



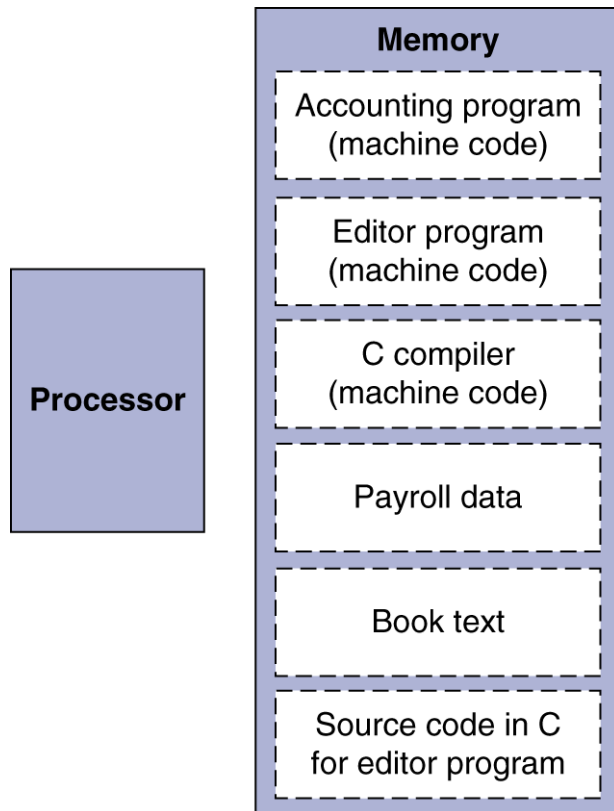
## Topic 3

---

### **Assembly Programming** **- Function (Procedure) Call**

# Stored Program

## The BIG Picture

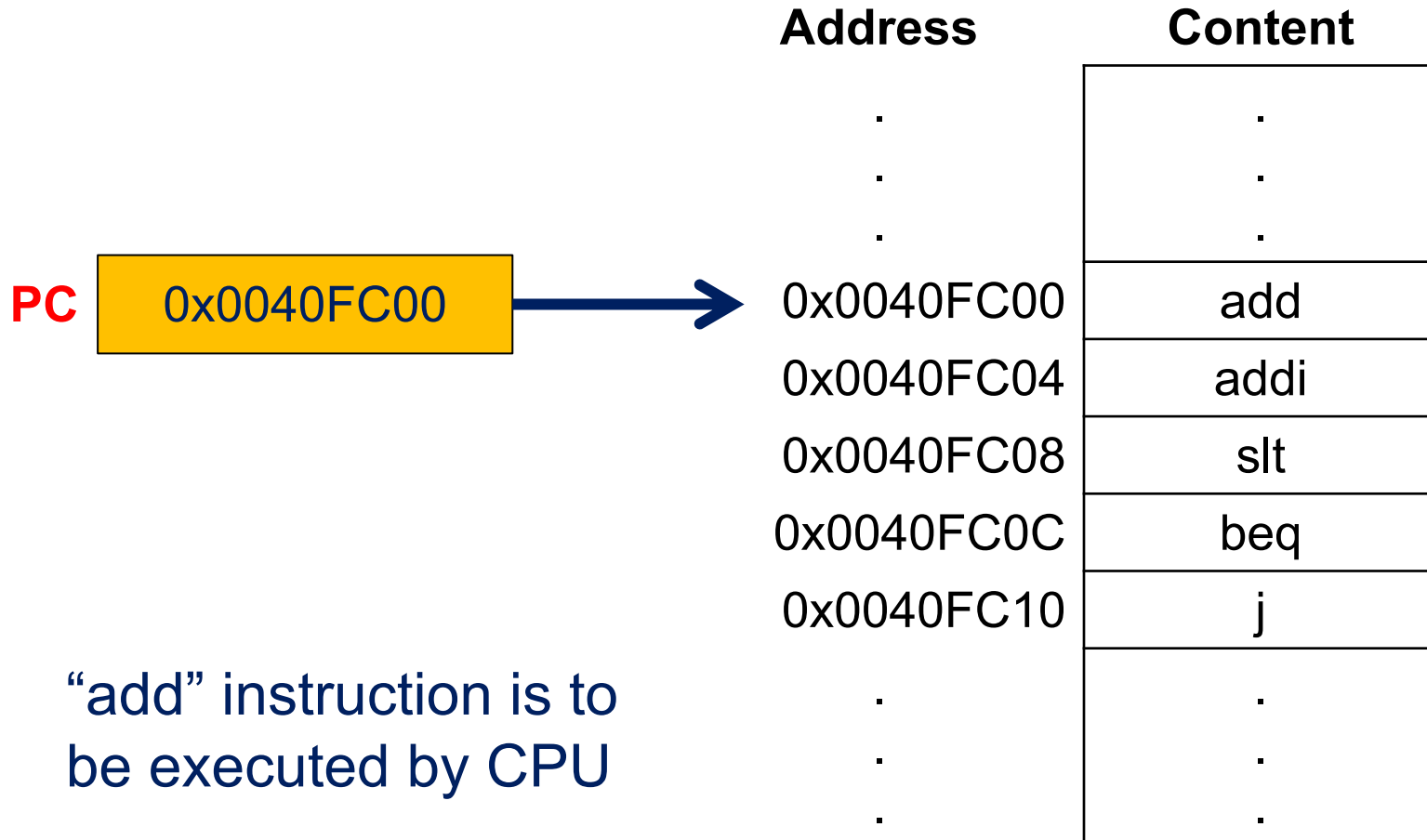


- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, ...

# Program Counter (PC)

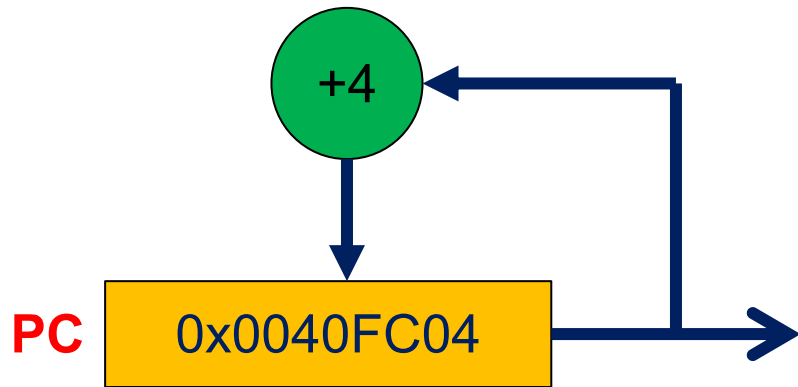
- Each instruction is stored as a word in program memory
  - has an address
  - when labeled, the label is equal to the address
