MIPS Programming

Design of Digital Circuits 2014
Srdjan Capkun
Frank K. Gürkaynak

http://www.syssec.ethz.ch/education/Digitaltechnik_14

In This Lecture

- Small review from last week
- Programming (continued)
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembling, and Loading
- Odds and Ends

Assembly Language

- **To command a computer, you must understand its language**
  - ***Instructions:*** words in a computer's language
  - ***Instruction set:*** the vocabulary of a computer's language
- **Instructions indicate the operation to perform and the operands to use**
  - ***Assembly language:*** human-readable format of instructions
  - ***Machine language:*** computer-readable format (1's and 0's)
- **MIPS architecture:**
  - Developed by John Hennessy and colleagues at Stanford in the 1980's
  - Used in many commercial systems (Silicon Graphics, Nintendo, Cisco)
- **Once you've learned one architecture, it's easy to learn others**

Operands: Registers

- **Main Memory is slow**
- **Most architectures have a small set of (fast) registers**
  - MIPS has thirty-two 32-bit registers
- **MIPS is called a 32-bit architecture because it operates on 32-bit data**
  - A 64-bit version of MIPS also exists, but we will consider only the 32-bit version

The MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

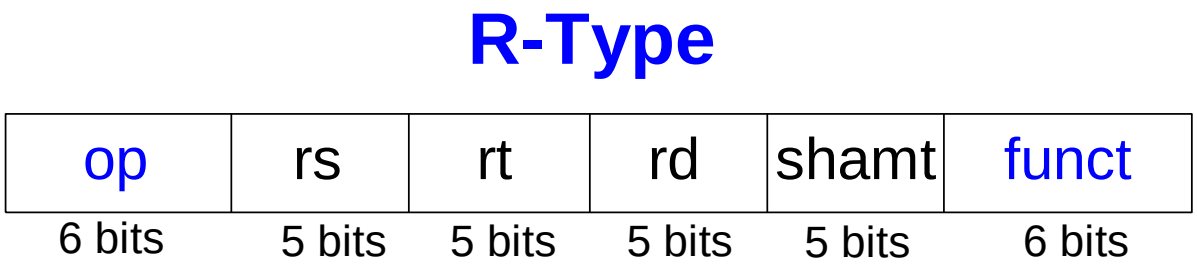
Operands: Memory

- **Too much data to fit in only 32 registers**
- **Store more data in memory**
  - Memory is large, so it can hold a lot of data
  - But it's also slow
- **Commonly used variables kept in registers**
- **Using a combination of registers and memory, a program can access a large amount of data fairly quickly**

Machine Language

- **Computers only understand 1's and 0's**
- **Machine language: binary representation of instructions**
- **32-bit instructions**
  - Again, simplicity favors regularity: 32-bit data, 32-bit instructions, and possibly also 32-bit addresses
- **Three instruction formats:**
  - **R-Type**: register operands
  - **I-Type**: immediate operand
  - **J-Type**: for jumping (we'll discuss later)

R-Type



- **Register-type, 3 register operands:**
  - **rs**, **rt**: source registers
  - **rd**: destination register
- **Other fields:**
  - **op**: the operation code or opcode (0 for R-type instructions)
  - **funct**: the function together, the opcode and function tell the computer what operation to perform
  - **shamt**: the shift amount for shift instructions, otherwise it's 0

