These slides are mostly taken verbatim, or with minor changes, from those prepared by Mary Jane Irwin (www.cse.psu.edu/~mji) of The Pennsylvania State University [Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - **Irwin CSE331** = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - **Irwin CSE431** = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - **Hegner UU** = slide by Stephen J. Hegner at Umeå University.
Review: R Format Instructions

- **R format**
  
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **Arithmetic instructions**

  add $t0, $s1, $s2  
  sub $t0, $s1, $s2

  | 0x00 | 17 | 18 | 8 | 0 | 0x20 | add |
  | 0x00 | 17 | 18 | 8 | 0 | 0x22 | sub |

  sll $t0, $s1, 4  
  srl $t0, $s1, 4  
  sra $t0, $s1, 4

  | 0x00 | 17 | 8 | 4 | 0x00 | sll |
  | 0x00 | 17 | 8 | 4 | 0x02 | srl |
  | 0x00 | 17 | 8 | 4 | 0x03 | sra |

  and $t0, $s1, $s2  
  or $t0, $s1, $s2  
  nor $t0, $s1, $s2

  | 0x00 | 17 | 18 | 8 | 0 | 0x24 | and |
  | 0x00 | 17 | 18 | 8 | 0 | 0x25 | or |
  | 0x00 | 17 | 18 | 8 | 0 | 0x27 | nor |
Review: I Format Instructions

- **I format**
  - Table: op, rs, rt, two's compl number
  - Bit counts: 6 bits, 5 bits, 5 bits, 16 bit

- **Data transfer instructions**
  - Example: `lw $t0, 24($s2)`
  - Binary values:
    - 0x23
    - 0x2b
  - Decimals:
    - 18
    - 8
    - 24

- **Immediate instructions**
  - Example: `addi $t0, $s1, 9`
  - Binary values:
    - 0x08
  - Decimals:
    - 17
    - 8
    - 9
  - Other immediate instructions:
    - `andi $t0, $s1, 0xff00`
    - `ori $t0, $s1, 0xff00`
MIPS Control Flow Instructions

- **MIPS conditional branch instructions:**
  
  bne $s0, $s1, Lbl  # go to Lbl if $s0≠$s1  
  beq $s0, $s1, Lbl  # go to Lbl if $s0=$s1

- **Ex:** if (i==j) h = i + j;
  
  bne $s0, $s1, Lbl1
  add $s3, $s0, $s1
  Lbl1:  ...

- **Instruction Format (I format):**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>16</td>
<td>17</td>
<td>???</td>
</tr>
</tbody>
</table>

- **How is the branch destination address specified?**
Specifying Branch Destinations

- Could specify the memory address of the branch target - but that would require a 32-bit field

- Could use a “base” register and add to it the 16-bit offset

  - which register?
    - Instruction Address Register (PC = program counter) - its use is automatically implied by branch
    - PC gets updated (PC+4) during the Fetch cycle so that it holds the address of the next instruction

  - limits the branch distance to $-2^{15}$ to $+2^{15}-1$ instructions from the (instruction after the) branch
    - but most branches are local anyway

```
PC → bne $s0,$s1,Lbl1
    add $s3,$s0,$s1
Lbl1:  ...
```
**Disassembling Branch Destinations**

- The contents of the updated PC (PC+4) is
  - added to the 16 bit branch offset;
  - which is converted into a 32-bit value by concatenating two low-order zeros to make it a word address;
  - and then sign-extending those 18 bits from the low order 16 bits of the branch instruction.

- The result is written into the PC if the branch condition is true as part of the **Exec** cycle - before the next **Fetch** cycle
Offset Tradeoffs

- Why not just store the word offset in the low order 16 bits? Then the two low order zeros wouldn’t have to be concatenated, it would be less confusing, …

- That would limit the branch distance to $-2^{13}$ to $+2^{13}-1$ instructions from the (instruction after the) branch

- And concatenating the two zero bits costs us very little in additional hardware and has no impact on the clock cycle time
Assembling Branches Example

- **Assembly code**
  
  ```
  bne $s0, $s1, Lbl1
  add $s3, $s0, $s1
  
  Lbl1: ... 
  ```

- **Machine Format of `bne`:**
  
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

- **Remember**
  
  - After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction.
  - The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC.
Assembling Branches Example

- Assembly code
  
  ```
  bne $s0, $s1, Lbl1
  add $s3, $s0, $s1
  ```
  
  Lbl1: ... 

