CprE 381: Computer Organization and Assembly Level Programming

MIPS Misc

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- HW3 due on Feb 11 at 11:59pm
- Exam 1: T-10 days

Why procedures?

- Procedures (subroutines, functions) allow the programmer to structure programs making them
  - easier to **understand and debug** and
  - allowing code to be **reused** (even across programmers and organizations)
- Procedures allow the programmer to concentrate on one portion of the code at a time
  - parameters act as **the interface** between the procedure and the rest of the program and data, allowing the procedure to be passed values (arguments) and to return values (results)
  - need a convention for this interface

Six Steps in Execution of a Procedure

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

WARNING: Lot’s of Moving Parts

Instruction for Calling a Procedure

- MIPS procedure call instruction:

```
jal ProcAddress #jump and link
```

- Saves PC+4 in register `$ra` as the link to the following instruction to set up the procedure return
- Machine format:

	6	26
Instruction Bits	**OP**	**target**
	Operation Code	Jump Target Address

- Then can do procedure return with just

```
jr  $ra       #return
```

Instruction for Calling a Procedure

- MIPS procedure call instruction:

```
jal ProcAddress #jump and link
```

- Saves PC+4 in register `$ra` as the link to the

In-class Assessment!

Access Code: sigh

Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating

Operation Code | Jump Target Address

- Then can do procedure return with just

```
jr  $ra         #return
```

Basic Procedure Flow

- For a procedure that computes the GCD of two values `i` (in `$t0`) and `j` (in `$t1`)

```
gcd(i,j);
```

- The caller puts the `i` and `j` (the parameter values) in `$a0` and `$a1` and issues a

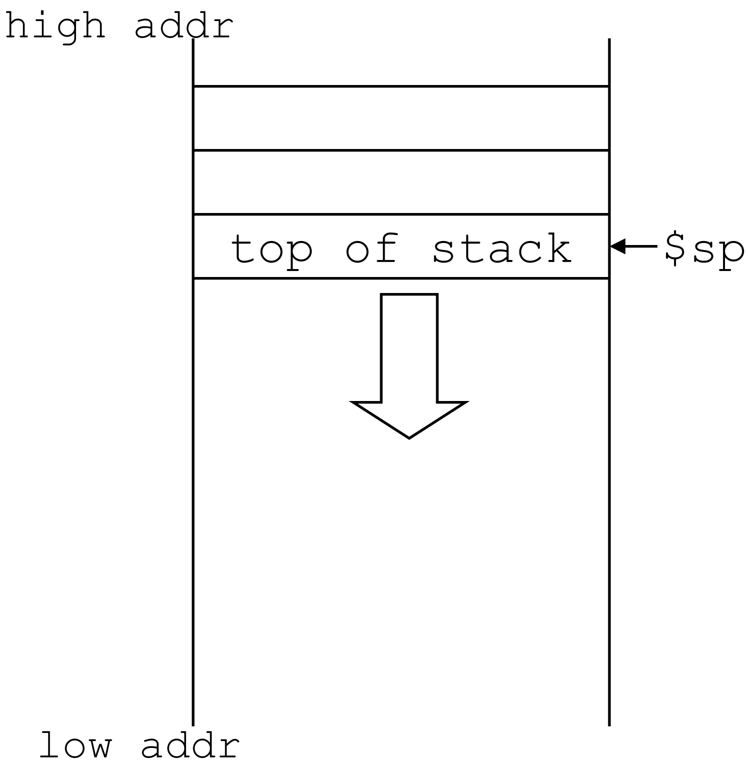
```
jal gcd    #jump to routine gcd
```

- The callee computes the GCD, puts the result in `$v0`, and returns control to the caller using

```
gcd: . . .      #code to compute gcd
  jr  $ra    #return
```

Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - callee uses a stack – a last-in-first-out structure



- One of the general registers, `$sp` (`$29`), is used to address the stack (which "grows" from high addresses to low addresses)
  - add data onto the stack – **push**

