Computer Architecture & Assembly Language 14:332:331

Lecture 3

Logical Operation, Branches, & Procedures

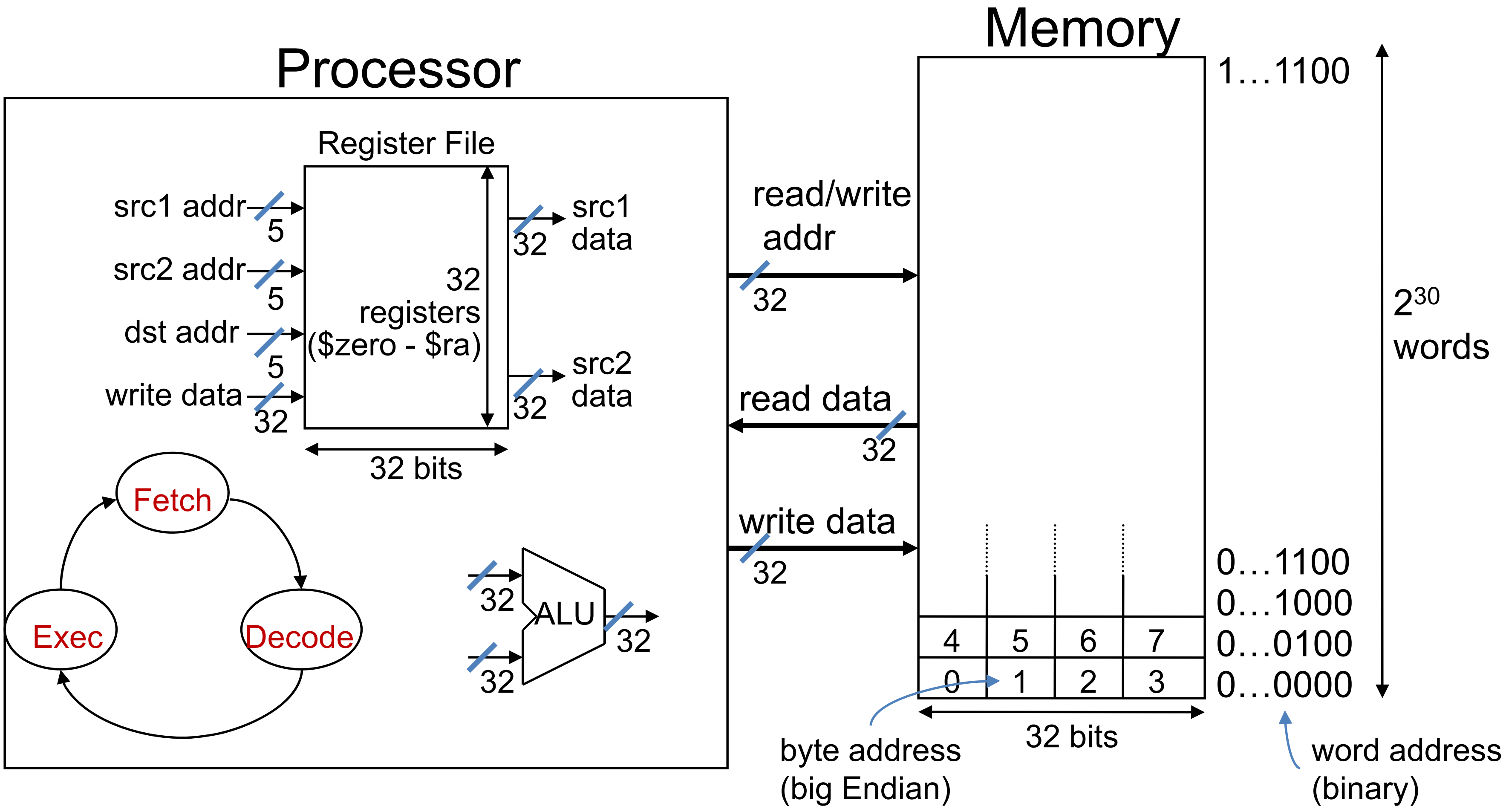
Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier
and Mary Jane Irwin's slides from Penn State University.

