C335
Computer Structures

MIPS Instructions (Part #6)

Dr. Liqiang Zhang

Department of Computer and Information Sciences

Adapted from Morgan Kaufmann and others

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 accept passed values (arguments) and to return values (results)

Six Steps in Execution of a Procedure

- Main routine (caller) places 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

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.
- Register conventions:
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`
- The stack is also used; more later.

Instruction Support for Functions (1/5)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

address
1000
1004
1008
1012
1016
2000
2004

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/5)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

address

```
1000 add  $a0,$s0,$zero  # x = a
1004 add  $a1,$s1,$zero  # y = b
1008 addi $ra,$zero,1016 #$ra=1016
1012 j    sum            #jump to sum
1016 ...

2000 sum: add $v0,$a0,$a1
2004 jr   $ra     # return to the caller
```

Instruction Support for Functions (3/5)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

- Question: Why use `jr` here? Why not simply use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra            # return to the caller
```

Instruction Support for Functions (4/5)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum                  #goto sum
```

- After:

```
1008 jal sum  # $ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls are very common. (Also, you don't have to know where the code is loaded into memory with `jal`.)

Instruction Support for Functions (5/5)

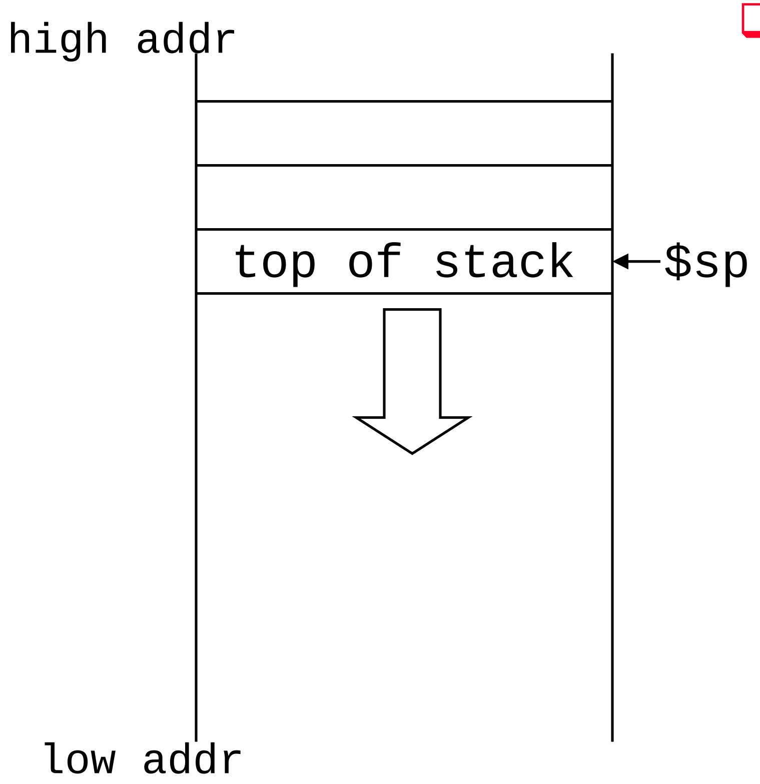
- Syntax for `jal` (jump and link) is same as for `j` (jump):

  ```
  jal  label
  ```

- `jal` should really be called `laj` for "link and jump":
  - Step 1 (link): Save address of *next* instruction into `$ra` (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label

Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - it 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
    store the data on stack at new $sp

  - remove data from the stack – pop

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

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     $s0,4($sp)    #save $s0 on stack
          sw     $s1,0($sp)    #save $s1 on stack
          add    $s0,$a0,$a1
          add    $s1,$a2,$a3
          sub    $s1,$s0,$s1
          add    $v0, $s1, $zero
          lw     $s1,0($sp)    #restore $s1
          lw     $s0,4($sp)    #restore $s0
          addi   $sp,$sp,8     #adjust stack ptr
          jr     $ra
```

Nested Procedures (1/2)

