MIPS Programming

Design of Digital Circuits 2017
Srdjan Capkun
Onur Mutlu
(Lecture by Der-Yeuan Yu)

http://www.syssec.ethz.ch/education/Digitaltechnik_17

In This Lecture

- Small review from last week

- Programming (continued)

- Addressing Modes

- Lights, Camera, Action: Compiling, Assembling, and Loading

- Odds and Ends

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 (:).

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**

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

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

13

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?   15

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)

```c
// 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

```c
// 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
# 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**

Address	Data	
7FFFFFFC	12345678	←—$sp
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	←—$sp
7FFFFFF0		
⋮	⋮	

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
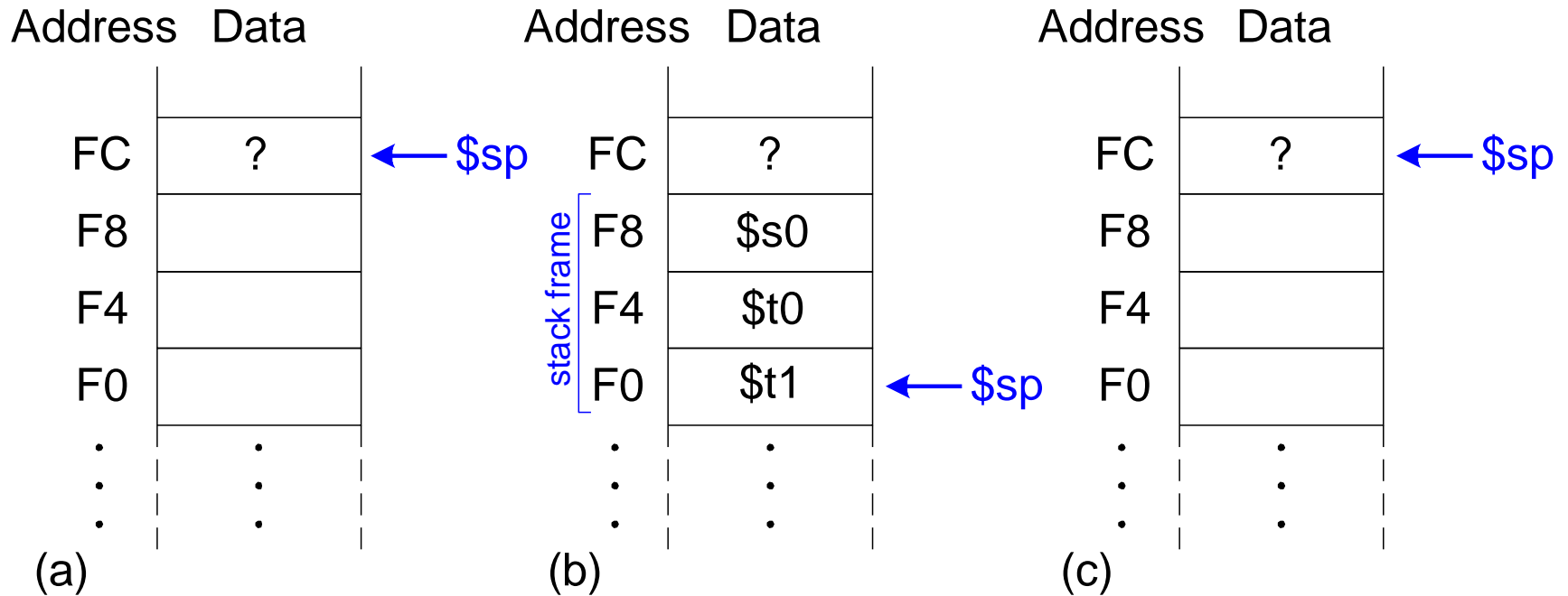
Registers

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

Address   Data

FC	
F8	
F4	
F0	
EC	
E8	
E4	
E0	
DC	

← $sp

Address   Data

FC	
F8	$a0 (0x3)
F4	$ra
F0	$a0 (0x2)
EC	$ra (0xBC)
E8	$a0 (0x1)
E4	$ra (0xBC)
E0	
DC	

← $sp (FC)
← $sp (F4)
← $sp (EC)
← $sp (E4)

Address   Data

FC	
F8	$a0 (0x3)
F4	$ra
F0	$a0 (0x2)
EC	$ra (0xBC)
E8	$a0 (0x1)
E4	$ra (0xBC)
E0	
DC	

