MIPS Introduction

Reading: Chapter 3. Appendix A.

- Language of the Machine

- More primitive than higher level languages

  e.g., no sophisticated control flow such as for and while

  only simple branch, jump, and jump subroutine

- Very restrictive

  e.g., MIPS Arithmetic Instructions, two operands, one result

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

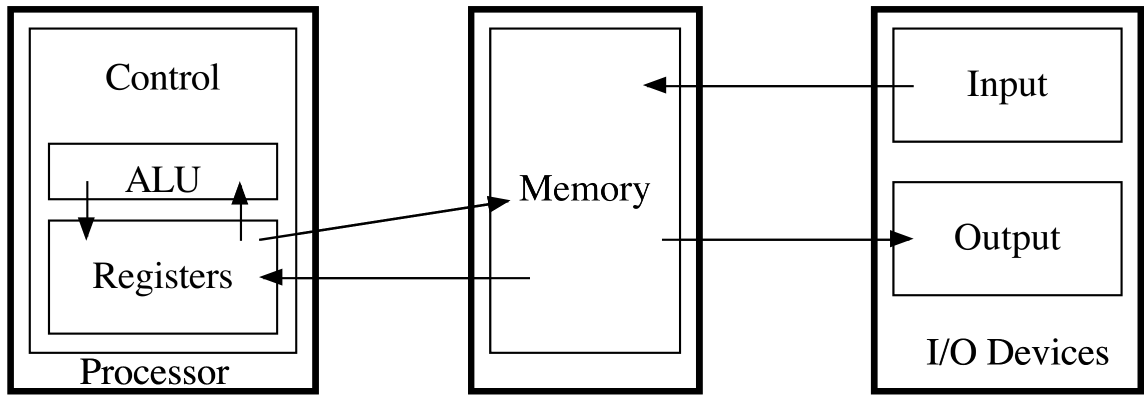
Design goals:

maximize performance and minimize cost,

reduce design time

Basic CPU Organization

- Simplified picture of a computer:



- Three components:
  - Processor (or Central Processing Unit or CPU); MIPS R2000 in our case
  - Memory — contains the program instructions to execute and the data for the program
  - I/O Devices — how the computer communicates to the outside world. Keyboard, mouse, monitor, printer, etc.
- CPU contains three components:
  - Registers — Hold data values for CPU
  - ALU — Arithmetic Logic Unit; performs arithmetic and logic functions. Takes values from and returns values to the registers
  - Control — Determines what operation to perform, directs data flow to/from memory, directs data flow between registers and ALU. Actions are determined by the current Instruction.

Memory Organization

- Viewed as a large, single-dimension array, with an address for each element — *byte* — of the array.
- A memory address is an *index* into the array
- “Byte addressing” — the index points to a byte, 8 bits in today’s computers, of memory.
- MIPS addresses 4 Gigabytes of memory:
  - Bytes are numbered from 0 to 2^{32} - 1, or 0 to 4,294,967,295
- Bytes are nice, but most data items use larger “words”

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
…	8 bits of data
4,294,967,293	8 bits of data
4,294,967,294	8 bits of data
4,294,967,295	8 bits of data

- For MIPS, a word is 32 bits or 4 bytes
  - Each register in the CPU holds 32 bits
  - Not just a coincidence!

- 2^{32} bytes with byte addresses from 0 to 2^{32} - 1

- 2^{30} words with byte addresses 0, 4, 8, …, 2^{32} - 4

- Words are “aligned”

  i.e., what are the least 2 significant bits of a word address in binary?

0	32 bits, 4 bytes, of data
4	32 bits, 4 bytes, of data
8	32 bits, 4 bytes, of data
12	32 bits, 4 bytes, of data
…	32 bits, 4 bytes, of data
4,294,967,284	32 bits, 4 bytes, of data
4,294,967,288	32 bits, 4 bytes, of data
4,294,967,292	32 bits, 4 bytes, of data

Notes:

2^{10} = 1024 = 1 Kilo

2^{20} = 1 Mega

2^{30} = 1 Giga

Registers vs. Memory

- Registers can be thought of as a type of memory.
- Registers are the "closest" memory to the CPU, since they are inside the CPU
- Principal advantages of registers vs. memory:
  - Fast access
  - Fast access
  - Fast access
- Principal advantages of memory vs. registers:
  - Lower cost
  - Lower cost
  - Lower cost
- An intermediate type of memory: Cache
  - Different "flavors" depending on size, physical location
  - Level 1 cache "closest" to the CPU
    - Usually installed on the chip as part of the CPU
    - Typically small: 32K, 64K
  - Level 2 cache between the CPU and the memory
    - Physically separate, but installed close to the CPU (i.e., "backside cache")
    - Typically a few Megabytes.
  - If you are curious, see Sections 7.2 and 7.3

Register Organization

Figure 3.13, page 140: (see also Appendix A, page A-23)

Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$at	1	reserved for assembler	n.a.
$v0-$v1	2-3	values for results & expression evaluation	no
$a0-$a3	4-7	arguments	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

These are the "General Registers". MIPS also has:

- PC (program counter) register and Status register
- Floating point registers

MIPS Arithmetic

- All arithmetic instructions have at most **3** operands

- All arithmetic is done in registers!
  - Can not, for example, add a number to a value stored in memory.
  - Must load value from memory into a register, then add the number to it.

- Operand order is fixed
  - Destination operand is first

Example:

C code:

```
A = B + C;
```

MIPS code:

```
add     $s0, $s1, $s2
```

adds contents of `$s1` and `$s2`, placing result in `$s0`.

MIPS Arithmetic (con't)

Design Principles:

1. Simplicity favors regularity.

2. Smaller is faster.

3. Good design demands good compromises.

4. Make the common case fast.

Why?

- Of course, this complicates some things…

  C code:

```
A = B + C + D;
E = F - A;
```

  MIPS code:

```
add $t0, $s1, $s2   # $t0 = $s1 + $s2, put result “temporarily” in $t0
add $s0, $t0, $s3   # $s0 = $t0 + $s3, now we can use the “temporary” result from $t0
sub $s4, $s5, $s0   # $s4 = $s5 - $s0
```

  note: use of $t0 to hold "temporary" result

- Operands must be registers — only 32 registers provided

Design Principles:

1. Simplicity favors regularity.
2. ***Smaller is faster.***
3. Good design demands good compromises.
4. Make the common case fast.

Why?

Clock cycle faster vs. More registers

- The amount of time it takes to get a value from a register into the ALU, or from the ALU into a register, is proportional to the exponent of 2. That is, the time for 32 registers is twice the time of 16 registers and 1/2 the time of 64 registers.

Registers vs. Memory

- Arithmetic instructions — operands must be registers
  - only 32 registers
- Compiler associates variables with registers

- What about programs with lots of variables?
  - Must move values between memory and registers

Load and Store Instructions

- Example:

  C or Java code, where z, w, and y are 4-byte, signed, two's-complement, integers:

```
z = w + y;
```

  MIPS code:

```
la      $t0, w            # put address of w into $t0
lw      $s0, 0($t0)       # put contents of w into $s0
la      $t1, y            # put address of y into $t1
lw      $s1, 0($t1)       # put contents of y into $s1
add     $s2, $s0, $s1     # add w + y, put result in $s2
la      $t2, z            # put address of z into $t2
sw      $s2, 0($t2)       # put contents of $s2 into z
```

- **`la`** = "load address" **`lw`** = "load word" **`sw`** = "store word"
- Assembly language allows us to use variable names to represent locations in memory.
  - Saves the hassle of computing the addresses ourselves!
- Must load address (**`la`**) to get the address of the memory location into a register
- **`0($t0)`** means "go 0 bytes from the address specified by **`$t0`**"

- Example:

  C or Java code:

  ```
  A[8] = h + A[8];
  ```

  MIPS code:

  ```
  la      $t3, A              # load address of start of array A
  lw      $t0, 32($t3)        # load contents of A[8]
  add     $t0, $s2, $t0
  sw      $t0, 32($t3)
  ```

  `32($t3)` means:

  `$t3` holds the starting address of the array

  Take the value stored in **`$t3`**, add **`32`** to it

  (32 = 4 bytes for one integer * position 8 in array)

  Use the sum of 32 and **`$t3`** as the memory address.

  Go to this location in memory.

  Get the contents of the 4 bytes (one word, lw = "load <u>word</u>") starting at that address.

  Put the contents into register **`$t0`**.

- Store word has the destination <u>last</u>
- Remember: Arithmetic operands are <u>registers</u>, ***not memory***!

Example

- Swap two adjacent values in the array **`v[]`**. **`k`** is the index of the first of the two adjacent values.

