5DV008
Computer Architecture
Umeå University
Department of Computing Science

Stephen J. Hegner

Topic 2: Instructions

Part C: Control Flow

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

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- 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
```

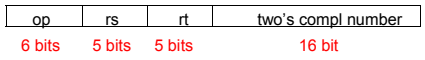
0x00		17	8	4	0x00	sll
0x00		17	8	4	0x02	srl
0x00		17	8	4	0x03	sra

```
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

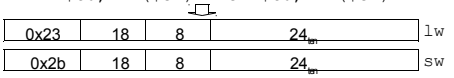
Review: I Format Instructions

- I format



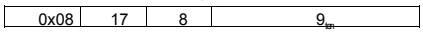
- Data transfer instructions

```
lw $t0, 24($s2)     sw $t0, 24($s2)
```

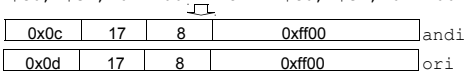


- Immediate instructions

```
addi  $t0, $s1, 9
```




```
andi $t0, $s1, 0xff00     ori  $t0, $s1, 0xff00
```



MIPS Control Flow Instructions

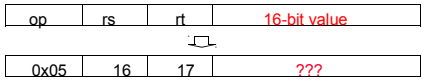
- MIPS conditional branch instructions:

```
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:   ...
```

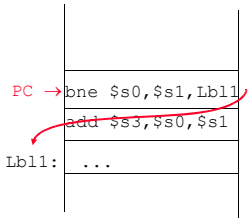
- Instruction Format (I format):



- How is the branch destination address specified?

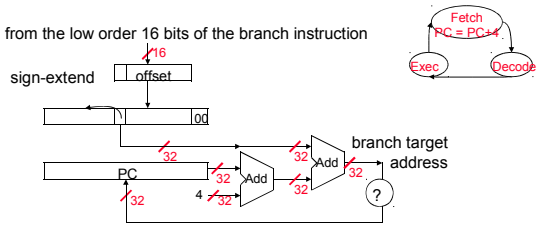
Specifying Branch Destinations

- 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 -2^5 to $+2^5-1$ instr's from the (instruction after the) branch
    - but most branches are local anyway



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
- 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

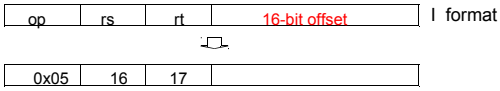
- Why not just store the word offset in the low order 16 bits? Then the two low order zeros wouldn't have to be concatenated, it would be less confusing, …
- That would limit the branch distance to -2^{13} to $+2^{13}-1$ instructions from the (instruction after the) branch
- And concatenating the two 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`:



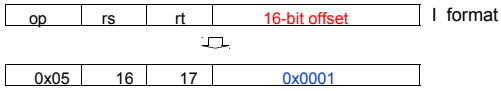
- Remember
  - After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction
  - The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC

Assembling Branches Example

- Assembly code

```
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

- Machine Format of `bne`:



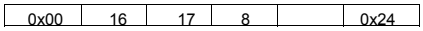
- Remember
  - After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction
  - The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC

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:

```
slt $t0, $s0, $s1    # if $s0 < $s1    then
                     # $t0 = 1         else
                     # $t0 = 0
```

- Instruction format (R format):



- Alternate versions of `slt`

```
slti $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1   # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25   # if $s0 < 25 then $t0=1 ...
```


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`
  - great than or equal to `bge $s1, $s2, Label`
- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$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
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
  - Its 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;
```

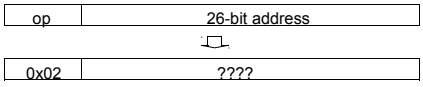
```
        beq  $s0, $s1, Else
        add  $s3, $s0, $s1
        j    Exit
Else:   sub  $s3, $s0, $s1
Exit:   ...
```


Assembling Jumps

- Instruction:

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

- Machine Format (J format):