- PC holds address of an instruction to be executed
  - 32 bits register
  - Usually increased by 4
- PC is a special register in CPU
  - Different from the registers in register file

# Program Counter (PC)



**Program stored in memory**

# Program Counter (PC)

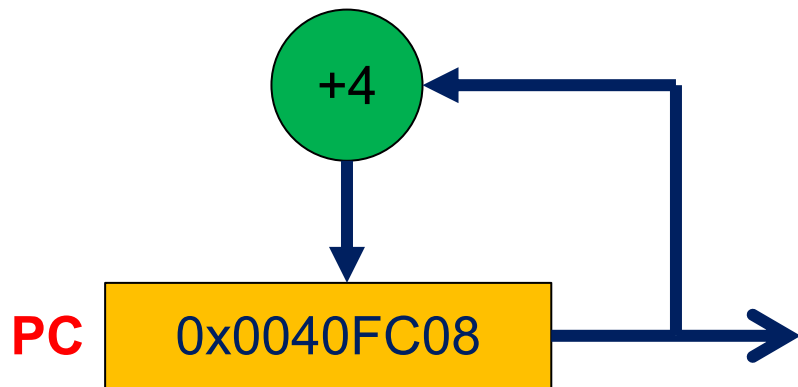


“addi” instruction is to be executed by CPU

Address	Content
.	.
.	.
.	.
0x0040FC00	add
0x0040FC04	addi
0x0040FC08	slt
0x0040FC0C	beq
0x0040FC10	j
.	.
.	.
.	.