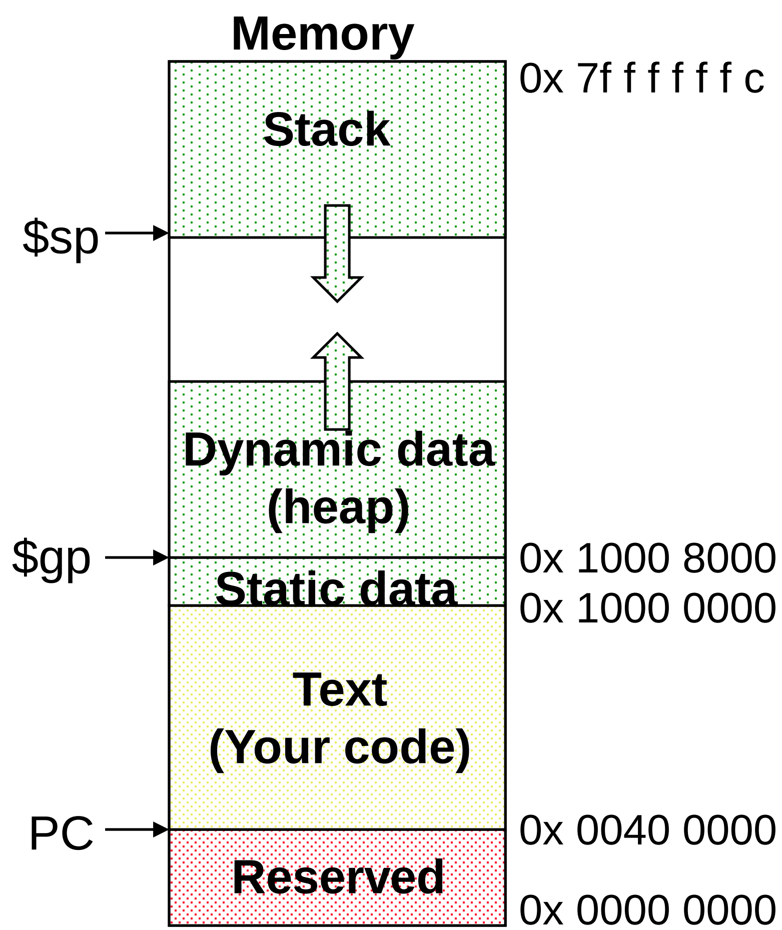
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there's a value in `$ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`.

Nested Procedures (2/2)

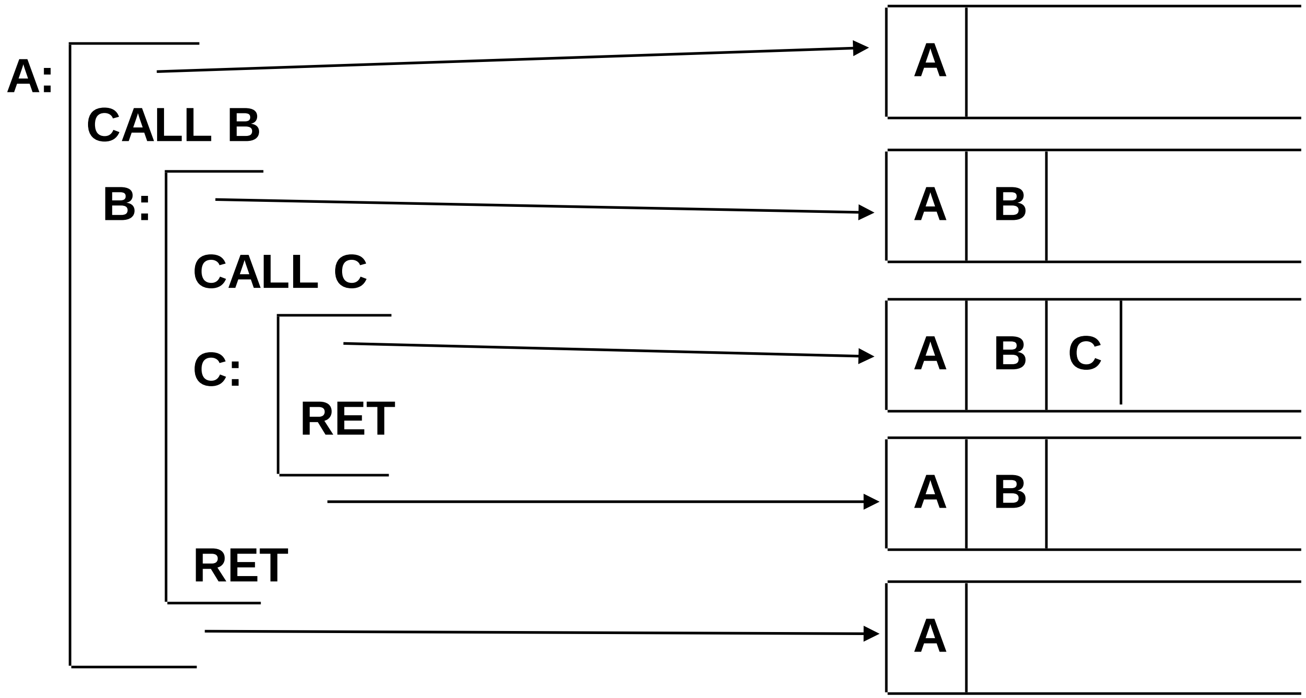
- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes. (C global variables).
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values

