# EECE 2322: Fundamentals of Digital Design and Computer Organization

## Lecture 8_3: MIPS ISA

Xiaolin Xu
Department of ECE
Northeastern University

# How about "Working Together"?

- Asking Caller and callee to each be responsible for a specific set of registers
- *Caller* is responsible for saving and restoring any of the following caller-saved registers that it cares about.
  - **$t0-$t9 $a0-$a3 $v0-$v1**
  - **Callee may freely modify these registers, under the assumption that the caller already saved them if necessary.**
- *Callee* is responsible for saving and restoring any of the following callee-saved registers that it uses. (Remember that $ra is "used" by jal.)
  - **$s0-$s7 $ra**

# Summary

- Any values in the preserved registers MUST be saved-and-restored, while a function can change the non-preserved registers freely! —> making it more efficient
  - The **caller saves any non-preserved registers ($t0–$t9 and $a0–$a3)** that are needed after the call.
  - The **callee saves any of the pre- served registers ($s0–$s7 and $ra)** that it intends to modify

# Summary

- It is the **caller's responsibility** to save-and-restore any values in the **non-preserved registers**
- It is the **callee's responsibility** to save-and-restore any values in the **preserved registers**
- **Like priority over which type of registers)**

# Summary

- It is the **caller's responsibility** to save-and-restore any values in the **non-preserved registers**

- It is the **callee's** save-and-resto **preserved regi**

- **Like priority o registers)**

| Preserved *Callee-Saved* | Nonpreserved *Caller-Saved* |
|---|---|
| $s0-$s7 | $t0-$t9 |
| $ra | $a0-$a3 |
| $sp<br>stack above $sp | $v0-$v1<br>stack below $sp |

# Summary

- It is the **caller's responsibility** to save-and-restore any values in the **non-preserved registers**
- It is the **callee's** save-and-restor **preserved regi**
- **Like priority o registers)**

Tricky

| **Preserved** *Callee-Saved* | **Nonpreserved** *Caller-Saved* |
|---|---|
| `$s0-$s7` | `$t0-$t9` |
| `$ra` | `$a0-$a3` |
| `$sp`<br>stack above `$sp` | `$v0-$v1`<br>stack below `$sp` |

# Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # make space on stack to
           # store one register
  sw   $s0, 0($sp)    # save $s0 on stack
                      # no need to save $t0 or $t1
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4   # deallocate stack space
  jr   $ra            # return to caller
```

# Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # make space on stack to
                # store one register
  sw   $s0, 0($sp)   # save $s0 on stack
                     # no need to save $t0 or $t1
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)   # restore $s0 from stack
  addi $sp, $sp, 4   # deallocate stack space
  jr   $ra           # return to caller
```

# Recursive Function Call

## High-level code

```
int factorial(int n)
   {
  if (n <= 1)
    return 1;
  else
    return (n *
   factorial(n-1));
}
```

## MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC            addi $sp, $sp, 8   # restore $sp
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC            lw   $ra, 0($sp)   # restore $ra
0xC0            lw   $a0, 4($sp)   # restore $a0
0xC4            addi $sp, $sp, 8   # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

# Recursive Function Call

## High-level code

```
int factorial(int n)
  {
  if (n <= 1)
    return 1;
  else
    return (n *
   factorial(n-1));
}
```

## MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC            addi $sp, $sp, 8   # restore $sp
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC            lw   $ra, 0($sp)   # restore $ra
0xC0            lw   $a0, 4($sp)   # restore $a0
0xC4            addi $sp, $sp, 8   # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

# Recursive Function Call

## High-level code

```
int factorial(int n)
   {
  if (n <= 1)
    return 1;
  else
    return (n *
   factorial(n-1));
}
```