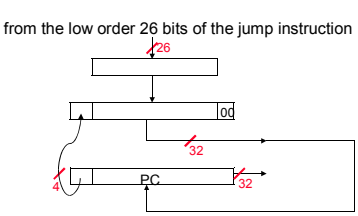


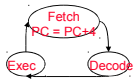
- How is the jump destination address specified?
  - As an absolute address formed by
    - concatenating 00 as the 2 low-order bits to make it a word address
    - concatenating the upper 4 bits of the current PC (now PC+4)

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

```
        beq  $s0, $s1, L1
```

becomes

```
        bne  $s0, $s1, L2
        j    L1
L2:
```


Assembling Branches and Jumps

- Assemble the MIPS machine code for the following code sequence. Assume that the addr of the `beq` instr is $0x00400020_{hex}$

```
        beq  $s0, $s1, Else
        add  $s3, $s0, $s1
        j    Exit
Else:   sub  $s3, $s0, $s1
Exit:   ...
```

Assembling Branches and Jumps

- Assemble the MIPS machine code for the following code sequence. Assume that the addr of the `beq` instr is $0x00400020_{hex}$

```
          beq  $s0, $s1, Else
          add  $s3, $s0, $s1
          j    Exit
Else:     sub  $s3, $s0, $s1
Exit:     ...
```

```
0x00400020          4     16    17          2
0x00400024          0     16    17    19    0   0x20
0x00400028          2     0000 0100 0 ... 0 0011 00₂
```

jmp dst = (0x0) $0x040003\ 00_2(00_2)$

= 0x00400030

```
0x0040002c          0     16    17    19    0   0x22
0x00400030          ...
```


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;
```

```
Loop:   beq  $s0, $s2, Exit
        add  $s0, $s0, $s1
        j    Loop
Exit:   . . .
```

- 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`

```
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`

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

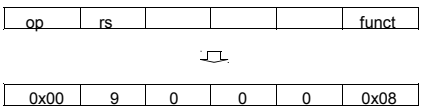
```
Loop:   sll  $t1, $s0, 2
        add  $t1, $t1, $s2
        lw   $t0, 0($t1)
        bne  $t0, $s1, Exit
        addi $s0, $s0, 1
        j    Loop
Exit:   . . .
```


Yet Another Instruction for Changing Flow

- Most higher level languages have `case` or `switch` statements allowing the code to select one of many alternatives depending on a single value
- Instruction:

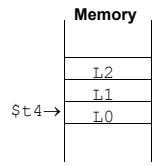
```
jr  $t1        #go to address in $t1
```

- Machine format (R format):



Compiling a Case (Switch) Statement

```
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
L0:     add    $s3, $s0, $s1       #k=0 so h=i+j
        j      Exit
L1:     add    $s3, $s0, $s3       #k=1 so h=i+h
        j      Exit
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

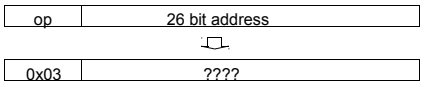
Nick Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for OS	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

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):



- 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`)

```
gcd(i,j);
```

- The caller puts the `i` and `j` (the parameters values) in `$a0` and `$a1` and issues a

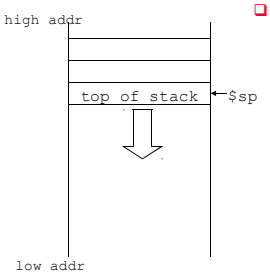
```
jal gcd      #jump to routine gcd
```

- The callee computes the GCD, puts the result in `$v0`, and returns control to the caller using

```
gcd: . . .            #code to compute gcd
     jr  $ra      #return
```

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

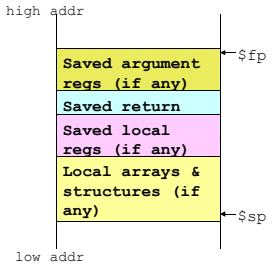


- One of the general registers, `$sp` (`$29`), is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    `$sp` = `$sp` – 4
    data on stack at new $sp
  - remove data from the stack – pop

    data from stack at $sp
    `$sp` = `$sp` + 4

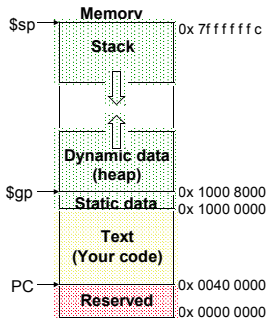
Allocating Space on the Stack



- 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

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



Compiling a C Leaf Procedure

- 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

```
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

```
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?