**Program stored in memory**

# Program Counter (PC)



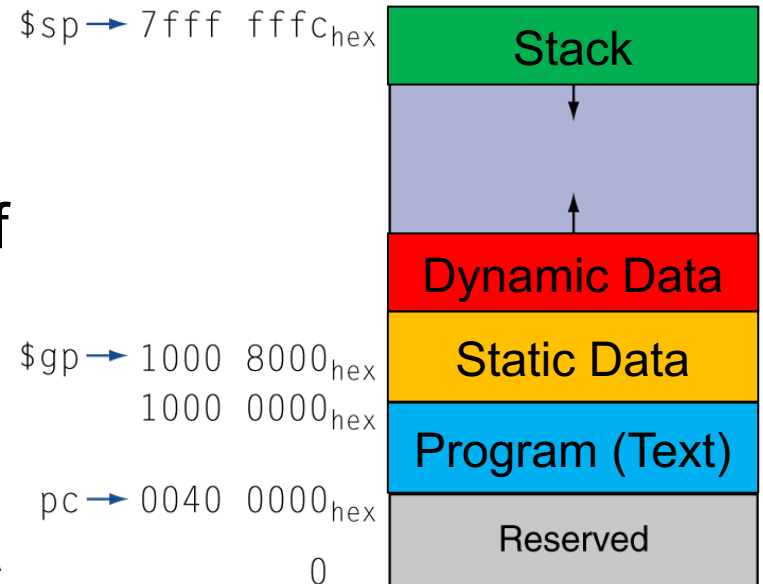
“slt” instruction is to be executed by CPU

Address	Content
.	.
.	.
.	.
0x0040FC00	add
0x0040FC04	addi
0x0040FC08	slt
0x0040FC0C	beq
0x0040FC10	j
.	.
.	.
.	.

**Program stored in memory**

# Memory Layout

- Text: program code
  - PC initialized to 0x00400000
- Static data: global/static variables
  - \$gp initialized to the middle of this segment, 0x10008000 allowing  $\pm$  offset
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: storage for temporary variable in functions
  - \$sp initialized to 0x7ffffffc, growing towards low address



# Function Calling

- Used to improve reusability and manageability
- Steps of function call operation
  1. Place parameters in parameter registers
  2. Transfer control to the function
  3. Acquire storage for the function in stack
  4. Perform function's operations
  5. Place results in result register(s) for caller
  6. Release storage
  7. Return to the place before the function call



# Function Call Instructions

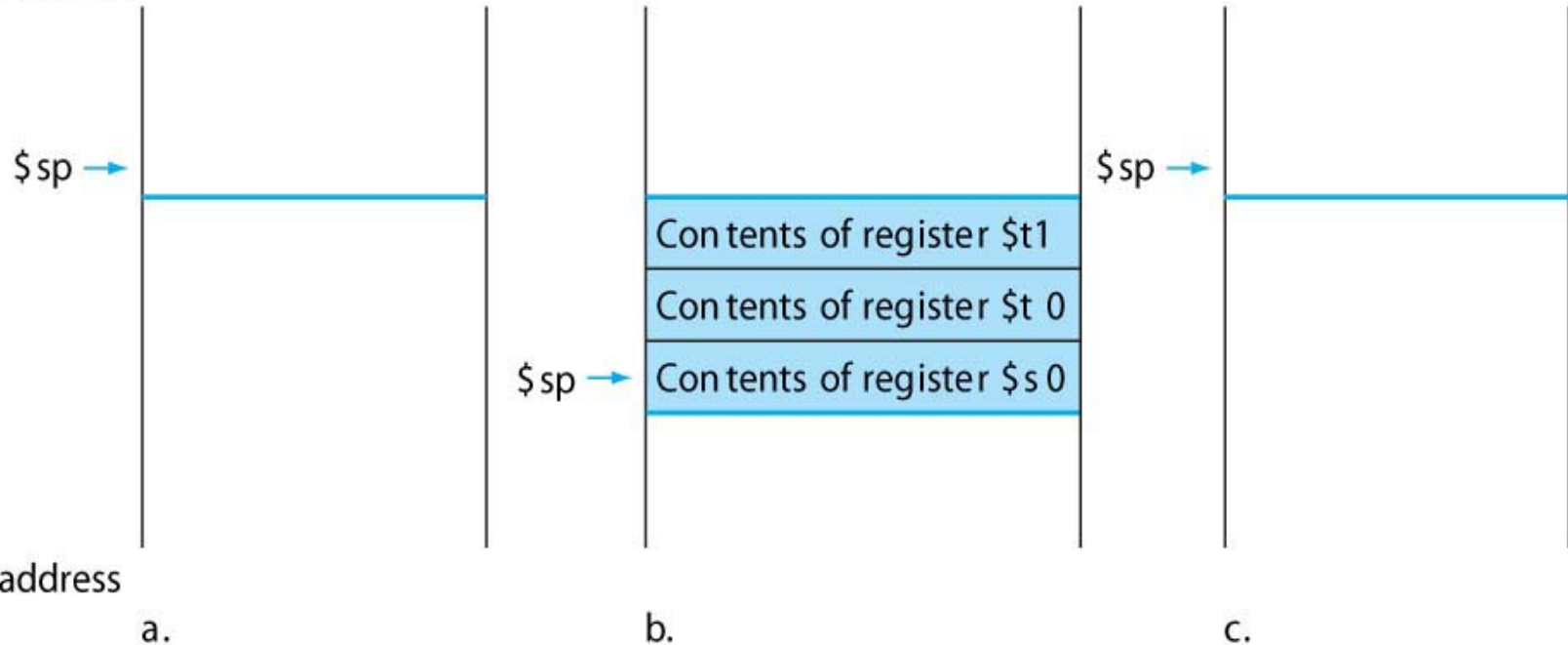
- Function call operations: jump and link  
`jal FunctionLabel` (J-type)
  - $\$ra = PC+4$ ; Address of following instruction put in  $\$ra$
  - $PC = \text{target address}$
- Function return operations: jump register  
`jr $ra` (R-type)
  - $PC = \$ra$ ; Copies  $\$ra$  to program counter
  - Can also be used for computed jumps (to any other register)
    - e.g., for case/switch statements