← $sp $v0 = 6

← $sp $a0 = 3
        $v0 = 3 x 2

← $sp $a0 = 2
        $v0 = 2 x 1

← $sp $a0 = 1
        $v0 = 1 x 1
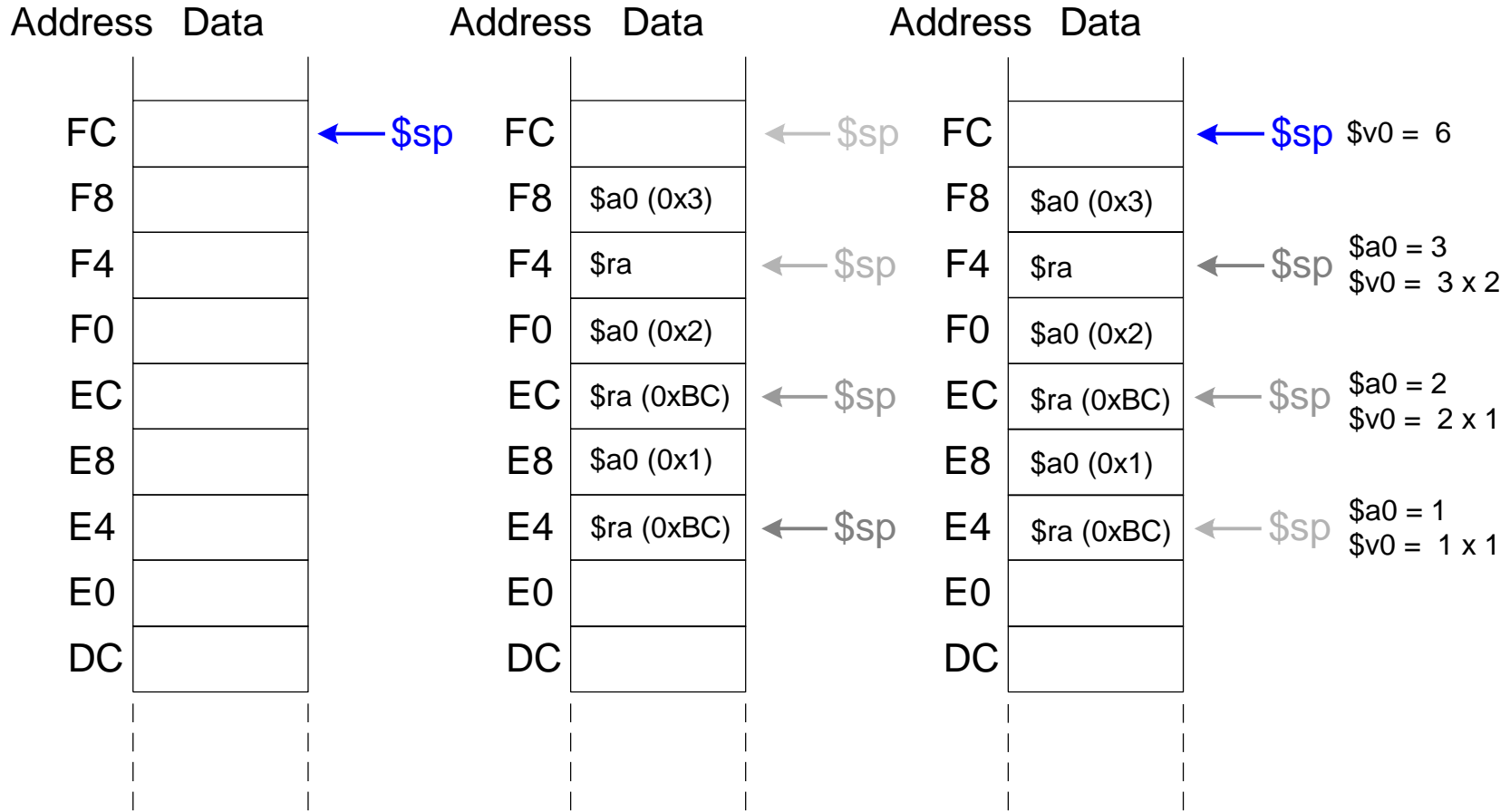
Procedure Call Summary

- **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 $t2, -24($t1)   Address = **$t1 - 24**

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
```

Assembly Code

beq $t0, $0, else

(beq $t0, $0, 3)

Field Values

op	rs	rt	imm
4	8	0	3

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

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?

High Level Code

↓