```
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:   . . .
```

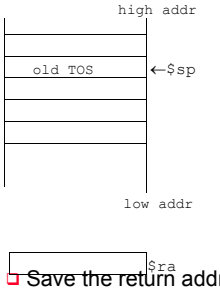
Nested Procedures Outcome

```
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 `rt_1`, the return address (`next` in the caller routine) gets stored in `$ra`. What happens to the value in `$ra` (when `i != 0`) when `rt_1` makes a call to `rt_2`?

Saving the Return Address, Part 1

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

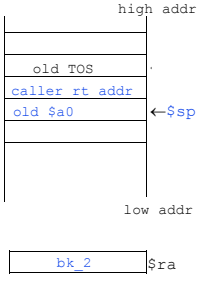



```
rt_1:  bne   $a0, $zero, to_2
       add   $v0, $zero, $zero
       jr    $ra
to_2:  addi  $sp, $sp, -8
       sw    $ra, 4($sp)
       sw    $a0, 0($sp)
       addi  $a0, $a0, -1
       jal   rt_2
bk_2:  lw    $a0, 0($sp)
       lw    $ra, 4($sp)
       addi  $sp, $sp, 8
       jr    $ra
```

- Save the return address (and arguments) on the stack

Saving the Return Address, Part 1

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

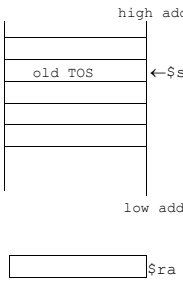



```
$pc →  rt_1:  bne   $a0, $zero, to_2
                    add   $v0, $zero, $zero
                    jr    $ra
$pc →  to_2:  addi  $sp, $sp, -8
$pc →         sw    $ra, 4($sp)
$pc →         sw    $a0, 0($sp)
                    addi  $a0, $a0, -1
$pc →         jal   rt_2
       bk_2:  lw    $a0, 0($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`)

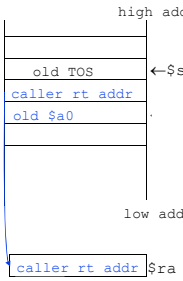



```
rt_1:  bne   $a0, $zero, to_2
       add   $v0, $zero, $zero
       jr    $ra
to_2:  addi  $sp, $sp, -8
       sw    $ra, 4($sp)
       sw    $a0, 0($sp)
       addi  $a0, $a0, -1
       jal   rt_2
bk_2:  lw    $a0, 0($sp)
       lw    $ra, 4($sp)
       addi  $sp, $sp, 8
       jr    $ra
```

- 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`)




```
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)
$pc →  lw    $ra, 4($sp)
$pc →  addi  $sp, $sp, 8
       jr    $ra
```

- Save the return address (and arguments) on the stack

Compiling a Recursive Procedure

- A procedure for calculating factorial

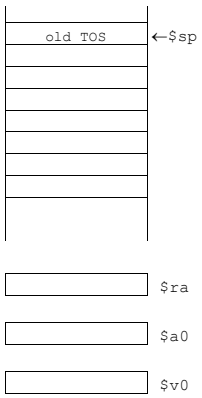
```
int fact (int n) {
if (n < 1) return 1;
else return (n * fact (n-1)); }
```

- A recursive procedure (one that calls itself!)
  - 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 `$a0`; result returned in `$v0`

Compiling a Recursive Procedure

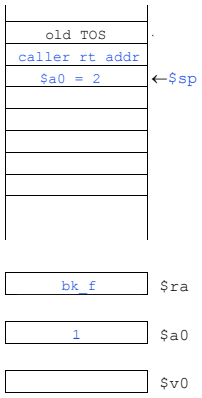
```
fact: addi  $sp, $sp, -8     #adjust stack pointer
      sw    $ra, 4($sp)      #save return address
      sw    $a0, 0($sp)      #save argument n
      slti  $t0, $a0, 1      #test for n < 1
      beq   $t0, $zero, L1   #if n >=1, go to L1
      addi  $v0, $zero, 1    #else return 1 in $v0
    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