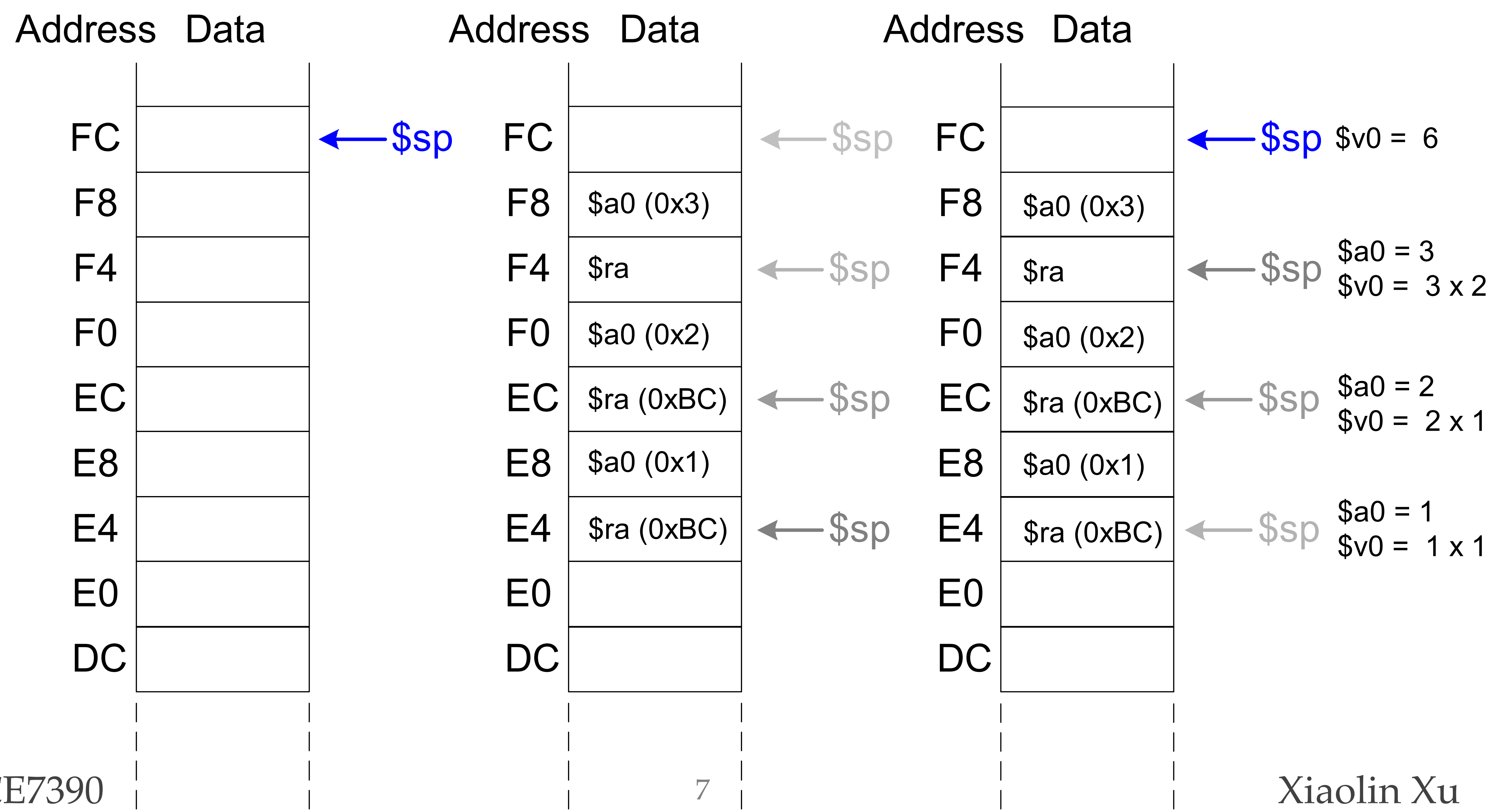
Why?

## MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4      else: addi $a0, $a0, -1   # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra            # return
```

# Stack During Recursive Call

- The stack when executing factorial(3), i.e., n=3



# Function Call Take-away

- **Caller**
  - Put arguments in $a0-$a3
  - Save any needed registers ($ra, maybe $t0-t9)
  - jal callee
  - Restore registers
  - Look for result in $v0
- **Callee**
  - Save registers that might be disturbed ($s0-$s7)
  - Perform function
  - Put result in $v0
  - Restore registers
  - jr $ra

# MIPS Addressing Mode

## Five operand addressing modes

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

# Addressing Mode: R/W Operands

## Register Only —> All R-type instructions (note jar)

- Operands found in registers
  - **Example:** `add $s0, $t2, $t3`
  - **Example:** `sub $t8, $s1, $0`

## Immediate —> I-type instructions

- 16-bit immediate used as an operand
  - **Example:** `addi $s4, $t5, -73`
  - **Example:** `ori  $t3, $t7, 0xFF`

# Addressing Mode: R/W Operands

## Base Addressing

- Address of operand is:

  `base address + sign-extended immediate`

  - **Example:** `lw  $s4, 72($0)`
    - address = `$0 + 72`
  - **Example:** `sw  $t2, -25($t1)`
    - address = `$t1 - 25`

# Addressing Mode: R/W Operands

## Base Addressing

- Address of operand is:
  `base address + sign-extended immediate`

  - **Example:** `lw $s4, 72($0)`
    - address = `$0 + 72`

  - **Example:** `sw $t2, -25($t1)`
    - address = `$t1` $\boxed{\texttt{- 25}}$

# Addressing Mode: R/W Operands

## Base Addressing

- Address of operand is:

  `base address + sign-extended immediate`

  - **Example:** `lw   $s4, 72($0)`
    - address = `$0 + 72`
  - **Example:** `sw   $t2, -25($t1)`
    - address = `$t1 - 25`

Ok?

# Addressing Mode: Writing Program Counter and Label

