Procedures

Overview

- HLL Functions -> MIPS Procedures
  - Passing arguments
  - Function calls
  - The stack
  - Nested functions
  - Register Conventions
- Be aware:
  - There are many ways to program
  - **This might be the most complicated topic in MIPS**

HLL Functions to MIPS Procedure

```
// a simple function in C
int sum (int x, int y ) {
    return x + y ;
}
void main () {
    int a = sum ( 3, 6 ) ;
}
```

How do we convert this into MIPS assembly?

- What instructions can accomplish this?
- What information must compiler / programmer keep track of?

Passing Arguments in MIPS Function Calls

The Caller and the Callee

```
int sum (int x, int y ) {
    return x + y ;
}
void main () {
    int a = sum ( 3, 6 ) ;
}
```

main is the parent/caller
sum is the child/callee

- The **parent/caller** needs to
  - Save arguments to **child/callee**
  - Save return address to **child/callee**
  - Branch to **child/callee**
- The **child/callee** needs to
  - Save the return value for **parent/caller**
  - Branch back to **parent/caller**

Register Conventions on Passing Arguments

- Registers play a major role in keeping track of information
- Register conventions
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`

```
int sum (int x, int y ) {
    return x + y ;
}
void main () {
    int a = sum ( 3, 6 ) ;
}
```

- Parent/Caller should save the arguments in $a0, $a1, $a2, $a3
- Child/Callee should save the return values in $v0, $v1

Register Conventions on Passing Arguments

```c
int sum (int x, int y ) {
  return x + y ;
}
void main () {
  int a = sum ( 3, 6 ) ;
    ...
}
```

```
# Add $a0 + $a1
# Save the result in $v0
add  $v0, $a0, $a1
```

```
# Save argument to $a0
li $a0, 3

# Save argument to $a1
li $a1, 6
```

First Try... with Jump

```c
// C code
int sum (int x, int y ) {
  return x + y ;
}

void main () {
  int a; // $s0
  a = sum(3,6) + 3;
      ...
}
```

```
                 # MIPS code
sum:    add  $v0, $a0, $a1   # $t0 = $a0 + $a1
        j    AfterSum        # jump to AfterSum

Main:   li   $a0, 3          # $a0 = 3
        li   $a1, 6          # $a1 = 6
        j    sum             # jump to sum

AfterSum:
        addi $s0, $v0, 3     # a = sum(3,6) + 3
```

Not perfect!

```
// C code
int sum (int x, int y ) {
  return x + y ;
}
void main () {
  int a; // $s0
  a = sum(3,6) + 3;
      …
  int b; // $s1
  b = sum(4,8) + 7 ;
}
```

If Sum has multiple callers, how does it return to the right place?

```
                # MIPS code
sum:   add  $v0, $a0, $a1   # $t0 = $a0 + $a1
       j    ???             # jump to ???
```

Returning (properly) from Function Calls

Label = Address of the Instructions

```
              # MIPS code from the last example
0x1000 sum:   add  $v0, $a0, $a1    # $t0 = $a0 + $a1
0x1008        j    AfterSum         # jump to AfterSum

0x2000 Main:  li   $a0, 3           # $a0 = 3
0x2004        li   $a1, 6           # $a1 = 6
0x2008        j    sum              # jump to sum

0x200C AfterSum: addi $s0, $v0, 3 # a = sum(3,6) + 3
```

Label = Address of the Instructions

```
               # MIPS code from the last example
0x1000 sum:    add  $v0, $a0, $a1     # $t0 = $a0 + $a1
0x1008         j    0x200C            # jump to AfterSum

0x2000 Main:   li   $a0, 3            # $a0 = 3
0x2004         li   $a1, 6            # $a1 = 6
0x2008         j    0x1000            # jump to sum

0x200C AfterSum: addi $s0, $v0, 3 # a = sum(3,6) + 3
```

Save the return address of the caller, so we can jump back

Register Conventions for Jump

- Register for the return address $ra

```
int sum (int x, int y ) {
    return x + y ;
}
void main () {
    int a = sum ( 3, 6 ) ;
}
```

- Parent/Caller saves the returning address in $ra before jumping
- Child/Callee jumps to the $ra after done