- Machine Format of `bne`:
  
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>16</td>
<td>17</td>
<td>0x0001</td>
</tr>
</tbody>
</table>

- Remember
  
  - After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction.
  
  - The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC.
In Support of Branch Instructions

- We have beq, bne, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, slt

- Set on less than instruction:

  \[
  \text{slt } t_0, s_0, s_1 \quad \# \text{ if } s_0 < s_1 \quad \text{then} \\
  \quad \# \quad t_0 = 1 \quad \text{else} \\
  \quad \# \quad t_0 = 0
  \]

- Instruction format (R format):

  | 0x00 | 16 | 17 | 8 | 0x24 |

- Alternate versions of slt

  \[
  \text{slti } t_0, s_0, 25 \quad \# \text{ if } s_0 < 25 \text{ then } t_0=1 \ldots \\
  \text{sltu } t_0, s_0, s_1 \quad \# \text{ if } s_0 < s_1 \text{ then } t_0=1 \ldots \\
  \text{sltiu } t_0, s_0, 25 \quad \# \text{ if } s_0 < 25 \text{ then } t_0=1 \ldots
  \]
More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than: `blt $s1, $s2, Label`

  - less than or equal to: `ble $s1, $s2, Label`

  - greater than: `bgt $s1, $s2, Label`

  - greater than or equal to: `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - It is why the assembler needs a reserved register (`$at`)
More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than: `blt $s1, $s2, Label`
  - `slt $at, $s1, $s2`#$at set to 1 if $s1 < $s2
  - `bne $at, $zero, Label`#$s1 < $s2
  - less than or equal to: `ble $s1, $s2, Label`
  - greater than: `bgt $s1, $s2, Label`
  - great than or equal to: `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - It is why the assembler needs a reserved register ($at)
Another Instruction for Changing Flow

- MIPS also has an unconditional branch instruction or jump instruction:

```
j Lbl #go to Lbl
```

- Example:

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
beq $s0, $s1, Else
add $s3, $s0, $s1
j Exit
```

```
Else: sub $s3, $s0, $s1
Exit: ...
```
Assembling Jumps

- **Instruction:**
  
  \[
  j \text{ Lbl} \quad \# \text{go to Lbl}
  \]

- **Machine Format (J format):**

  \[
  \begin{array}{c|c}
  \text{op} & \text{26-bit address} \\
  \hline
  0x02 & \text{????}
  \end{array}
  \]

- **How is the jump destination address specified?**
  - As an absolute address formed by
    - concatenating 00 as the 2 low-order bits to make it a word address
    - concatenating the upper 4 bits of the current PC (now PC+4)
Disassembling Jump Destinations

- The low-order 26 bits of the jump instruction is converted into a 32-bit jump destination address by concatenating two low-order zeros to create an 28 bit (word) address and then concatenating the upper 4 bits of the current PC (now PC+4) to create a 32 bit (word) address that is put into the PC prior to the next Fetch cycle.
Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

  \[
  \text{beq} \quad \$s0, \quad \$s1, \quad \text{L1}
  \]

  becomes

  \[
  \text{bne} \quad \$s0, \quad \$s1, \quad \text{L2}
  \]

  \[
  \text{j} \quad \text{L1}
  \]

  \[
  \text{L2:}
  \]
Assembling Branches and Jumps

Assemble the MIPS machine code for the following code sequence. Assume that the addr of the `beq` instr is $0x00400020$

```
beq $s0, $s1, Else
add $s3, $s0, $s1
j Exit
Else: sub $s3, $s0, $s1
Exit: ...
```
Assemble the MIPS machine code for the following code sequence. Assume that the addr of the \texttt{beq} instr is \texttt{0x00400020}_{\text{hex}}.

\begin{verbatim}
beq  $s0, $s1, Else
add  $s3, $s0, $s1
j    Exit
Else: sub  $s3, $s0, $s1
Exit: ...
\end{verbatim}

\begin{verbatim}
0x00400020  4  16  17  2
0x00400024  0  16  17  19  0  0x20
0x00400028  2  0000 0100 0 ... 0 0011 00_2
\end{verbatim}

\begin{equation}
\text{jmp dst} = (0x0) 0x040003 00_2(00_2) = 0x00400030
\end{equation}
Compiling While Loops

- Compile the assembly code for the C `while` loop where `i` is in `$s0`, `j` is in `$s1`, and `k` is in `$s2`