Review: MIPS Organization

- Arithmetic instructions – to/from the register file
- Load/store word and byte instructions – from/to memory



Review: MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
Data transfer (I format)	load word	35	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	43	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	32	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	40	sb $s1, 101($s2)	Memory($s2+101) = $s1

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

R format

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

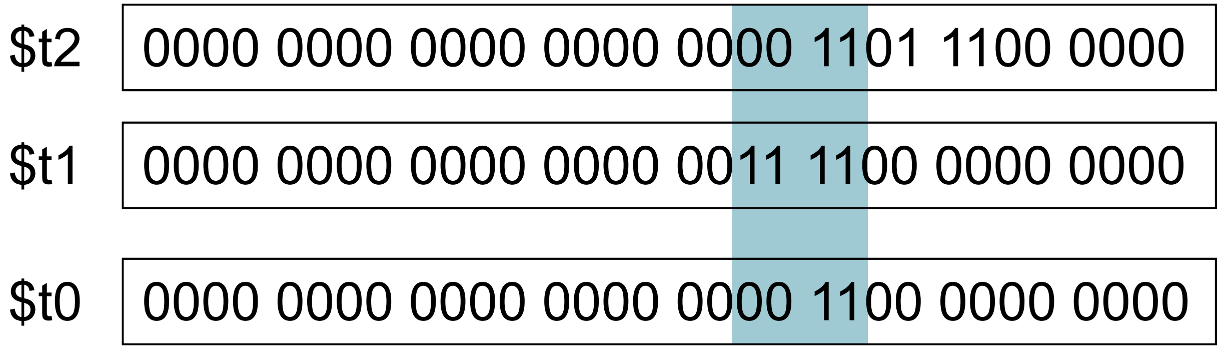
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

R format

```
and $t0, $t1, $t2
```



$t2 0000 0000 0000 0000 0000 1101 1100 0000

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 0000 0000 0000 0000 0000 1100 0000 0000

i format

```
andi $t0, $t1, 0xFF00  #$t0 = $t1 & ff00
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

R format

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

i format

```
ori $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations (Control Flow Instructions)

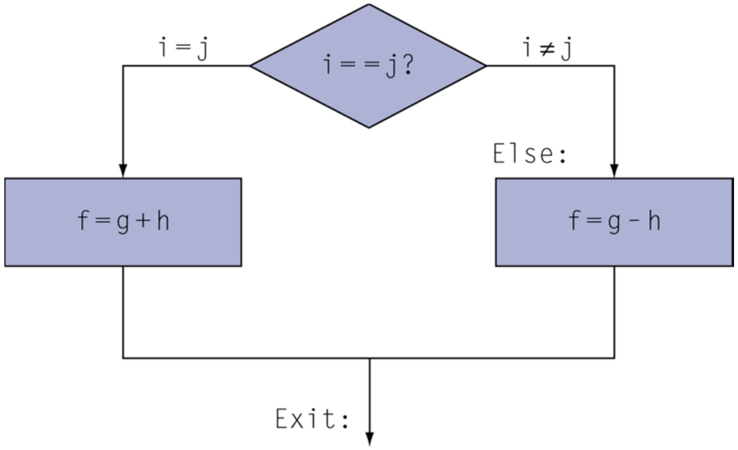
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Conditional Branches

- **Instructions:**

```
bne $s0, $s1, Label      #go to Label if $s0≠$s1
beq $s0, $s1, Label      #go to Label if $s0=$s1
```

- Machine Formats:

I format

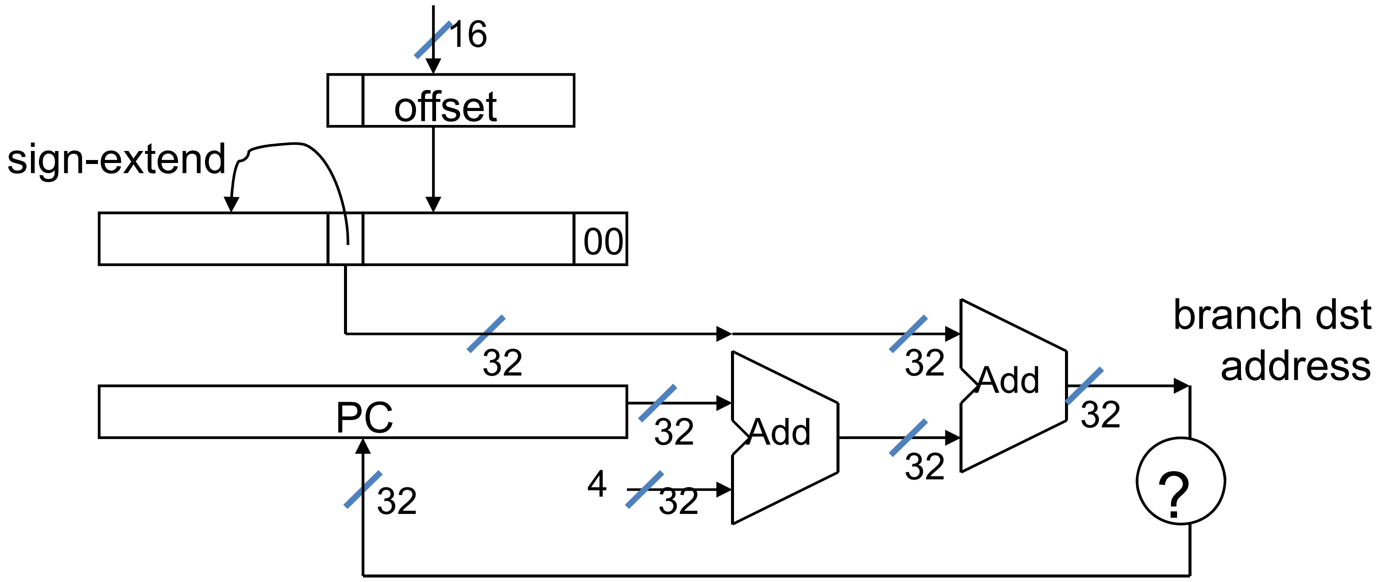
6 bits	5 bits	5 bits	
op	rs	rt	16 bit number
5	16	17	????
4	16	17	????