# Register Usage

- \$zero: constant 0 (reg 0)
- \$at: Assembler Temporary (reg 1)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$t0 – \$t9: temporaries (reg's 8 – 15)
  - Can be overwritten by callee
- \$s0 – \$s7: saved (reg's 16 – 23)
  - Must be saved/restored by callee
- \$t8, \$t9: temporaries (reg's 24 and 25)
- \$k0, \$k1: reserved for OS kernel (reg's 26 and 27)
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

# Uses of Stack in Function Call

High address



**Before calling**

**During function**

- For storing important registers
- For temporary variables

**After calling**

- Important registers restored
- Temporary variables destroyed

# Leaf Function

- Functions that don't call other functions

- C code:

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

- Assumptions:

- Arguments g, ..., j in \$a0, ..., \$a3
- f in \$s0 (need to save \$s0 before it's overwritten)
- Result in \$v0

# Leaf Function Example

## ■ MIPS code

leaf\_example:

addi \$sp, \$sp, -12 #create spaces on stack

*sw \$t1, 8(\$sp)* #store data on stack

*sw \$t0, 4(\$sp)*

sw \$s0, 0(\$sp)

add \$t0, \$a0, \$a1

add \$t1, \$a2, \$a3

sub \$s0, \$t0, \$t1

add \$v0, \$s0, \$zero

*lw \$s0, 0(\$sp)* #restore data from stack

*lw \$t0, 4(\$sp)*

*lw \$t1, 8(\$sp)*

addi \$sp, \$sp, 12 #destroy spaces on stack

jr \$ra #return from function

*Unnecessary*

# String Copy Example

- C code:

- Assuming null-terminated string

```
void strcpy (char x[], char y[])  
{ int i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

- Base addresses of x, y in \$a0, \$a1
- i in \$s0

# String Copy Example

## ■ MIPS code:

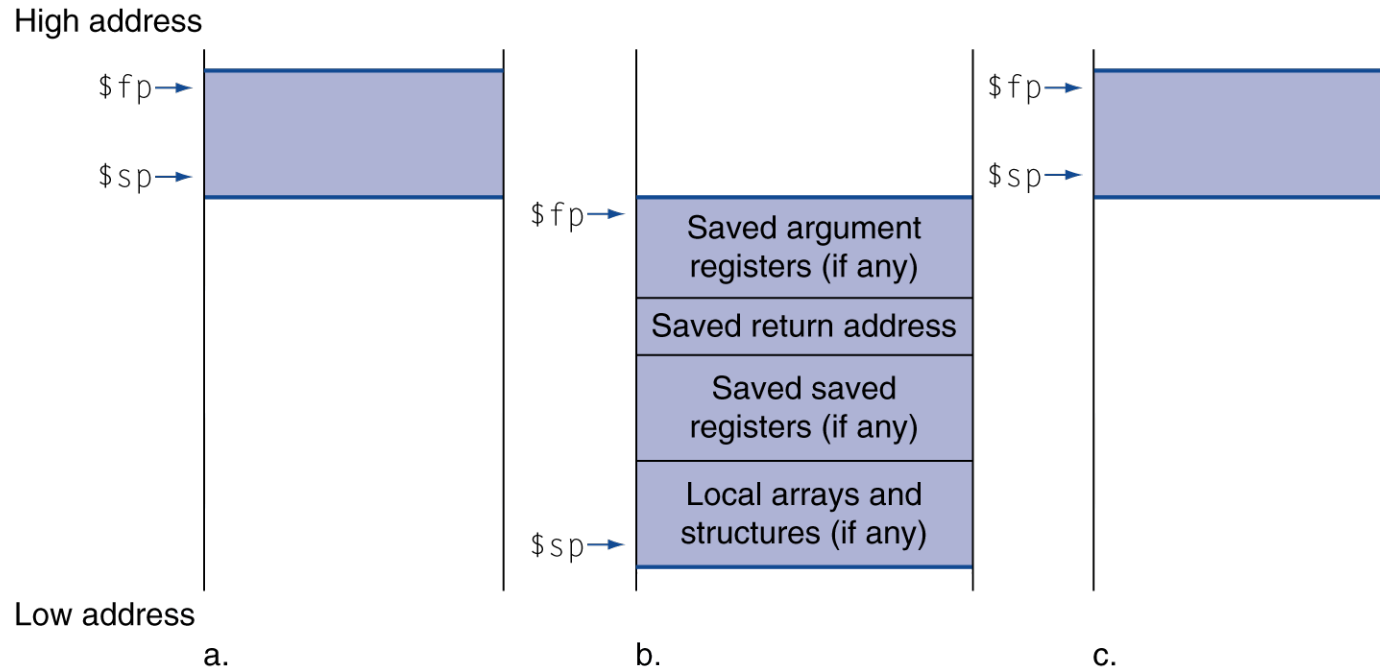
strcpy:		
	addi \$sp, \$sp, -4	# adjust stack for 1 item
	sw \$s0, 0(\$sp)	# save \$s0
	add \$s0, \$zero, \$zero	# i = 0
L1:	add \$t1, \$s0, \$a1	# addr of y[i] in \$t1
	lbu \$t2, 0(\$t1)	# \$t2 = y[i]
	add \$t3, \$s0, \$a0	# addr of x[i] in \$t3
	sb \$t2, 0(\$t3)	# x[i] = y[i]
	beq \$t2, \$zero, L2	# exit loop if y[i] == 0
	addi \$s0, \$s0, 1	# i = i + 1
	j L1	# next iteration of loop
L2:	lw \$s0, 0(\$sp)	# restore saved \$s0
	addi \$sp, \$sp, 4	# pop 1 item from stack
	jr \$ra	# and return

# Non-Leaf Functions

- Functions that call other functions
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call



# Local Data on the Stack



- Function frame (activation record)
  - Saved registers
  - Local data allocated by function
- Two pointers manages stack
  - \$sp manages frames
  - \$fp manages elements in each frame

# Non-Leaf Function Example

- C code:

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

- Argument n in \$a0
- Result in \$v0

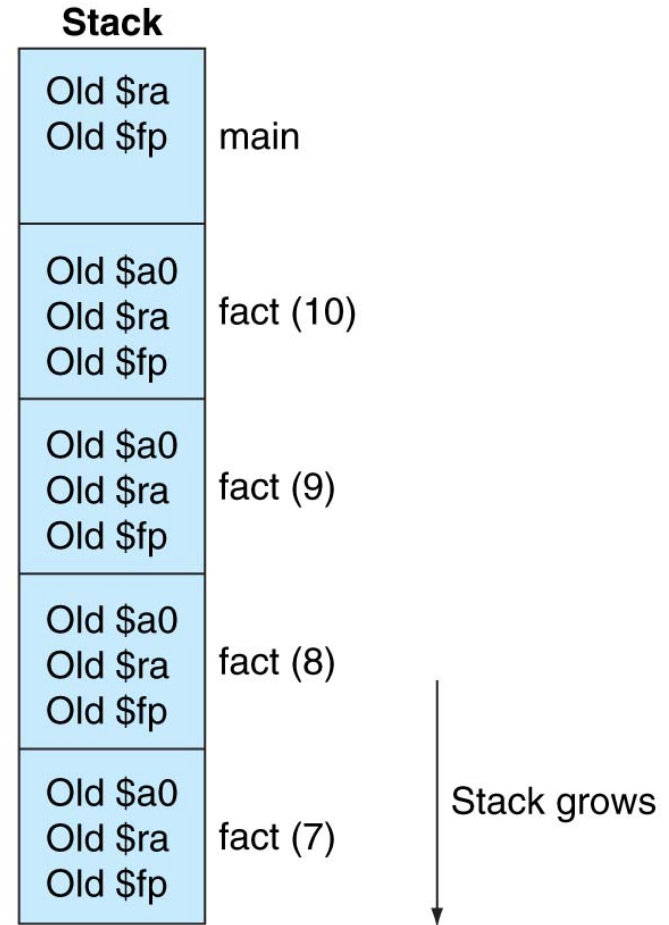
# Non-Leaf Function Example

- MIPS code:

fact:		
addi	\$sp, \$sp, -8	# adjust stack for 2 items
sw	\$ra, 4(\$sp)	# save return address
sw	\$a0, 0(\$sp)	# save argument
slti	\$t0, \$a0, 1	# test for n < 1
beq	\$t0, \$zero, L1	
addi	\$v0, \$zero, 1	# if so, result is 1
addi	\$sp, \$sp, 8	# release stack
jr	\$ra	# and return
L1:	addi \$a0, \$a0, -1	# else decrement n
	jal fact	# recursive call
lw	\$a0, 0(\$sp)	# restore original n
lw	\$ra, 4(\$sp)	# and return address
addi	\$sp, \$sp, 8	# pop 2 items from stack
mul	\$v0, \$a0, \$v0	# multiply to get result
jr	\$ra	# and return

# Usage of Stack Frames

- `fact (int n)` is a function, can be called recursively
- Note: `$fp` wasn't used in previous example