Jump Register Operation

- Jump Register `jr`
  - Syntax: `jr register`

```
j label
```

```
jr $ra
```

J takes a label (fixed location)

Jr takes an address (variable location)

  - Only useful if we know exact address to jump to
  - Rarely applicable in situations other than return from function

Jump Register Operation

```c
int sum (int x, int y ) {
  return x + y ;
}
void main () {
  int a = sum ( 3, 6 ) ;
    ...
}
```

```
Address                # MIPS code

0x1000  sum:   add  $v0, $a0, $a1   # $v0 = $a0 + $a1
0x1008         jr   $ra             # jump to address $ra

0x2000  Main:  li   $a0, 3          # $a0 = 3
0x2004         li   $a1, 6          # $a1 = 6
0x2008         li   $ra, 0x2010     # save address to $ra
0x200C         j    Sum             # jump to sum
0x2010  AfterSum:
```

☹ **Need to find the address of the next instruction**

Load Address

- Pseudo-instruction: Load Address la Register Label

```
sum:  add  $v0, $a0, $a1       # $v0 = $a0 + $a1
      jr   $ra                 # jump to address $ra

Main: li   $a0, 3              # $a0 = 3
      li   $a1, 6              # $a1 = 6
      la   $ra, AfterSum       # load address to $ra
      j    Sum                 # jump to sum
AfterSum:
```

Too complicated? There are Simpler Instructions

Jump and Link Instruction

- Jump and Link syntax : **`jal label`**
  - Step 1 (link): Save address of ***next* instruction** into $ra (Why next?)
  - Step 2 (jump): Jump to the given label
  - `jal` should really be called `laj` for “link and jump”

- Very useful for function calls:
  - `jal` stores return address in $ra
  - `jr` jumps back to that address

Jump and Link Instruction

```
int sum (int x, int y ) {
  return x + y ;
}
void main () {
  int a = sum ( 3, 6 ) ;
    ...
}
```

A little bit simpler with jr and jal

Address

```
                                # MIPS code
0x1000  sum:  add  $v0, $a0, $a1    # $v0 = $a0 + $a1
0x1008        jr   $ra              # jump to address $ra

0x2000  Main: li   $a0, 3           # $a0 = 3
0x2004        li   $a1, 6           # $a1 = 6
0x2008        jal  Sum              # jump to sum
0x200C
```

What about Nested Procedures?

```
int square (int x ) {
  return x * x ;
}
int sumSquare (int x, int y ) {
  return square(x) + y ;
}
void main () {
  int a = sumSquare ( 3, 6 ) ;
    ...
}
```

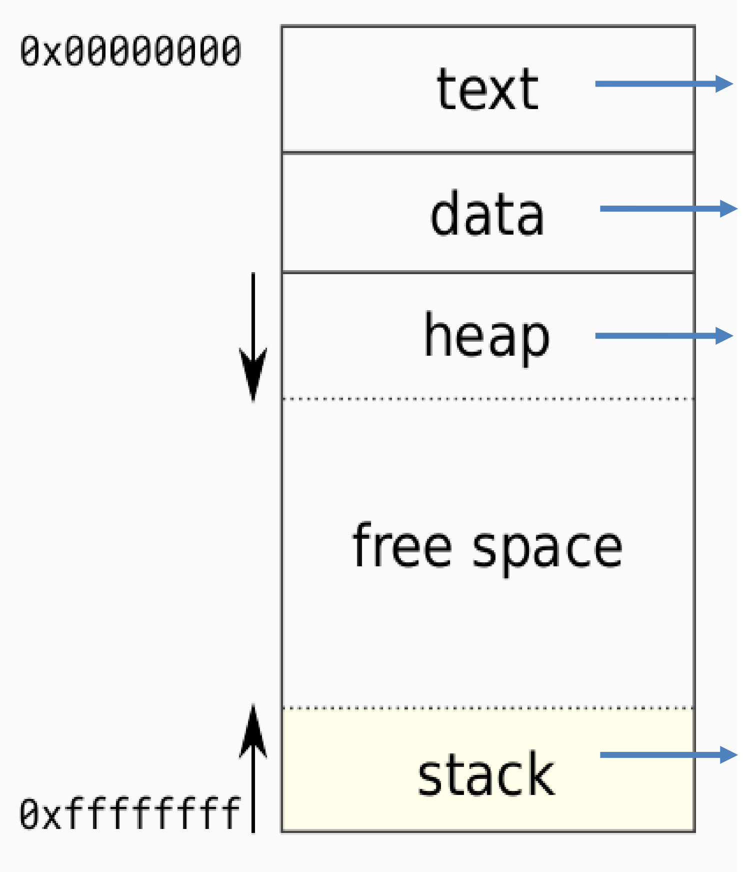
☹

- main called `sumSquare`, save the returning address in $ra
- `SumSquare` called `square`, but $ra will be overwritten when calling `square`

Need to save more information?