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register added to the 16-bit offset
  - which register? Instruction Address Register (the **PC**)
    - 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 $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

from the low order 16 bits of the branch instruction



More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

More Branch Instructions

- Can use **slt, beq, bne**, and the fixed value of 0 in register $zero to create other conditions

Example: less than `blt $s1, $s2, Label`

Solution:

```
slt  $at, $s1, $s2  #$at set to 1 if
bne  $at, $zero, Label   #$s1 < $s2
```

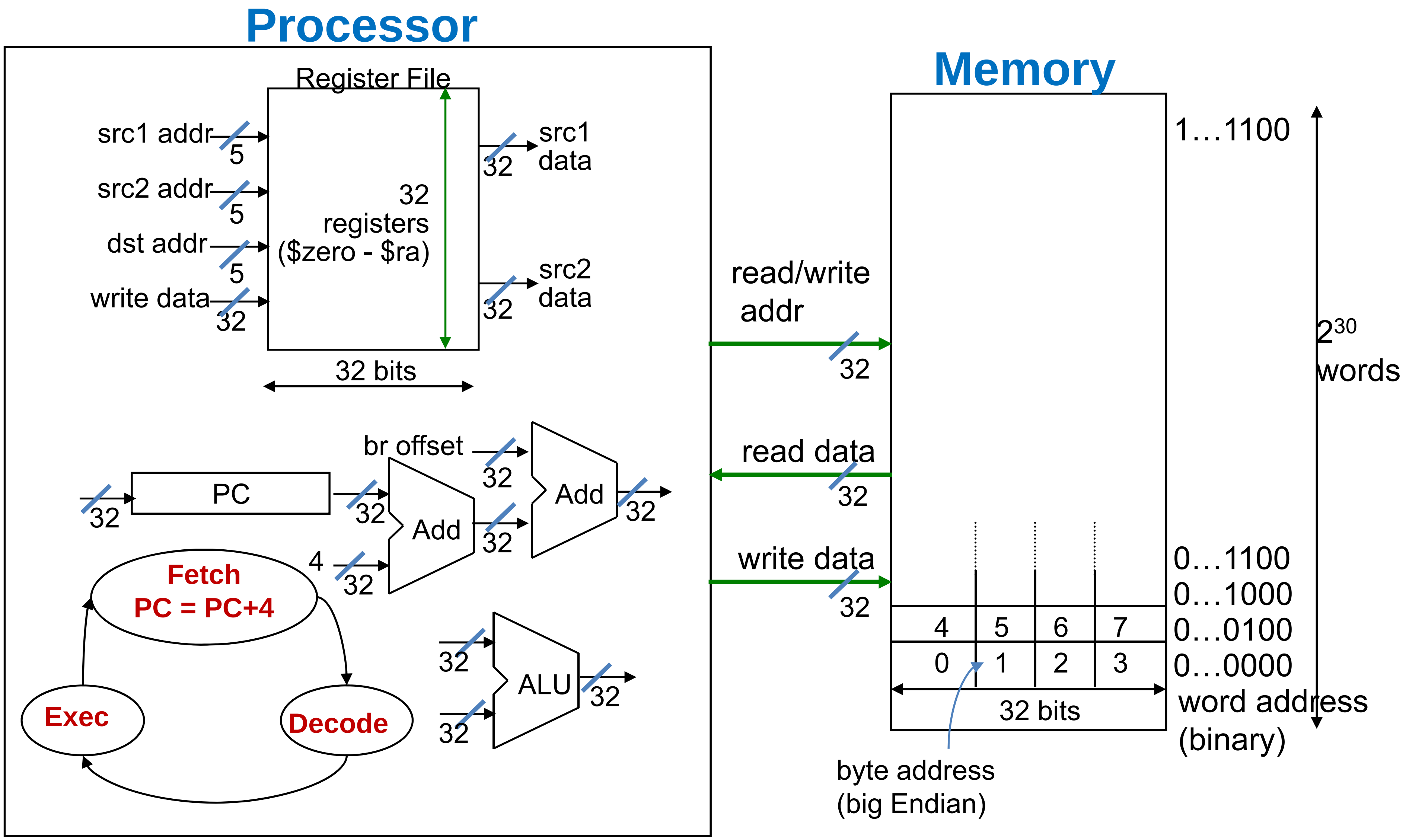
- Similarly, you can implement the followings:

  – 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`)

MIPS Organization



Processor

Register File

src1 addr 5

src2 addr 5

dst addr 5

write data 32

32 registers ($zero - $ra)

32 src1 data

32 src2 data

32 bits

br offset 32

PC 32

32 Add

4 32

32 Add 32

Fetch
PC = PC+4

Decode

Exec

32 32 ALU 32

Memory

read/write addr 32

read data 32

write data 32

4	5	6	7
0	1	2	3

32 bits

1…1100

0…1100

0…1000

0…0100

0…0000

2^{30} words

word address (binary)

byte address (big Endian)

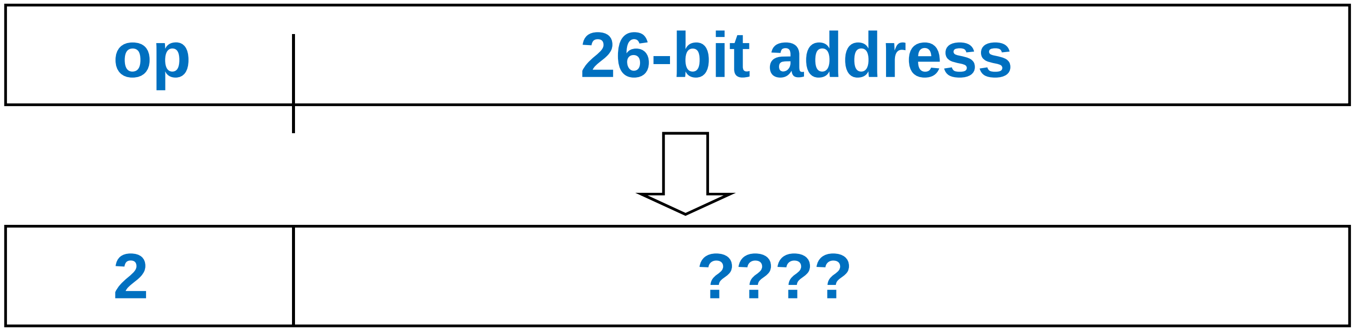
Assembling Jumps

- Instruction:

```
    j   label       #go to label
```

- Machine Format:

J format



op

26-bit address

2

????

- How is the jump destination address specified?

  As an absolute address formed by

  - concatenating the upper 4 bits of the current PC (now PC+4) to the 26-bit address and
  - concatenating 00 as the 2 low-order bits

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

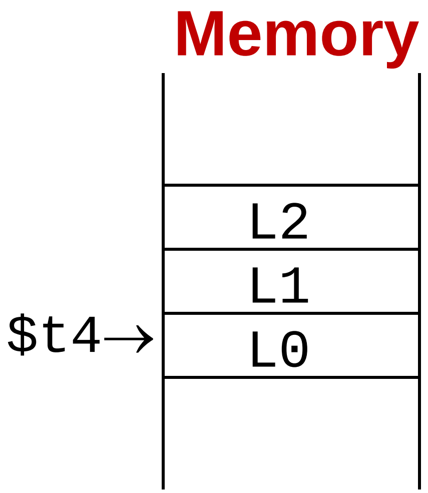
```
          beq  $s0, $s1, L1
```

becomes

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

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*/
```



- Assuming 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: . . .
```

Procedures

```
int leaf_example (int g, int h, int i, int j) {
    int   f;
    f = (g+h) – (i+j);
    return f;
}
```

CALLEE

```
void main(){
   int   f;
   f = leaf_example(1, 2, 3, 4);
   f ++;
}
```