<pre>int k = 7;
int v[12] =
  {-87, 15, 13, ...,
   -6};
int temp;
 	
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;</pre>
See file **`swap.s`** from examples link on class web page.
	<pre>.data # a section of memory used for variables
	k: .word 7
	v: .word -87 # v[0]
	        .word 15 # v[1]
	        .word 13 # v[2]
	... # etc., until
	        .word -6 # v[11], last position in v
	.text # marks a section of assembly instructions
	swap:
	   la $s1, k # Put address of k in $s1
	   lw $s1, 0($s1) # Put contents of k in $s1
	   add $t0, $s1, $s1 # Start with $t0 = k + k
	   add $t0, $t0, $t0 # Now, $t0 = 4k = 2k + 2k
	   la $s0, v # Put address of v in $s0
	   add $t0, $s0, $t0 # Now, $t0 = v + 4k
	   lw $t1, 0($t0) # load contents of v[k]
	   lw $t2, 4($t0) # load contents of v[k+1] by
	                        # going 4 bytes beyond v[k]
	   sw $t2, 0($t0) # store contents of $t2 in v[k]
	   sw $t1, 4($t0) # store contents of $t1 in v[k+1]</pre>

Complete MIPS program for swap example:

```
.data   # a section of memory used for variables

k:
  .word   7
v:
  .word   -87  # v[0]
  .word   15   # v[1]
  .word   13   # v[2]
  .word   -7   # v[3]
  .word   27   # v[4]
  .word   41   # v[5]
  .word   42   # v[6]
  .word   43   # v[7]
  .word   -5   # v[8]
  .word   16   # v[9]
  .word   42   # v[10]
  .word   -6   # v[11], last position in v

.text # marks a section of assembly instructions

main:

  # Function prologue -- even main has one
  subu  $sp, $sp, 24  # allocate stack space --
                      # default of 24 here
  sw    $fp, 0($sp)   # save caller's frame pointer
  sw    $ra, 4($sp)   # save return address
  addiu $fp, $sp, 24  # setup main's frame pointer
```

```
swap:
  la    $s1, k        # Put address of k in $s1
  lw    $s1, 0($s1)   # Put contents of k in $s1
  add  $t0, $s1, $s1  # Start with $t0 = k + k
  add  $t0, $t0, $t0  # Now, $t0 = 4k = 2k + 2k
  la   $s0, v         # Put address of v in $s0
  add  $t0, $s0, $t0  # Now, $t0 = v + 4k
  lw   $t1, 0($t0)    # load contents of v[k]
  lw   $t2, 4($t0)    # load contents of v[k+1] by
                      # going 4 bytes beyond v[k]
  sw   $t2, 0($t0)    # store contents of $t2 in v[k]
  sw   $t1, 4($t0)    # store contents of $t1 in v[k+1]

done: # Epilogue for main -- restore stack & frame
      # pointers and return
  lw   $ra, 4($sp)  # get return address from stack
  lw   $fp, 0($sp)  # restore caller's frame pointer
  addiu $sp,$sp,24  # restore caller's stack pointer
  jr   $ra          # return to caller's code
```

So far we've learned:

- MIPS
  - loading *words* but addressing *bytes*
  - arithmetic performed on *registers only*

Instruction	English	Meaning
`add $s1, $s2, $s3`	"add"	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	"subtract"	`$s1 = $s2 - $s3`
`la  $t0, xray`	"load address"	`$t0 = address of label xray`
`lw  $s1, 24($s2)`	"load word"	`$s1 = Memory[$s2 + 24]`
`sw  $s1, 72($s2)`	"store word"	`Memory[$s2+72] = $s1`

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers: $t0 = 9, $s1 = 17, $s2 = 18
  - see Figure 3.13, page 140.
- Instruction Format:

op	rs	rt	rd	shamt	funct
000000	10001	10010	01001	00000	100000

Can you guess what the field names stand for?

- **`op`** — basic operation of the instruction, the *opcode*
- **`rs`** — first register source operand
- **`rt`** — second register source operand
- **`rd`** — register destination operand, it gets the result
- **`shamt`** — shift amount (not used until Chapter 4)
- **`funct`** — Function. Selects the specific variant of the opcode. (See Figure A.19, page A-54)

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?

Design Principles:

1. Simplicity favors regularity.
2. Smaller is faster.
3. ***Good design demands good compromises.***
4. Make the common case fast.

- Introduce a new type of instruction format
  - I-type for data transfer instructions

op	rs	rt	16 bit number
35	18	8	32

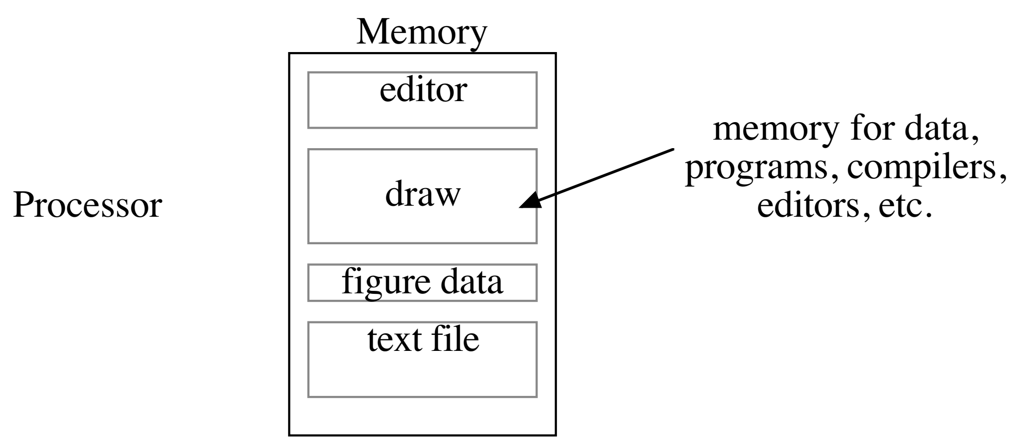
  - the previous format was R-type for register
- Example: **`lw $t0, 32($s2)`**

Where's the compromise?

```
Keep all instructions at 32 bits
Offset limited to 16-bits, ± 32K
```

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register — the *instruction register*
    - not one of the 32 general registers
  - Bits in this register “control” the subsequent actions
  - Fetch the “next” instruction and continue

Control

- Decision making instructions
  - alter the control flow
  - I.e, change the "next" instruction to be executed
- MIPS conditional branch instructions, two versions

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example:

```
if ( i == j )              bne $s0, $s1, Label
   h = i + j;              add $s3, $s0, $s1
                    Label: ...
```

Note the reversal of the condition from equality to inequality!

beq	op = 4	rs	rt	16 bit offset
bne	op = 5	rs	rt	16 bit offset

Program Flow Control

- MIPS unconditional branch instructions:

  `j  label`

op	26 bit address

- Example:

```
if ( i != j )                      beq $s4, $s5, Lab1  # compare i, j
   h = i + j;                      add $s3, $s4, $s5   # h = i + j
                                   j Lab2              # skip false part
else                         Lab1:
   h = i - j;                      sub $s3, $s4, $s5   # h = i - j
                             Lab2:
k = h + i;                         add $s6, $s3, $4    # k = h + i
```

Set Less Than

- New instruction that will compare two values and put a result in a register.
- Comparison is less than

```
slt     $t0, $t1, $t2
```

- First register listed is the destination — `$t0` in this case
- Second and third registers are compared with less than: `$t1 < $t2`
- Result is 1 if comparison is true
- Result is 0 if comparison is false

op	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	101010

For Loop Example

- C code:

```
sum = 0;
for (i = 0; i < y; i++)
   sum = sum + x;
```

- MIPS:

  Assume: **x**, **y**, and **sum** are in **$s0**, **$s1**, and **$s2** respectively.