```
0x10            beq   $t0, $0, else
0x14            addi  $v0, $0, 1
0x18            addi  $sp, $sp, i
0x1C            jr     $ra
0x20      else:     addi  $a0, $a0, -1
0x24            jal   factorial
```

**Assembly Code** | **Field Values**

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

| op | rs | rt | imm |
|---|---|---|---|
| 4 | 8 | 0 | 3 |

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

**The imm = 3 denotes the number of instructions *between the branch and target instruction***

# Addressing Mode: Writing Program Counter and Label

```
0x10          beq   $t0, $0, else
0x14          addi  $v0, $0, 1
0x18          addi  $sp, $sp, i
0x1C          jr     $ra
0x20   else:     addi  $a0, $a0, -1
0x24          jal   factorial
```

| Assembly Code | Field Values | | | | | |
|---|---|---|---|---|---|---|
| | op | rs | rt | imm | | |
| beq $t0, $0, else<br>(beq $t0, $0, 3) | 4 | 8 | 0 | 3 | | |
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

**The imm = 3 denotes the number of instructions *between the branch and target instruction***

# Practice: PC-Relative Addressing

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

| Assembly Code | Field Values | | | |
|---|---|---|---|---|
| | op | rs | rt | imm |
| `bne $t1, $0, loop` | 5 | 9 | 0 | |
| | 6 bits | 5 bits | 5 bits | 16 bits |

# Practice: PC-Relative Addressing

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

**Assembly Code** / **Field Values**

| Assembly Code | op | rs | rt | imm |
|---|---|---|---|---|
| `bne $t1, $0, loop` | 5 | 9 | 0 | -6 |
| | 6 bits | 5 bits | 5 bits | 16 bits |

# Addressing Mode: Pseudo-direct Addressing

- Why pseudo-direct Addressing: no enough address bit!
  - Specifically used for J-type instructions , j and jal
- Jump target address (JTA) needs 32-bit, but only 26-bit available. How to achieve this?
  - The two least significant bits (1:0) of JTA are 0s and can be saved

# Addressing Mode: Pseudo-direct Addressing

- Why pseudo-direct Addressing: no enough address bit!
  - Specifically used for J-type instructions , j and jal
- Jump target address (JTA) needs 32-bit, but only 26-bit available. How to achieve this?
  - The two least significant bits (1:0) of JTA are 0s and can be saved
  - The middle 27:2 are directly applied

# Addressing Mode: Pseudo-direct Addressing

- Why pseudo-direct Addressing: no enough address bit!
  - Specifically used for J-type instructions , j and jal
- Jump target address (JTA) needs 32-bit, but only 26-bit available. How to achieve this?
  - The two least significant bits (1:0) of JTA are 0s and can be saved
  - The middle 27:2 are directly applied
  - The four most significant bits (31:28) are borrowed from PC+4

# Example: Pseudo-direct Addressing

## Pseudo-direct Addressing

```
0x0040005C          jal    sum
...
0x004000A0   sum:   add    $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

**Field Values**

| op | imm |
|---|---|
| 3 | 0x0100028 |
| 6 bits | 26 bits |

**Machine Code**

| op | addr | |
|---|---|---|
| 000011 | 00 0001 0000 0000 0000 0010 1000 | (0x0C100028) |
| 6 bits | 26 bits | |