R-Type Examples

Assembly Code

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

sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

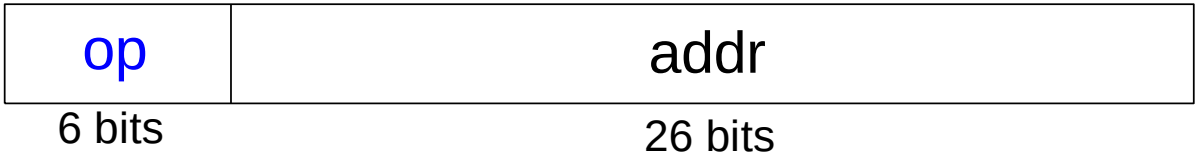
```
add rd, rs, rt
```

Review: Instruction Formats

R-Type

I-Type

J-Type



Branching

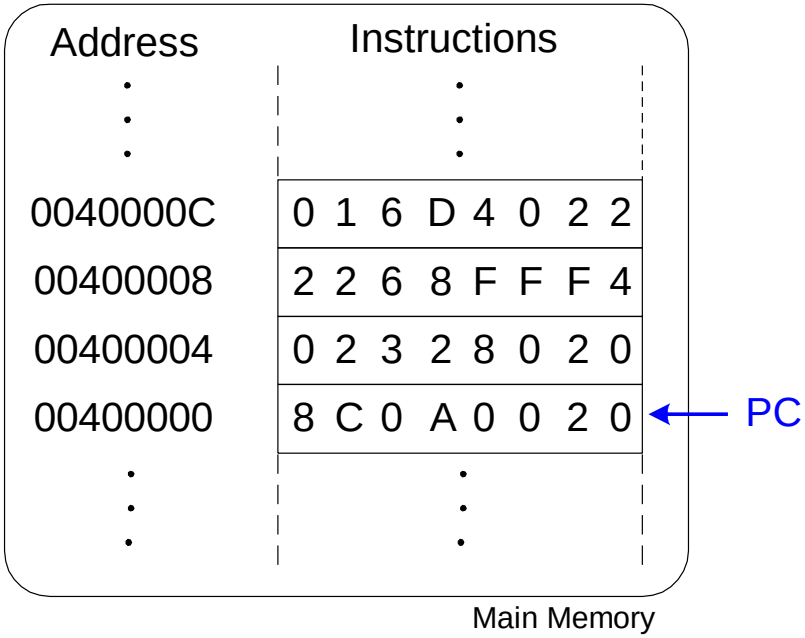
- **Allows a program to execute instructions out of sequence**
- ***Conditional branches***
  - branch if equal: **`beq`** (I-type)
  - branch if not equal: **`bne`** (I-type)
- ***Unconditional branches***
  - jump: **`j`** (J-type)
  - jump register: **`jr`** (R-type)
  - jump and link: **`jal`** (J-type)

these are the only two J-type instructions

Review: The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Stored Program



Conditional Branching (beq)

```
# MIPS assembly
 addi $s0, $0, 4
 addi $s1, $0, 1
 sll  $s1, $s1, 2
 beq  $s0, $s1, target
 addi $s1, $s1, 1
 sub  $s1, $s1, $s0

target:
 add  $s1, $s1, $s0
```

Blackboard

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

Conditional Branching (beq)

```
# MIPS assembly
 addi $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target    # branch is taken
 addi $s1, $s1, 1         # not executed
 sub  $s1, $s1, $s0       # not executed

target: .                 # label
 add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

The Branch Not Taken (bne)

```
# MIPS assembly
  addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne    $s0, $s1, target    # branch not taken
  addi   $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub    $s1, $s1, $s0       # $s1 = 5 – 4 = 1

target:
  add    $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching / Jumping (j)

```
# MIPS assembly
      addi   $s0, $0, 4       # $s0 = 4
      addi   $s1, $0, 1       # $s1 = 1
      j      target           # jump to target
      sra    $s1, $s1, 2      # not executed
      addi   $s1, $s1, 1      # not executed
      sub    $s1, $s1, $s0    # not executed

  target:
      add    $s1, $s1, $s0    # $s1 = 1 + 4 = 5
```

Unconditional Branching (jr)

```
# MIPS assembly
0x00002000  addi $s0, $0, 0x2010   # load 0x2010 to $s0
0x00002004  jr   $s0               # jump to $s0
0x00002008  addi $s1, $0, 1        # not executed
0x0000200C  sra  $s1, $s1, 2       # not executed
0x00002010  lw   $s3, 44($s1)      # program continues
```

High-Level Code Constructs

- `if` statements
- `if/else` statements
- `while` loops
- `for` loops

If Statement

High-level code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

If Statement

High-level code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

- Notice that the assembly tests for the opposite case (i != j) than the test in the high-level code (i == j)

If / Else Statement

High-level code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

If / Else Statement

High-level code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

While Loops

High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x
```

While Loops

High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

       addi $s0, $0, 1
       add  $s1, $0, $0
       addi $t0, $0, 128
while: beq  $s0, $t0, done
       sll  $s0, $s0, 1
       addi $s1, $s1, 1
       j    while
done:
```

- **Notice that the assembly tests for the opposite case (pow == 128) than the test in the high-level code (pow != 128)**

For Loops

The general form of a for loop is:

```
for (initialization; condition; loop operation)
  loop body
```

- `initialization`: executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `loop body:` executes each time the condition is met

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

- **Notice that the assembly tests for the opposite case (i == 10) than the test in the high-level code (i != 10)**

Less Than Comparisons

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

Less Than Comparisons

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

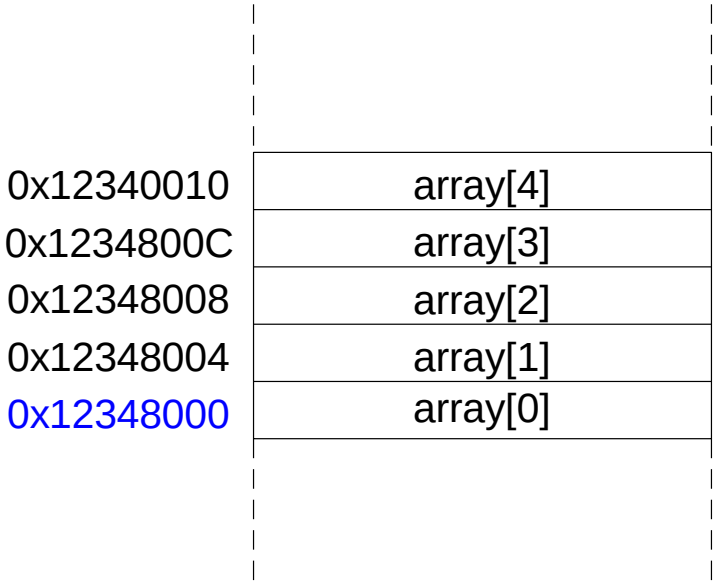
- **$t1 = 1 if i < 101**

Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by index
- Array size: number of elements in the array

Arrays

- **5-element array**
- **Base address = `0x12348000`**
  (address of the first array element, array[0])
- **First step in accessing an array:**
  - Load base address into a register



Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
```

How to get a 32-bit address into register $s0?

Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
lui  $s0, 0x1234      # upper $s0
ori  $s0, $s0, 0x8000 # lower $s0
```

How to load a[0] and a[1] into a register?

Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
lui  $s0, 0x1234      # upper $s0
ori  $s0, $s0, 0x8000 # lower $s0

lw   $t1, 0($s0)   # $t1=array[0]
sll  $t1, $t1, 1   # $t1=$t1*2
sw   $t1, 0($s0)   # array[0]=$t1

lw   $t1, 4($s0)   # $t1=array[1]
sll  $t1, $t1, 1   # $t1=$t1*2
sw   $t1, 4($s0)   # array[1]=$t1
```

Arrays Using For Loops

High-level code

```
// high-level code
int arr[1000];
int i;

for (i = 0; i < 1000; i = i + 1)
    arr[i] = arr[i] * 8;
```

MIPS Assembly code

```
# $s0 = array base, $s1 = i
lui  $s0, 0x23B8      # upper $s0
ori  $s0, $s0, 0xF000 # lower $s0
```

Arrays Using For Loops

High-level code

```
// high-level code
int arr[1000];
int i;

for (i = 0; i < 1000; i = i + 1)
    arr[i] = arr[i] * 8;
```

MIPS Assembly code

```
# $s0 = array base, $s1 = i
lui  $s0, 0x23B8      # upper $s0
ori  $s0, $s0, 0xF000 # lower $s0

addi $s1, $0, 0    # i = 0
addi $t2, $0, 1000 # $t2 = 1000

loop:
slt  $t0, $s1, $t2 # i < 1000?
beq  $t0, $0, done # if not done
sll  $t0, $s1, 2   # $t0=i * 4
add  $t0, $t0, $s0 # addr of arr[i]
lw   $t1, 0($t0)   # $t1=arr[i]
sll  $t1, $t1, 3   # $t1=arr[i]*8
sw   $t1, 0($t0)   # arr[i] = $t1
addi $s1, $s1, 1   # i = i + 1
j    loop          # repeat
done:
```

Procedures

- **Definitions**
  - ***Caller:*** calling procedure (in this case, main)
  - ***Callee:*** called procedure (in this case, sum)

```
// High level code
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

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

Procedure Calling Conventions

- **Caller:**
  - passes arguments to callee
  - jumps to the callee
- **Callee:**
  - performs the procedure
  - returns the result to caller
  - returns to the point of call
  - must not overwrite registers or memory needed by the caller

MIPS Procedure Calling Conventions

- **Call procedure:**
  - jump and link (`jal`)
- **Return from procedure:**
  - jump register (`jr`)
- **Argument values:**
  - `$a0 - $a3`
- **Return value:**
  - `$v0`

Procedure Calls

High-level code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS Assembly code

```
0x00400200 main: jal simple
0x00400204       add $s0,$s1,$s2


...
0x00401020 simple: jr $ra
```

- ***void*** **means that simple doesn't return a value**

Procedure Calls

High-level code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS Assembly code

```
0x00400200 main: jal simple
0x00400204       add $s0,$s1,$s2


...
0x00401020 simple: jr $ra
```

- `jal`: **jumps to** `simple` **and saves PC+4 in the return address register (`$ra`)**
  - In this case, `$ra` = `0x00400204` after `jal` executes
- `jr $ra`: **jumps to address in `$ra`**
  - in this case jump to address `0x00400204`

Input Arguments and Return Values

- **MIPS conventions:**
  - Argument values: `$a0 - $a3`
  - Return value: `$v0`

Input Arguments and Return Values

```
// High-level code
int main()
{
  int y;
  ...
  // 4 arguments
  y = diffofsums(2, 3, 4, 5);
  ...
}

int diffofsums(int f, int g,
  int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result; // return value
}
```

```
# MIPS assembly code
# $s0 = y

main:
  ...
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call procedure
  add  $s0, $v0, $0  # y = returned value
  ...

# $s0 = result
diffofsums:
  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
  jr  $ra            # return to caller
```

Input Arguments and Return Values

```
# $s0 = result
diffofsums:
  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
  jr  $ra            # return to caller
```