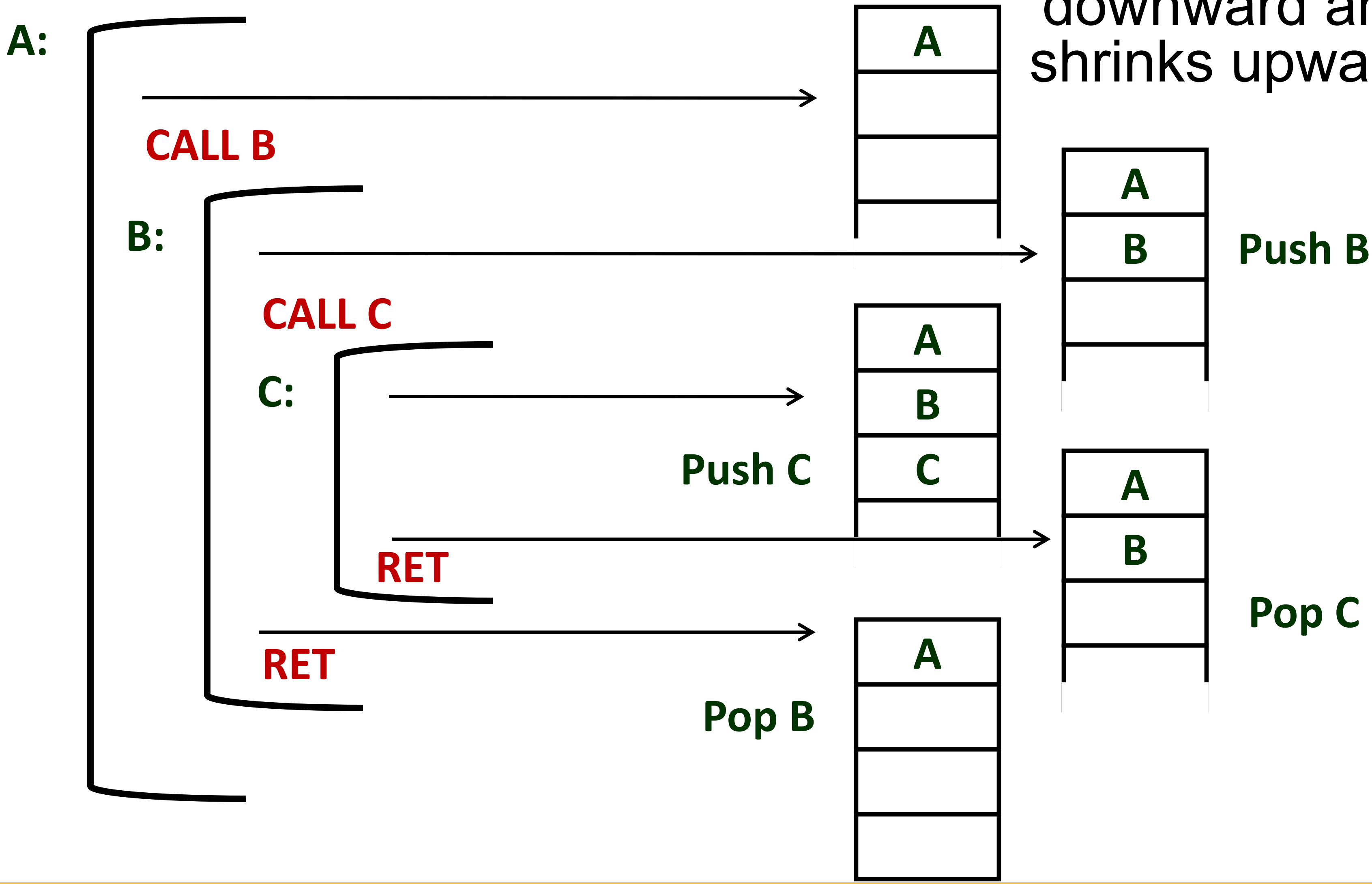
    `$sp` = `$sp` – 4 data on stack at new $sp

  - remove data from the stack – **pop**

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

Stack with Nested Calls

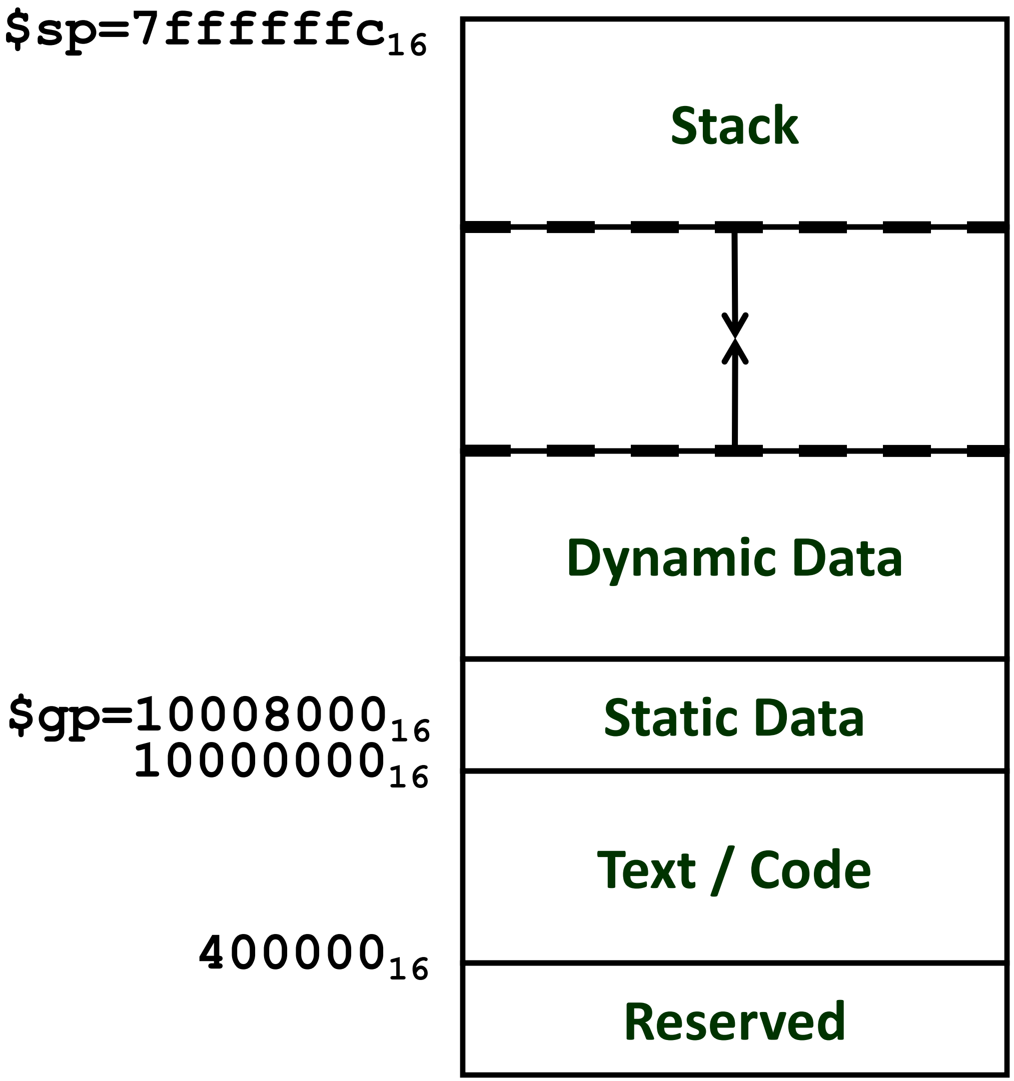
The stack grows downward and shrinks upward



