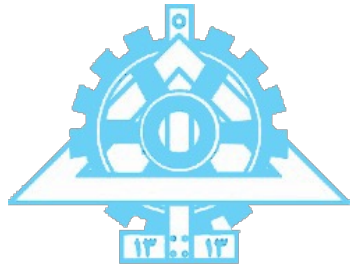


Saeed Safari,

Advanced Computer Architecture

MIPS ISA

ECE School,

Faculty of Engineering,

University of Tehran

Outline

- General purpose registers
- Integer/Floating point instructions
  - Arithmetic/Logical/Shift
  - Load/Store
  - Control flow
- High-level programming constructs
- Miscellaneous instructions

General Purpose Registers

Register name	Register #	Use	Preserved across a call?
$zero	0	Constant zero	
$at	1	Assembler temporary (reserved for assembler)	
$v0 to $v1	2 to 3	Function return values	
$a0 to $a3	4 to 7	Function parameters	
$t0 to $t7	8 to 15	Temporaries	
$s0 to $s7	16 to 23	Saved temporaries	Yes
$t8 to $t9	24 to 25	Temporaries	
$k0 to $k1	26 to 27	Reserved for OS kernel	
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

General Purpose Registers

- $zero
  - Can be read, cannot be written
- $at
  - Used by the assembler for pseudo instructions
- $v0-$v1
  - Function return value
  - Parameter passing to *syscall*
- $a0-$a3
  - Used to pass parameters to a function

General Purpose Registers

- $t0-$t9
  - Used to store temporary variables
  - No need to preserve during function call
- $s0-$s8
  - Used to store saved values
  - Preserved during function call
- $k0-$k1
  - Used by operating system
  - Not available for user program

General Purpose Registers

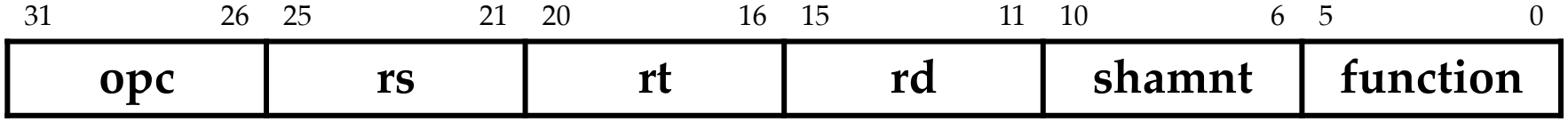
- $gp
  - Global pointer
  - Used to access global variables
- $sp
  - Stack pointer
  - Points to the last location in use on the stack
  - Used to access function parameters/local variables

General Purpose Registers

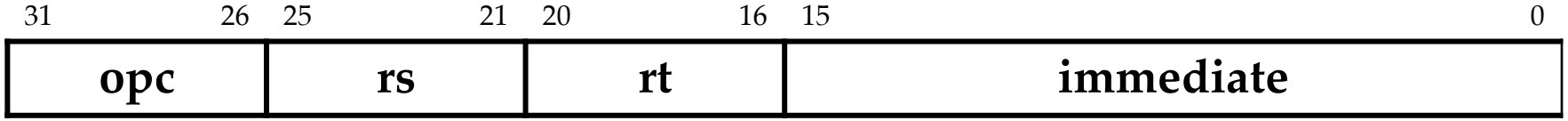
- $fp
  - Fame pointer
  - Used as an alternate way to keep track of the stack
- $ra
  - Return address
  - During a function call, holds the address to which control should be returned

MIPS Instruction Format

- Fixed length (4-byte aligned)
- Instruction Formats:
  - R-Type:



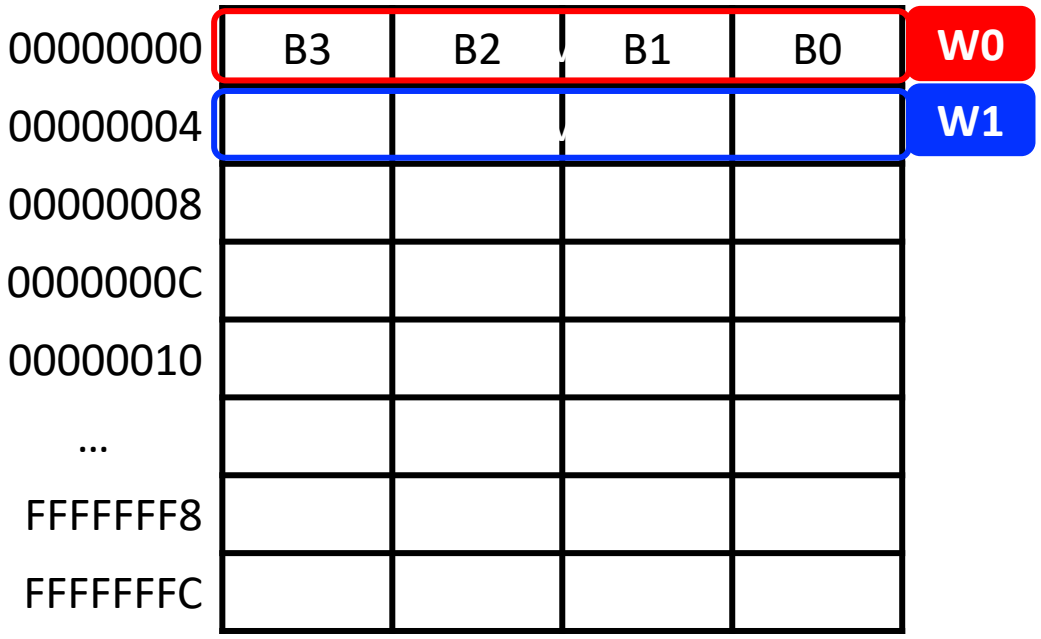
  - I-Type:



  - j-Type:

Memory Organization

- Byte/word addressable
  - Address is a multiple of 4
- Little endian
- Both data and instruction are aligned
- Addressing space:
  - 4GB or
  - 1GW



Arithmetic Instructions

Instruction	Description	Instruction Format
add rd, rs, rt	Add	R-Type
addu rd, rs, rt	Add unsigned	R-Type
sub rd, rs, rt	Subtract	R-Type
subu rd, rs, rt	Subtract unsigned	R-Type
slt rd, rs, rt	Set on less than	R-Type
sltu rd, rs, rt	Set on less than unsigned	R-Type
and rd, rs, rt	AND	R-Type
or rd, rs, rt	OR	R-Type
xor rd, rs, rt	XOR	R-Type
nor rd, rs, rt	NOR	R-Type

Arithmetic Instructions

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

```
add   $t0, $s2, $s3
add   $t1, $s4, $s5
sub   $s1, $t0, $t1
```

f	$s1
g	$s2
h	$s3
i	$s4
j	$s5

Shift Instructions

Instruction				Description	Instruction Format
sll	rd,	rs,	shamnt	Shift logical left	R-Type
srl	rd,	rs,	shamnt	Shift logical right	R-Type
sla	rd,	rs,	shamnt	Shift right arithmetic	R-Type
sllv	rd,	rs,	rt	Shift logical left variable	R-Type
srlv	rd,	rs,	rt	Shift logical right variable	R-Type
slav	rd,	rs,	rt	Shift right arithmetic variable	R-Type

Arithmetic (Immediate) Instructions

Instruction				Description	Instruction Format
addi	rd,	rs,	Imm_16bit	Add immediate	I-Type
addiu	rd,	rs,	Imm_16bit	Add immediate unsigned	I-Type
slti	rd,	rs,	Imm_16bit	Set on less than immediate	I-Type
sltiu	rd,	rs,	Imm_16bit	Set on less than immediate unsigned	I-Type
andi	rd,	rs,	Imm_16bit	AND immediate	I-Type
ori	rd,	rs,	Imm_16bit	OR immediate	I-Type
xori	rd,	rs,	Imm_16bit	XOR immediate	I-Type
lui	rd,	Imm_16bit		Load upper immediate	I-Type