CALLER

Six Steps in Execution of a Procedure

- ❑ Main routine (**caller**) places actual parameters in a place where the procedure (**callee**) can access them
  - $a0 - $a3: four argument registers
- ❑ **Caller** transfers control to the **callee**
- ❑ **Callee** acquires the storage resources needed
- ❑ **Callee** performs the desired task
- ❑ **Callee** places the result value in a place where the caller can access it
  - $v0 - $v1: two value registers for result values
- ❑ **Callee** returns control to the **caller**
  - $ra: one return address register to return to the point of origin

Instruction for Calling a Procedure

- MIPS **procedure call** instruction (caller):

```
jal    ProcedureAddress   #jump and link
```

Saves PC+4 in register $ra

Jump to address ProcedureAddress

- Then (**callee**) can do procedure return with just

```
jr $ra        #return
```

Register Usage

- $a0 – $a3: arguments (reg’s 4 – 7)
- $v0, $v1: result values (reg’s 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

MIPS Register Convention

Name	Register Number	Usage	Should preserve on call?
$zero	**0**	**the constant 0**	**no**
$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**
$gp	**28**	**global pointer**	**yes**
$sp	**29**	**stack pointer**	**yes**
$fp	**30**	**frame pointer**	**yes**
$ra	**31**	**return address**	**yes**

Procedure Call Instructions

- Procedure call: jump and link

  ```
  jal ProcedureLabel
  ```

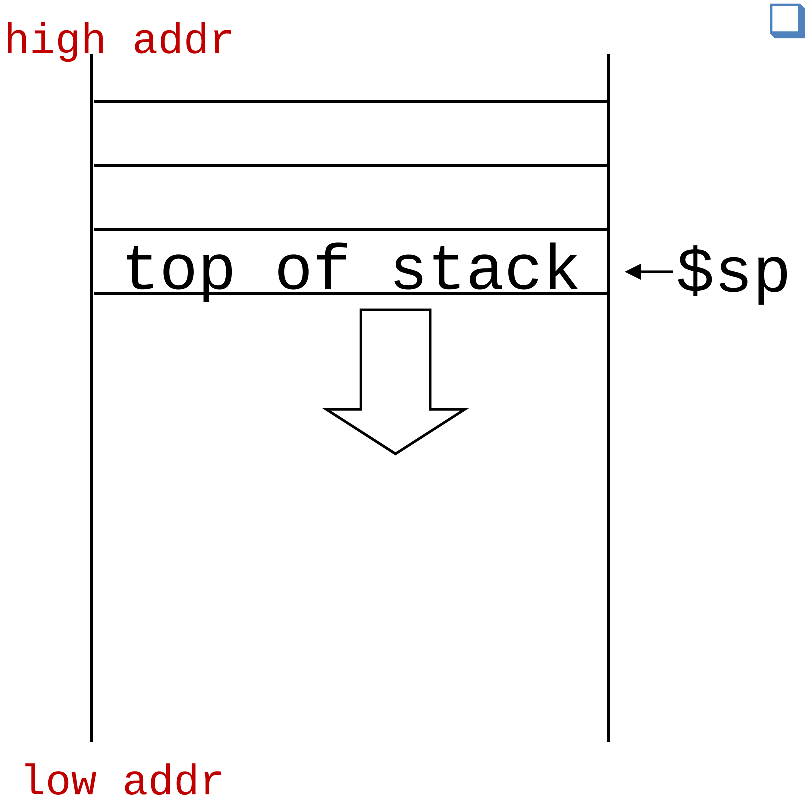
  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  ```
  jr $ra
  ```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

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`

Procedure Example

- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Procedure Example (Cont'd)

- MIPS code:

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

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); }
```

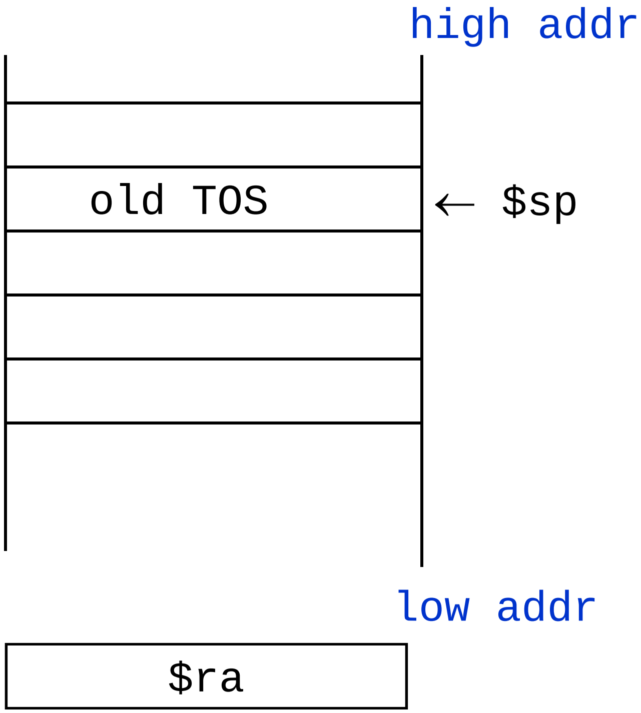
```
      caller:     jal   rt_1
next: . . .

      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: . . .
```

Saving the Return Address

- 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

Example: A Recursive Procedure

- Calculating factorial:

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

- 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
  . . .