Stacks

- Data is pushed onto the stack to store it and popped from the stack when no longer needed
  - MIPS does not support in hardware (use loads/stores)
  - Procedure calling convention requires one
- Calling convention
  - Common rules across procedures required
  - Recent machines are set by software convention and earlier machines by hardware instructions
- Using stacks
  - Stacks can grow up or down
  - Stack grows down in MIPS
  - Entire stack frame is pushed and popped, rather than single elements

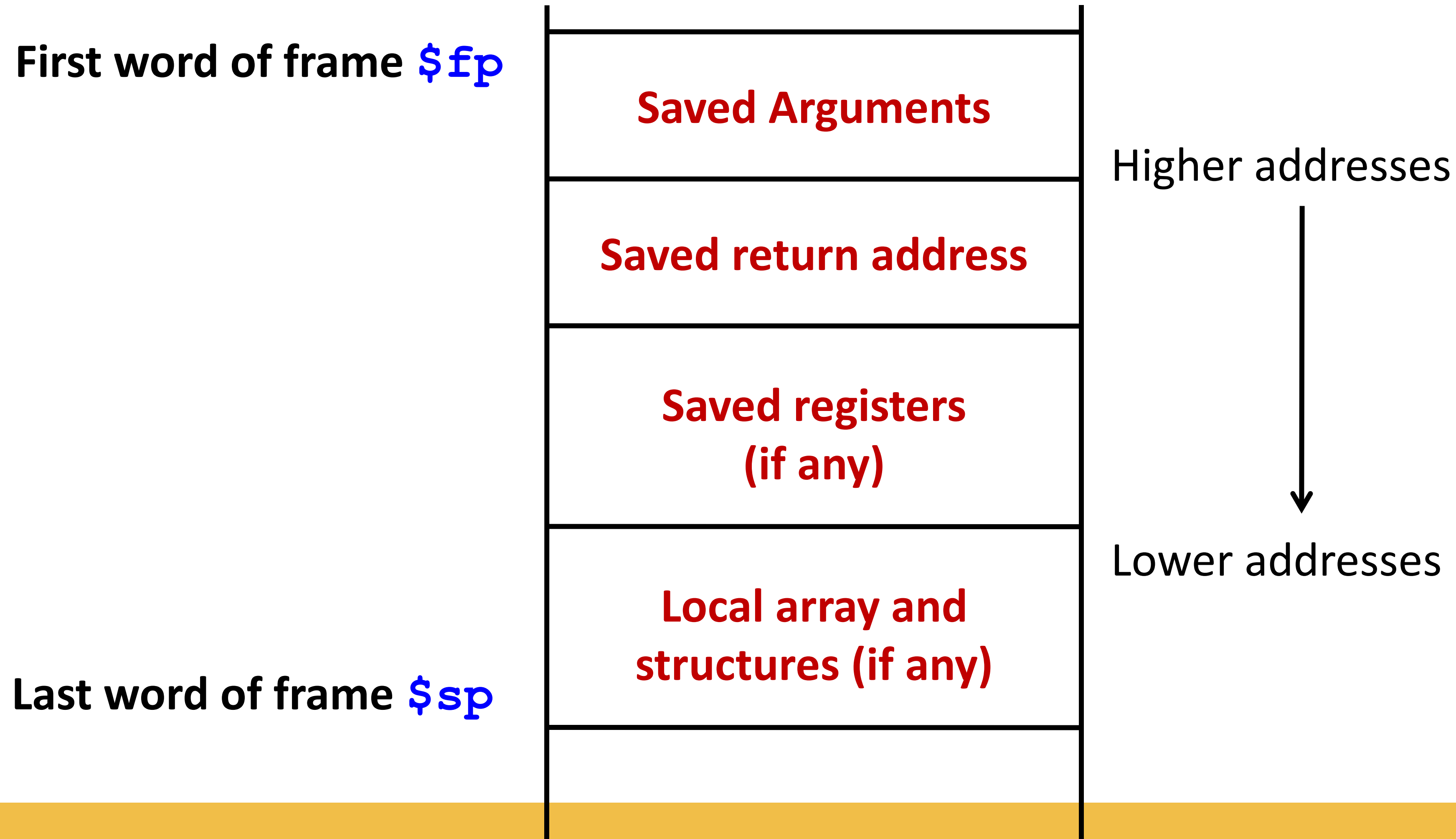
MIPS Storage Layout



- Stack and dynamic area grow towards one another to maximize storage before collision