  Will use **$t0** for **i** and **$t1** for the constant **1**.

```
        add     $s2, $zero, $zero    # sum = 0
        add     $t0, $zero, $zero    # i = 0
LoopBegin:
        slt     $t2, $t0, $s1
        beq     $t2, $zero, LoopEnd  # is i < y  ??
        add     $s2, $s2, $s0        # sum = sum + x
        add     $t0, $t0, $t1        # i++
        j       LoopBegin
LoopEnd:
```

Complete MIPS program for loop example. Available as for1.s on examples link from the class web page

```
.data

x:        .word  42
y:        .word   8
sum:      .word   0

one:      .word   1

answer:   .asciiz  "The sum is "
newline:  .asciiz "\n"

.text

main:
  # Function prologue -- even main has one
  subu  $sp, $sp, 24  # allocate stack space --
                      # default of 24 here
  sw    $fp, 0($sp)   # save caller's frame pointer
  sw    $ra, 4($sp)   # save return address
  addiu $fp, $sp, 24  # setup main's frame pointer

  # Put x into $s0
  la $t0, x
  lw $s0, 0($t0)

  # Put y into $s1
  la $t0, y
  lw $s1, 0($t0)

  # Put the constant 1 into $t1
  la $t0, one
  lw $t1, 0($t0)

  add $s2, $zero, $zero    # sum = 0
  add $t0, $zero, $zero    # i = 0

LoopBegin:
   slt $t2, $t0, $s1        # $t2 = (i < y)
   # branch out of loop if (i == y)
   beq $t2, $zero, LoopEnd
   add $s2, $s2, $s0        # sum = sum + x
   add $t0, $t0, $t1        # i++
   j   LoopBegin

LoopEnd:
   # Print message
   la $a0, answer
   li $v0, 4
   syscall

   # Print the sum
   add $a0, $s2, $zero
   li $v0, 1
   syscall

   # Print newline
   la $a0, newline
   li $v0, 4
   syscall

done:  # Epilogue for main -- restore stack & frame
       # pointers and return
  lw   $ra, 4($sp)   # get return address from stack
  lw   $fp, 0($sp)   # restore caller's frame pointer
  addiu $sp,$sp,24 # restore caller's stack pointer
  jr    $ra        # return to caller's code
```

While Loop

- C while loop (from example on page 127 in the textbook)

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

- MIPS version:

```
.data

save:   .word  42
        .word  42
        .word  42
        .word  42
        .word  42
        .word  42
        .word  42
        .word  93
        .word  -2
k:      .word  42
i:      .word  3
jj:     .word  2      # can’t use ‘j’ for a variable since ‘j’ is “jump”
str:    .asciiz "The final value of i = "
newline:.asciiz "\n"


.text
```

```
main:
        # Function prologue -- even main has one
        subu  $sp, $sp, 24    # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        addiu $fp, $sp, 24    # setup main's frame pointer

        la    $s6, save     # $s6 = address of save[0], beginning of array
        la    $t0, i
        lw    $s3, 0($t0)   # $s3 = value of i
        la    $t0, jj
        lw    $s4, 0($t0)   # $s4 = value of jj
        la    $t0, k
        lw    $s5, 0($t0)   # $s5 = value of k

LoopBegin:
     # Loop Test
        add   $t1,$s3,$s3   # quadruple i to get offset for save[i]
        add   $t1,$t1,$t1
        add   $t1,$t1,$s6   # compute address of save[i]
        lw    $t0, 0($t1)   # $t0 = value stored at save[i]
        bne   $t0,$s5,LoopEnd # end loop if save[i] != k

     # Loop body
        add   $s3,$s3,$s4   # i = i + jj
        j     LoopBegin
```

```
LoopEnd:
        sw    $s3,i          # store value of i into memory

        la    $a0,str        # $a0 = address of start of string
        li    $v0,4
        syscall

        add   $a0,$s3,$zero # $a0 = value of i
        li    $v0,1
        syscall

        la    $a0,newline   # $a0 = address of newline string
        li    $v0,4
        syscall

done:    # Epilogue for main -- restore stack & frame pointers and return
         lw    $ra, 4($sp)     # get return address from stack
         lw    $fp, 0($sp)     # restore the caller's frame pointer
         addiu $sp, $sp, 24    # restore the caller's stack pointer
         jr    $ra             # return to caller's code
```

So far:

Instruction	Meaning
add $s1, $s2, $s3	$s1 = $s2 + $s3
sub $s1, $s2, $s3	$s1 = $s2 - $s3
lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]
sw $s1, 100($s2)	Memory[$s2 + 100] = $s1
slt $s2, $t0, $t1	$s2 = $t0 < $t1, put 1 in $s2 if true, else 0
bne $s4, $s5, Label	Next instruction is at Label if $s4 ≠ $s5
beq $s4, $s5, Label	Next instruction is at Label is $s4 = $s5
j Label	Next instruction is at Label

- Formats

R	op	26 bit address				
I	op	rs	rt	rd	shamt	funct
J	op	rs	rt	16 bit number		

Control Flow

- We have: beq, bne, what about Branch-if-less-than (and other options)?
- New instruction, Set if less than:

`slt $t0, $s1, $s2`	`if $s1 < $s2 then`
	`$t0 = 1`
	`else`
	`$t0 = 0`

- Can use this instruction to build “branch if less than”

```
slt     $t0, $s1, $s2        # set $t0 to 1 if $s1 < $s2
bne     $t0, $zero, ToHere   # branch if less than
```

- Can use this instruction to build “branch if greater than”

```
slt     $t0, $s2, $s1        # set $t0 to 1 if $s1 > $s2
bne     $t0, $zero, ToHere   # branch if greater than
```

- Can build other control structures in a similar fashion
- Use of two instructions is faster than a single instruction, in this case,
  - given the complexity that would be required for blt, bgt, etc.
  - would increase CPI for branches and/or lower clock speed

Constant or Immediate Operands

- Section 3.8, pages 145-147
- Many times, one operand of an arithmetic instruction is a small constant
  - 50% or more in some programs
- Possible solutions:
  - put typical constants in memory and load them when needed
  - hard-wire registers to hold common values, i.e., **`$zero`**

Design Principles:

1. Simplicity favors regularity.
2. Smaller is faster.
3. Good design demands good compromises.
4. ***Make the common case fast.***

- MIPS Solution:
  - Add “a few” opcodes that allow one operand to be stored in the instruction:
    - addi
    - slti
    - li — a *pseudoinstruction*

- Example:

```
addi    $t1, $t1, 4
```

op	rs	rt	16 bit immediate operand
8	9	9	4

Note: There is NOT **`subi, muli`**

16 bits = 2^{16} = 64K, but have to allow for negative constants, so range is limited to ±32K

How to handle large constants, those needing more than 16 bits?

Requires a two-step process:

to put 0000 0001 0011 1110 0000 1010 0000 0011_{two} = $0x013E0A03_{hex}$ into $s0

```
lui     $s0, 0x013E          # put values into upper 16-bits of $s0
addi    $s0, $s0, 0x0A03     # add the lower 16-bits to $s0
```

Thus far, Sections 3.1 through 3.5, pages 106—131:

Name	Example	Comments
32 registers	`$s0, $s1, …, $s7,` `$t0, $t1, …, $t7,` `$zero`	Fast locations for data. Data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0.
2^{30} memory words	`Memory[0],` `Memory[4], …,` `Memory[4294967292]`	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures (arrays, spilled registers, etc.)

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3e`	`$s1 = $s2 + $s3`	3 operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	3 operands; data in registers
Data transfer	load word	`lw  $s1, 48($s2)`	`$s1 = Memory[$s2+48]`	Data fm memory to register
	store word	`sw  $s1, 52($s2)`	`Memory[$s2+52] = $s1`	Data fm register to memory
Conditional branch	branch on equal	`beq $s1, $s2, L`	`if ($s1==$s2) goto L`	Equal test and branch
	branch on not equal	`bne $s1, $s2, L`	`if ($s1!=$s2) goto L`	Not equal test and branch
	set on less than	`slt $s1, $s2, $s3`	`if ($s2 < $s3) $s1 = 1; else $s1 = 0`	Compare less than; used with `beq`, `bne`
Unconditional jump	jump	`j   2500`	`goto 10000`	Jump to target address

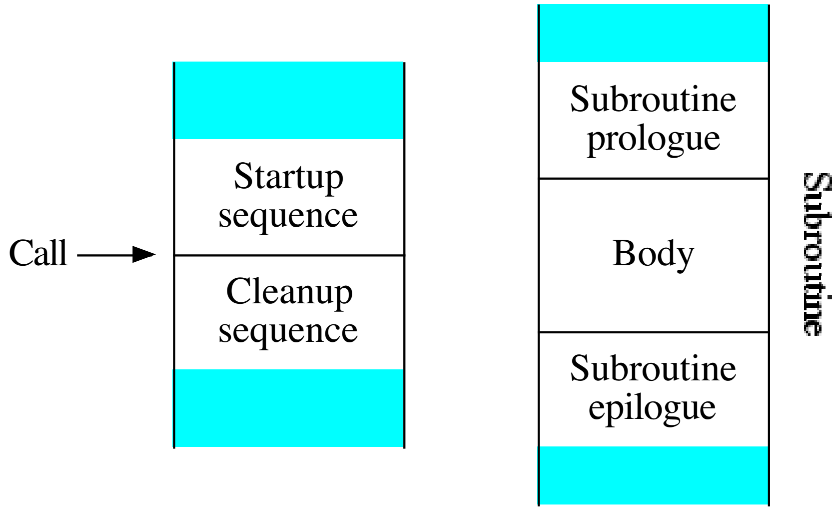
Name	Format	Example						Comments
`add`	R	0	18	19	17	0	32	`add $s1,$s2,$s3`
`sub`	R	0	18	19	17	0	34	`sub $s1,$s2,$s3`
`lw`	I	35	18	17	48			`lw $s1,48($s2)`
`sw`	I	43	18	17	52			`sw $s1,52($s2)`
`beq`	I	4	17	18	25			`beq $s1,$s2,100`
`bne`	I	5	17	18	25			`bne $s1,$s2,100`
`slt`	R	0	18	19	17	0	42	`slt $s1,$s2,$s3`
`j`	J	2	2500					`j 10000 (p. 150)`
`Field size`		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
`R-format`	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
`I-format`	I	op	rs	rt	address			Data transfer, branch format

Procedures in MIPS

Reading: Section 3.6, pages 132 to 141, and Section A.6, pages A-22 to A-32.

