

In the name of Allah

## Midterm Overview

Zohre Soorani  
Mahdi Haghverdi  
Hussein Hussein  
Hosna Rajaei



Isfahan University

November 14, 2023

# Content

Computer Abstraction and Technology

Operations of the Computer Hardware

Operands of the Computer Hardware

- Memory Operands

- Constant or Immediate Operands

Representing Instructions

Logical Operations

Instructions for Making Decisions

Supporting Procedures in Computer Hardware

MIPS Addressing for 32-Bit immediates and addresses

A C Sort Example to Put It All Together

# Computer Abstraction and Technology

# Computer Abstraction and Technology

# Operations of the Computer Hardware

# Operations of the Computer Hardware

Figure: Arithmetic Instructions in MIPS

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants

# Operations of the Computer Hardware (Cont'd)

load word	lw \$s1, 20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
store word	sw \$s1, 20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
load byte	lb \$s1, 20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
load byte unsigned	lbu \$s1, 20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
store byte	sb \$s1, 20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
load upper immed	lui \$s1, 20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits

Table: Data Transfer Instructions in MIPS

# Operations of the Computer Hardware (Cont'd)

Figure: Logical Instructions in MIPS

Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 \mid 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant



# Operations of the Computer Hardware (Cont'd)

Figure: Conditional Branch Instructions in MIPS

Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned

# Operations of the Computer Hardware (Cont'd)

Figure: Unconditional Jump Instructions in MIPS

Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

## Example - Compiling a Complex C Assignment into MIPS

A somewhat complex statement contains the five variables `f`, `g`, `h`, `i`, and `j`:

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

What might a C compiler produce?

# Answer

- add t0, g, h # temporary variable t0 contains  $g + h$
- add t1, i, j # temporary variable t1 contains  $i + j$
- sub f, t0, t1 # f gets  $t0 - t1$ , which is  $(g + h) - (i + j)$

# Operands of the Computer Hardware

## Example - Compiling a C Assignment Using Registers

It is the compiler's job to associate program variables with registers.

Take, for instance, the assignment statement from our earlier example:

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

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively.

What is the compiled MIPS code?

# Answer

- `add $t0, $s1, $s2` # register \$t0 contains  $g + h$
- `add $t1, $s3, $s4` # register \$t1 contains  $i + j$
- `sub $s0, $t0, $t1` # f gets  $\$t0 - \$t1$ , which is  $(g + h) - (i + j)$

## Example - Compiling Using Load and Store

Assume variable `h` is associated with register `$s2` and the base address of the array `A` is in `$s3`.

What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```



# Answer

- `lw $t0, 32($s3)` # Temporary reg \$t0 gets A[8]
- `add $t0, $s2, $t0` # Temporary reg \$t0 gets h + A[8]
- `sw $t0, 48($s3)` # Stores h + A[8] back into A[12]

# Constant or Immediate Operands

- `addi $s3, $s3, 4`    `# $s3 = $s3 + 4`

# Representing Instructions

# Instructions Big Picture

Name	Format	Example						Comments
Field Size		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
I-format	I	op	rs	rt	address			Data transfer format
lw	I	35	18	17	100			lw \$s1,100(\$s2)
J-format	J	op	address					Unconditional Branch
j	J	8	300					jump to address

# Logical Operations

# Logical Operations Big Picture

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

# Instructions for Making Decisions

# Instructions for Making Decisions

- `beq register1, register2, L1`

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`.

The mnemonic `beq` stands for *branch if equal*.

- `bne register1, register2, L1`

It means go to the statement labeled `L1` if the value in `register1` does not equal the value in `register2`.

The mnemonic `bne` stands for *branch if not equal*.



## Example - Compiling *if-then-else* into Conditional Branches

In the following code segment, **f**, **g**, **h**, **i**, and **j** are variables.

If the five variables, **f** through **j**, correspond to the five registers **\$s0** through **\$s4**, what is the compiled MIPS code for this C if statement?

---

```
1  if i == j:
2      f = g + h
3  else:
4      f = g - h
```

---

# Answer

---

```
1  bne $s3, $s4, Else      # go to Else if i != j
2  add $s0, $s1, $s2       # f = g + h (skipped if i != j)
3  j    Exit               # go to Exit
4  Else:
5      sub $s0, $s1, $s2    # f = g - h (skipped if i = j)
6  Exit:
```

---

## Example - Compiling a *while* Loop in C

Here is a traditional loop in C:

---

```
1 while (save[i] == k){  
2     i += 1;  
3 }
```

---

Assume that *i* and *k* correspond to registers *\$s3* and *\$s5* and the base of the array *save* is in *\$s6*.

What is the MIPS assembly code corresponding to this C segment?