  ```
  while (i!=k)
    i=i+j;
  ```

- **Basic block** – A sequence of instructions without branches (except at the end) and without branch targets (except at the beginning)
Compiling While Loops

- Compile the assembly code for the C while loop where i is in $s0, j is in $s1, and k is in $s2

  ```
  while (i!=k)
    i=i+j;
  ```

- **Basic block** – A sequence of instructions without branches (except at the end) and without branch targets (except at the beginning)
Compiling Another While Loop

- Compile the assembly code for the C `while` loop where 
  i is in $s0, k is in $s1, and the base address of the array 
  `save` is in $s2

  ```c
  while (save[i] == k)
    i += 1;
  ```
Compiling Another While Loop

- Compile the assembly code for the C `while` loop where `i` is in `$s0`, `k` is in `$s1`, and the base address of the array `save` is in `$s2`

  ```c
  while (save[i] == k)
    i += 1;
  ```

  
  ```assembly
  Loop:  sll  $t1, $s0, 2
           add  $t1, $t1, $s2
           lw   $t0, 0($t1)
           bne  $t0, $s1, Exit
           addi $s0, $s0, 1
           j    Loop
  Exit:  . . .
  ```
Most higher level languages have case or switch statements allowing the code to select one of many alternatives depending on a single value.

Instruction:

```
jr $t1 #go to address in $t1
```

Machine format (R format):

```
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>9</td>
<td>0x08</td>
</tr>
</tbody>
</table>
```
### Compiling a Case (Switch) Statement

```c
switch (k) {
    case 0:  h=i+j;  break; /*k=0*/
    case 1:  h=i+h;  break; /*k=1*/
    case 2:  h=i-j;  break; /*k=2*/
}
```

- Assume three sequential words in memory starting at the address in $t4$ have the addresses of the labels L0, L1, and L2 and $k$ is in $s2$.

```
add $t1, $s2, $s2  #$t1 = 2*k
add $t1, $t1, $t1  #$t1 = 4*k
add $t1, $t1, $t4  #$t1 = addr of JumpT[k]
lw  $t0, 0($t1)  #$t0 = JumpT[k]
jr   $t0   #jump based on $t0
```

```c
L0:  add $s3, $s0, $s1  #k=0 so h=i+j
    j   Exit
```

```c
L1:  add $s3, $s0, $s3  #k=1 so h=i+h
    j   Exit
```

```c
L2:  sub $s3, $s0, $s1  #k=2 so h=i-j
```

Exit: . . .
Programming Styles

- Procedures (subroutines, functions) allow the programmer to structure programs making them
  - easier to understand and debug and
  - allowing code to be reused

- Procedures allow the programmer to concentrate on one portion of the code at a time
  - parameters act as barriers between the procedure and the rest of the program and data, allowing the procedure to be passed values (arguments) and to return values (results)
Six Steps in Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - $a0 - $a3: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - $v0 - $v1: two value registers for result values
6. Callee returns control to the caller
   - $ra: one return address register to return to the point of origin
## Review: MIPS Register Naming Convention

<table>
<thead>
<tr>
<th>Nick Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>reserved for OS</td>
<td>n.a.</td>
</tr>
<tr>
<td>$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 addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Instruction for Calling a Procedure

- MIPS procedure call instruction:

  jal  ProcAddress  #jump and link

- Saves PC+4 in register $ra as the link to the following instruction to set up the procedure return

- Machine format (J format):

<table>
<thead>
<tr>
<th>op</th>
<th>26 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>????</td>
</tr>
</tbody>
</table>

- Then can do procedure return with just

  jr  $ra  #return
Basic Procedure Flow

- For a procedure that computes the GCD of two values $i$ (in $t0$) and $j$ (in $t1$)
  
  \[ \text{gcd}(i,j); \]

- The **caller** puts the $i$ and $j$ (the parameters values) in $a0$ and $a1$ and issues a
  
  \[ \text{jal gcd} \quad \# \text{jump to routine gcd} \]

- The **callee** computes the GCD, puts the result in $v0$, and returns control to the **caller** using
  
  \[ \begin{align*}
  \text{gcd:} & \quad \ldots \quad \# \text{code to compute gcd} \\
  \text{jr} & \quad \text{ra} \quad \# \text{return}
  \end{align*} \]
Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - callee uses a stack – a last-in-first-out queue

- One of the general registers, $sp ($29), is used to address the stack (which “grows” from high address to low address)

  - add data onto the stack – push
    \[ sp = sp - 4 \]
    data on stack at new $sp

  - remove data from the stack – pop
    \[ sp = sp + 4 \]
    data from stack at $sp
### Allocating Space on the Stack

- The segment of the stack containing a procedure’s saved registers and local variables is its **procedure frame** (aka **activation record**)
  - The frame pointer ($fp$) points to the first word of the frame of a procedure – providing a stable “base” register for the procedure
  - $fp$ is initialized using $sp$ on a call and $sp$ is restored using $fp$ on a return

<table>
<thead>
<tr>
<th>high addr</th>
<th>low addr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Saved argument regs (if any)</strong></td>
<td>$fp$</td>
</tr>
<tr>
<td><strong>Saved return</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Saved local regs (if any)</strong></td>
<td>$sp$</td>
</tr>
<tr>
<td><strong>Local arrays &amp; structures (if any)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Allocating Space on the Heap

- There is a static data segment area for storing constants and other static variables (e.g., arrays)
- And a dynamic data segment (aka heap) area for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()` in C
Leaf procedures are ones that do not call other procedures. Give the MIPS assembler code for
int leaf_ex (int g, int h, int i, int j)
{  int f;
   f = (g+h) - (i+j);
   return f; }
where g, h, i, and j are in $a0, $a1, $a2, $a3
Compiling a C Leaf Procedure

- Leaf procedures are ones that do not call other procedures. Give the MIPS assembler code for