MIPS Memory Allocation for Program and Data



Procedure Call and Stack

Stacking of Subroutine Calls & Returns and Environments:



Using the Stack (1/2)

- So we have a register `$sp` which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
 return mult(x,x)+ y;
}
```

Using the Stack (2/2)

- Hand-compile

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

"push"

"pop"

```
sumSquare:
        addi $sp,$sp,-8       # space on stack
        sw $ra, 4($sp)        # save ret addr
        sw $a1, 0($sp)        # save y

        add $a1,$a0,$zero     # mult(x,x)
        jal mult              # call mult

        lw $a1, 0($sp)        # restore y
        add $v0,$v0,$a1       # mult()+y
        lw $ra, 4($sp)        # get ret addr
        addi $sp,$sp,8        # restore stack
        jr $ra
mult: ...
```

Steps for Making a Procedure Call

1) Save necessary values onto stack.

2) Assign argument(s), if any.

3) `jal` call

4) Restore values from stack.

Rules for Procedures

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions

  So what are they?

MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	the constant 0	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no*
$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*

MIPS Register Convention

- $at: may be used by the assembler at any time; unsafe to use
- $k0-$k1: may be used by the OS at any time; unsafe to use
- $gp, $fp: don’t worry about them
- Note: Feel free to read up on $gp and $fp in Appendix A, but you can write perfectly good MIPS code without them.

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

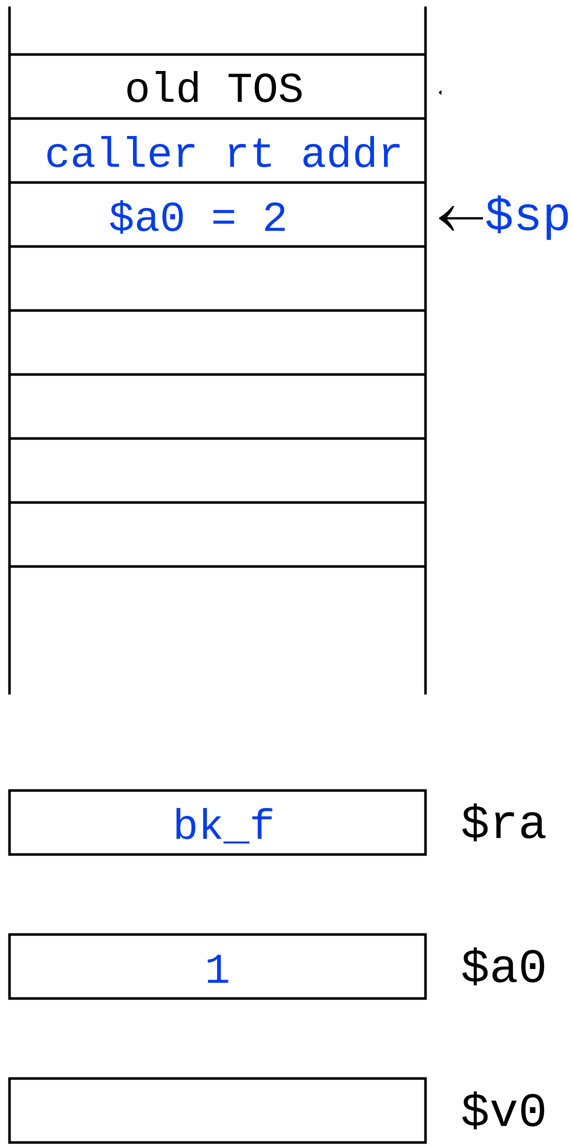
  . . .

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

Compiling a Recursive Procedure

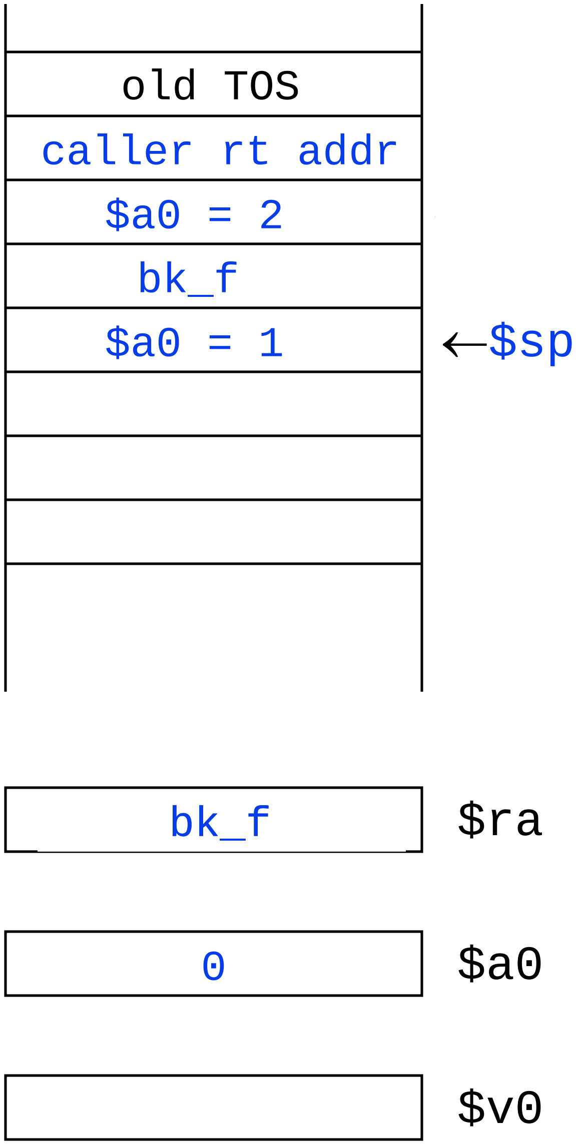
```
fact: addi  $sp, $sp, -8       #adjust stack pointer
      sw    $ra, 4($sp)        #save return address
      sw    $a0, 0($sp)        #save argument n
      slti  $t0, $a0, 1        #test for n < 1
      beq   $t0, $zero, L1     #if n >=1, go to L1
      addi  $v0, $zero, 1      #else return 1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

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



- Stack state after execution of first encounter with the `jal` instruction (*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 second encounter with the `jal` instruction (*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