Stack Them!

Memory Layout



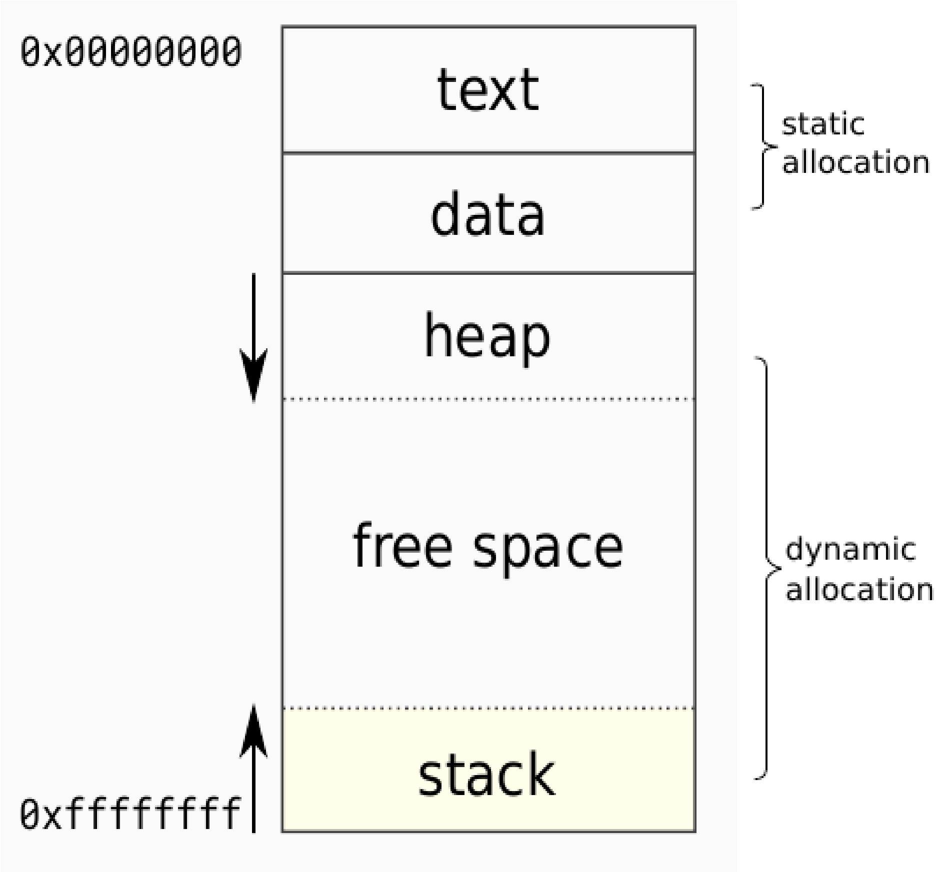
Where Instructions are stored (.text in MARS)

Statically allocated variables (.data in MARS)

Dynamically declared variables.
Explicitly created space, e.g., malloc(); new;