Procedure Activation Record (Frame)

- Each procedure creates an activation record on the stack
  - P&H version differs from SGI / GCC compiler output



Register Assignments

Name	Register Number	Usage
`$zero`	0	the constant value 0
`$v0 – $v1`	2-3	values for results
`$a0 – $a3`	4-7	arguments
`$t0 – $t7`	8-15	temporaries
`$s0 – $s7`	16-23	saved registers
`$t8 – $t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Caller vs. Callee Saved Registers

- Preserved:
  - Saved registers (`$s0 – $s7`)
  - Stack/frame pointer (`$sp, $fp, $gp`)
  - Return address (`$ra`)
- Not preserved:
  - Temporary registers (`$t0 – $t9`)
  - Argument registers (`$a0 – $a3`)
  - Return values (`$v0 – $v1`)

- Preserved registers (Callee Save)
  - Save register values on stack prior to use
  - Restore registers before return
- Not preserved registers (Caller Save)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

A Simple Example

```
int foo(int num) {                int bar(int num) {
  return(bar(num + 1));             return(num + 1);
}  foo:                           }
      addiu $sp, $sp, -32      # push frame
      sw    $ra, 20($sp)       # Save $ra
      sw    $fp, 16($sp)       # Save $fp
      addiu $fp, $sp, 28       # Set new $fp
      addiu $a0, $a0, 1        # num + 1
      jal   bar                # call bar
      lw    $fp, 16($sp)       # Restore $fp
      lw    $ra, 20($sp)       # Restore $ra
      addiu $sp, $sp, 32       # pop frame
      jr    $ra                # return
   bar:
      addiu $v0, $a0, 1        # leaf procedure
      jr    $ra                # with no frame
```

A Simple Example

```
int foo(int num) {
  return(bar(num + 1));
}
```

```
int bar(int num) {
  return(num + 1);
}
```

```
foo:
    addiu $sp, $sp, -32     # push frame
```

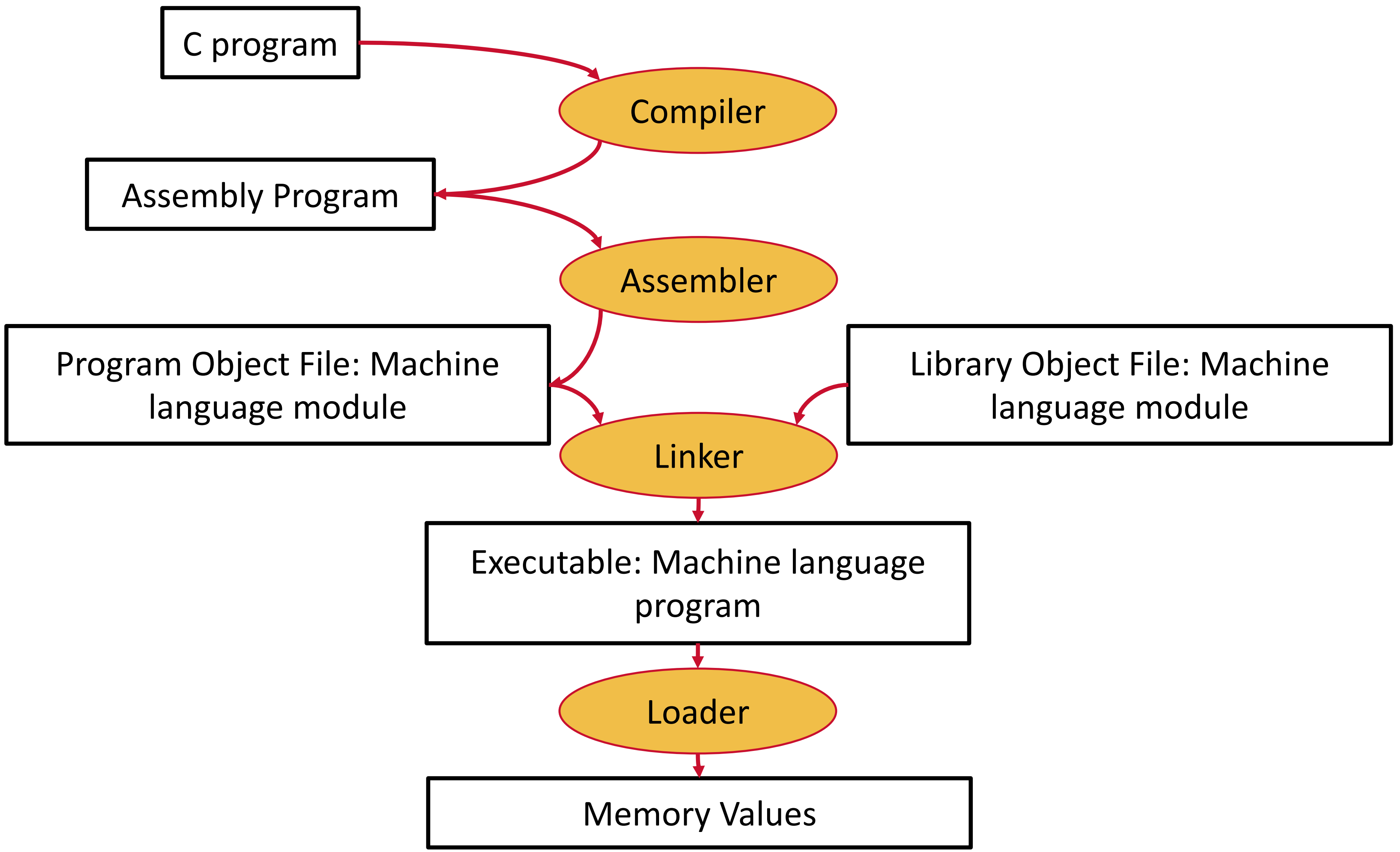
In-class Assessment!