- Overview
  - Structure programs:
    - make them easier to understand, and
    - make code segments easier to re-use
- Problems:
  - Want to call the procedure from anywhere in the code
  - Want to pass arguments to the subroutine that may be different each time the procedure is called
  - Want the procedure to return to the point from which it was called
  - (May) want the procedure to return a value (technically, such a "procedure" is actually a "function")
- Issues in implementing subroutines:
  - How does the subroutine return to the caller's location?
  - Where/how is the result returned?
  - Where are the parameter(s) passed?
  - Where are the registers used (i.e., overwritten) by the subroutine saved?
  - Where does the subroutine store its local variables?
- Issues must be agreed upon by both the caller and callee in order to work.
- Termed the *calling conventions*. Not enforced by hardware but expected to be followed by all programs.
- Information shared between caller and callee also termed the *subroutine linkage*.

Calling Subroutines



- The caller establishes part of the subroutine linkage in the *startup sequence*.
- The callee establishes the remainder of the linkage in the *subroutine prologue*
- The *subroutine epilogue* contains instructions that return to the caller.
- The *cleanup sequence* contains instructions to clean up the linkage.

Indicating the Return Address

- The calling convention describes the allocation, construction and deallocation of a subroutine linkage.
- Perhaps the most simple calling convention stores the return address in a register
  - In MIPS, this is $ra, register $31.
- And then provides an instruction that can jump to the address contained in a register
  - In MIPS, this is the jr (jump register) instruction:

```
        jr      $ra
```

- Example: We could do a simple subroutine with only the MIPS instructions we've learned:

```
# Startup sequence:
        la      $ra, ReturnHere         # Put return address in $ra
        j       SubBegin                # Jump to beginning of subroutine
ReturnHere:
        # ... code that follows subroutine call
        # Cleanup sequence
        # None needed this time...
        ...     # continue w/ code following subroutine call

        # can do it again...
        la      $ra, ComeBackHere
        j       SubBegin
ComeBackHere:
        ... more MIPS code here ...
```

```
# Somewhere else in .text segment:
SubBegin:
        # Subroutine prologue:
        # no prologue needed this time...

        # Subroutine body goes here...

        # Subroutine epilogue:
        jr      $ra                   # jump to address stored in register $ra
```

Using the JAL (Jump and Link) Instruction

- To support subroutines, machines provide an instruction that stores the return address and jumps to the start of the subroutine
  - Also called JSR (Jump to Subroutine) and BL (Branch and Link).
- Example: use the JAL instruction to implement a subroutine call:

```
Startup sequence:
                                      # other instructions
        jal     SubBegin              # store return addr & jump to beginning of subroutine
```

- Do not need to specify which register to use; **`jal`** will always put return address in **`$ra`**

Registers and Parameters

- The registers must be considered as global memory locations among the different subroutines.
- Someone needs to insure after the subroutine returns, that registers contain the old values that they had before it was called.
- Multiple possible approaches:
  - Before every subroutine call, the caller saves all the registers that it will need (regardless of the ones used by the callee), and restores them after the subroutine returns, or
  - The callee saves (in its prologue) the registers that it will use in its body, and restores all of them in its epilogue (regardless of the ones used by its caller).
- MIPS: A compromise: Divide registers between those saved by caller (*t* registers) and those saved by callee (*s* registers).

- Done by the Caller

  Startup sequence:

  - Save the *t* registers used by the caller
  - Save the arguments sent to subroutine
  - Store return address and jump to subroutine (**`jal`**)

  Cleanup sequence:

  - Restore the *t* registers used by the caller

- Done by the Callee

  Subroutine prologue:

  - Save the *s* registers used in the subroutine body
  - Save the return address (**`$ra`**), if necessary

  Subroutine epilogue:

  - Restore the *s* registers saved in the prologue
  - Restore value of **`$ra`**, if necessary
  - Return

Where does all this go? On the Stack, of course :-)

Stack and Frame Pointers

- The MIPS calling conventions dictate that “t” registers are saved by the caller and “s” registers by the callee.
  - Both caller and callee use the stack to save these.
- The stack pointer, SP, in MIPS is register **`$sp`** (**`$r29`**).
- The frame pointer, FP, is **`$fp`** (**`$r30`**)
  - points to the word after the last word (highest address) of the frame.

Passing parameters

- In general, the parameters to a subroutine are put on the stack by the caller, and loaded from there by the subroutine.
- Note: if the caller has a parameter in a register it must store it to the stack, then the subroutine must load it from the stack to get it back in a register.
- MIPS optimizes this by passing the first four parameters in the registers **`$a0`** - **`$a3`**; the remainder are passed on the stack.
- Space must be reserved for *all* parameters (including those in **`$a0 - $a3`**) on the stack in case the callee wants to store them to memory before making calls of its own.

Putting It All Together

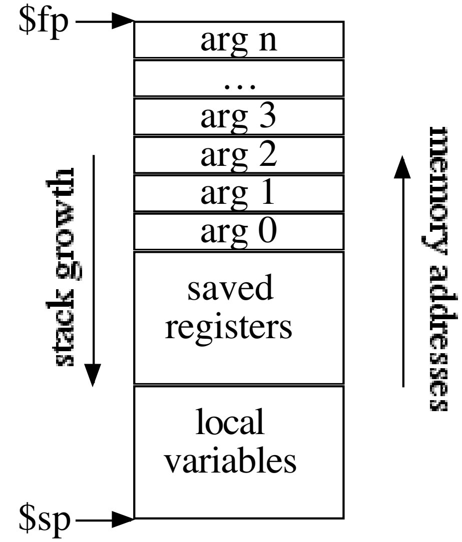
- Recall the four major steps in calling a subroutine:
  - Caller executes *startup* code to set things up for the subroutine and invokes the subroutine.
  - Subroutine executes *prologue* code to manage the stack frame.
  - Subroutine executes *epilogue* code prior to returning to undo the stack frame, then returns to the caller.
  - Caller executes *cleanup* code to clean up after the call.

Startup

- Save the caller-saved registers into the "saved registers" area of the current stack frame.
  - **`$t0`** - **`$t9`** registers that will be needed after the call.
  - The **`$a0`** - **`$a3`** registers, and
  - Any additional arguments being passed to the subroutine beyond the first four
- Pass the arguments to the subroutine.
  - The first four are in registers **`$a0`** - **`$a3`**, the rest are put on the stack starting with the last argument first.
  - Arguments are stored by the caller at negative offsets from the stack pointer.
- Use the **`jal`** instruction to jump to the subroutine.

Prologue

- Allocate a stack frame by subtracting the frame size from the stack pointer. Once set-up, a function’s stack will be:



    - The stack pointer must always be double-word aligned, so round the frame size to a multiple of 8.
    - The minimum frame size is 24 bytes (space for **`$a0`** - **`$a3`**, **`$fp`**, and **`$ra`**) and is always the minimum that must be allocated.

- Save the callee-saved registers into the frame, including **`$fp`**. Save **`$ra`** if the subroutine might call another subroutine, and save any of **`$s0`** - **`$s7`** that are used.
- Set the frame pointer to **`$sp`** plus the frame size.

Epilogue

- Restore any registers that were saved in the prologue, including **`$fp`**.
- Pop the stack frame by adding the frame size to **`$sp`**.
- Return by jumping to the address in **`$ra`**.