Space for local variables and registers in procedure calls

Memory Layout



Size of the text and data are fixed once the program is compiled

Heap starts at lower address and grows downward toward higher memory locaion

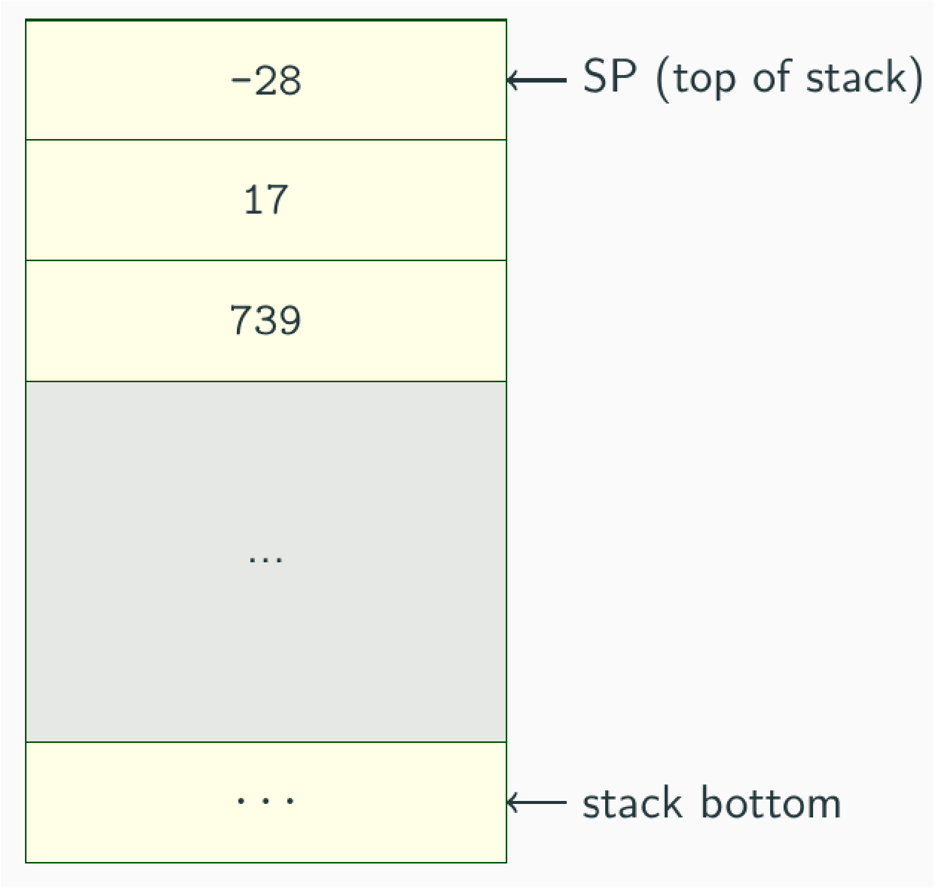
Stack starts at a fixed address and grows upward toward lower memory location

Stack v.s Heap

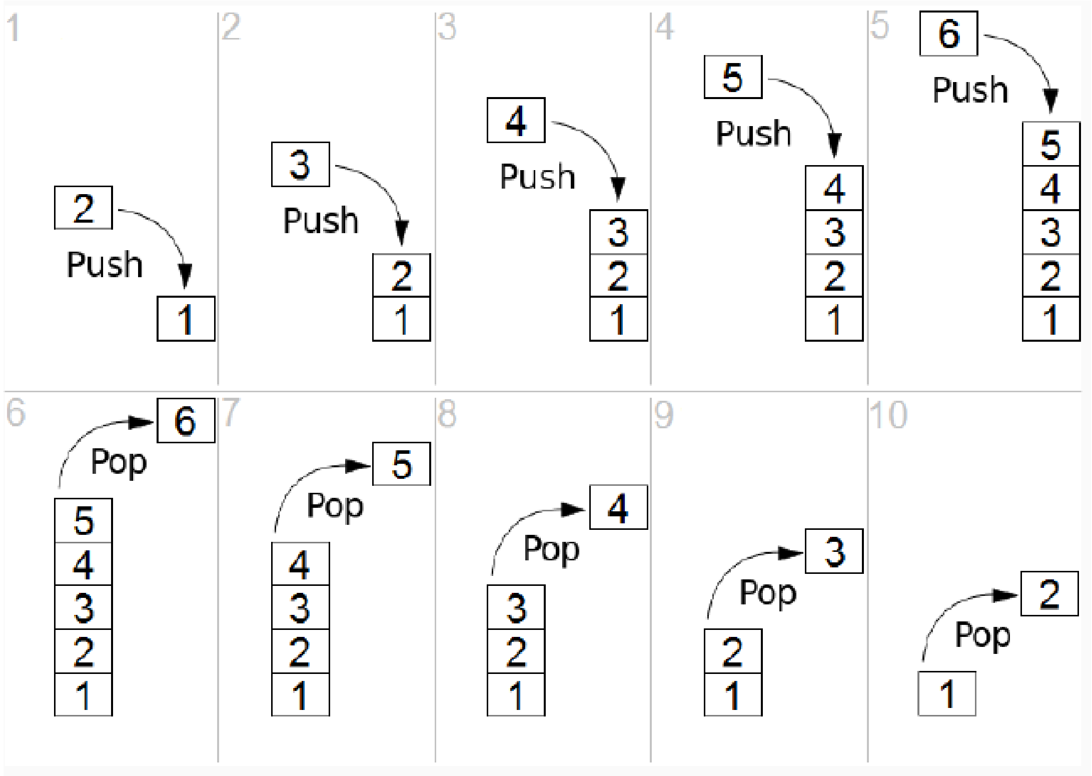
Stack is very organized. It is used to save data for procedures