```
Compiler
```

Assembly Code

↓

```
Assembler
```

Object File

Object Files
Library Files

↓

```
Linker
```

↓

Executable

↓

```
Loader
```

↓
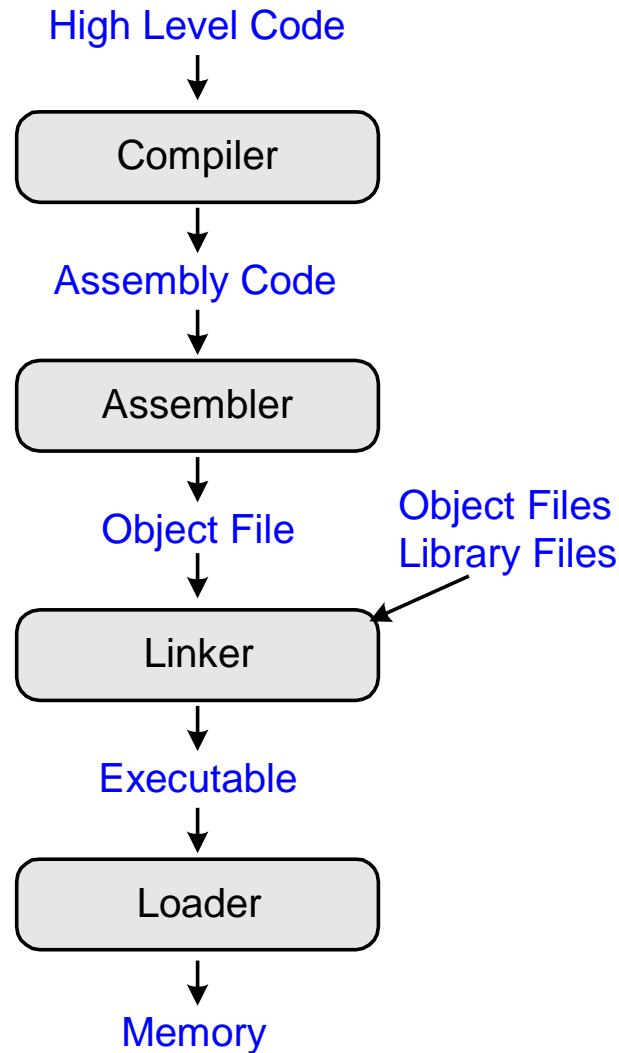
Memory

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

Address	Segment
	Address　　**Segment**

0xFFFFFFFC

Reserved

0x80000000

0x7FFFFFFC

Stack
↓

Dynamic Data

↑
Heap

0x10010000

0x1000FFFC

Static Data

0x10000000

0x0FFFFFFC

Text

0x00400000

0x003FFFFC

Reserved

0x00000000

48
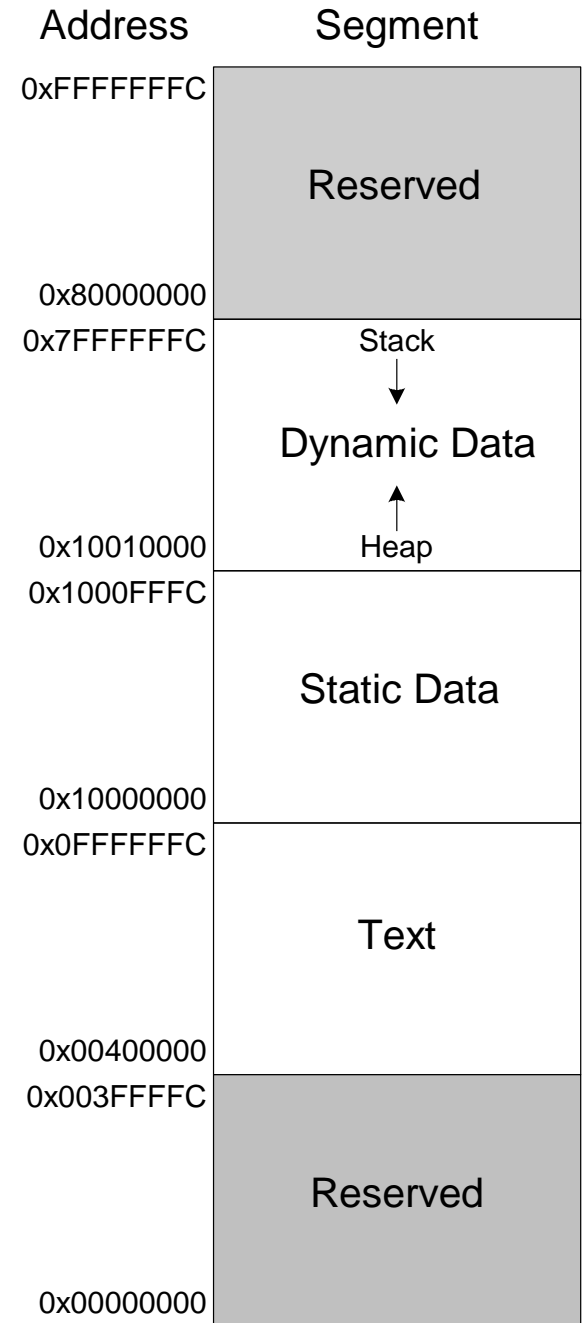
Example Program: C Code