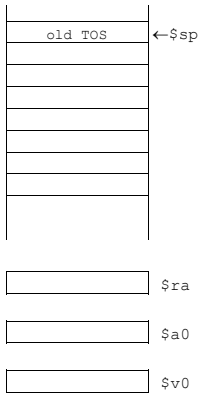
- Stack state after execution of the first encounter with `jal` (*second* call to fact routine with `$a0` now holding 1)
  - saved return address to caller routine (i.e., location in the main routine where *first* call to fact is made) on the stack
  - saved original value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 1



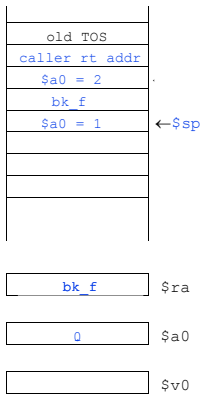
- Stack state after execution of the first encounter with `jal` (*second* call to fact routine with `$a0` now holding 1)
  - saved return address to caller routine (i.e., location in the main routine where *first* call to fact is made) on the stack
  - saved original value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 2



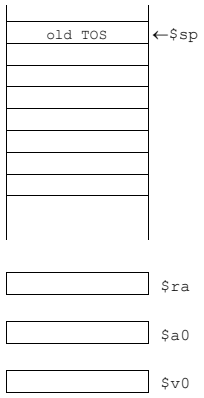
- Stack state after execution of the second encounter with `jal` (*third* call to fact routine with `$a0` now holding 0)
  - save return address of instruction in caller routine (instruction after `jal`) on the stack
  - save previous value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 2



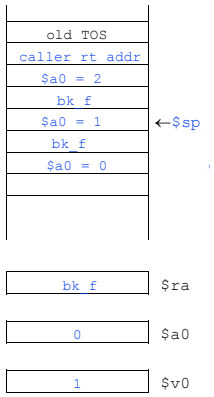
- Stack state after execution of the second encounter with `jal` (*third* call to fact routine with `$a0` now holding 0)
  - saved return address of instruction in caller routine (instruction after `jal`) on the stack
  - saved previous value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 3



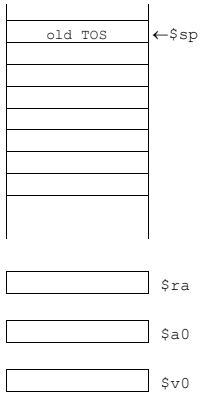
- Stack state after execution of the first encounter with the first `jr` (`$v0` initialized to 1)
  - stack pointer updated to point to *third* call to fact

A Look at the Stack for $a0 = 2, Part 3



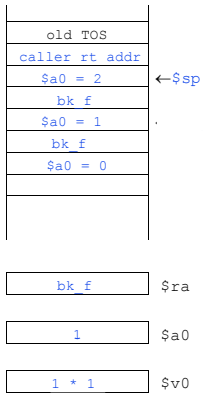
- Stack state after execution of the first encounter with the first `jr` (`$v0` initialized to 1)
  - stack pointer updated to point to *third* call to fact