```c
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

where \( g, h, i, \) and \( j \) are in \$a0, \$a1, \$a2, \$a3

```asm
leaf_ex:  addi       $sp,$sp,-8   #make stack room
sw        $t1,4($sp)  #save $t1 on stack
sw        $t0,0($sp)  #save $t0 on stack
add       $t0,$a0,$a1
add       $t1,$a2,$a3
sub       $v0,$t0,$t1
lw        $t0,0($sp)  #restore $t0
lw        $t1,4($sp)  #restore $t1
addi      $sp,$sp,8   #adjust stack ptr
jr         $ra
```
Nested Procedures

- What happens to return addresses with nested procedures?

```c
int rt_1(int i) {
    if (i == 0) return 0;
    else return rt_2(i-1);
}
```

caller: jal rt_1
next:    . . .

```
rt_1:    bne $a0, $zero, to_2
         add $v0, $zero, $zero
         jr $ra

to_2:    addi $a0, $a0, -1
         jal rt_2
         jr $ra
```

rt_2:    . . .
On the call to \texttt{rt\_1}, the return address (\texttt{next} in the \texttt{caller} routine) gets stored in \texttt{$ra$}. What happens to the value in \texttt{$ra$} (when \texttt{i \neq 0}) when \texttt{rt\_1} makes a call to \texttt{rt\_2}?
Saving the Return Address, Part 1

- Nested procedures (i passed in $a0, return value in $v0)

Save the return address (and arguments) on the stack

```
r
<table>
<thead>
<tr>
<th>high addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>old TOS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>low addr</td>
</tr>
</tbody>
</table>

rt_1: bne $a0, $zero, to_2
add $v0, $zero, $zero
jr $ra
to_2: addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
addi $a0, $a0, -1
jal rt_2
bk_2: lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra
```
Saving the Return Address, Part 1

- Nested procedures (i passed in $a0, return value in $v0)

```

| high addr | $pc → rt_1: bne $a0, $zero, to_2 |
|           | add $v0, $zero, $zero |
|           | jr $ra |

| low addr  | $pc → to_2: addi $sp, $sp, -8 |
|           | $pc → sw $ra, 4($sp) |
|           | $pc → sw $a0, 0($sp) |
|           | $pc → addi $a0, $a0, -1 |
|           | $pc → jal rt_2 |
|           | $pc → bk_2: lw $a0, 0($sp) |
|           | lw $ra, 4($sp) |
|           | addi $sp, $sp, 8 |
|           | jr $ra |

```

- Save the return address (and arguments) on the stack
Saving the Return Address, Part 2

- Nested procedures ($i$ passed in $a0$, return value in $v0$)
  