Stack

- Stack is used to support subroutines
- Register $29 or $sp is used as a **Stack Pointer** points to the top of the stack
- The elements on the stack are words



Push and Pop for Stack

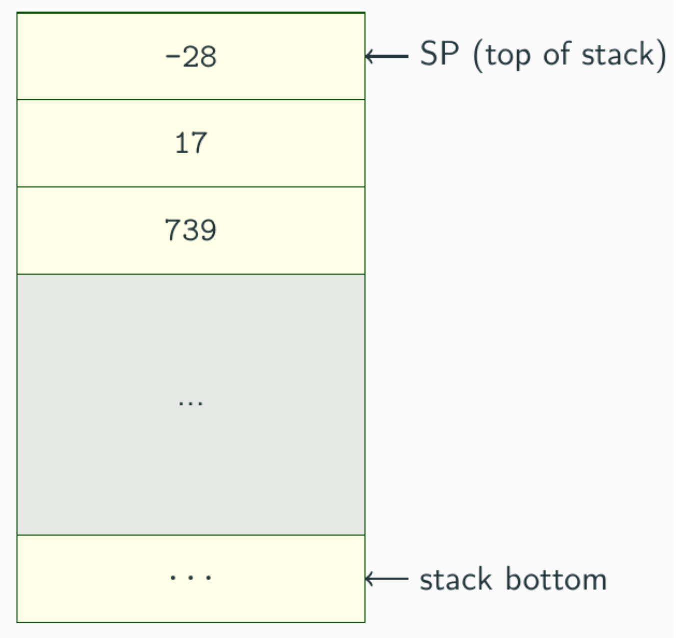


- **Push**: add an item on the top of the stack
- **Pop**: take the top element and remove it from the stack
- **Stack pointer** $sp always points to the last used space in the stack.

Push the Stack

- Register $sp always points to the last used space in the stack.
- To push, decrement this pointer by 4, then fill it with data.

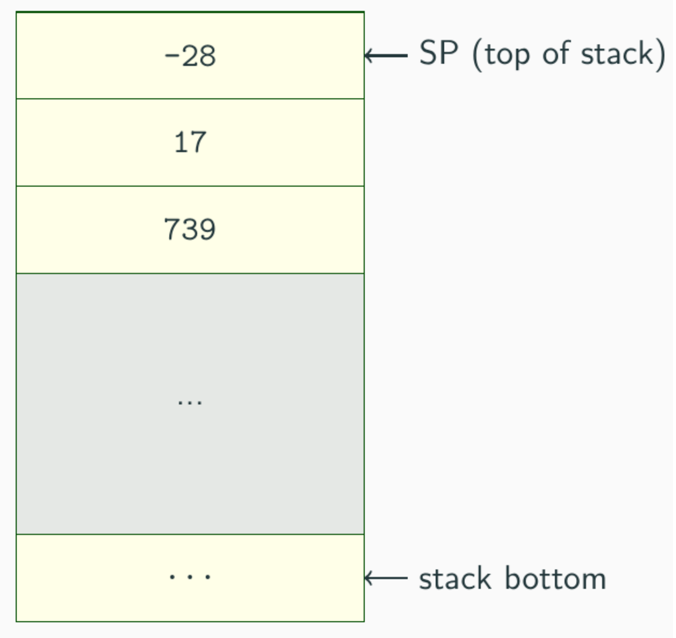
```
# move the stack pointer down
addi $sp, $sp, -4
# store the content of $s0 into address $sp
sw   $s0, 0($sp)
```