# Function Calling Convention

- Three places in function calling when conventions apply
  - Immediately before the function is called
  - In function, but before it starts executing
  - Immediately before the function finishes

# Function Calling Convention

- Before the function is called
  - Pass arguments to \$a0-\$a3
    - more arguments on stack
  - Save registers that should be saved by caller,
    - such as \$a0-\$a3 (non-leaf function), \$t0-\$t9 (if necessary)
  - `jal`

# Function Calling Convention

- Before function starts executing
  - Allocate memory of frame's size
    - by moving `$sp` downwards for frame's size
  - Save registers that should be saved by the function in the frame, before they are overwritten
    - `$s0-$s7` (if to be used), `$fp` (if used), `$ra` (non-leaf function),
  - Establish `$fp` (if desired),  $\text{\$fp} = \text{\$sp} + \text{frame's size} - 4$

# Function Calling Convention

- Before function finishes
  - If necessary, place function result to \$v0, \$v1
  - Restore registers saved by the function
    - Pop from frame
  - Destroy stack frame by moving \$sp upword
  - `jr $ra`



# C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap function (leaf)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in \$a0, k in \$a1, temp in \$t0

# The Function Swap

swap:	sll \$t1, \$a1, 2	# \$t1 = k * 4
	add \$t1, \$a0, \$t1	# \$t1 = v+(k*4)
		# (address of v[k])
	lw \$t0, 0(\$t1)	# \$t0 (temp) = v[k]
	lw \$t2, 4(\$t1)	# \$t2 = v[k+1]
	sw \$t2, 0(\$t1)	# v[k] = \$t2 (v[k+1])
	sw \$t0, 4(\$t1)	# v[k+1] = \$t0 (temp)
	jr \$ra	# return to calling routine

# The Sort Function in C

- Non-leaf function (calls swap)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in \$a0, n in \$a1, i in \$s0, j in \$s1

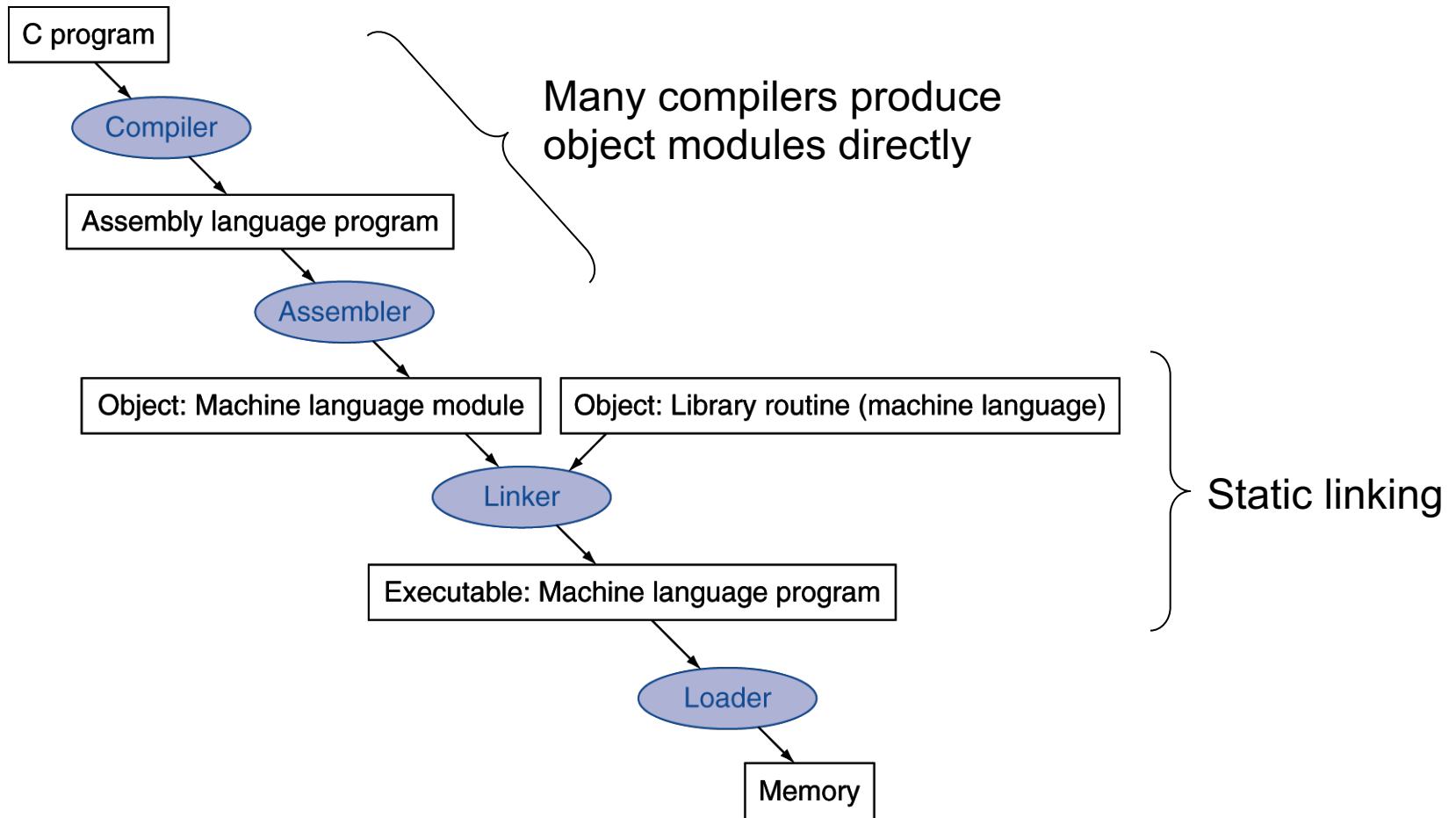
# The Full Function

sort:	addi \$sp,\$sp, -20	# make room on stack for 5 # registers
	sw \$ra, 16(\$sp)	# save \$ra on stack
	sw \$s3,12(\$sp)	# save \$s3 on stack