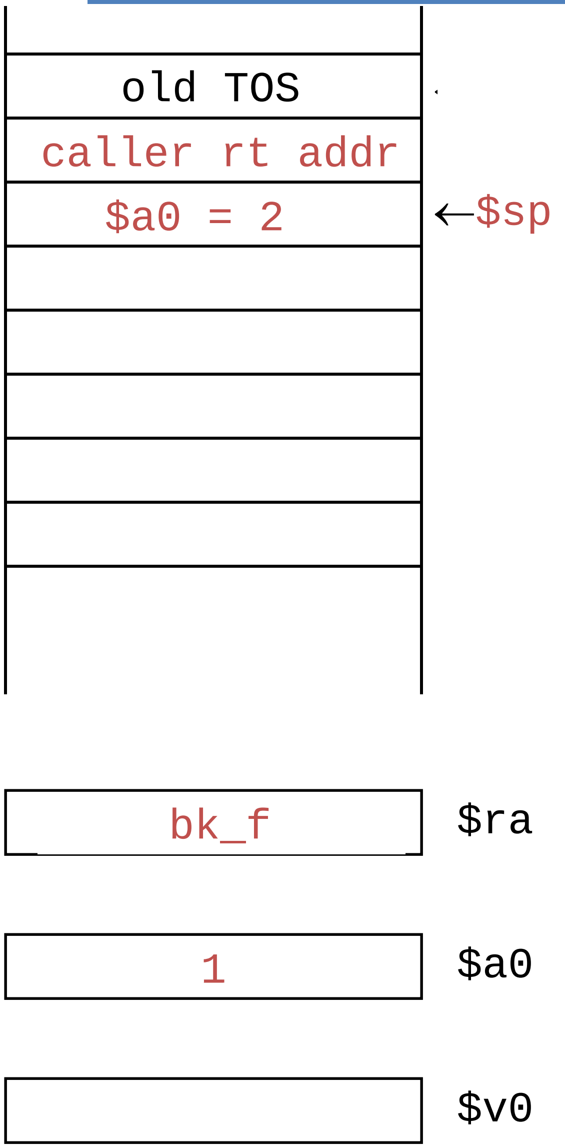
- Assume n is passed in `$a0`; result returned in `$v0`

Example: A Recursive Procedure (Cont’d)

```
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

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
```

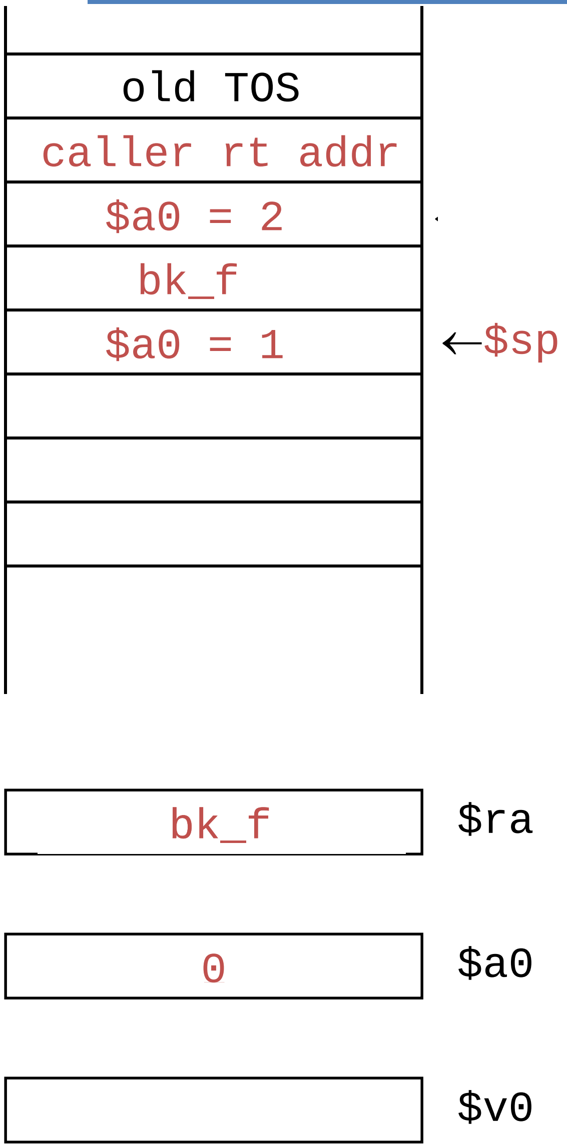
A Look at the Stack for $a0 = 2, Part 1



```
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

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 addr
      addi  $sp, $sp, 8      #adjust stack pointer
      mul   $v0, $a0, $v0    #$v0 = n * fact(n-1)
      jr    $ra             #return to caller
```