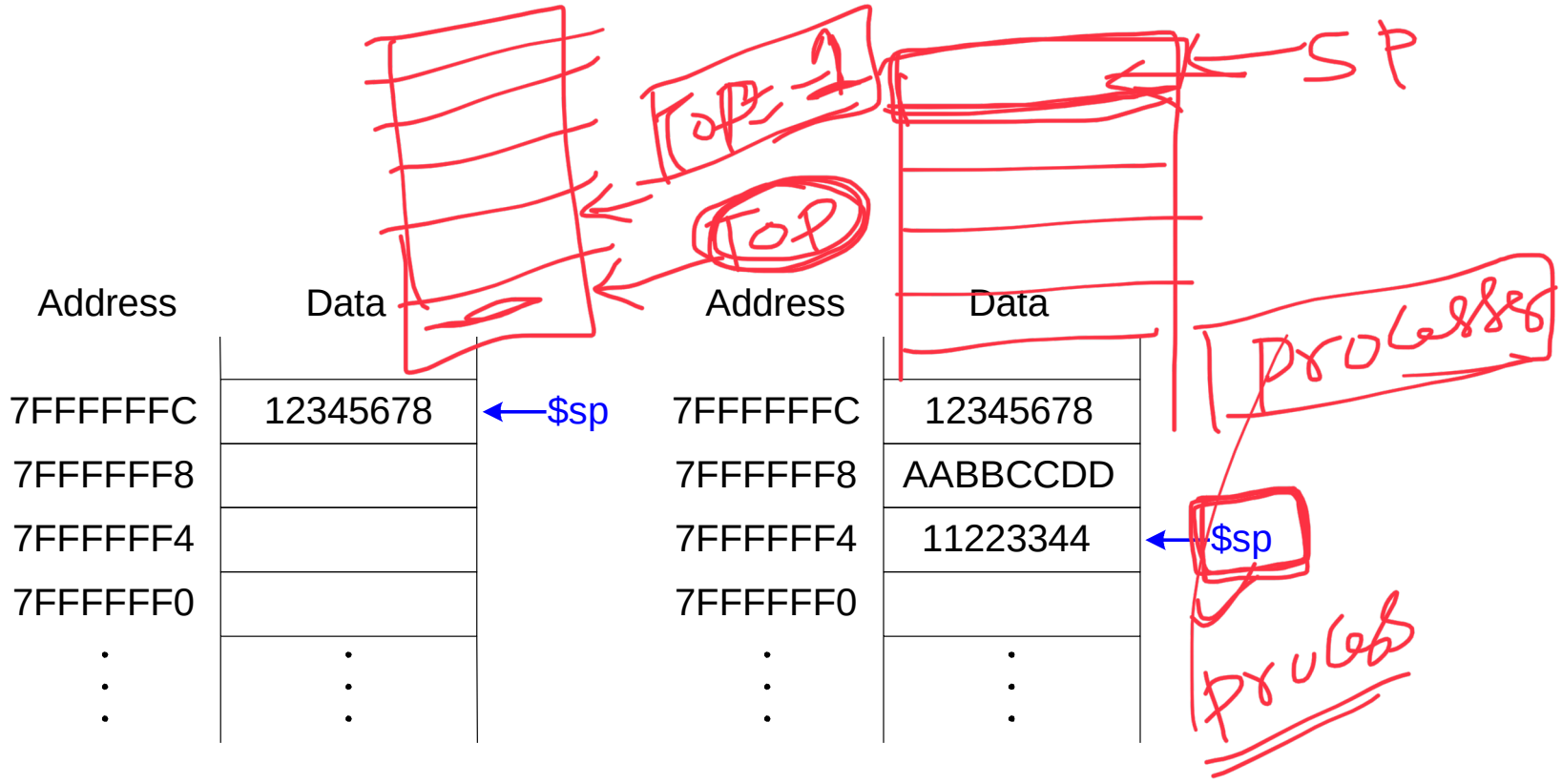
- `diffofsums` **overwrote 3 registers: `$t0`, `$t1`, and `$s0`**
- `diffofsums` **can use the *stack* to temporarily store registers (comes next)**

The Stack

- Memory used to temporarily save variables
- Like a stack of dishes, last-in-first-out (LIFO) queue
- *Expands:* uses more memory when more space is needed
- *Contracts:* uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp`, points to top of the stack