	sw \$s2, 8(\$sp)	# save \$s2 on stack
	sw \$s1, 4(\$sp)	# save \$s1 on stack
	sw \$s0, 0(\$sp)	# save \$s0 on stack
...		# function body
...		
exit1:	lw \$s0, 0(\$sp)	# restore \$s0 from stack
	lw \$s1, 4(\$sp)	# restore \$s1 from stack
	lw \$s2, 8(\$sp)	# restore \$s2 from stack
	lw \$s3,12(\$sp)	# restore \$s3 from stack
	lw \$ra,16(\$sp)	# restore \$ra from stack
	addi \$sp,\$sp, 20	# restore stack pointer
	jr \$ra	# return to calling routine

# The Function Body

	move \$s2, \$a0	# save \$a0 into \$s2	Move params
	move \$s3, \$a1	# save \$a1 into \$s3	
for1tst:	move \$s0, \$zero	# i = 0	Outer loop
	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 ( $i \geq n$ )	
for2tst:	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 ( $i \geq n$ )	Inner loop
	addi \$s1, \$s0, -1	# j = i - 1	
	slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 ( $j < 0$ )	
	bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 ( $j < 0$ )	
	sll \$t1, \$s1, 2	# \$t1 = j * 4	
	add \$t2, \$s2, \$t1	# \$t2 = v + (j * 4)	
	lw \$t3, 0(\$t2)	# \$t3 = v[j]	
	lw \$t4, 4(\$t2)	# \$t4 = v[j + 1]	
	slt \$t0, \$t4, \$t3	# \$t0 = 0 if \$t4 ≥ \$t3	Pass params & call
	beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
	move \$a0, \$s2	# 1st param of swap is v (old \$a0)	Inner loop
	move \$a1, \$s1	# 2nd param of swap is j	
	jal swap	# call swap function	
	addi \$s1, \$s1, -1	# j -= 1	Outer loop
	j for2tst	# jump to test of inner loop	
exit2:	addi \$s0, \$s0, 1	# i += 1	Outer loop
	j for1tst	# jump to test of outer loop	

# Translation and Startup



# Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - *Header*: described contents of object module
  - *Text segment*: translated instructions
  - *Static data segment*: data allocated for the life of the program
  - *Relocation info*: for contents that depend on absolute location of loaded program
  - *Symbol table*: global definitions and external refs
  - *Debug info*: for associating with source code

# Example of Object Modules