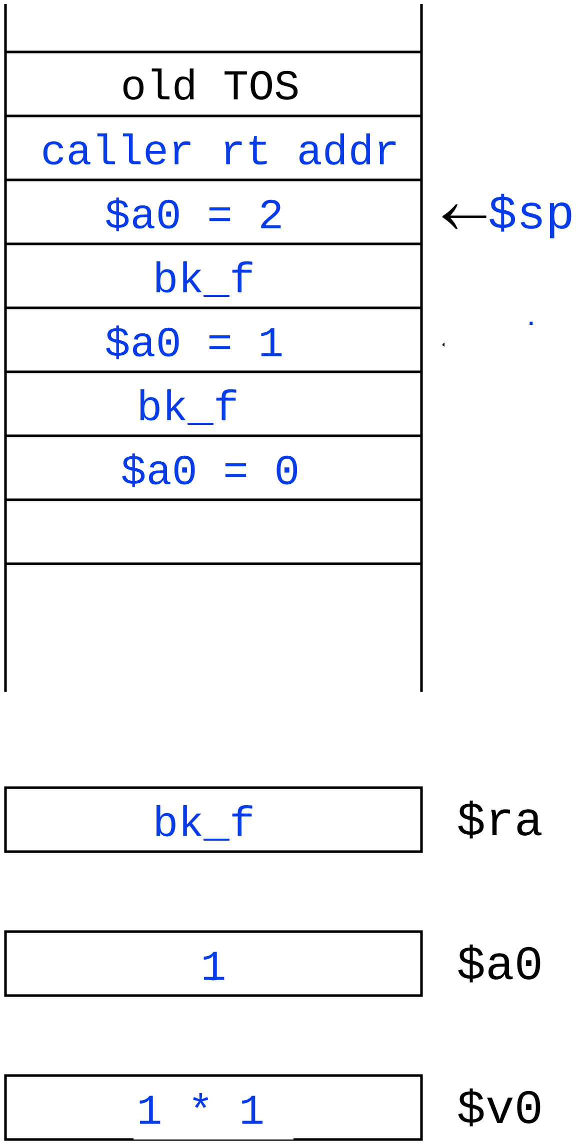
A Look at the Stack for $a0 = 2, Part 3

old TOS	
caller rt addr	
$a0 = 2	
bk_f	
$a0 = 1	←$sp
bk_f	
$a0 = 0	

bk_f	$ra
0	$a0
1	$v0

- Stack state after execution of first encounter with the first `jr` instruction (`$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 first encounter with the second `jr` instruction (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

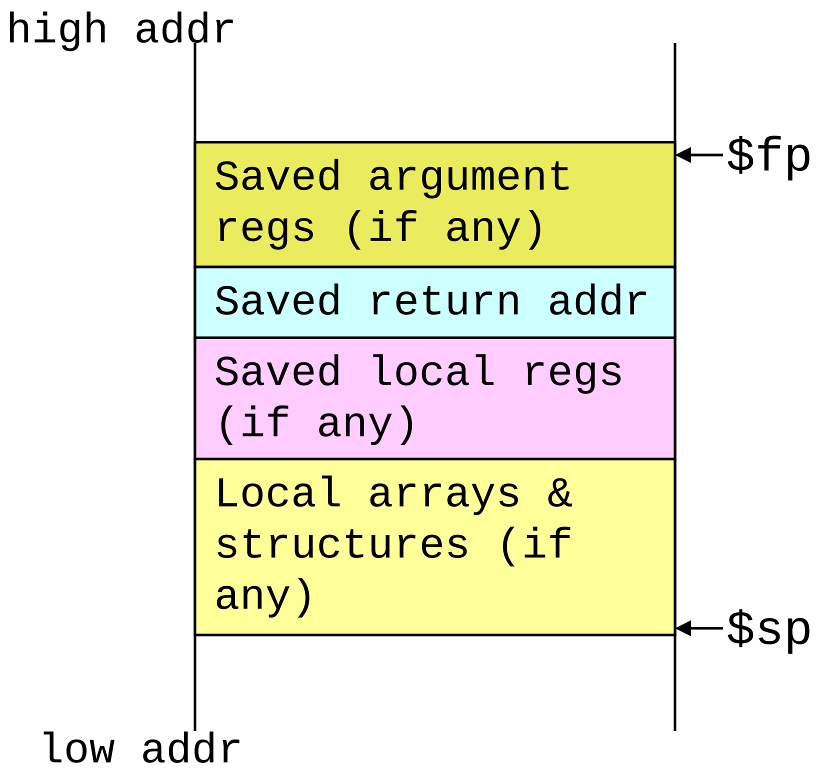
A Look at the Stack for $a0 = 2, Part 5

Stack	
old TOS	←$sp
caller rt addr	
$a0 = 2	
bk_f	
$a0 = 1	
bk_f	
$a0 = 0	

Value	Register
caller rt addr	$ra
2	$a0
2 * 1 * 1	$v0

- Stack state after execution of second encounter with the second `jr` instruction (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