Arithmetic (Immediate) Instructions

y = 0X1234;

```
addi $s1, $zero, 0X1234
```

y	$s1
z	$s2

z = 0XABCD1234;

```
lui  $s2, $zero, 0XABCD
ori  $s2, $s2, 0X1234
```

Pseudo Instruction:

```
li   $s2, 0XABCD1234
# $s2 ← 0XABCD1234
```

Multiply/Divide Instructions

Instruction	Description		Instruction Format
mult rs, rt	Multiply	{hi, lo} ⟵ rs × rt	R-Type
multu rs, rt	Multiply unsigned	{hi, lo} ⟵ rs × rt	R-Type
div rs, rt	Divide	hi ⟵ rs % rt, lo ⟵ rs / rt	R-Type
divu rs, rt	Divide unsigned	hi ⟵ rs % rt, lo ⟵ rs / rt	R-Type
mfhi rd	Move from hi		R-Type
mthi rs	Move to hi		R-Type
mflo rd	Move from lo		R-Type
mtlo rs	Move to lo		R-Type

Load/Store Instructions

Instruction			Description	Instruction Format
lb	rd,	adr_16b (rs)	Load byte	I-Type
lbu	rd,	adr_16b (rs)	Load byte unsigned	I-Type
lh	rd,	adr_16b (rs)	Load halfword	I-Type
lhu	rd,	adr_16b (rs)	Load halfword unsigned	I-Type
lw	rd,	adr_16b (rs)	Load word	I-Type
sb	rt,	adr_16b (rs)	Store byte	I-Type
sh	rt,	adr_16b (rs)	Store halfword	I-Type
sh	rt,	adr_16b (rs)	Store word	I-Type

Load/Store Instructions

```
lw    $s1, 0($zero)      # $s1←0X03020100
```

00000000	03	02	01	00
00000004	07	06	05	04
00000008	0B	0A	09	08
0000000C				
00000010				
…				
00000124	F3	F2	F1	F0
…				

```
$t1 = 00000024
lw    $s1, 100($t1)      # $s1←0XF3F2F1F0
lbu   $s1, 101($t1)      # $s1←0X000000F1
lb    $s1, 101($t1)      # $s1←0XFFFFFFF1
```

Load/Store Instructions

```
lw $t0, 4($zero)          # OK
lw $t0, 7($zero)          # BAD: 7 mod 4 = 1
lw $t0, 8($zero)          # OK

lh $t0, 2($zero)          # OK
lh $t0, 5($zero)          # BAD: 5 mod 2 = 1
lh $t0, 12($zero)         # OK
```

Jump/Branch Instructions

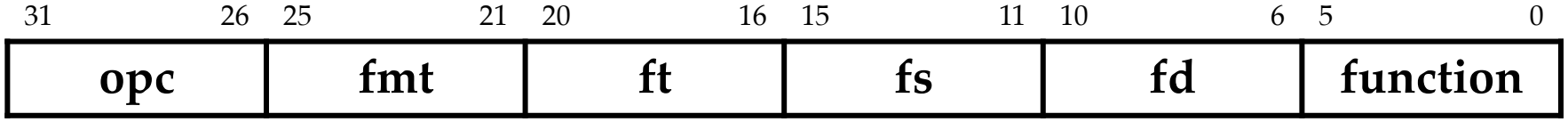
Instruction	Description	Instruction Format
j adr_26bit	Jump	J-Type
jal adr_26bit	Jump and link	J-Type
jr rs	Jump register	R-Type
jalr rs	Jump and link register	R-Type
beq rs, rt, adr_16bit	Branch equal	I-Type
bne rs, rt, adr_16bit	Branch not equal	I-Type

Floating Point (FP) Registers

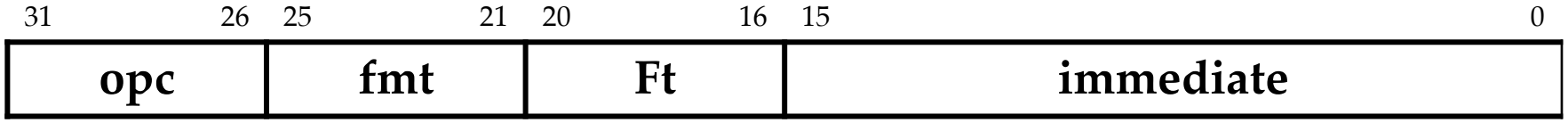
- $f0-$f31
  - $f0 is not hardwired to 0.0
  - Each can be used as a 32-bit single precision FP (SPFP) number
    - $f0, $f1, $f2, …
  - Two consecutive registers considered as a 64-bit double precision FP (DPFP) number
    - $f0, $f2, $f4