Access Code: yikes!

Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating

```
    lw    $ra, 20($sp)      # Restore $ra
    addiu $sp, $sp, 32      # pop frame
    jr    $ra               # return
bar:
    addiu $v0, $a0, 1       # leaf procedure
    jr    $ra               # with no frame
```

Translation & Startup



Assembling

- Covert assembly code into 1's and 0's in memory
  - **Expand pseudoinstructions**
  - **Calculate offsets**

Pseudoinstructions

- Assembler expands pseudoinstructions

```
move $t0, $t1         # Copy $t1 to $t0

addu $t0, $zero, $t1 # Actual instruction
```

- Some pseudoinstructions need a temporary register:
  – Cannot use `$t`, `$s`, etc. since they may be in use
  – The `$at` register is reserved for the assembler

```
blt $t0, $t1, L1    # Goto L1 if $t0 < $t1

slt $at, $t0, $t1   # Set $at = 1 if $t0 < $t1
bne $at, $zero, L1 # Goto L1 if $at != 0
```

Register Assignments

Name	Register Number	Usage
`$zero`	0	the constant value 0
`$at`	1	temporary assembler
`$v0 – $v1`	2-3	values for results
`$a0 – $a3`	4-7	arguments
`$t0 – $t7`	8-15	temporaries
`$s0 – $s7`	16-23	saved registers
`$t8 – $t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

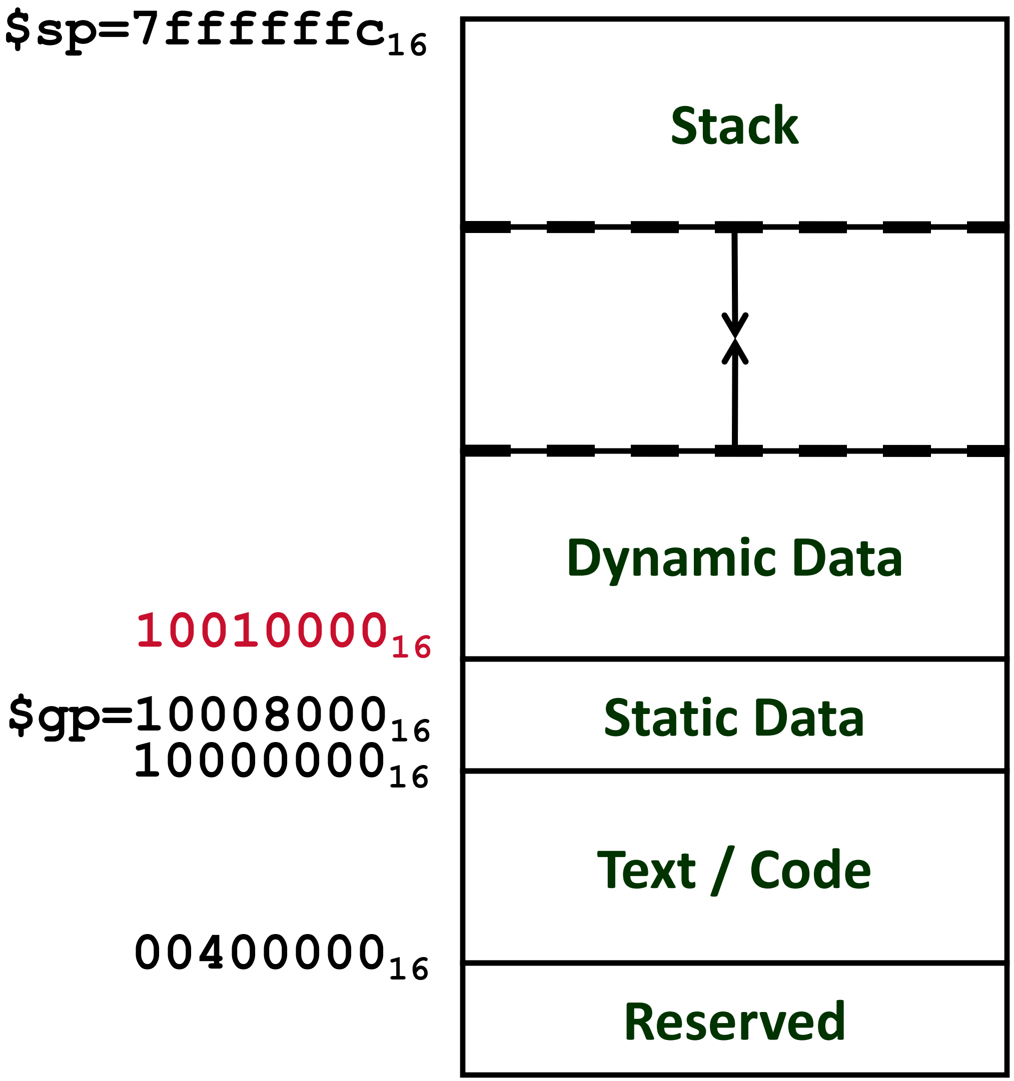
Assembler Pass 1

```
.data
ArrayA: .word 0,1,2,3,4,5,6,7,8,9
.text
   li $s0, 0 //i=0 → ori $s0, $zero, 0
   la $s1, ArrayA → lui $s1, ArrayAU
                    ori $s1, $s1, ArrayAL
LOOP:
   sll $t0, $s0, 2 //4*i
   add $t1, $t0, $s1 //addr of A[i]
   lw $t2, 0($t1) //load A[i]
   addi $t2, $t2, 1
   sw $t2, 0($t1) //store A[i]+1
   addi $s0, $s0, 1 //i++
   slti $t1, $s0, 10 //(i < 10)?
   bne $t1, $Zero, LOOP
```

