two} = 25_{\text{ten}}
\]

• If executed \( \text{sra} \ $s0, \ $s0, \ 4 \), result is:

\[
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \ \text{two} = 1_{\text{ten}}
\]
Shift right arithmetic moves \( n \) bits to the right (insert high order sign bit into empty bits)

- For example, if register \( s0 \) contained
  \[
  1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{\text{two}} = -25_{\text{ten}}
  \]
- If executed \( \text{sra}\ s0,\ s0,\ 4 \), result is:
  \[
  1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}
  \]
- Unfortunately, this is NOT same as dividing by \( 2^n \)
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 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?
  - slti (set less than immediate)?
C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn’t. Why?

- Logical operations AND and OR do & and | while conditional branches do && and ||
- The previous statement has it backwards: && and || logical ops, & and | are branches
- They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C
Peer Instruction Answer

• C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn’t. Why?

  Logical operations AND and OR implement & and | while conditional branches implement && and ||

Reason:

  e.g., && is logical and: true && true is true and everything else is false

  & is bitwise and: e.g., \((1010_{\text{two}} \& 1000_{\text{two}}) = 1000_{\text{two}}\)
Administrivia

• HW #3 Due Sunday @ 11:59:59
• Project #1 Part 1 Due Sunday @ 11:59:59
• Midterm on the horizon:
  – 10/17 (4 weeks), 6-9 PM
  – It’s going to be completely multiple choice!

“And in Conclusion, ...”

• 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 numInteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
• Floating point is an approximation of reals
• Everything is a (binary) number in a computer
  – Instructions and data; stored program concept