  ```
  rt_1: bne $a0, $zero, to_2
        add $v0, $zero, $zero
        jr $ra
  to_2: addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        addi $a0, $a0, -1
        jal rt_2
  bk_2: lw $a0, 0($sp)
        lw $ra, 4($sp)
        addi $sp, $sp, 8
        jr $ra
  ```

- Save the return address (and arguments) on the stack
Saving the Return Address, Part 2

- Nested procedures (i passed in $a0, return value in $v0)

```assembly
   high addr
    old TOS <- $sp
    caller rt addr
    old $a0

   low addr
    $pc ->

   caller rt addr $ra

   rt_1:  bne   $a0, $zero, to_2
          add $v0, $zero, $zero
          jr   $ra

   to_2:  addi  $sp, $sp, -8
          sw   $ra, 4($sp)
          sw   $a0, 0($sp)
          addi $a0, $a0, -1
          jal  rt_2

   bk_2:  lw   $a0, 0($sp)
          lw   $ra, 4($sp)
          addi $sp, $sp, 8
          jr   $ra
```

- Save the return address (and arguments) on the stack
A procedure for calculating factorial

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

A recursive procedure (one that calls itself!)

```c
fact (0) = 1
fact (1) = 1 * 1 = 1
fact (2) = 2 * 1 * 1 = 2
fact (3) = 3 * 2 * 1 * 1 = 6
fact (4) = 4 * 3 * 2 * 1 * 1 = 24
```

Assume \( n \) is passed in \( \texttt{a0} \); result returned in \( \texttt{v0} \)
Compiling a Recursive Procedure

```
fact: addi $sp, $sp, -8     #adjust stack pointer
    sw $ra, 4($sp)          #save return address
    sw $a0, 0($sp)          #save argument n
    slti $t0, $a0, 1       #test for n < 1
    beq $t0, $zero, L1     #if n >=1, go to L1
    addi $v0, $zero, 1     #else return 1 in $v0
    addi $sp, $sp, 8       #adjust stack pointer
    jr $ra                 #return to caller (1st)

L1: addi $a0, $a0, -1       #n >=1, so decrement n
    jal fact               #call fact with (n-1)
    #this is where fact returns

bk_f: lw $a0, 0($sp)        #restore argument n
    lw $ra, 4($sp)         #restore return address
    addi $sp, $sp, 8       #adjust stack pointer
    mul $v0, $a0, $v0      #$v0 = n * fact(n-1)
    jr $ra                 #return to caller (2nd)
```
A Look at the Stack for $a0 = 2, Part 1

- Stack state after execution of the first encounter with `jal`
  (second call to fact routine with $a0 now holding 1)
  - saved return address to caller routine (i.e., location in the main routine where first call to fact is made) on the stack
  - saved original value of $a0 on the stack
## A Look at the Stack for $a0 = 2, Part 1

- Stack state after execution of the first encounter with `jal` *(second call to fact routine with $a0 now holding 1)*
  - saved return address to caller routine (i.e., location in the main routine where *first* call to fact is made) on the stack
  - saved original value of $a0 on the stack

<table>
<thead>
<tr>
<th>old TOS</th>
<th>caller rt addr</th>
<th>$a0 = 2</th>
<th>←$sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bk_f</td>
<td>$ra</td>
<td>$a0</td>
<td>$v0</td>
</tr>
</tbody>
</table>
A Look at the Stack for $a0 = 2, Part 2

<table>
<thead>
<tr>
<th>old TOS</th>
<th>←$sp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stack state after execution of the second encounter with `jal` (third call to fact routine with $a0$ now holding 0)
  - save return address of instruction in caller routine (instruction after `jal`) on the stack
  - save previous value of $a0$ on the stack
A Look at the Stack for $a0 = 2$, Part 2

- Stack state after execution of the second encounter with `jal` (*third* call to fact routine with $a0$ now holding 0)
  - saved return address of instruction in caller routine (instruction after `jal`) on the stack
  - saved previous value of $a0$ on the stack

<table>
<thead>
<tr>
<th></th>
<th>old TOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller rt addr</td>
<td>$a0 = 2$</td>
</tr>
<tr>
<td>bk_f</td>
<td>$a0 = 1$</td>
</tr>
</tbody>
</table>
A Look at the Stack for $a0 = 2$, Part 3

- Stack state after execution of the first encounter with the first $jr$ ($v0$ initialized to 1)
  - stack pointer updated to point to third call to fact