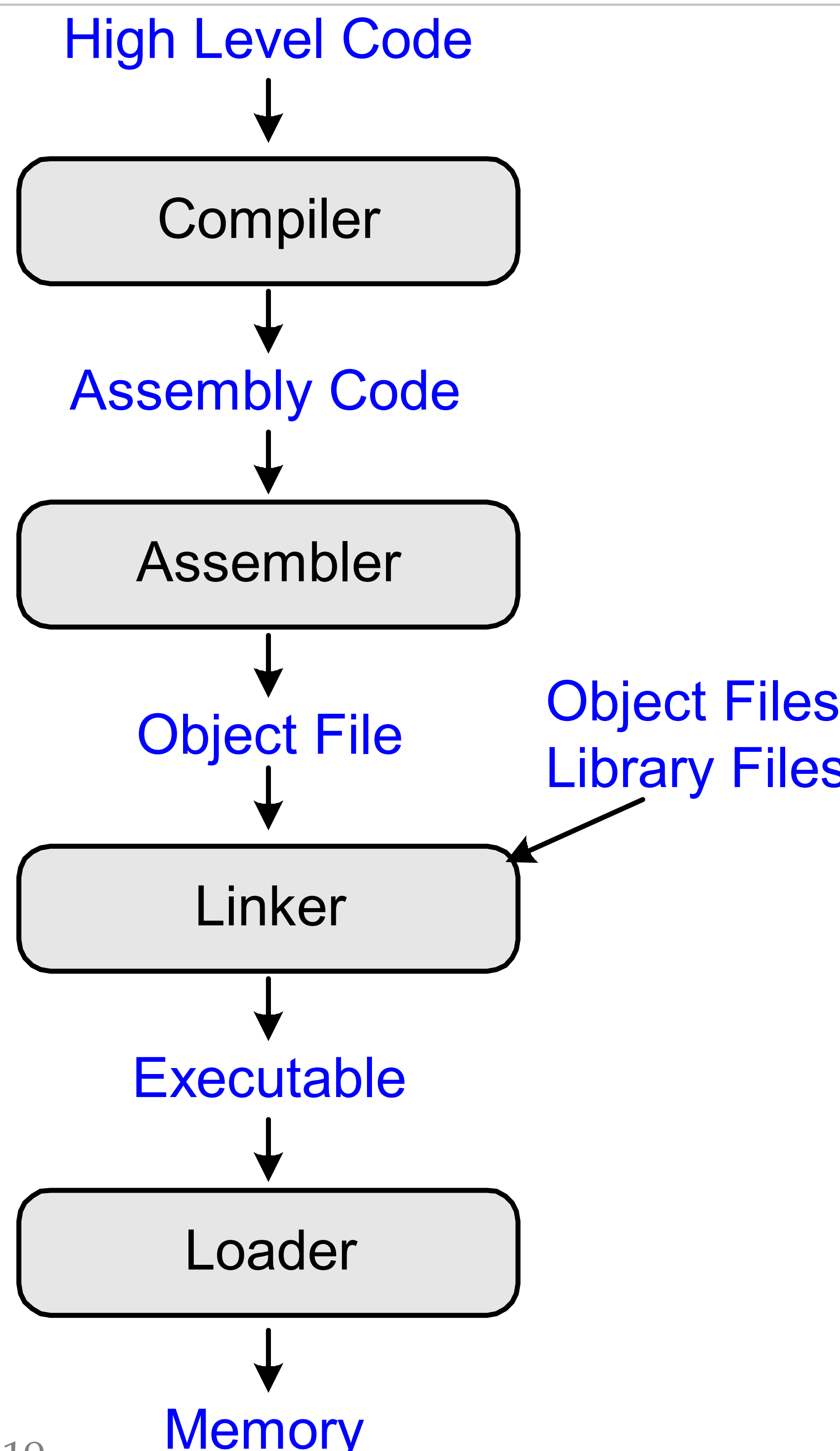
# Summary: Pseudo-direct Addressing

- The effective address will always be word-aligned
  - The two least significant bits are 00
- The range of jump target is constrained
  - Anywhere within the current 256 MB block of code
  - Since the upper 4 bits of the PC are used
- What if to jump anywhere within the 4 GB space

# Summary: Pseudo-direct Addressing

- The effective address will always be word-aligned
  - The two least significant bits are 00
- The range of jump target is constrained
  - Anywhere within the current 256 MB block of code
  - Since the upper 4 bits of the PC are used
- What if to jump anywhere within the 4 GB space
  - R-type instructions jr and jalr are used , where the complete 32 - bit target address is specified in a register