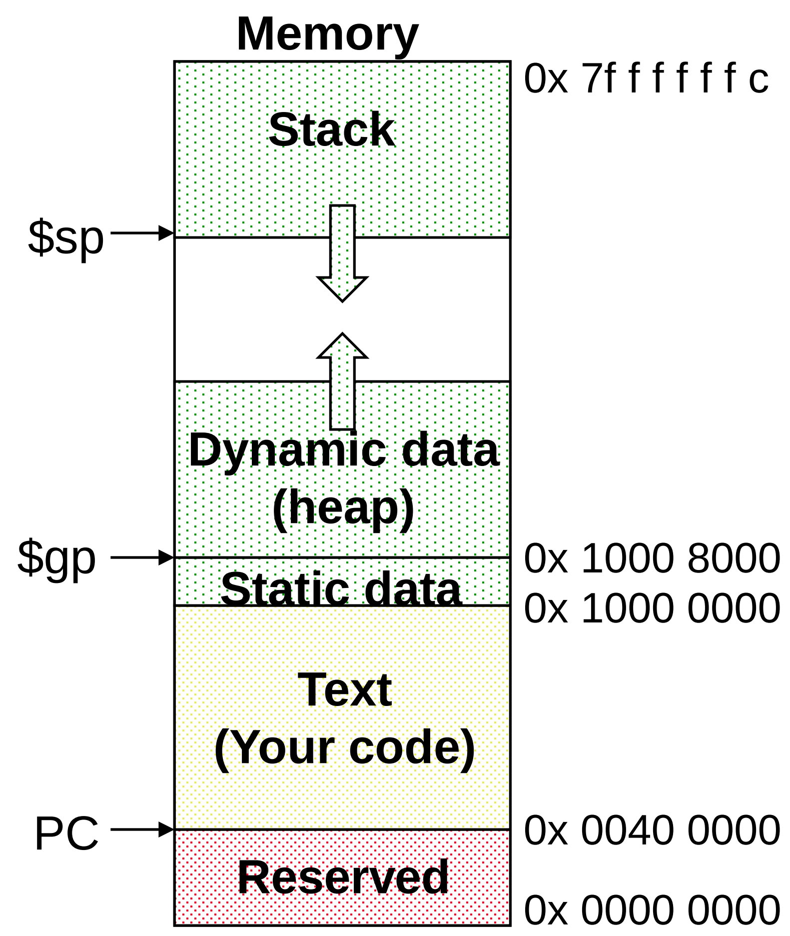
Allocating Space on the Stack



- 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

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()`
  - Or `new()/delete()`



MACROs

- The macro directive allows the programmer to write a named block of source statements, then use that name in the source file to represent the group of statements. During the assembly phase, the assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.
- Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if used repeatedly. Procedures or subroutines take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower.

MACROs

Advantages

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages

- In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

MACROs: example

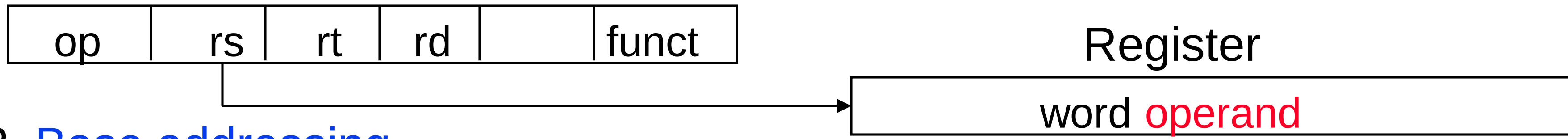
- See Textbook, Appendix A.2, page A-15 – A-17