<table>
<thead>
<tr>
<th>old TOS</th>
<th>←$sp$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$ra$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$v0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
A Look at the Stack for $a0 = 2$, Part 3

<table>
<thead>
<tr>
<th>old TOS</th>
<th>$ra$</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller rt addr</td>
<td>bk_f</td>
</tr>
<tr>
<td>$a0 = 2$</td>
<td></td>
</tr>
<tr>
<td>bk_f</td>
<td></td>
</tr>
<tr>
<td>$a0 = 1$</td>
<td>←$sp$</td>
</tr>
<tr>
<td>bk_f</td>
<td></td>
</tr>
<tr>
<td>$a0 = 0$</td>
<td></td>
</tr>
</tbody>
</table>

- Stack state after execution of the first encounter with the first `jr` ($v0$ initialized to 1)
  - stack pointer updated to point to *third* call to fact
A Look at the Stack for $a0 = 2, Part 4

- Stack state after execution of the first encounter with the second `jr` (return from fact routine after updating $v0 to 1 * 1)
  - return address to caller routine (`bk_f` in fact routine) restored to $ra from the stack
  - previous value of $a0 restored from the stack
  - stack pointer updated to point to `second` call to fact
A Look at the Stack for $a0 = 2$, Part 4

<table>
<thead>
<tr>
<th>old TOS</th>
<th>caller rt addr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a0 = 2$</td>
</tr>
<tr>
<td></td>
<td>$bk_f$</td>
</tr>
<tr>
<td></td>
<td>$a0 = 1$</td>
</tr>
<tr>
<td></td>
<td>$bk_f$</td>
</tr>
<tr>
<td></td>
<td>$a0 = 0$</td>
</tr>
</tbody>
</table>

Stack state after execution of the first encounter with the second $jr$ (return from fact routine after updating $v0$ to $1 * 1$)

- return address to caller routine ($bk_f$ in fact routine) restored to $ra$ from the stack
- previous value of $a0$ restored from the stack
- stack pointer updated to point to *second* call to fact
# A Look at the Stack for $a0 = 2, Part 5

- Stack state after execution of the second encounter with the second `jr` (return from fact routine after updating $v0 to $2 \times 1 \times 1$)
  - return address to caller routine (main routine) restored to $ra$ from the stack
  - original value of $a0$ restored from the stack
  - stack pointer updated to point to first call to fact
A Look at the Stack for $a0 = 2$, Part 5

<table>
<thead>
<tr>
<th></th>
<th>caller rt addr</th>
<th>$a0 = 2$</th>
<th>$v0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bk_f</td>
<td>$a0 = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bk_f</td>
<td>$a0 = 0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stack state after execution of the second encounter with the second jr (return from fact routine after updating $v0$ to $2 * 1 * 1$)
  - return address to caller routine (main routine) restored to $ra$ from the stack
  - original value of $a0$ restored from the stack
  - stack pointer updated to point to first call to fact
## Review: MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OpC</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong> (R &amp; I format)</td>
<td><strong>add</strong></td>
<td>0 &amp; 20</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td><strong>subtract</strong></td>
<td>0 &amp; 22</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td><strong>add immediate</strong></td>
<td>8</td>
<td>addi $s1, $s2, 4</td>
<td>$s1 = $s2 + 4</td>
</tr>
<tr>
<td></td>
<td><strong>shift left logical</strong></td>
<td>0 &amp; 00</td>
<td>sll $s1, $s2, 4</td>
<td>$s1 = $s2 &lt;&lt; 4</td>
</tr>
<tr>
<td></td>
<td><strong>shift right logical</strong></td>
<td>0 &amp; 02</td>
<td>srl $s1, $s2, 4</td>
<td>$s1 = $s2 &gt;&gt; 4 (fill with zeros)</td>
</tr>
<tr>
<td></td>
<td><strong>shift right arithmetic</strong></td>
<td>0 &amp; 03</td>
<td>sra $s1, $s2, 4</td>
<td>$s1 = $s2 &gt;&gt; 4 (fill with sign bit)</td>
</tr>
<tr>
<td></td>
<td><strong>and</strong></td>
<td>0 &amp; 24</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td><strong>or</strong></td>
<td>0 &amp; 25</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td><strong>nor</strong></td>
<td>0 &amp; 27</td>
<td>nor $s1, $s2, $s3</td>
<td>$s1 = not ($s2</td>
</tr>
<tr>
<td></td>
<td><strong>and immediate</strong></td>
<td>c</td>
<td>and $s1, $s2, ff00</td>
<td>$s1 = $s2 &amp; 0xff00</td>
</tr>
<tr>
<td></td>
<td><strong>or immediate</strong></td>
<td>d</td>
<td>or $s1, $s2, ff00</td>
<td>$s1 = $s2</td>
</tr>
</tbody>
</table>
## Review: MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>OpC</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer (I format)</td>
<td>load word</td>
<td>23</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory($s2+100)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>2b</td>
<td>sw $s1, 100($s2)</td>
<td>Memory($s2+100) = $s1</td>
</tr>
<tr>
<td>Cond. branch (I &amp; R format)</td>
<td>br on equal</td>
<td>4</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1===$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>br on not equal</td>
<td>5</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 !=$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>set on less than immediate</td>
<td>a</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2&lt;100) $s1=1; else $s1=0</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>0 &amp; 2a</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2&lt;$s3) $s1=1; else $s1=0</td>
</tr>
<tr>
<td>Uncond. jump</td>
<td>jump</td>
<td>2</td>
<td>j 2500</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>0 &amp; 08</td>
<td>jr $t1</td>
<td>go to $t1</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>3</td>
<td>jal 2500</td>
<td>go to 10000; $ra=PC+4</td>
</tr>
</tbody>
</table>
Review: MIPS R3000 ISA