How Procedures use the Stack

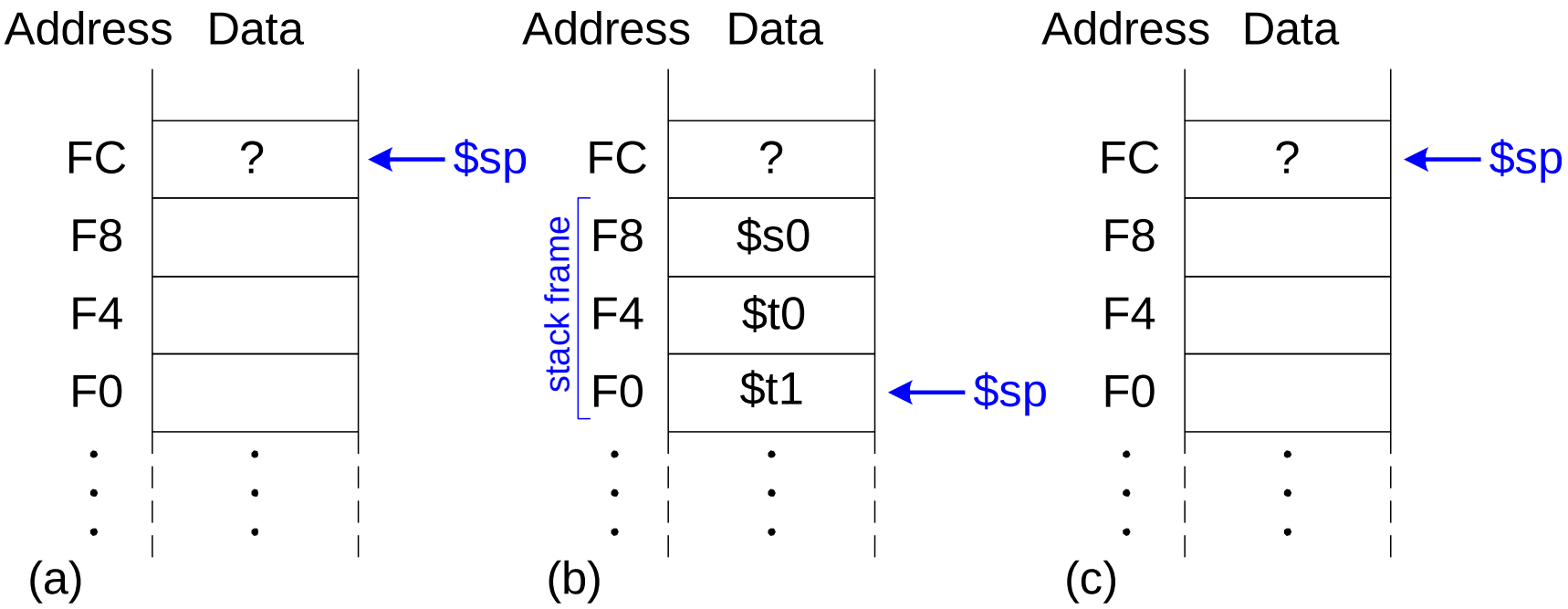
- **Called procedures must have no other unintended side effects**
- **But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`**

```
# MIPS assembly
# $s0 = result
diffofsums:
  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
  jr  $ra            # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # make space on stack
                      # to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  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   $t1, 0($sp)    # restore $t1 from stack
  lw   $t0, 4($sp)    # restore $t0 from stack
  lw   $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr   $ra            # return to caller
```

The Stack during `diffofsums` Call



Registers

Preserved Callee-saved = Callee must preserve	Nonpreserved Caller-saved = Callee can overwrite
$s0 - $s7	$t0 - $t9
$ra	$a0 - $a3
$sp	$v0 - $v1
stack above $sp	stack below $sp

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:




  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


  jr  $ra             # return to caller
```

which of these registers may not be overwritten by `diffofsums`?

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
```

which of these registers may not be overwritten by diffofsums?

`$s0` – hence it has to be stored on the stack and restored

Multiple Procedure Calls

```
proc1:
  addi $sp, $sp, -4    # make space on stack
  sw   $ra, 0($sp)     # save $ra on stack
  jal  proc2
  ...
  lw   $ra, 0($sp)     # restore $s0 from stack
  addi $sp, $sp, 4     # deallocate stack space
  jr  $ra              # return to caller
```

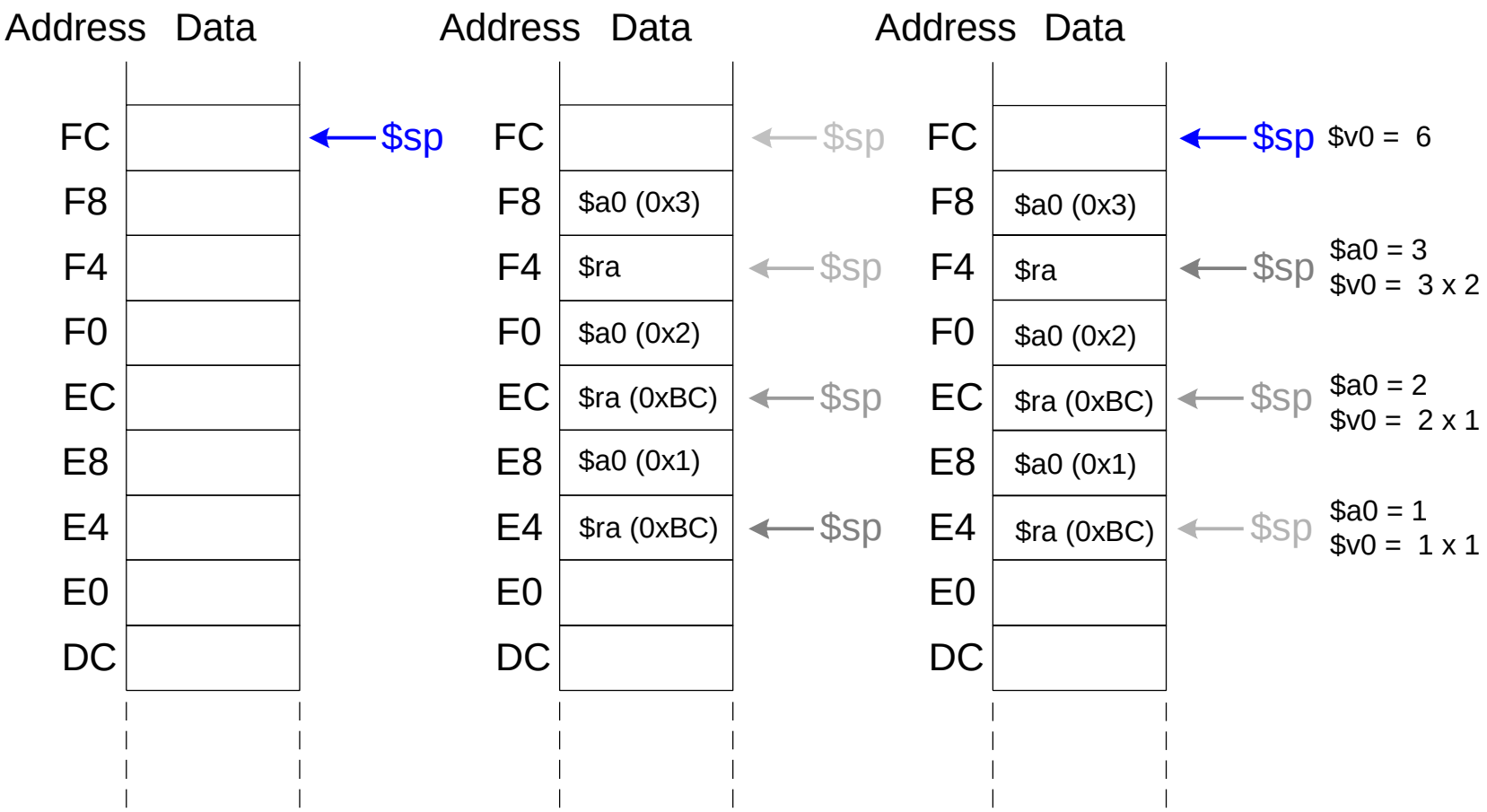
Recursive Procedure Call

```
// High-level code

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

Recursive Procedure Call