Function Call Example 1

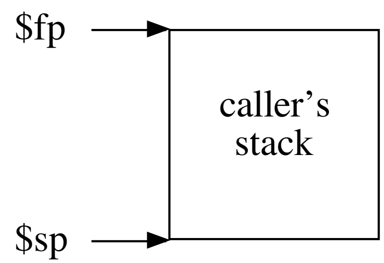
- Want to call a function named **`zap1`** that takes one int as an argument, and returns an **`int`** as its result:

```
int zap1( int x )
```

- Want to call the function **`zap1`** with **`x`** as 15:

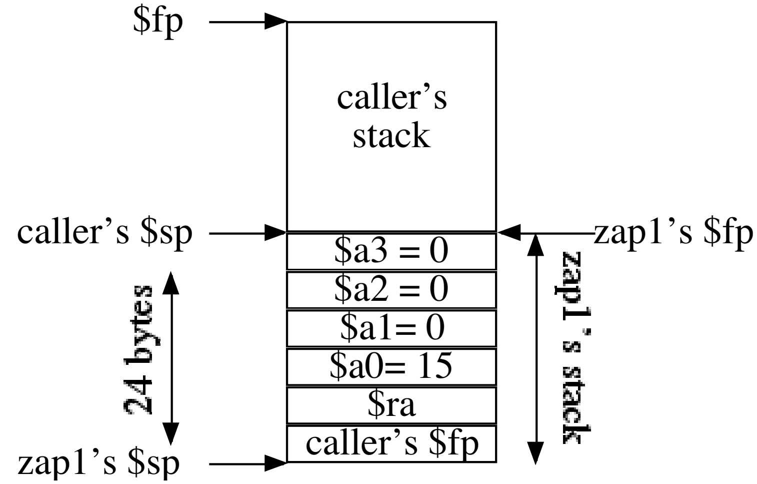
```
y = zap1( 15 );
```

- Calling code:




```
...
addi    $a0, $zero, 15      # put value into $a0 to pass to zap1
jal     zap1                # calling function zap1
# get result from function
add     $t1, $v0, $zero     # put result of function in register $t1
...
```

- The Function:




```
zap1:   subu    $sp, $sp, 24        # make enough room for zap1's needs on the stack
        sw      $fp, 0($sp)         # save the caller's frame pointer on the stack
        sw      $ra, 4($sp)         # save the return address on the stack
        sw      $a0, 8($sp)         # save $a0 on the stack
        addiu   $fp, $sp, 24        # set zap1's frame pointer
        # $a1-$a3 not used here
        # body of zap1 here...
        # assuming zap1 does not use any s registers and does not call any other functions
        # somewhere in body, zap1 puts the return value into $v0
        lw      $a0, 8($sp)         # restore original value of $a0 from the stack
        lw      $ra, 4($sp)         # get return address so we can return
        lw      $fp, 0($sp)         # restore caller's $fp
        addiu   $sp, $sp, 24        # restore caller's $sp
        jr      $ra                 # return to caller's code
```

```
.data
main1String: .asciiz  "Inside main, after call to zap1, returned value = "
zap1String:  .asciiz  "Inside function zap1, quadrupled value = "
newline:     .asciiz  "\n"

.text
main:
        # Function prologue -- even main has one
        subu  $sp, $sp, 24    # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        addiu $fp, $sp, 24    # setup zap1's frame pointer

        # body of main

        # call function zap1 with 15
        addi  $a0, $zero, 15
        jal   zap1

        add   $t0, $v0, $zero   # save return value in $t0

        la    $a0, main1String
        li    $v0, 4
        syscall
        add   $a0, $t0, $zero
        li    $v0, 1
        syscall
```

```
        la    $a0, newline
        li    $v0, 4
        syscall

        # call function zap1 with 42
        addi  $a0, $zero, 42
        jal   zap1

        add   $t0, $v0, $zero   # save return value in $t0

        la    $a0, main1String
        li    $v0, 4
        syscall
        add   $a0, $t0, $zero
        li    $v0, 1
        syscall
        la    $a0, newline
        li    $v0, 4
        syscall

done:   # Epilogue for main -- restore stack & frame pointers and return
        lw    $ra, 4($sp)      # get return address from stack
        lw    $fp, 0($sp)      # restore the caller's frame pointer
        addiu $sp, $sp, 24     # restore the caller's stack pointer
        jr    $ra              # return to caller's code
```

```
zap1:
        # Function prologue
        subu  $sp, $sp, 24    # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        sw    $a0, 8($sp)     # save parameter value
        addiu $fp, $sp, 24    # setup zap1's frame pointer

        # something for zap to do
        add   $t0, $a0, $a0   # double the parameter
        add   $t0, $t0, $t0   # quadruple the parameter

        # print results
        la    $a0, zap1String # print the string
        li    $v0, 4
        syscall
        add   $a0, $t0, $zero # print the quadruple'd value
        li    $v0, 1
        syscall
        la    $a0, newline
        li    $v0, 4
        syscall
```

```
# put result of function in $v0
# Note: could not do this before printing!
add   $v0, $t0, $zero

# Function epilogue -- restore stack & frame pointers and return
lw    $a0, 8($sp)      # restore original value of $a0 for caller
lw    $ra, 4($sp)      # get return address from stack
lw    $fp, 0($sp)      # restore the caller's frame pointer
addiu $sp, $sp, 24     # restore the caller's stack pointer
jr    $ra              # return to caller's code
```

Function Call Example 2

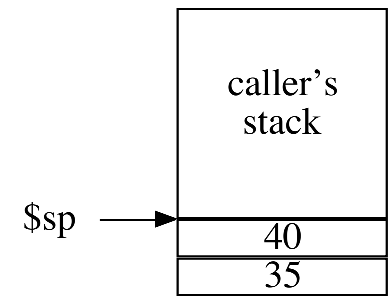
- Want to call a function that takes more than four arguments:

  `int zap2( int a, int b, int c, int d, int e, int f)`

- Need to put a, b, c, and d into $a0-$a3.
- Where to put e and f? On the stack!

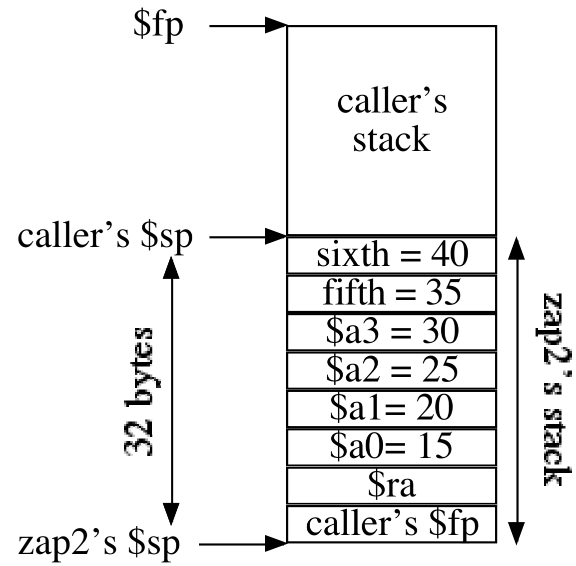
Caller's code:

```
...
li      $a0, 15             # put value into $a0 for zap2
li      $a1, 20             # put value into $a1 for zap2
li      $a2, 25             # put value into $a2 for zap2
li      $a3, 30             # put value into $a3 for zap2
li      $t0, 40
sw      $t0, -4($sp)        # put value onto stack for zap2
li      $t1, 35
sw      $t1, -8($sp)        # put value onto stack for zap2
jal     zap2                # calling function zap2
# get result from function
add     $t1, $v0, $zero     # put result of function in register $t1
...
```



- The Function:

```
zap2:   # make enough room for zap2 on the stack
        subu    $sp, $sp, 32
        # save the caller's frame pointer on the stack
        sw      $fp, 0($sp)
        # save the return address on the stack
        sw      $ra, 4($sp)
        # save parameter values $a0-$a3 on the stack
        sw      $a0, 8($sp)
        sw      $a1, 12($sp)
        sw      $a2, 16($sp)
        sw      $a3, 20($sp)
        # set zap2's frame pointer
        add     $fp, $sp, 32
        # assuming zap2 does not use any s registers and does not call any other functions
        # add up all six values:
        add     $t0, $a0, $a1       # add $a0 + $a1
        add     $t0, $t0, $a2       # add $a2
        add     $t0, $t0, $a3       # add $a3
        lw      $t1, 24($sp)        # get 5th argument
        add     $t0, $t0, $t1       # add 5th argument
        lw      $t1, 28($sp)        # get 6th argument
        add     $t0, $t0, $t1       # add 6th argument
```

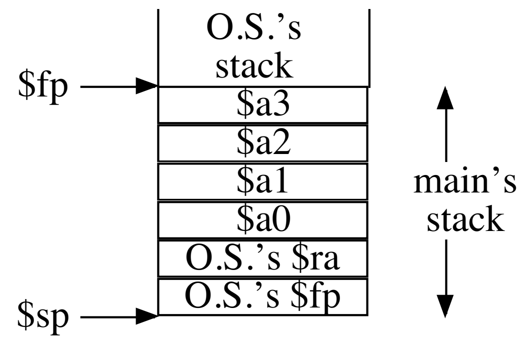


```
# zap2 puts the return value into $v0
add     $v0, $t0, $zero
# zap2 did not change $a0-$a3, so we do not need to restore them
lw      $fp, 0($sp)          # restore caller’s $fp
lw      $ra, 4($sp)          # get return address so we can return
addiu   $sp, $sp, 32         # restore caller’s $sp
jr      $ra                  # return to caller’s code
```

`main:` is a Function!