# Answer

---

```
1 Loop:    sll      $t1,$s3,2      # Temp reg $t1 = i * 4
2          add      $t1,$t1,$s6    # $t1 = address of save[i]
3          lw       $t0,0($t1)     # Temp reg $t0 = save[i]
4          bne      $t0,$s5, Exit  # go to Exit if save[i] != k
5          addi     $s3,$s3,1      # i = i + 1
6          j        Loop          # go to Loop
7 Exit:
```

---

# Supporting Procedures in Computer Hardware

# Six Steps

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

# Provided Registers

MIPS software follows the following convention for procedure calling in allocating its 32 registers

- `$a0-$a3`: four argument registers in which to pass parameters
- `$v0-$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

## Provided Registers (Cont'd)

- In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures:
- *It jumps to an address and simultaneously saves the address of the following instruction in register \$ra.*
- The *jump-and-link* instruction (`jal`) is simply written:
- `jal ProcedureAddress`



## Provided Registers (Cont'd)

- The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This “link”, stored in register \$ra (register 31), is called the return address.
- The return address is needed because the same procedure could be called from several parts of the program.
- To support such situations, computers like MIPS use jump register instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:
- `jr $ra`

## Example - Compiling a C Procedure That Doesn't Call Another Procedure

---

```
1  int leaf_example (int g, int h, int i, int j)
2  {
3      int f;
4      f = (g + h) - (i + j);
5      return f;
6  }
```

---

# Answer

---

```
leaf_example:
    addi $sp, $sp, -12    # adjust stack to make room for 3 items
    sw   $t1, 8($sp)      # save register $t1 for use afterwards
    sw   $t0, 4($sp)      # save register $t0 for use afterwards
    sw   $s0, 0($sp)      # save register $s0 for use afterwards

    add  $t0, $a0, $a1     # register $t0 contains g + h
    add  $t1, $a2, $a3     # register $t1 contains i + j
    sub  $s0, $t0, $t1     # f = $t0 - $t1, which is (g + h) - (i + j)

    add  $v0, $s0, $zero   # returns f ($v0 = $s0 + 0)
```

---

## Answer (Cont'd)

---

```
lw    $s0, 0($sp)    # restore register $s0 for caller
lw    $t0, 4($sp)    # restore register $t0 for caller
lw    $t1, 8($sp)    # restore register $t1 for caller
addi  $sp, $sp, 12    # adjust stack to delete 3 items

jr    $ra             # jump back to calling routine
```

---

## Example - Compiling a Recursive C Procedure, Showing Nested Procedure Linking

---

```
1  int fact (int n)
2  {
3      if (n < 1) {
4          return 1;
5      }
6      else {
7          return (n * fact(n - 1));
8      }
9  }
```

---

# Answer

---

fact:

```
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 4($sp)     # save the return address
    sw   $a0, 0($sp)     # save the argument n

    slti $t0, $a0, 1     # test for n < 1
    beq  $t0, $zero, L1  # if n >= 1, go to L1

    addi $v0, $zero, 1   # return 1
    addi $sp, $sp, 8     # pop 2 items off stack
    jr   $ra             # return to caller
```

---

## Answer (Cont'd)

---

L1:

```
addi $a0, $a0, -1    # n >= 1: argument gets (n - 1)
jal  fact            # call fact with (n - 1)

lw  $a0, 0($sp)      # return from jal: restore argument n
lw  $ra, 4($sp)      # restore the return address
addi $sp, $sp, 8     # adjust stack pointer to pop 2 items

mul $v0, $a0, $v0    # return n * fact (n - 1)

jr  $ra              # return to the caller
```

---

## MIPS Addressing for 32-Bit Immediates and Addresses



## Example - Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

0000 0000 0011 1101 0000 1001 0000 0000

# Answer

---

```
1  lui $s0, 61  # 61 decimal = 0000 0000 0011 1101 binary
2
3  # value of $s0
4  # 0000 0000 0011 1101 0000 0000 0000 0000
5
6  ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
7
8  # final value of $s0
9  # 0000 0000 0011 1101 0000 1001 0000 0000
```

---

# Illustration of the five MIPS addressing modes

## 1. Immediate addressing



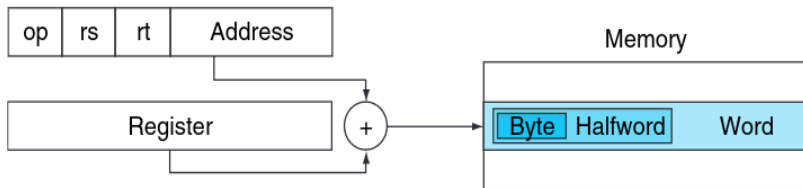
## Illustration of the five MIPS addressing modes (Cont'd)

### 2. Register addressing



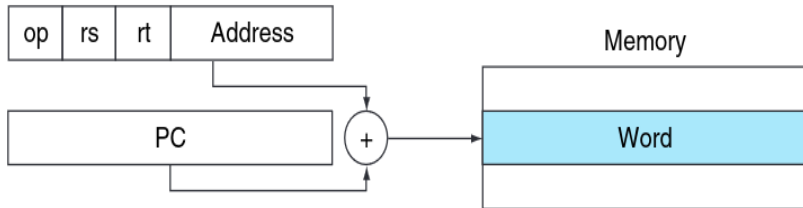
## Illustration of the five MIPS addressing modes (Cont'd)

### 3. Base addressing



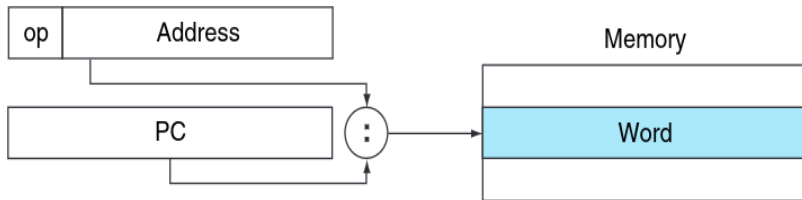
## Illustration of the five MIPS addressing modes (Cont'd)

### 4. PC-relative addressing



## Illustration of the five MIPS addressing modes (Cont'd)

### 5. Pseudodirect addressing



## A C Sort Example to Put It All Together