```
# 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



Procedure Call Summary

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

Addressing Modes

- **How do we address the operands?**
  - Register Only
  - Immediate
  - Base Addressing
  - PC-Relative
  - Pseudo Direct

Register Only Addressing

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

Immediate Addressing

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

Base Addressing

- **Address of operand is:**

  `base address + sign-extended immediate`

  - ***Example:***

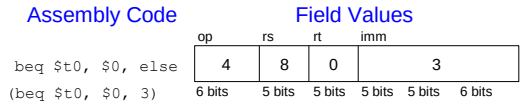
    `lw  $s4, 72($0)` Address = `$0 + 72`

  - ***Example:***

    `sw  $t2, -25($t1)` Address = `$t1 - 25`

PC-Relative Addressing

```
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
```



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

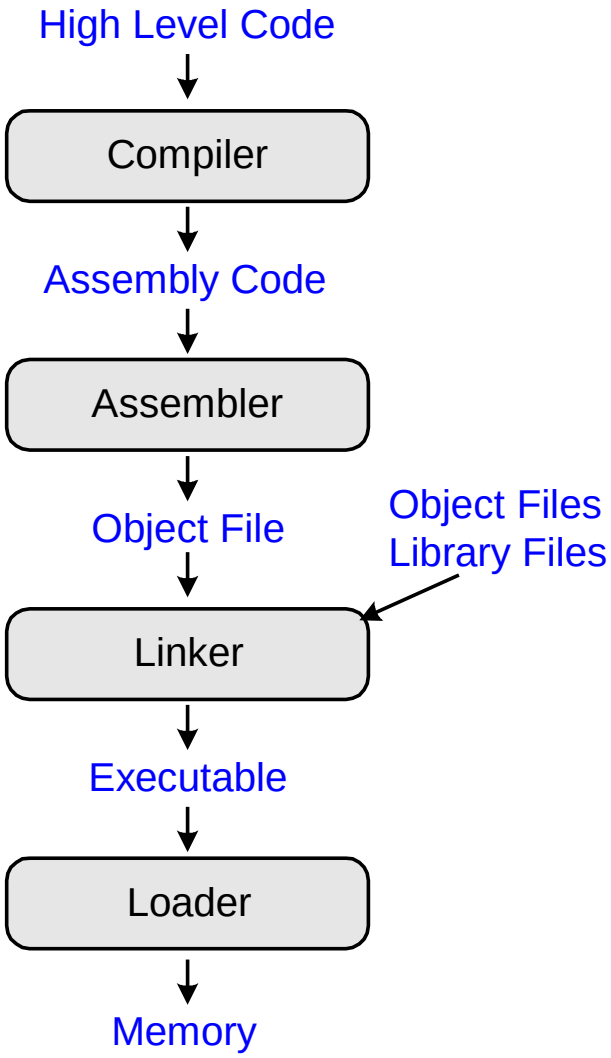
Field Values