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



old TOS

caller rt addr

$a0 = 2

bk_f

$a0 = 1

←$sp

bk_f

$ra

0

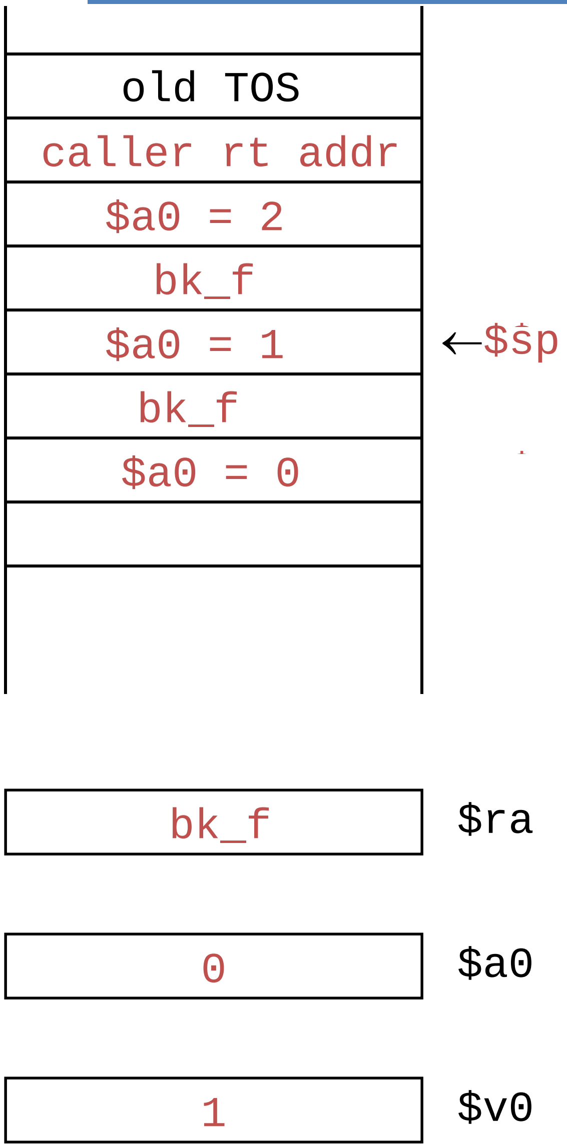
$a0

$v0

```
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

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 addr
      addi  $sp, $sp, 8       #adjust stack pointer
      mul   $v0, $a0, $v0    #$v0 = n * fact(n-1)
      jr    $ra              #return to caller
```