- The "outside world" does a function call to **`main`** to start our program running
  - "outside world" can be the O.S., can be a command-line shell
  - parameters can be passed to our program from the outside.
- Have to set up main's stack correctly
  - First code in main will always be:

```
main:
        # Prologue: set up stack and frame pointers for main
        subu    $sp, $sp, 24      # allocate stack space
        sw      $fp, 0($sp)       # save frame pointer
        sw      $ra, 4($sp)       # save return address
        addiu   $fp, $sp, 24      # establish main's $fp
```



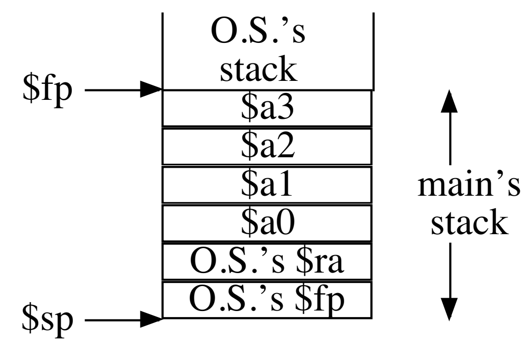
  - Final code in main will always be:

```
#       Epilogue: restore stack and frame pointers and return
        lw      $ra, 4($sp)       # restore return address
        lw      $fp, 0($sp)       # restore caller's frame pointer
        addiu   $sp, $sp, 24      # restore caller's stack pointer
        jr      $ra               # return  MAIN ENDS HERE
```

Following example is available to copy as: **`funcExample2.s`**

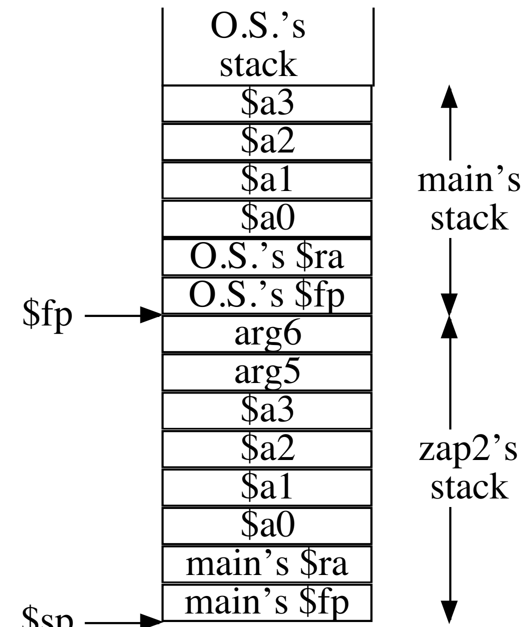
- on lectura: **`~cs252/fall03/SPIMexamples/funcExample2.s`**
- on Win2k: **`Rotis -> cs252/source/SPIMexamples/funcExample2.s`**

```
.data
str1:   .asciiz "Result of call #1 to function zap2 is "
str2:   .asciiz "Result of call #2 to function zap2 is "
nl:     .asciiz "\n\n"
.text
main:   # Prologue: set up stack and frame pointers for main
        subu    $sp, $sp, 24     # allocate stack space
        sw      $fp, 0($sp)      # save caller's frame pointer
        sw      $ra, 4($sp)      # save return address
        addiu   $fp, $sp, 24     # setup main's $fp

        # add up some numbers using zap2 and print result
        li      $a0, 15          # put value into $a0 for zap2
        li      $a1, 20          # put value into $a1 for zap2
        li      $a2, 25          # put value into $a2 for zap2
        li      $a3, 30          # put value into $a3 for zap2
        li      $t0, 40
        sw      $t0, -4($sp)     # put value onto stack for zap2
        li      $t1, 35
        sw      $t1, -8($sp)     # put value onto stack for zap2
        jal     zap2             # calling function zap2

        # print result from function
        add     $a0, $v0, $zero  # put result of function in register $a0
        addi    $a1, $zero, 1    # indicate which result this is
```

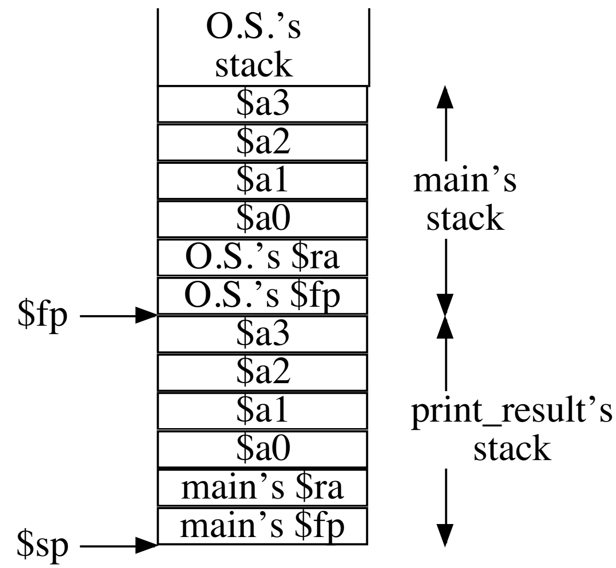


```
        jal     print_result

        # and, to show we can do it again...
        # add up some numbers using zap2 and print result
        li      $a0, -15          # put value into $a0 for zap2
        li      $a1, -20          # put value into $a1 for zap2
        li      $a2, -25          # put value into $a2 for zap2
        li      $a3, -30          # put value into $a3 for zap2
        li      $t0, -40
        sw      $t0, -4($sp)      # put value onto stack for zap2
        li      $t1, -35
        sw      $t1, -8($sp)      # put value onto stack for zap2
        jal     zap2              # calling function zap2

        # print result from function
        add     $a0, $v0, $zero   # put result of function in register $a0
        addi    $a1, $zero, 2     # indicate which result this is
        jal     print_result

done:   # Epilogue: restore stack and frame pointers and return
        lw      $ra, 4($sp)       # restore return address
        lw      $fp, 0($sp)       # restore caller's frame pointer
        addiu   $sp, $sp, 24      # restore caller's stack pointer
        jr      $ra               # return  MAIN ENDS HERE
```

```
zap2:   # Prologue: set up stack and frame pointers for zap2
        subu    $sp, $sp, 32
        # save the caller's frame pointer on the stack
        sw      $fp, 0($sp)
        # save the return address on the stack
        sw      $ra, 4($sp)
        # set zap2's frame pointer
        add     $fp, $sp, 32
        # zap2 doesn't use s registers or call functions

        # add up all six values:
        add     $t0, $a0, $a1      # add $a0 + $a1
        add     $t0, $t0, $a2      # add $a2
        add     $t0, $t0, $a3      # add $a3
        lw      $t1, 24($sp)       # get 5th argument
        add     $t0, $t0, $t1      # add 5th argument
        lw      $t1, 28($sp)       # get 6th argument
        add     $t0, $t0, $t1
        # zap2 puts the return value into $v0
        add     $v0, $t0, $zero
        # zap2 did not change $a0-$a3, so we do not need to restore them
        lw      $fp, 0($sp)        # restore caller's $fp
        lw      $ra, 4($sp)        # get return address
        addiu   $sp, $sp, 32       # restore caller's $sp
        jr      $ra                # return to caller's code ZAP2 ENDS HERE
```



```
print_result:
        # Prologue: set up stack and frame pointers for print_result
        subu    $sp, $sp, 24
        # save the caller's frame pointer on the stack
        sw      $fp, 0($sp)
        # save the return address on the stack
        sw      $ra, 4($sp)
        # set-up our frame pointer
        addi    $fp, $sp, 24
        # save parameter values $a0-$a1 on the stack
        # syscall's below use $a0, so save $a0 on stack
        # can also save $a1, but not necessary...
        sw      $a0, 8($sp)

        # second parameter tells us which string to print
        beq     $a1, 2, second
        la      $a0, nl         # print some blank lines
        li      $v0, 4
        syscall
        la      $a0, str1       # print first message
        li      $v0, 4
        syscall
        j       printSum
```