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	

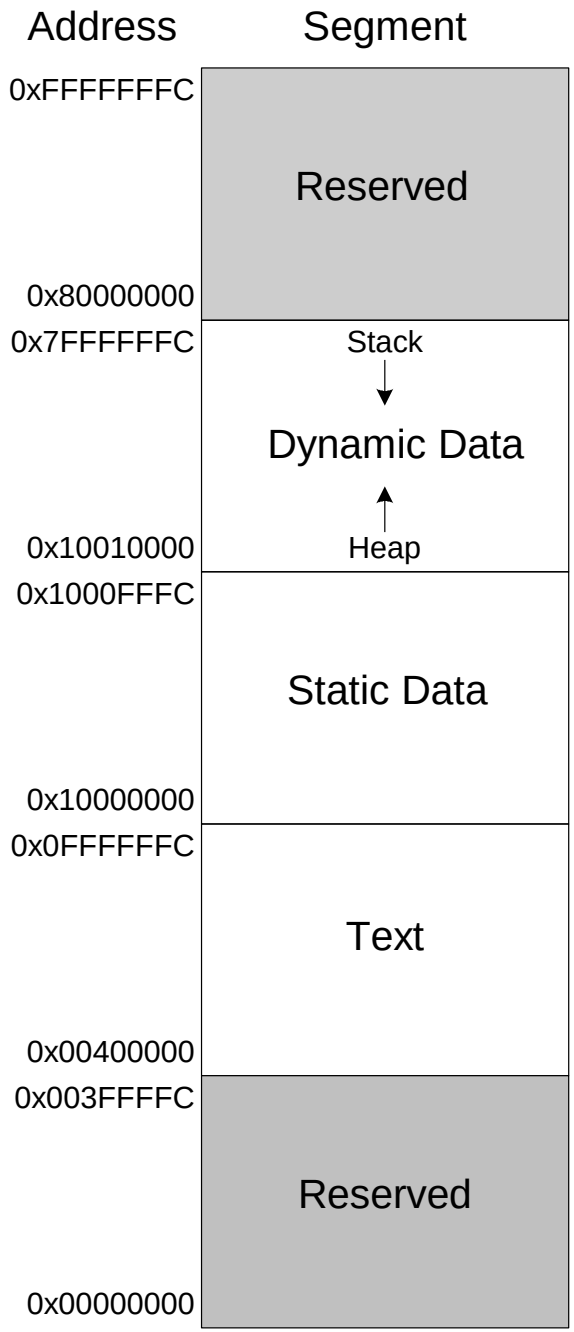
How Do We Compile & Run an Application?



What needs to be 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`

The MIPS Memory Map



Example Program: C Code

```
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);
}
```

Example Program: Assembly Code

```
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
      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)  # rest. $ra
      addi $sp, $sp, 4  # rest. $sp
      jr   $ra          # return
sum:  add  $v0, $a0, $a1 # $v0= a+b
      jr   $ra          # return
```

Example Program: Symbol Table

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

Example Program: Executable

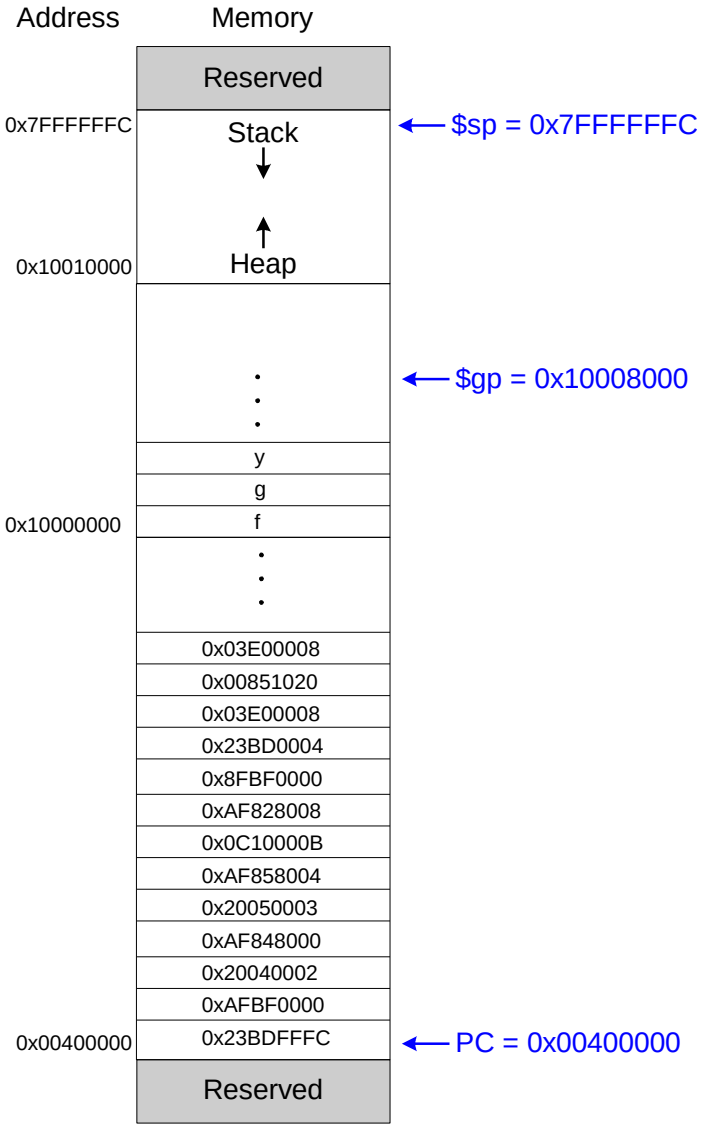
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E0008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Example Program: In Memory