Pop the Stack

- Register $sp always points to the last used space in the stack.
- To pop, load the data and increment the stack pointer by 4

```
# load the content in address $sp to $s0
lw    $s0, 0($sp)
# move the stack pointer up
addi $sp, $sp, 4
```



Nested Call

- If you are a subroutine that will call another subroutine, follow this convention
  - Before you call anyone, push the return address in \$ra on the stack
  - When you are done calling, pop the return address of the stack into \$ra

Nested Call

```c
// Nested functions in C
int square (int x ) {
  return x * x ;
}
int sumSquare (int x, int y ) {
  return square(x) + y ;
}
void main () {
  int a = sumSquare ( 3, 6 ) ;
    ...
}
```

```
Main:
  li   $a0, 3
  li   $a1, 6
  jal  SumSquare
```

```
SumSquare:
  # 1. push $ra onto the stack
  addi $sp, $sp, -4    # move stack pointer
  sw   $ra, 0($sp)     # store $ra
  # 2. Jump to Square ($ra is overwritten)
  jal  Square
  # 3. Return from Square. $v0 has the result
  add  $v0, $v0, $a1   # Calculate square(x) + y
  # 4. pop $ra so we can return to the main
  lw   $ra, 0($sp)
  addi $sp, $sp, 4     # move stack pointer
  # 5. jump back to main
  jr   $ra
```

```
Square:
some code here
```

Stack for Variables

The stack is your friend: Use it to save anything you need.
Just be sure to leave it the way you found it.

Saved Registers in Procedure

```
int sum (int x, int y ) {
  int z = 5 ; // Use $s0 for z
  return z + x + y ;
}

void main () {
  int a = 2 ; // Use $s0 for a
  int b = sum ( 3, 6 ) ;
  int c = a + b;
    ...
}
```

- What if the callee needs some registers for local variables?
- Can we make sure that the callee does not overwrite important data?

Saved Registers in Procedure

```
int sum (int x, int y ) {
  int z = 4 ; // Use $s0 for z
  return z + x + y ;
}

void main () {
  int a = 2 ; // Use $s0 for a
  int b = sum ( 3, 6 ) ;
  int c = a + b;
     ...
}
```

For the Callee:

1. Push the registers to the stack
2. Execute the callee instructions
3. Before returning to the caller, restore the registers from the stack

Saved Registers in Procedure

```c
int sum (int x, int y ) {
  int z = 5 ; // Use $s0 for z
  return z + x + y ;
}

void main () {
  int a = 2 ; // Use $s0 for a
  int b = sum ( 3, 6 ) ;
  int c = a + b;
    ...
}
```

```
Sum:
  # 1. push $s0 to the stack
  addi $sp, $sp, -4
  sw   $s0, 0($sp)      # Store $s0
  # 2. Execute callee's instructions
  li   $s0, 5           # $s0 = 5
  add  $s0, $s0, $a0    # $s0 = z + x
  add  $v0, $s0, $a1    # $v0 = z + x + y
  # 3. Restore $s0 for the caller
  lw   $s0, 0($sp)      # Restore $s0
  addi $sp, $sp, 4
  jr   $ra             # jump back to main
```

Rules for Procedures

When should the callee save the registers?

Must follow ***register conventions***

- **Do this even for functions that only you will call!!!**
- What are they?

Register Convention

Assembly Variables: MIPS Registers

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0–\$v1	2–3	Values for results and expression evaluation	no
\$a0–\$a3	4–7	Arguments	no
\$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

Register Conventions in Procedure Call

If the ***callee*** is going to use some ***saved registers***, it must save those saved registers on the stack

The **callee** doesn't need to save the **temporary registers**

If the ***caller*** wants to preserve the **temporary registers**, it must save those **tempororay registers** on the stack

Steps for Procedure Call

Caller

1. Push the temp variable(s) if needed onto stack
2. Assign argument(s), if any
3. Jump and Link `jal`
4. Restore temp variable(s) from stack

Callee

1. Push the saved registers onto stack if needed
2. Use the saved registers
3. Restore the values of saved registers from stack
4. Assign return value(s), if any
5. Jump back with `jr $ra`

Preserved on Call?