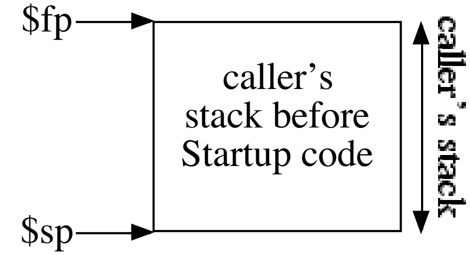


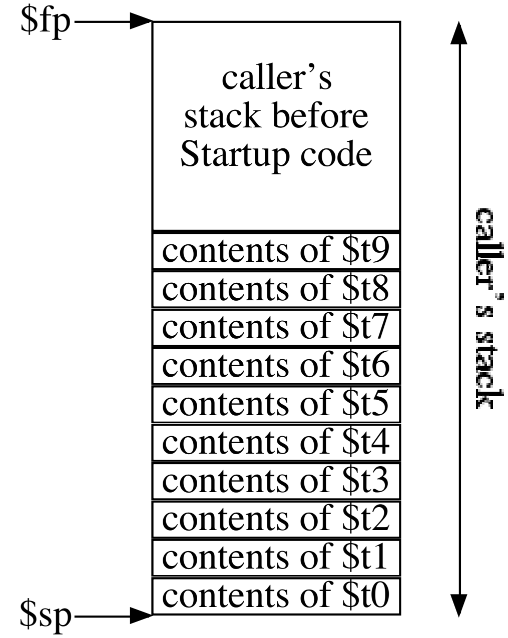
```
second: la      $a0, str2          # print second message
        li      $v0, 4
        syscall

printSum:
        lw      $a0, 8($sp)        # print the sum
        li      $v0, 1
        syscall
        la      $a0, nl            # print the newline's
        li      $v0, 4
        syscall

        # Epilogue: Restore stack and frame pointers and return
        # Since $a0 was modified by print_result, must restore $a0
        lw      $a0, 8($sp)
        lw      $fp, 0($sp)        # restore caller's frame pointer
        lw      $ra, 4($sp)        # get return address so we can return
        addiu   $sp, $sp, 24       # restore caller's stack pointer
        jr      $ra                # PRINTSUM ENDS HERE
```

Function Call Example 3 — Saving Registers

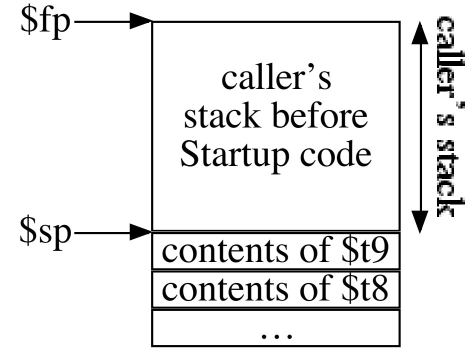
- The calling function is using all of the `t` registers and needs to preserve their contents:

```
# Startup sequence to call function zap3
# Save t registers on stack, need 4 bytes for each
subu    $sp, $sp, 40        # make room on my stack
sw      $t9, 36($sp)
sw      $t8, 32($sp)
sw      $t7, 28($sp)
sw      $t6, 24($sp)
sw      $t5, 20($sp)
sw      $t4, 16($sp)
sw      $t3, 12($sp)
sw      $t2, 8($sp)
sw      $t1, 4($sp)
sw      $t0, 0($sp)

# Two parameters, put in $a0 and $a1
lw      $a0, x
lw      $a1, y

jal     zap3                # call the function
```

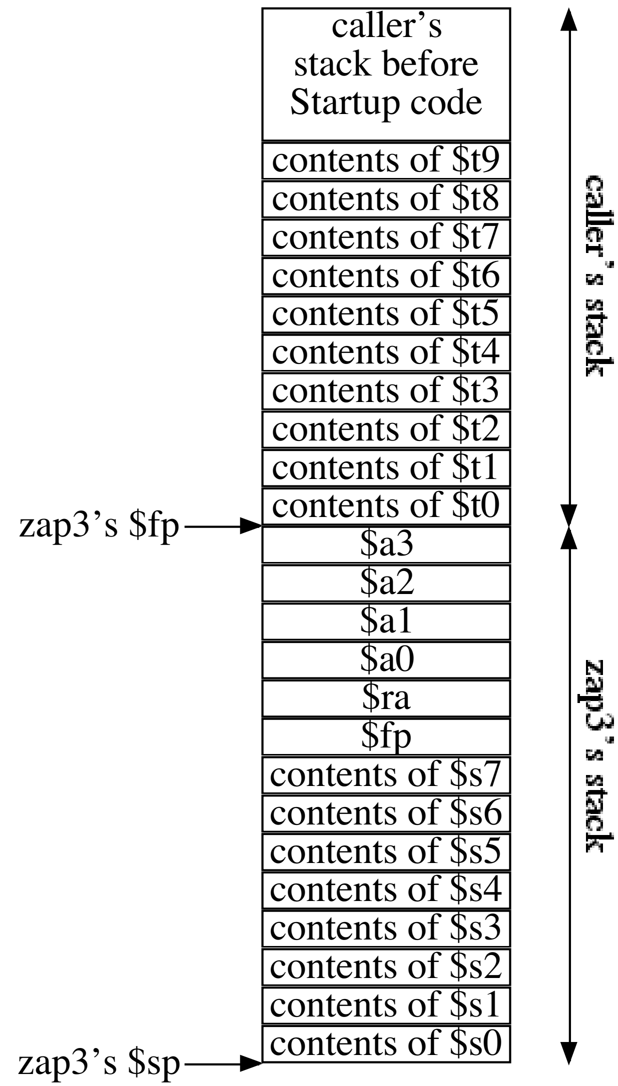




```
# Restore the t registers
lw      $t9, 36($sp)
lw      $t8, 32($sp)
lw      $t7, 28($sp)
lw      $t6, 24($sp)
lw      $t5, 20($sp)
lw      $t4, 16($sp)
lw      $t3, 12($sp)
lw      $t2, 8($sp)
lw      $t1, 4($sp)
lw      $t0, 0($sp)
addiu   $sp, $sp, 40        # Shrink stack
#... code that follows function call
```



```
zap3:   # zap3 may need to call another function, so must
        # save all the arguments on the stack and save the
        # s registers on the stack
        # Prologue code:
        subu    $sp, $sp, 56
        sw      $a1, 44($sp)
        sw      $a0, 40($sp)
        sw      $ra, 36($sp)
        sw      $fp, 32($sp)
        sw      $s7, 28($sp)
        sw      $s6, 24($sp)
        sw      $s5, 20($sp)
        sw      $s4, 16($sp)
        sw      $s3, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        addiu   $fp, $sp, 56        # set zap3's $fp

        # ... body of zap3 goes here ...

        # Epilogue code:
        # Must restore the a registers before returning
        lw      $a1, 44($sp)
        lw      $a0, 40($sp)
```



```
# Must restore the s registers before returning
lw      $ra, 36($sp)
lw      $fp, 32($sp)
lw      $s7, 28($sp)
lw      $s6, 24($sp)
lw      $s5, 20($sp)
lw      $s4, 16($sp)
lw      $s3, 12($sp)
lw      $s2, 8($sp)
lw      $s1, 4($sp)
lw      $s0, 0($sp)
addiu   $sp, $sp, 56
jr      $ra
```

Function Call Example 4 — Recursion

- Functions can call other functions, including themselves.
- Set up the stack in the same way as for the original function call — not really any different from what we have been doing!
- Fibonacci sequence:

$$f(N) = \begin{cases} 1, N = 1 \\ 1, N = 2 \\ f(N-1) + f(N-2), N \geq 3 \end{cases}$$

- Need to check the two base cases:

```
        # if N == 1, return 1
        li      $t0, 1
        bne     $a0, $t0, N2
        li      $v0, 1
        j       fibend

N2:     # if N == 2, return 1
        li      $t0, 2
        bne     $a0, $t0, N3
        li      $v0, 1
        j       fibend

N3:      # compute fibonacci(N-1) + fibonacci(N-2)
        ...
```