Odds and Ends

- Pseudoinstructions
- Exceptions
- Signed and unsigned instructions
- Floating-point instructions

Pseudoinstruction Examples

Pseudoinstruction	MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`mul $s0, $s1, $s2`	`mult $s1, $s2` `mflo $s0`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Exceptions

- **Unscheduled procedure call to the exception handler**
- **Caused by:**
  - Hardware, also called an interrupt, e.g. keyboard
  - Software, also called traps, e.g. undefined instruction
- **When exception occurs, the processor:**
  - Records the cause of the exception
  - Jumps to the exception handler at instruction address 0x80000180
  - Returns to program

Exception Registers

- **Not part of the register file.**
  - Cause
    - Records the cause of the exception
  - EPC (Exception PC)
    - Records the PC where the exception occurred
- **EPC and Cause: part of Coprocessor 0**
- **Move from Coprocessor 0**
  - mfc0 $t0, EPC
  - Moves the contents of EPC into $t0

Exception Causes

Exception	Cause
Hardware Interrupt	`0x00000000`
System Call	`0x00000020`
Breakpoint / Divide by 0	`0x00000024`
Undefined Instruction	`0x00000028`
Arithmetic Overflow	`0x00000030`

Exceptions

- **Processor saves cause and exception PC in Cause and EPC**
- **Processor jumps to exception handler (0x80000180)**
- **Exception handler:**
  - Saves registers on stack
  - Reads the Cause register
    - `mfc0 $t0, Cause`
  - Handles the exception
  - Restores registers
  - Returns to program
    - `mfc0 $k0, EPC`
    - `jr $k0`

Signed and Unsigned Instructions

- Addition and subtraction
- Multiplication and division
- Set less than

Addition and Subtraction

- ***Signed***: **`add, addi, sub`**
  - Same operation as unsigned versions
  - But processor takes exception on overflow
- ***Unsigned***: **`addu, addiu, subu`**
  - Doesn't take exception on overflow
  - Note: **`addiu`** sign-extends the immediate

Multiplication and Division

- *Signed*: `mult`, `div`
- *Unsigned*: `multu`, `divu`

Set Less Than

- ***Signed***: **`slt, slti`**
- ***Unsigned***: **`sltu, sltiu`**
  - Note: `sltiu` sign-extends the immediate before comparing it to the register

Loads

- ***Signed*: `lh, lb`**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- ***Unsigned*: `lhu, lbu`**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`

Floating-Point Instructions

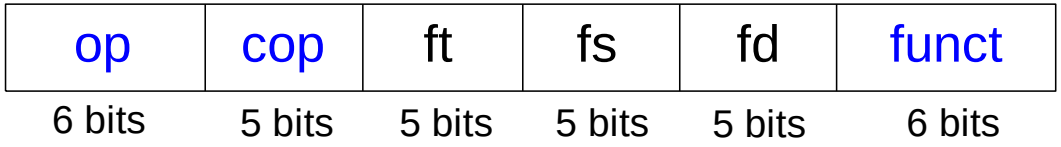
- **Floating-point coprocessor (Coprocessor 1)**
- **Thirty-two 32-bit floating-point registers (`$f0 - $f31`)**
- **Double-precision values held in two floating point registers**
  - e.g., `$f0` and `$f1`, `$f2` and `$f3`, etc.
  - So, double-precision floating point registers: `$f0`, `$f2`, `$f4`, etc.

Floating-Point Instructions

Name	Register Number	Usage
`$fv0 - $fv1`	`0, 2`	return values
`$ft0 - $ft3`	`4, 6, 8, 10`	temporary variables
`$fa0 - $fa1`	`12, 14`	procedure arguments
`$ft4 - $ft8`	`16, 18`	temporary variables
`$fs0 - $fs5`	`20, 22, 24, 26, 28, 30`	saved variables

F-Type Instruction Format

F-Type



- **Opcode = 17 $(010001)_2$**
- ***Single-precision*: cop = 16 $(010000)_2$**
  - `add.s, sub.s, div.s, neg.s, abs.s`, etc.
- ***Double-precision*: cop = 17 $(010001)_2$**
  - `add.d, sub.d, div.d, neg.d, abs.d`, etc.
- **3 register operands:**
  - fs, ft: source operands
  - fd: destination operands

Floating-Point Branches

- **Set/clear condition flag: fpcond**
  - Equality: `c.seq.s`, `c.seq.d`
  - Less than: `c.lt.s`, `c.lt.d`
  - Less than or equal: `c.le.s`, `c.le.d`
- **Conditional branch**
  - `bclf`: branches if fpcond is ***FALSE***
  - `bclt`: branches if fpcond is ***TRUE***
- **Loads and stores**
  - `lwc1: lwc1 $ft1, 42($s1)`
  - `swc1: swc1 $fs2, 17($sp)`

What Did We Learn?

- **How to translate common programming constructs**
  - Conditons
  - Loops
  - Procedure calls
- **Stack**
- **The compiled program**
- **Odds and Ends**
  - Floating point (F-type) instructions
- **What Next?**
  - Actually building the MIPS Microprocessor!!