A Look at the Stack for $a0 = 2, Part 4



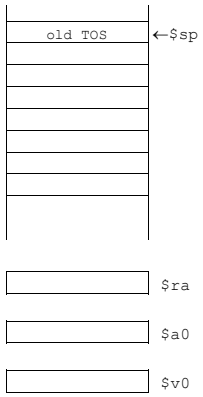
- Stack state after execution of the first encounter with the second `jr` (return from fact routine after updating `$v0` to 1 * 1)
  - return address to caller routine (`bk_f` in fact routine) restored to `$ra` from the stack
  - previous value of `$a0` restored from the stack
  - stack pointer updated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

A Look at the Stack for $a0 = 2, Part 4



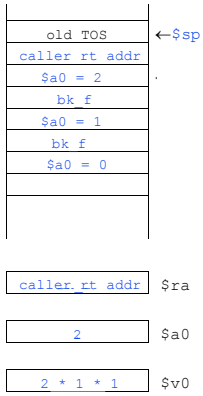
- Stack state after execution of the first encounter with the second `jr` (return from fact routine after updating `$v0` to 1 * 1)
  - return address to caller routine (`bk_f` in fact routine) restored to `$ra` from the stack
  - previous value of `$a0` restored from the stack
  - stack pointer updated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

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 `$v0` to 2 * 1 * 1)
  - return address to caller routine (main routine) restored to `$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

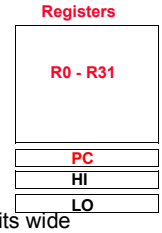
Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 4	$s1 = $s2 + 4
	shift left logical	0 & 00	sll $s1, $s2, 4	$s1 = $s2 << 4
	shift right logical	0 & 02	srl $s1, $s2, 4	$s1 = $s2 >> 4 (fill with zeros)
	shift right arithmetic	0 & 03	sra $s1, $s2, 4	$s1 = $s2 >> 4 (fill with sign bit)
	and	0 & 24	and $s1, $s2, $s3	$s1 = $s2 & $s3
	or	0 & 25	or $s1, $s2, $s3	$s1 = $s2 | $s3
	nor	0 & 27	nor $s1, $s2, $s3	$s1 = not ($s2 | $s3)
	and immediate	c	and $s1, $s2, ff00	$s1 = $s2 & 0xff00
	or immediate	d	or $s1, $s2, ff00	$s1 = $s2 | 0xff00

Review: MIPS Instructions, so far

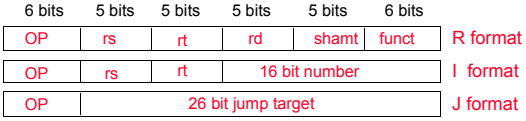
Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	2b	sw $s1, 100($s2)	Memory($s2+100) = $s1
Cond. branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to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

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



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`

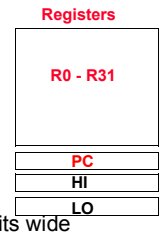
- If the contents of the memory location specified by the `ll` are changed before the `sc` to the same address occurs, the `sc` fails (returns a zero)

```
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.

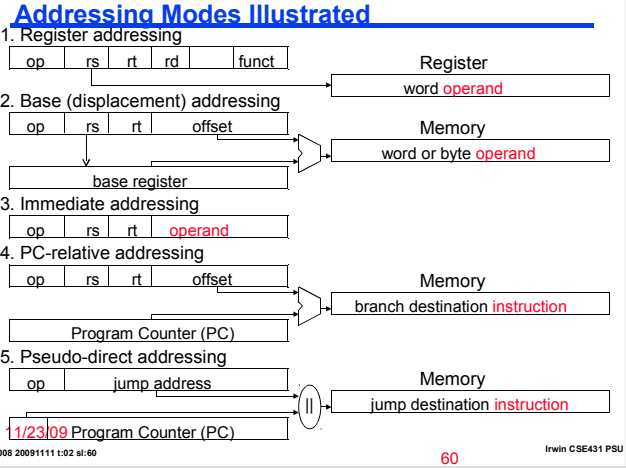
Review: MIPS R3000 ISA

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



- 3 Instruction Formats: all 32 bits wide

Addressing Modes Illustrated



MIPS Organization So Far