# Compile & Run a Program



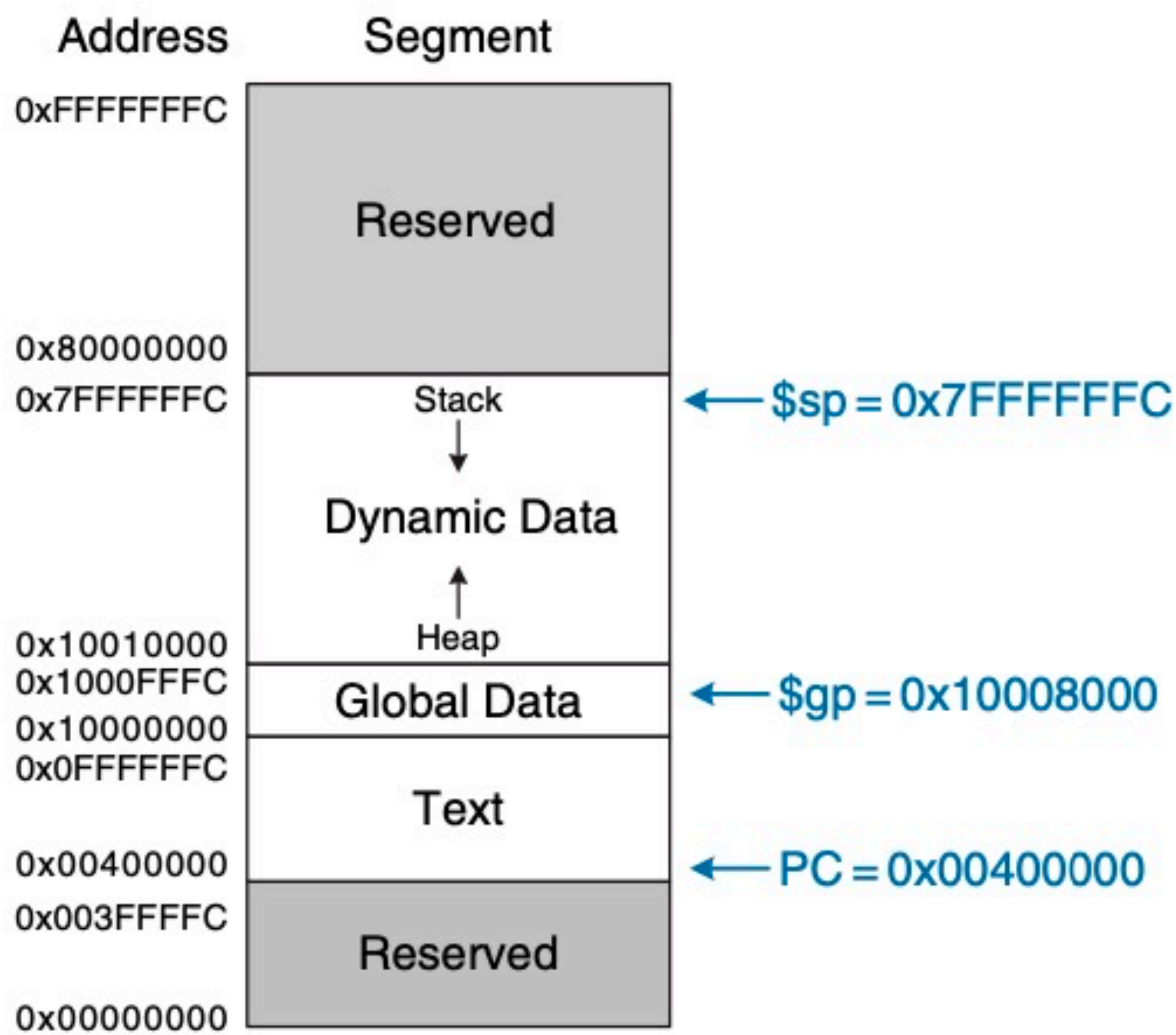
# What is Stored in Memory?

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

- How big is memory?
  - At most $2^{32}$ = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