```
int X[100], Y[100];  
Procedure_A(int m)  
{ m = X[0];  
  ...  
  Procedure_B(...);  
  ...  
}
```



```
lw $a0, offset1($gp)  
jal Procedure_B  
...
```

```
Procedure_B(int n)  
{ Y[0] = n;  
  ...  
  Procedure_A(...);  
  ...  
}
```



```
sw $a1, offset2($gp)  
jal Procedure_A  
...
```

- m and n are parameters to the C functions
- Array X and Y are global variables
- By default \$gp = 1000 8000<sub>hex</sub>



# Example of Object Modules

Object file header			
	Name	Procedure A	
	Text size	100 <sub>hex</sub>	
	Data size	20 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	---	
	B	---	

Object file header			
	Name	Procedure B	
	Text size	200 <sub>hex</sub>	
	Data size	30 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	---	
	A	---	

# Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external references

# Example of Linked Objects

Executable File Header		
	Text size	300 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text Segment	Address	Instruction
	0040 0000 <sub>hex</sub>	lw \$a0, 8000 <sub>hex</sub> (\$gp)
	0040 0004 <sub>hex</sub>	jal 40 0100 <sub>hex</sub>
	...	...
	0040 0100 <sub>hex</sub>	sw \$a1, 8020 <sub>hex</sub> (\$gp)
	0040 0104 <sub>hex</sub>	jal 40 0000 <sub>hex</sub>
	...	...
Data Segment	Address	
	1000 0000 <sub>hex</sub>	(X)
	...	...
	1000 0020 <sub>hex</sub>	(Y)
	...	...

# Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including `$sp`, `$fp`, `$gp`)
  6. Jump to startup routine
    - Copies arguments to `$a0`, ... and calls `main`
    - When `main` returns, do `exit` syscall

# Dynamic Linking

- Only link/load library function when it is called
  - Requires function code to be relocatable
  - Avoids big executable caused by static linking of all referenced libraries
    - Some of them may be never used
  - Automatically picks up new library versions

# MIPS R2000 Assembly Language

- Appendix B.10