Assembler Directives

- An operation that tells the assembler how to translate a program but does not produce machine instructions;
- always begins with a period.
- Examples: *.data <addr>, .text <addr>, .byte, .word*
- *.word 15, 20, 25 (3 words with values 15, 20, 25)*
- *.align 2 (align at 2^2 boundary – word)*
- *.extern myExternSymbol size; .globl myGlobalVar*
- *.space size (uninitialized space)*

MIPS Storage Layout



- Stack and dynamic area grow towards one another to maximize storage before collision

Assembler Pass 1

```
         .data
10010000 ArrayA: .word 0,1,2,3,4,5,6,7,8,9
         .text
00400000    li $s0, 0 //i=0 → ori $s0, $zero, 0
00400004    la $s1, ArrayA → lui $s1, ArrayAU
00400008                     ori $s1, $s1, ArrayAL
         LOOP:
0040000c    sll $t0, $s0, 2 //4*i
00400010    add $t1, $t0, $s1 //addr of A[i]
00400014    lw $t2, 0($t1) //load A[i]
00400018    addi $t2, $t2, 1
0040002c    sw $t2, 0($t1) //store A[i]+1
00400020    addi $s0, $s0, 1 //i++
00400024    slti $t1, $s0, 10 //(i < 10)?
00400028    bne $t1, $Zero, LOOP
```

Symbol	Address
ArrayA	??
LOOP	??

Assembler Pass 1

```
         .data
10010000 ArrayA: .word 0,1,2,3,4,5,6,7,8,9
         .text
00400000    li $s0, 0 //i=0 → ori $s0, $zero, 0
00400004    la $s1, ArrayA → lui $s1, ArrayAU
00400008                     ori $s1, $s1, ArrayAL
         LOOP:
0040000c    sll $t0, $s0, 2 //4*i
00400010    add $t1, $t0, $s1 //addr of A[i]
00400014    lw $t2, 0($t1) //load A[i]
00400018    addi $t2, $t2, 1
0040002c    sw $t2, 0($t1) //store A[i]+1
00400020    addi $s0, $s0, 1 //i++
00400024    slti $t1, $s0, 10 //(i < 10)?
00400028    bne $t1, $Zero, LOOP
```

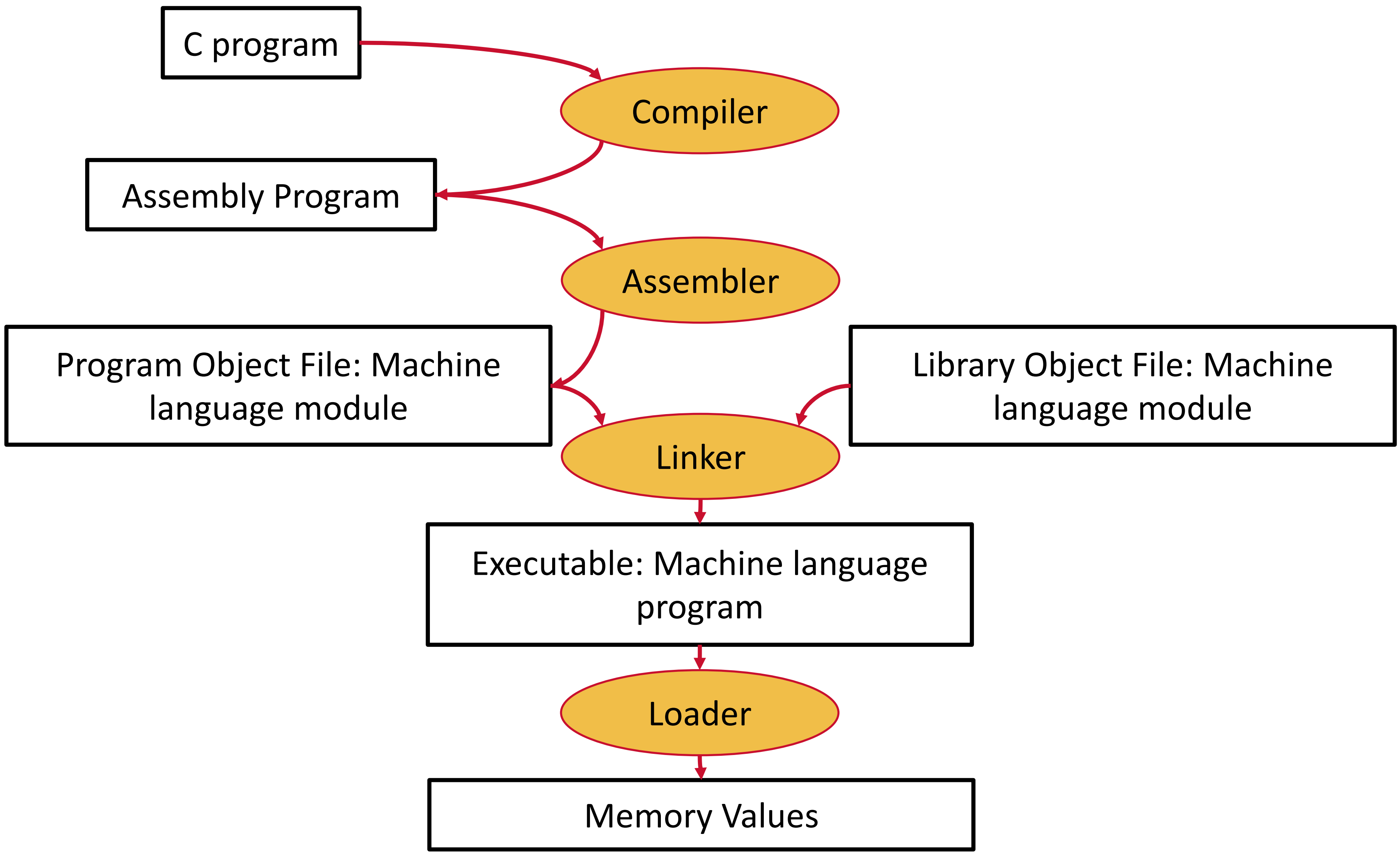
Symbol	Address
ArrayA	10010000
LOOP	004000c