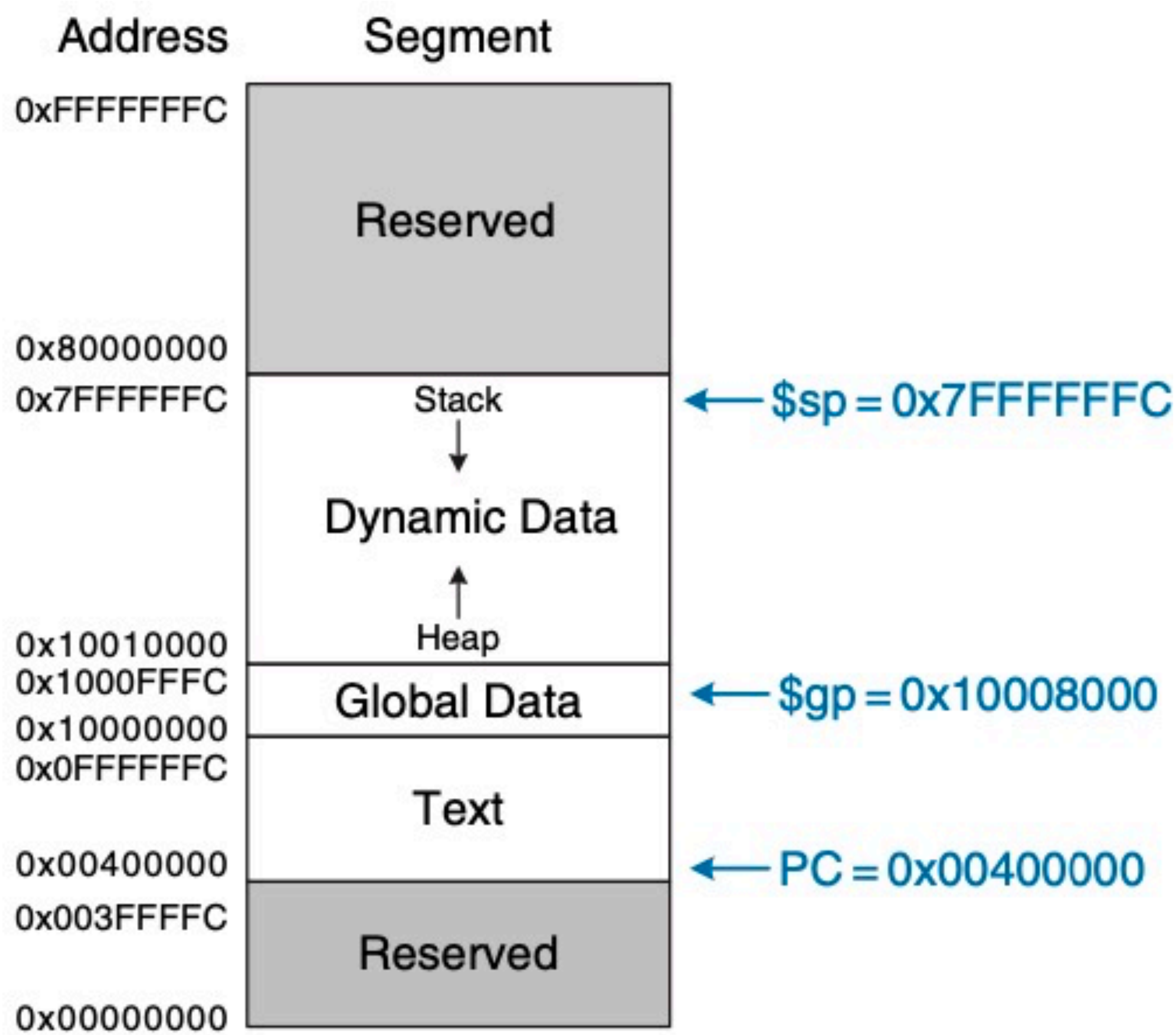
# MIPS Memory Map

- Why 0xFFFFFFFC?
- **Text segment stores the machine language program.**
  - 256 MB of code
  - Four most significant bits of the address in the text space are all 0