```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);
}
```

Example Program: Assembly Code

```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);
}
```

```asm
      .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	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	0x03E0008	jr $ra

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

Example Program: In Memory

Address   Memory

	Reserved
0x7FFFFFFC	Stack ↓
	↑ Heap
0x10010000	

←— $sp = 0x7FFFFFFC

←— $gp = 0x10008000

0x10000000	...
	y
	g
	f
	...

0x03E00008
0x00851020
0x03E00008
0x23BD0004
0x8FBF0000
0xAF828008
0x0C10000B
0xAF858004
0x20050003
0xAF848000
0x20040002
0xAFBF0000
0x23BDFFFC

0x00400000

←— PC = 0x00400000

Reserved

53
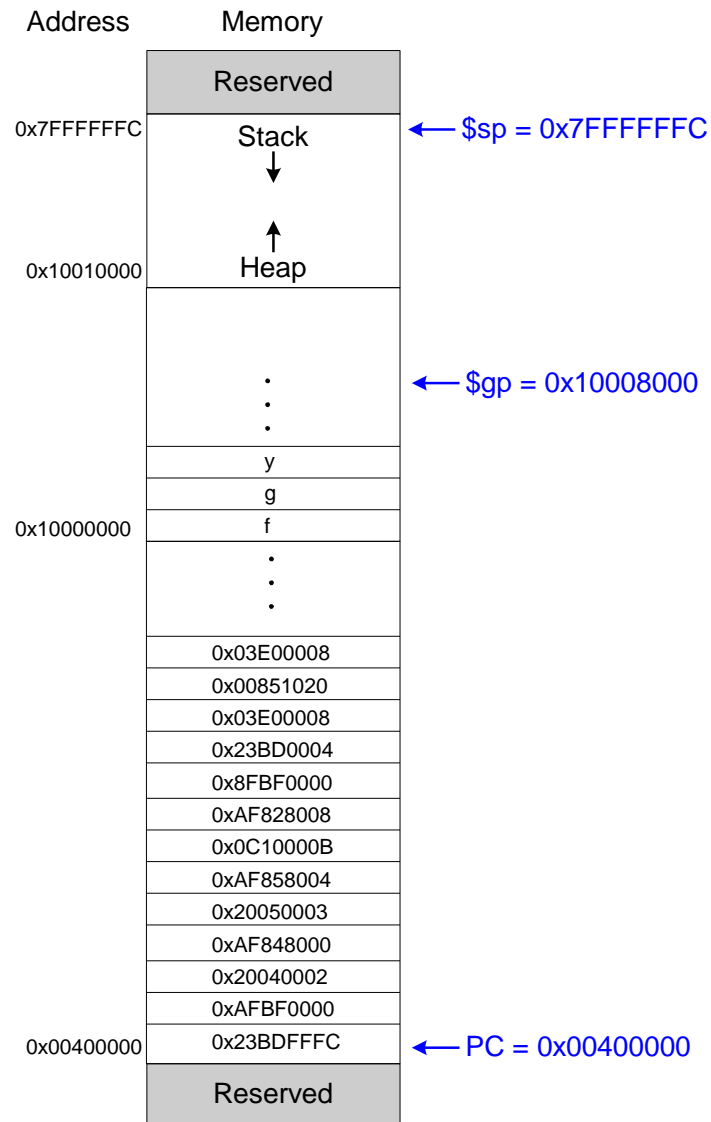
Odds and Ends

- **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

op	cop	ft	fs	fd	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **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

67

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**
    - `bc1f`: branches if fpcond is *FALSE*
    - `bc1t`: 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!!