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



old TOS

caller rt addr

$a0 = 2

bk_f

$a0 = 1 ←$sp

bk_f

$a0 = 0

bk_f $ra

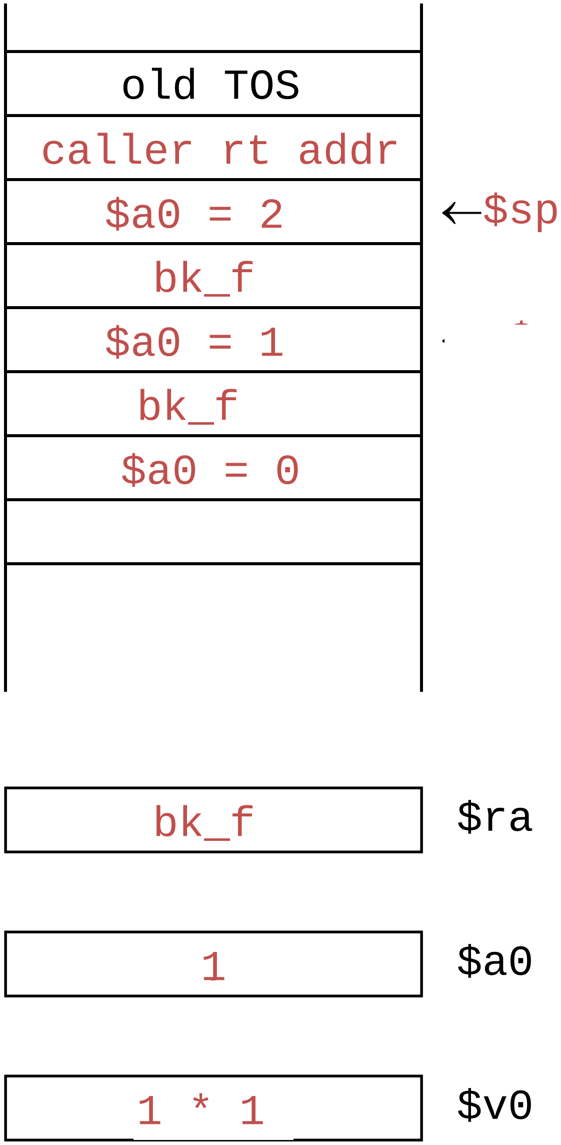
0 $a0

1 $v0

```
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

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 addr
      addi  $sp, $sp, 8      #adjust stack pointer
      mul   $v0, $a0, $v0   #$v0 = n * fact(n-1)
      jr    $ra             #return to caller
```

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



old TOS

caller rt addr

$a0 = 2

bk_f

$a0 = 1

bk_f

$a0 = 0

←$sp

bk_f

$ra

1

$a0

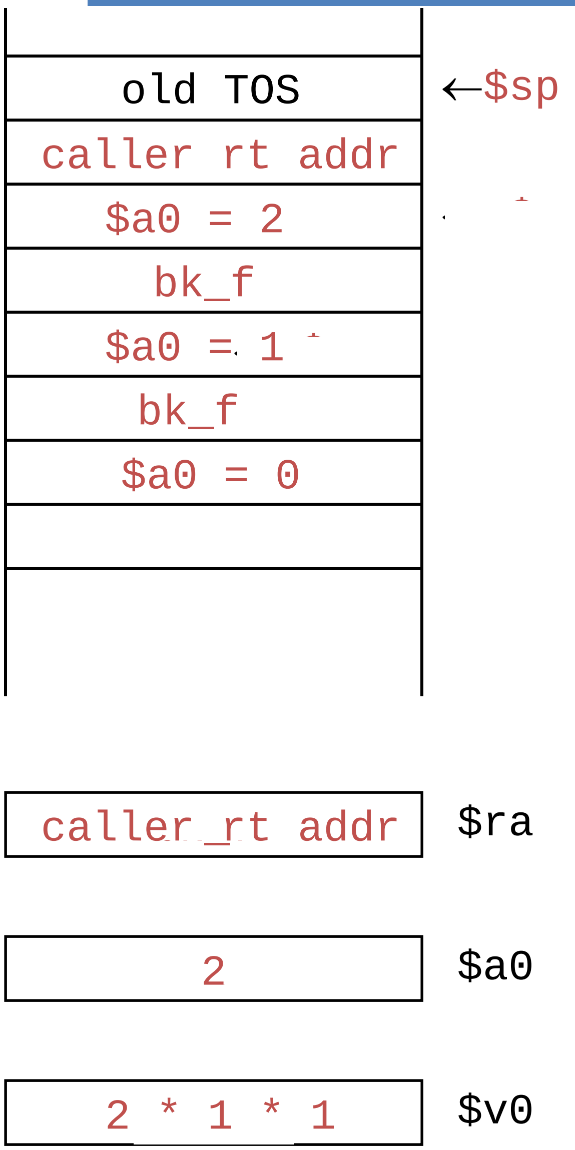
1 * 1

$v0

```
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

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 addr
      addi  $sp, $sp, 8     #adjust stack pointer
      mul   $v0, $a0, $v0   #$v0 = n * fact(n-1)
      jr    $ra            #return to caller
```

A Look at the Stack for $a0 = 2, Part 5



old TOS

←$sp

caller rt addr

$a0 = 2

bk_f

$a0 = 1

bk_f

$a0 = 0

caller_rt addr

$ra

2

$a0

2 * 1 * 1

$v0

```
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

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 addr
      addi  $sp, $sp, 8      #adjust stack pointer
      mul   $v0, $a0, $v0   #$v0 = n * fact(n-1)
      jr    $ra             #return to caller
```

Review: MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
Data transfer (I format)	load word	35	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	43	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	32	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	40	sb $s1, 101($s2)	Memory($s2+101) = $s1
Cond. Branch	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	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
Uncond. Jump	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4