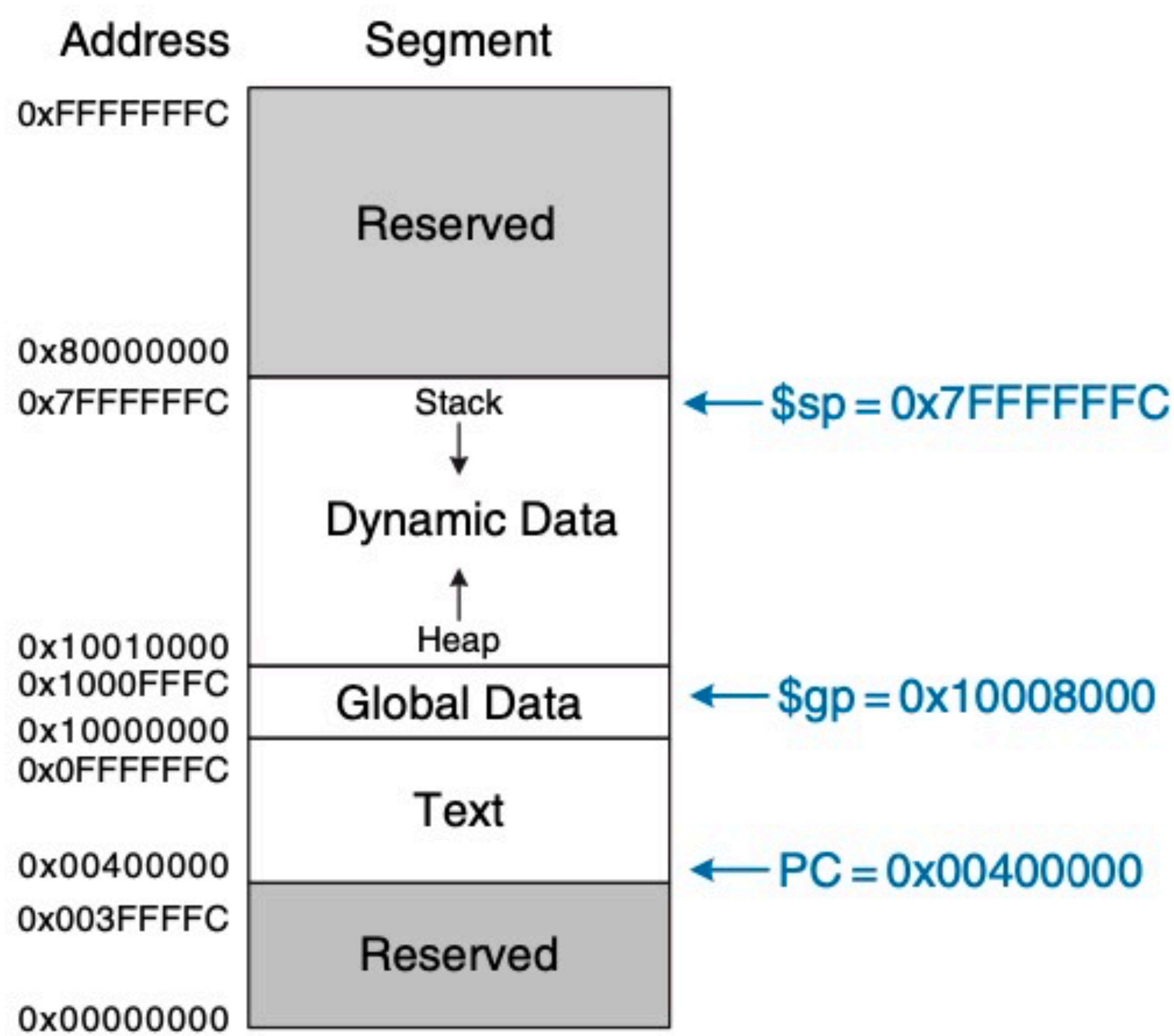
# MIPS Memory Map

- Why 0xFFFFFFFC?
- **Text segment stores the machine language program.**
  - 256 MB of code
  - Four most significant bits of the address in the text space are all 0
  - j instruction can directly jump to any address in the program.



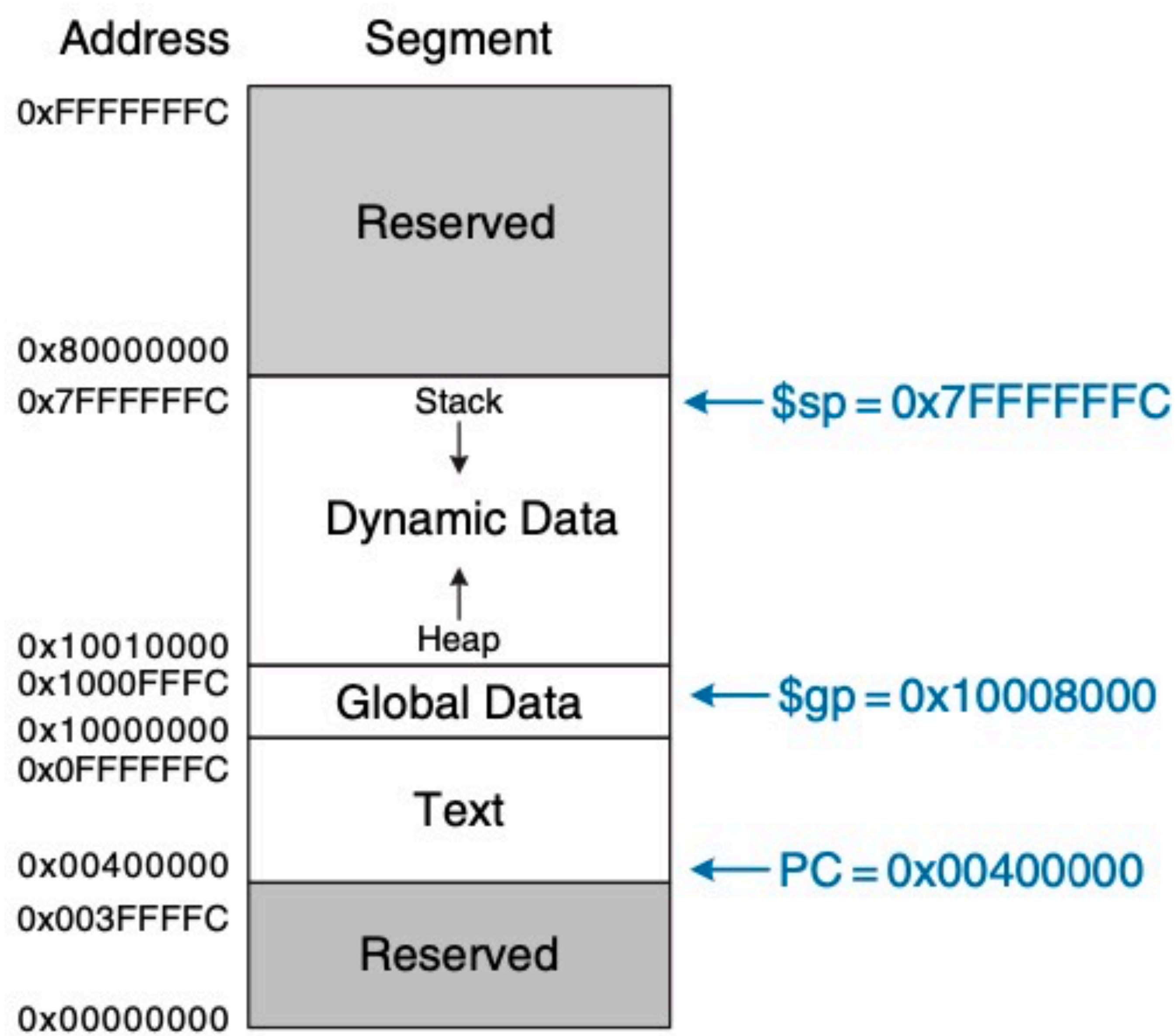
# MIPS Memory Map

- **Global data segment** stores global variables that, can be seen by all functions in a program
- **Dynamic data segment** holds the stack and the heap
  - The data are not known at start-up
  - Dynamically allocated and deallocated throughout the execution of the program