Name	Preserved on a call?
`$v0-v1`	**No**, they are expected to contain return values
`$a0-$a3`	**No**, they are arguments
`$t0-$t9`	**No**, any procedure may change them at any time
`$s0-$s7`	**YES**. If the callee changes these registers, it must restore the original values before returning. That's why they're called saved registers.

Preserved on Call?

Name	Preserved on a call?
`$sp`	**Yes**. The stack pointer must point to the same place before and after the jal call, or else the caller won't be able to restore values from the stack.
`$ra`	**Yes**, until the next `jal` call
`$a0-$a3`	**No**, they are arguments
`$zero`	Always 0

Lastly

- Each register has a purpose and limits to its usage. Learn these and follow them, even if you're writing all the code yourself

- If every developer follows the same standard, it would be easier to exchange code or build large software

Review

- Functions are called with `jal and jr`
- Stack is your friend
- Text Section 2.8

Purpose	Operations
Arithmetic	**`add, addi`**
Memory	**`lw, sw`**
Decision	**`beq, bne, slt, slti, sltu, sltiu`**
Unconditional Branches	**`j, jal, jr`**

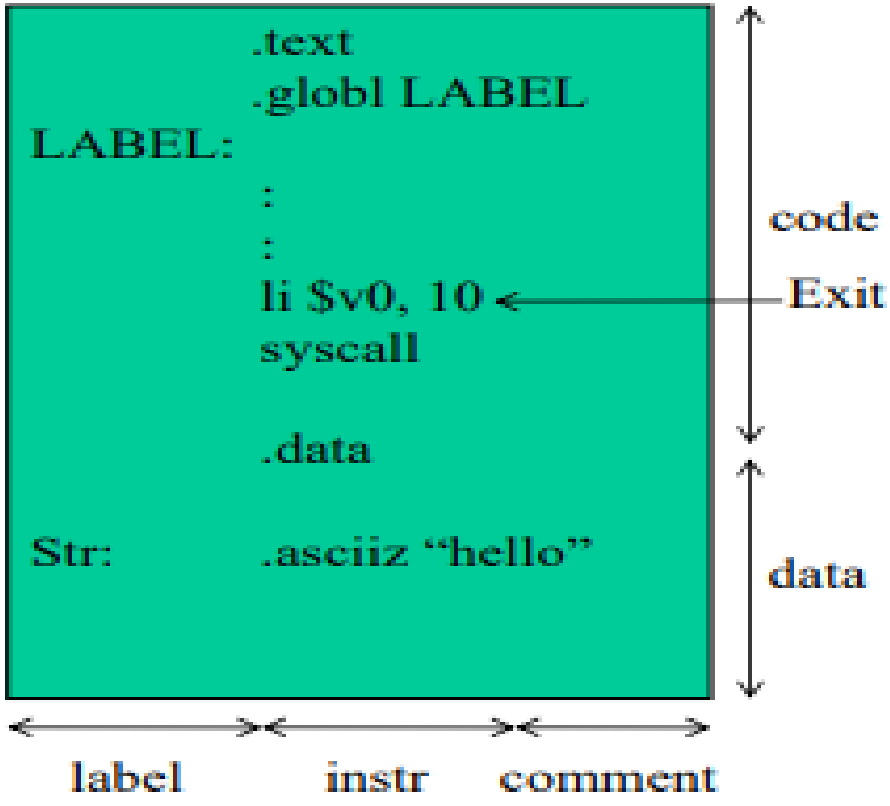
Data Transfer Instructions

Data transfer	load word	lw	$s1,100($s2)	$s1 = Memory[$s2+100]
	store word	sw	$s1,100($s2)	Memory[$s2 + 100] = $s1
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1
	load address	la	$s1, x	la register,label

You can reference data either using its **label** or its RAM **address**.

A label is converted to an address by the assembler.

MIPS assembly program structure



#	← comments
.command	← directive
Identifier:	← label
Identifier	← reference
INSTR arg1,arg2,…	← assembler

```
# my first code
.data
    x:      .word      5      # int x=5;

.text
.globl main                   # main()
    main:
    lw $s0, x                 #Reg = 5
    add $s0, $s0, 1           #Reg=Reg+1
    sw $s0, x                 #x = Reg
    li $v0, 10                #end prog.
    syscall                   #end prog.
```