MIPS Addressing Modes

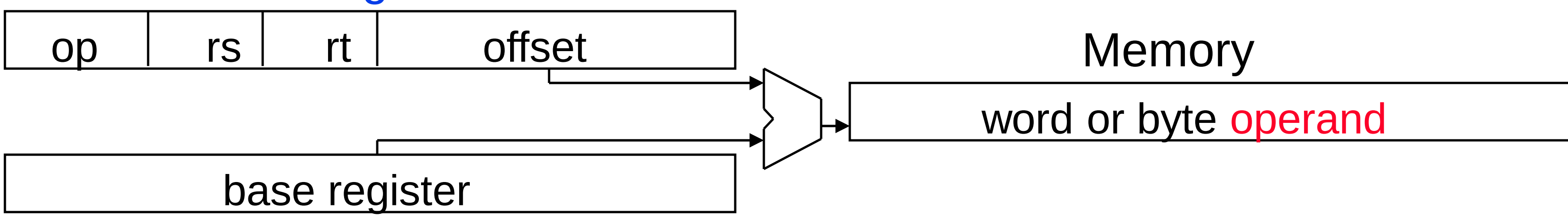
- Register addressing – operand is in a register
- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction
- Immediate addressing – operand is a 16-bit constant contained within the instruction
- PC-relative addressing –instruction address is the sum of the PC and a 16-bit constant contained within the instruction
- Pseudo-direct addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC

Addressing Modes Illustrated

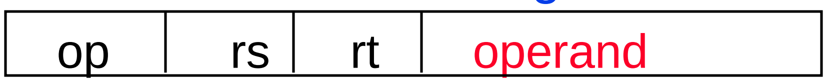
1. Register addressing



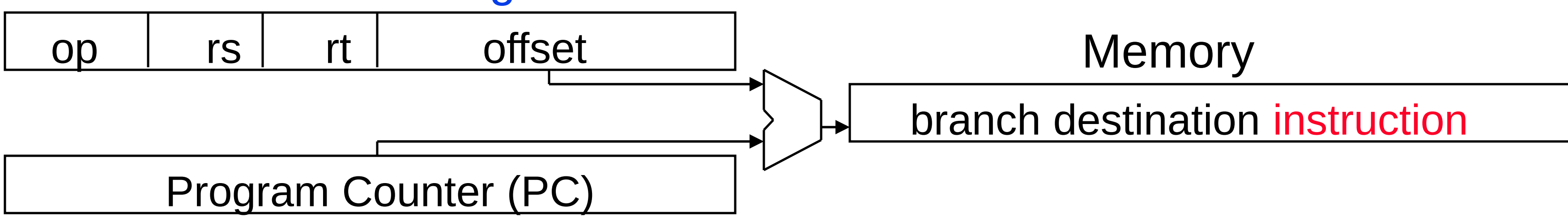
2. Base addressing



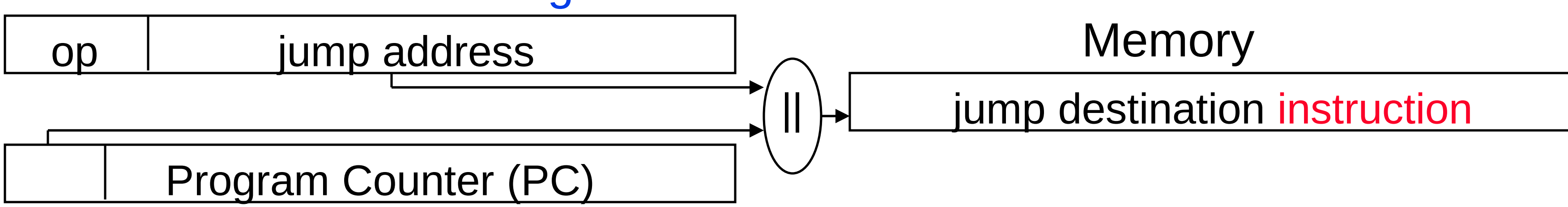
3. Immediate addressing



4. PC-relative addressing



5. Pseudo-direct addressing



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
	load upper immediate	f	lui $s1, 0xffff	$s1 = 0xffff0000

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
	load byte	20	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	28	sb $s1, 101($s2)	Memory($s2+101) = $s1
	load half	21	lh $s1, 101($s2)	$s1 = Memory($s2+102)
	store half	29	sh $s1, 101($s2)	Memory($s2+102) = $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

Registers

R0 - R31
PC
HI
LO

- 3 Instruction Formats: all 32 bits wide

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
OP	rs	rt	rd	shamt	funct	R format
OP	rs	rt	16 bit number			I format
OP	26 bit jump target					J format