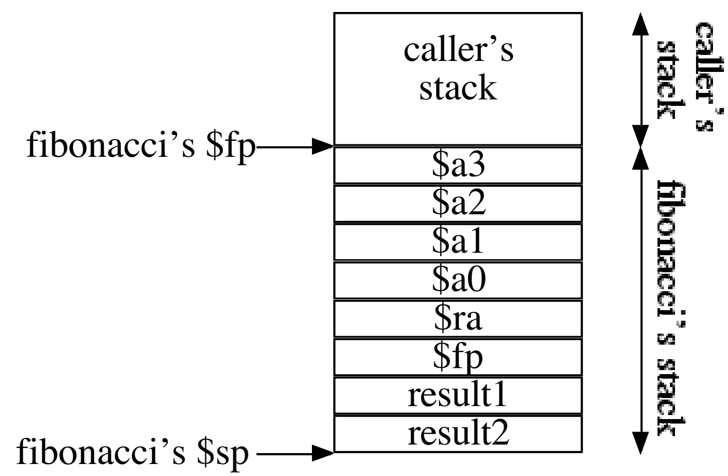
- Need to make two recursive calls:
  - Need to “remember” the value of \$a0 so we can restore it — we’ve done this before
  - Need to “remember” the results of the two recursive calls. Two ways to do this:
    - Use a register for each — \$t1 and \$t2 in my example
    - Save them as “local” variables
- Using two registers:

```
N3:     # compute $t1 = fibonacci(N-1)
        addi    $a0, $a0, -1     # compute N - 1
        jal     fibonacci
        add     $t1, $v0, $zero  # save result1 in $t1

        # compute $t2 = fibonacci(N-2)
        # save $t1 on the stack first
        # grow stack temporarily (double-word aligned means 8 bytes)
        subu    $sp, $sp, 8
        sw      $t1, 0($sp)
        addi    $a0, $a0, -1     # compute N - 2
        jal     fibonacci
        add     $t2, $v0, $zero  # save result2 in $t2
        # get $t1 off the stack and shrink the stack
        lw      $t1, 0($sp)
        addiu   $sp, $sp, 8

        add     $v0, $t1, $t2    # compute answer = result1 + result2
```

- Using “local” variables
  - Basic idea is to create enough space on the stack initially to hold locally-declared variables.
  - The C code would be:

```
int fibonacci( int N ) {
    int result1;
    int result2;
    /* test for base cases not shown here... */
    result1 = fibonacci( N - 1 );
    result2 = fibonacci( N - 2 );
    return result1 + result2;
}
```

  - For “local” variables (result1 and result2 in this case), create space on the stack
    - add enough space to the stack size, 8 bytes in this case.
    - add extra space, if needed, to meet double-word aligned requirement (not needed this time).
    - order of locals on the stack entirely up to the programmer — no convention for this.



• The code for the local variable case:

```
fibonacci:
        # Prologue: set up stack and frame pointers for fibonacci
        # Need two local variables to hold the results of the two
        # recursive calls to fibonacci
        subu    $sp, $sp, 32          # allocate stack space
        sw      $fp, 8($sp)           # save frame pointer
        sw      $ra,12($sp)           # save return address
        addi    $fp, $sp, 32          # set-up our frame pointer
        sw      $a0,16($sp)           # save $a0 on the stack

        # skip over the two base cases for now...

N3:     # compute result1 = fibonacci(N-1)
        addi    $a0, $a0, -1          # compute N - 1
        jal     fibonacci
        sw      $v0, 4($sp)           # save result1

        # compute result2 = fibonacci(N-2)
        addi    $a0, $a0, -1          # compute N - 2
        jal     fibonacci
        sw      $v0, 0($sp)           # save result2

        lw      $t1, 4($sp)           # $t1 = result1
        lw      $t2, 0($sp)           # $t2 = result2
        add     $v0, $t1, $t2         # compute answer = result1 + result2
```

```
# Epilogue: restore stack and frame pointers and return
lw      $a0,16($sp)         # restore $a0's value
lw      $fp, 8($sp)         # restore caller's frame pointer
lw      $ra,12($sp)         # restore return address
addiu   $sp, $sp, 32        # restore caller's stack pointer
jr      $ra                 # return  FIBONACCI ENDS HERE
```

- Note:
  - Using **`t`** registers choice creates space on the stack at the next function call, and removes that space just after the next function call.
  - Local variable choice creates space on the stack for the variables at the beginning (prologue) of the function, and removes that space at the end (epilogue) of the function.

- Complete program examples available for download:
  - from web page: **`http://www.cs.arizona.edu/classes/cs252/summer04/`**
  - from lectura: **`~cs252/summer04/SPIMexamples`**
  - from Win2K: **`Rotis -> cs252/source/SPIMexamples`**

Function Call Example 4 — Local Variables

Local variables

- If the subroutine has local variables that don't fit in registers, it reserves space for them in its stack frame.

```
int computeAverage( int zapArray[], int size ) {
    int sum;
    int count;
    ...
    return sum;
}
```

`zapArray`'s address in **`$a0`** and on the stack.

`size`'s value in **`$a1`** and on the stack.

`sum` and **`count`** on the local variable part of the stack.

- Consider the following C code:

```
int zap4 ( int start, int step ) {
    int i;  /* loop index */
    for ( i = start; i < finish; i += step ) {
```

-

Case/Switch Statement (see example program named switch.s)

- C code: (see pages 129-130)

```
switch (k) {
   case 0: f = i + jj; break;
   case 1: f = g + h;  break;
   case 2: f = g - h;  break;
   case 3: f = i - jj; break;
}
```

- MIPS version:

```
.data
jump:       .word   L0   # address of label for case 0
            .word   L1   # address of label for case 1
            .word   L2   # address of label for case 1
            .word   L3   # address of label for case 1
# init some variable values
g:          .word   42
h:          .word   37
i:          .word   15
jj:         .word   12   # Note: can't use j: as a label; conflicts w/ j opcode
# Useful messages
str0:       .asciiz "case 0: f = i + jj = "
str1:       .asciiz "case 1: f = g + h = "
str2:       .asciiz "case 2: f = g - h = "
str3:       .asciiz "case 3: f = i - jj = "
```

```
inputstr:  .asciiz "Enter a value for k between 0 and 3 "
smallstr:  .asciiz "k is too small, must be between 0 and 3\n\n"
largestr:  .asciiz "k is too large, must be between 0 and 3\n\n"
nl:        .asciiz "\n"

.text
main:
# Get value of k from stdin
inputK:
        la      $a0, inputstr    # print input query
        li      $v0, 4
        syscall
        li      $v0, 5           # read k from stdin
        syscall
        add     $t0, $v0, $zero  # put k in $t0
        la      $a0, nl          # print a blank line
        li      $v0, 4
        syscall
# test for valid input, 0 <= k <= 3
        slt     $t1, $t0, $zero  # Test if k < 0
        bne     $t1, $zero, toosmall
        slti    $t1, $t0, 4      # Test if k > 3
        beq     $t1, $zero, toolarge
        j       switch
```

```
toosmall:
        la      $a0, smallstr
        li      $v0, 4
        syscall
        j       inputK
toolarge:
        la      $a0, largestr
        li      $v0, 4
        syscall
        j       inputK

# switch statement
switch:

        la      $t1, jump           # load address of start of jump table

        # Compute offset from start of jump table
        add     $t0, $t0, $t0       # compute k = 4 * k
        add     $t0, $t0, $t0
        add     $t1, $t0, $t1       # add offset (4 * k) to start of jump
        lw      $t2, 0($t1)         # load address from jump[k]
        jr      $t2                 # jump to appropriate case
```

```
L0:     la      $a0, str0         # print string for this case
        li      $v0, 4
        syscall
        lw      $s1, i            # $s1 = i
        lw      $s2, jj           # $s2 = jj
        add     $a0, $s1, $s2     # f = i + jj
        li      $v0, 1            # print f
        syscall
        j       endswitch

L1:     la      $a0, str1         # print string for this case
        li      $v0, 4
        syscall
        lw      $s1, g            # $s1 = g
        lw      $s2, h            # $s2 = h
        add     $a0, $s1, $s2     # f = g + h
        li      $v0, 1            # print f
        syscall
        j       endswitch
```

```
L2:     la      $a0, str2         # print string for this case
        li      $v0, 4
        syscall
        lw      $s1, g            # $s1 = g
        lw      $s2, h            # $s2 = h
        sub     $a0, $s1, $s2     # f = g - h
        li      $v0, 1            # print f
        syscall
        j       endswitch
L3:     la      $a0, str3         # print string for this case
        li      $v0, 4
        syscall
        lw      $s1, i            # $s1 = i
        lw      $s2, jj           # $s2 = jj
        sub     $a0, $s1, $s2     # f = i - jj
        li      $v0, 1            # print f
        syscall

endswitch:
        la      $a0, nl           # print newline character
        li      $v0, 4
        syscall
done:   li      $v0, 10          # exit
        syscall
```