# MIPS Memory Map

- The **reserved** segments are used by the operating system and cannot directly be used by the program



# Example Program: C Code

```c
int f, g, y;  // global variables

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

# Step1: Compilation

```c
int f, g, y;  // global



int main(void)
{




  f = 2;
  g = 3;


  y = sum(f, g);
  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

# Step 2: Assembling

- Assembler turns the assembly language code into an object file containing machine language code
- Two passes:
  - 1, assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names

| Symbol | Address |
|---|---|
| `f` | 0x10000000 |
| `g` | 0x10000004 |
| `y` | 0x10000008 |
| `main` | 0x00400000 |
| `sum` | 0x0040002C |

# Step 2: Assembling

- Assembler turns the assembly language code into an object file containing machine language code
- Two passes:
  - 1, assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names

```
0x00400000 main: addi $sp, $sp, -4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

| Symbol | Address |
|---|---|
| f | 0x10000000 |
| g | 0x10000004 |
| y | 0x10000008 |
| main | 0x00400000 |
| sum | 0x0040002C |

# Step 2: Assembling

- Two passes:
  - 1, assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names
  - 2, the assembler produces the machine language code —> stored in the object file

# Step 2: Assembling

- Two passes:
  - 1, assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names
  - 2, the assembler produces the machine language code — > stored in the object file

```
0x00400000 main: addi $sp, $sp, -4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

# Step 3: Linking

- Most large programs contain **more than one file**
- Combine all of the object files into one machine language file called the executable
- Uses the information in the symbol tables to adjust the addresses of global variables and of labels that are relocated

| Executable file header | Text Size | Data Size | |
|---|---|---|---|
| | 0x34 (52 bytes) | 0xC (12 bytes) | |
| **Text segment** | **Address** | **Instruction** | |
| | 0x00400000 | 0x23BDFFFC | addi \$sp, \$sp, –4 |
| | 0x00400004 | 0xAFBF0000 | sw \$ra, 0(\$sp) |
| | 0x00400008 | 0x20040002 | addi \$a0, \$0, 2 |
| | 0x0040000C | 0xAF848000 | sw \$a0, 0x8000(\$gp) |
| | 0x00400010 | 0x20050003 | addi \$a1, \$0, 3 |
| | 0x00400014 | 0xAF858004 | sw \$a1, 0x8004(\$gp) |
| | 0x00400018 | 0x0C10000B | jal 0x0040002C |
| | 0x0040001C | 0xAF828008 | sw \$v0, 0x8008(\$gp) |
| | 0x00400020 | 0x8FBF0000 | lw \$ra, 0(\$sp) |
| | 0x00400024 | 0x23BD0004 | addi \$sp, \$sp, –4 |
| | 0x00400028 | 0x03E00008 | jr \$ra |
| | 0x0040002C | 0x00851020 | add \$v0, \$a0, \$a1 |
| | 0x00400030 | 0x03E00008 | jr \$ra |
| **Data segment** | **Address** | **Data** | |
| | 0x10000000 | f | |
| | 0x10000004 | g | |
| | 0x10000008 | y | |

# Example Program: Executable

| Executable file header | Text Size | Data Size | |
|---|---|---|---|
| | 0x34 (52 bytes) | 0xC (12 bytes) | |
| **Text segment** | **Address** | **Instruction** | |
| | 0x00400000 | 0x23BDFFFC | addi $sp, $sp, -4 |
| | 0x00400004 | 0xAFBF0000 | sw $ra, 0 ($sp) |
| | 0x00400008 | 0x20040002 | addi $a0, $0, 2 |
| | 0x0040000C | 0xAF848000 | sw $a0, 0x8000 ($gp) |
| | 0x00400010 | 0x20050003 | addi $a1, $0, 3 |
| | 0x00400014 | 0xAF858004 | sw $a1, 0x8004 ($gp) |
| | 0x00400018 | 0x0C10000B | jal 0x0040002C |
| | 0x0040001C | 0xAF828008 | sw $v0, 0x8008 ($gp) |
| | 0x00400020 | 0x8FBF0000 | lw $ra, 0 ($sp) |
| | 0x00400024 | 0x23BD0004 | addi $sp, $sp, -4 |
| | 0x00400028 | 0x03E00008 | jr $ra |
| | 0x0040002C | 0x00851020 | add $v0, $a0, $a1 |
| | 0x00400030 | 0x03E00008 | jr $ra |
| **Data segment** | **Address** | **Data** | |
| | 0x10000000 | f | |
| | 0x10000004 | g | |
| | 0x10000008 | y | |

# Step 4: Loading