Assembler Pass 2

```
         .data
10010000 ArrayA: .word 0,1,2,3,4,5,6,7,8,9
         .text
00400000    ori $s0, $zero, 0
00400004    lui $s1, 0x1001
00400008    ori $s1, $s1, 0x0000
         LOOP:
0040000c    sll $t0, $s0, 2 //4*i
00400010    add $t1, $t0, $s1 //addr of A[i]
00400014    lw $t2, 0($t1) //load A[i]
00400018    addi $t2, $t2, 1
0040002c    sw $t2, 0($t1) //store A[i]+1
00400020    addi $s0, $s0, 1 //i++
00400024    slti $t1, $s0, 10 //(i < 10)?
00400028    bne $t1, $Zero, 0xFFF8
```

Symbol	Address
ArrayA	10010000
LOOP	004000c

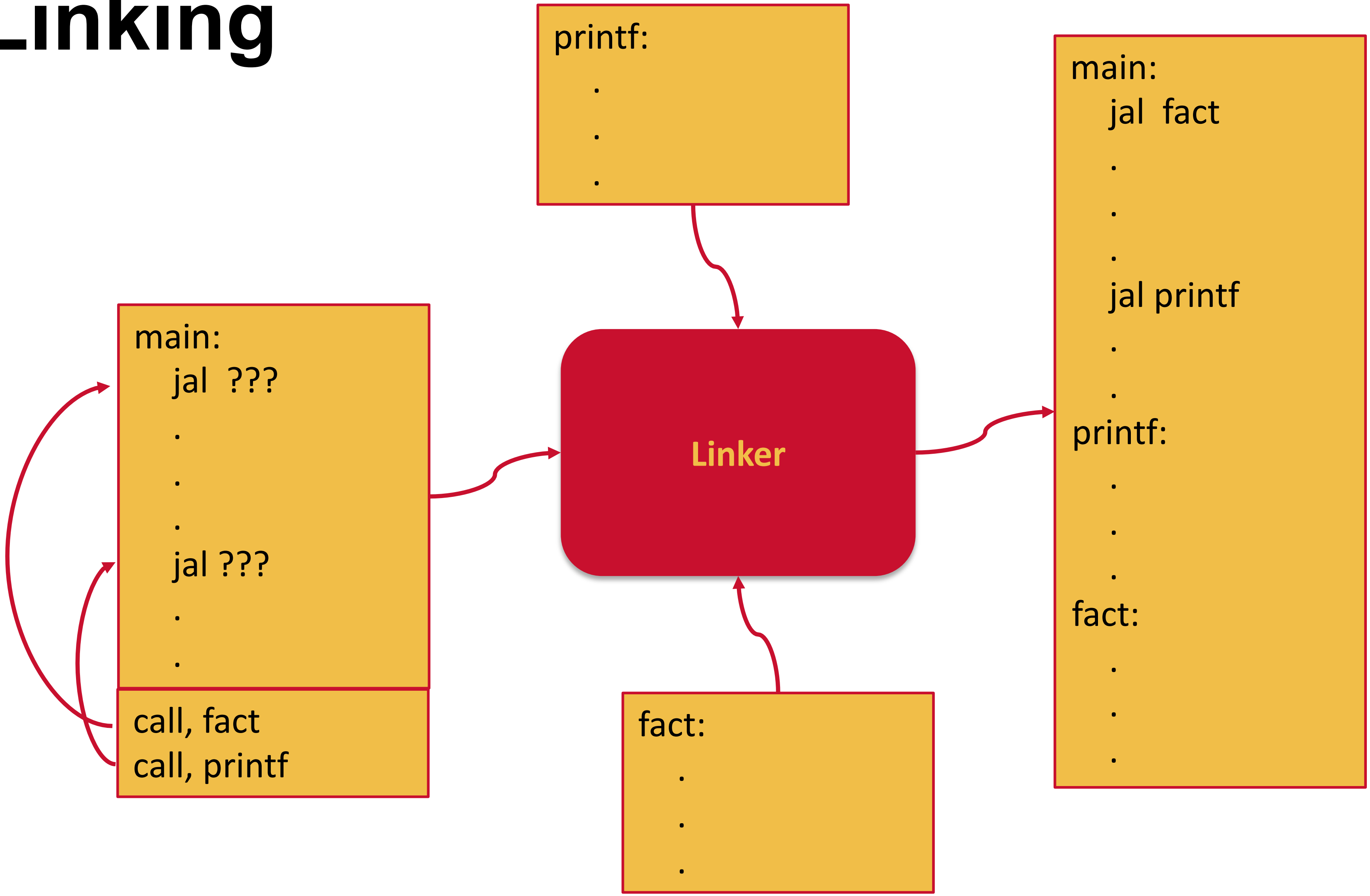
Translation & Startup



Object File

- 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

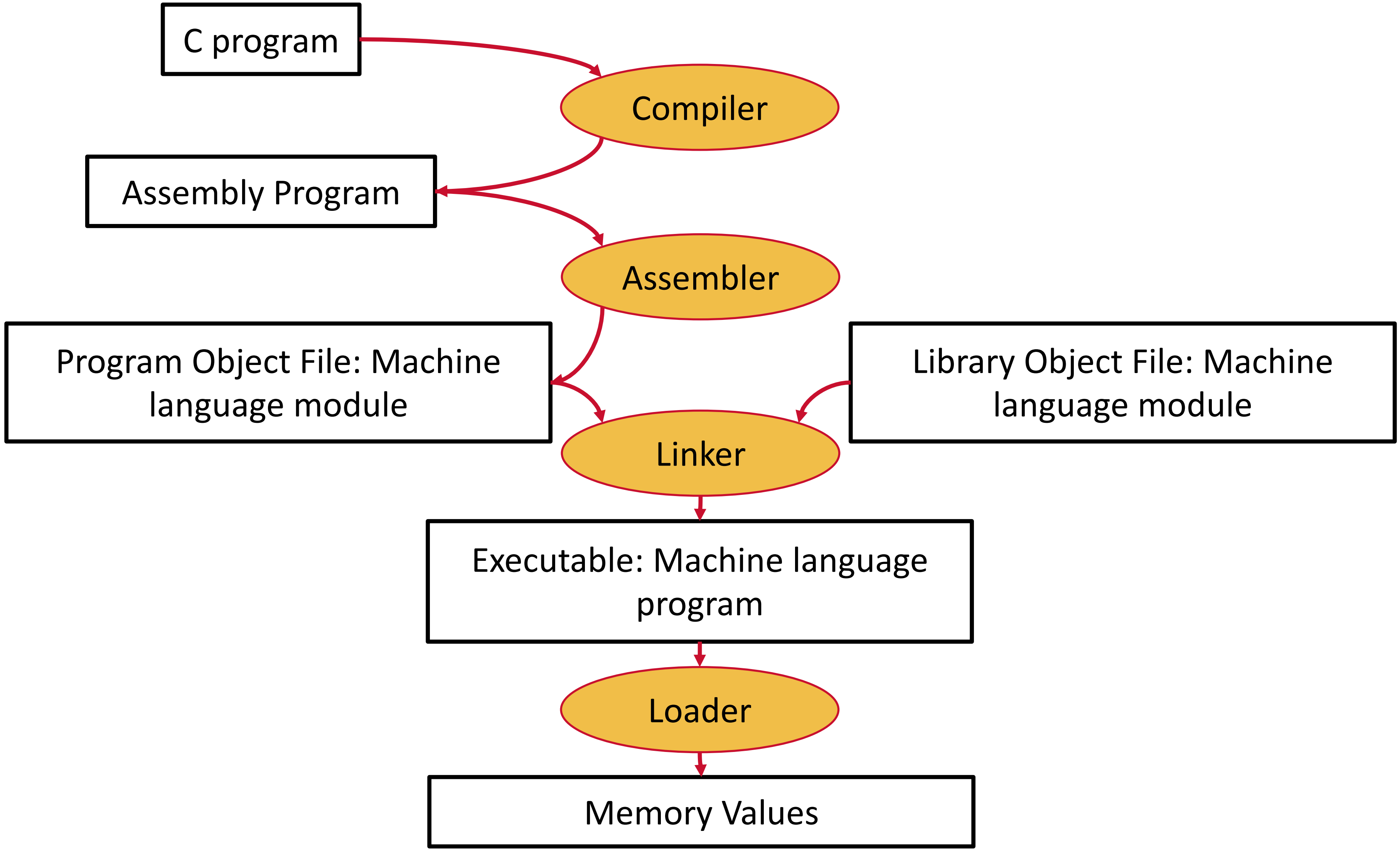
Linking


printf:
.
.
.
main:
jal ???
.
.
.
jal ???
.
.
call, fact
call, printf
Linker
fact:
.
.
.
main:
jal fact
.
.
.
jal printf
.
.
printf:
.
.
.
fact:
.
.
.

Loading

- 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

Translation & Startup



Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)