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

- 3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>16 bit number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit jump target</td>
</tr>
</tbody>
</table>

R format
I format
J format
Atomic Exchange Support

- Need hardware support for synchronization mechanisms to avoid data races where the results of the program can change depending on how events happen to occur
  - Two memory accesses from different threads to the same location, and at least one is a write

- Atomic exchange (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
  - Implementing an atomic exchange would require both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have a pair of specially configured instructions

```
ll  $t1, 0($s1)          #load linked
sc  $t0, 0($s1)          #store conditional
```
Atomic Exchange with `ll` and `sc`

- If the contents of the memory location specified by the `ll` are changed before the `sc` to the same address occurs, the `sc` fails (returns a zero).

- Swap `$s4` and memory(`$s1`):
  
  ```
  try:  add $t0, $zero, $s4  #$t0=$s4 (exchange value)
       ll $t1, 0($s1)  #load memory value to $t1
       sc $t0, 0($s1)  #try to store exchange
                      #value to memory, if fail
                      #$t0 will be 0
  beq $t0, $zero, try  #try again on failure
  add $s4, $zero, $t1  #load value in $s4
  ```

- If the value in memory between the `ll` and the `sc` instructions changes, then `sc` returns a 0 in `$t0` causing the code sequence to try again.
Review: MIPS R3000 ISA

- **Instruction Categories**
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

- **3 Instruction Formats:** all 32 bits wide

<table>
<thead>
<tr>
<th>Formats</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R format</strong></td>
<td>OP</td>
</tr>
<tr>
<td><strong>I format</strong></td>
<td>OP</td>
</tr>
<tr>
<td><strong>J format</strong></td>
<td>OP</td>
</tr>
</tbody>
</table>

- **Registers**
  - R0 - R31
  - PC
  - HI
  - LO
Addressing Modes Illustrated

1. Register addressing
   - op  rs  rt  rd  funct
   - Register
     - word operand

2. Base (displacement) addressing
   - op  rs  rt  offset
   - Memory
     - word or byte operand
   - base register

3. Immediate addressing
   - op  rs  rt  operand

4. PC-relative addressing
   - op  rs  rt  offset
   - Memory
     - branch destination instruction
     - Program Counter (PC)

5. Pseudo-direct addressing
   - op  jump address
   - Memory
     - jump destination instruction
     - Program Counter (PC)
MIPS Organization So Far

### Processor

- **Register File**
  - src1 addr
  - src2 addr
  - dst addr
  - write data
  - 32 registers ($zero - $ra)

- **Memory**
  - read/write addr
  - read data
  - write data

- **ALU**

### Memory

- **Word Address (Binary)**
  - 0…1100
  - 0…1000
  - 0…0100
  - 0…0000
  - 2^30 words

- **Fetch**
  - PC = PC+4

- **Decode**

- **Exec**

- **Add**