MIPS FP Instruction Format

- Fixed length (4-byte aligned)
- Instruction Formats:
  - FR-Type:



  -
  - FI-Type:



Floating Point Instructions (SP)

Instruction	Description	Instruction Format
add.s rd, rs, rt	Add single precision	FR-Type
sub.s rd, rs, rt	Sub single precision	FR-Type
mul.s rd, rs, rt	Multiply single precision	FR-Type
div.s rd, rs, rt	Divide single precision	FR-Type
abs.s rd, rs	Absolute value single precision	FR-Type
neg.s rd, rs	Negate single precision	FR-Type
l.s rt, adr_16bit	Load single precision	FI-Type
s.s rt, adr_16bit	Store single precision	FI-Type
c.eq.s rs, rt	Compare equal single precision	FR-Type
c.lt.s rs, rt	Compare less than single precision	FR-Type

Floating Point Instructions (SP)

Instruction	Description	Instruction Format
c.eq.s rs, rt	Compare equal single precision *if (rs == rt) cond_bit* = 1;* *else cond_bit = 0;*	FR-Type
c.lt.s rs, rt	Compare less than single precision *if (rs < rt) cond_bit = 1;* *else cond_bit = 0;*	FR-Type
c.le.s rs, rt	Compare less than or equal single precision *if (rs <= rt) cond_bit = 1;* *else cond_flag = 0;*	FR-Type

* Condition bit is in floating point unit

Floating Point Instructions (DP)

Instruction	Description	Instruction Format
add.d rd, rs, rt	Add double precision	FR-Type
sub.d rd, rs, rt	Sub double precision	FR-Type
mul.d rd, rs, rt	Multiply double precision	FR-Type
div.d rd, rs, rt	Divide double precision	FR-Type
abs.d rd, rs	Absolute value double precision	FR-Type
neg.d rd, rs	Negate double precision	FR-Type
l.d rt, adr_16bit	Load double precision	FI-Type
s.d rt, adr_16bit	Store double precision	FI-Type
c.eq.d rs, rt	Compare equal double precision	FR-Type
c.lt.d rs, rt	Compare less than double precision	FR-Type

Floating Point Instructions (DP)

Instruction	Description	Instruction Format
c.eq.d rs, rt	Compare equal double precision *if (rs == rt) cond_bit = 1;* *else cond_bit = 0;*	R-Type
c.lt.d rs, rt	Compare less than double precision *if (rs < rt) cond_bit = 1;* *else cond_bit = 0;*	R-Type
c.le.d rs, rt	Compare less than or equal double precision *if (rs <= rt) cond_bit = 1;* *else cond_flag = 0;*	R-Type

Conditional Branch (FP)

Instruction		Description	Instruction Format
bc1t	adr_16bit	*Branch if cond_bit is 1*	FI-Type
bc1f	adr_16bit	*Branch if cond_bit is 0*	FI-Type

Pseudo Instructions

Instruction	Description	Instruction (s)
b adr_16bit	Branch always	**beq** $zero, $zero, adr_16bit
beqz rs, adr_16bit	Branch if equal zero	**beq** rs, $zero adr_16bit
bge rs, rt, adr_16bit	Branch if grater than or equal	**slt** $at, rs, rt **beq** $at $zero adr_16bit
bgeu rs, rt, adr_16bit	Branch if grater than or equal unsigned	**sltu** $at, rs, rt **beq** $at $zero adr_16bit
bgt rs, rt, adr_16bit	Branch if grater than	**slt** $at, rt, rs **bne** $at $zero adr_16bit
bgtu rs, rt, adr_16bit	Branch if grater than unsigned	**sltu** $at, rt, rs **bne** $at $zero adr_16bit

Pseudo Instructions

Instruction	Description	Instruction (s)
li rd, data_32bit	Load immediate	**lui** rd, HI_16(data_32bit) **ori** rd, rd, LO_16(data_32bit)
move rd, rs	Move	**or** rd, rs, $zero
mul rd, rs, rt	Multiply	**mult** rs, rt **mflo** rd
negu rd, rs	Negate unsigned	**subu** rd, $zero, Rs
seq rd, rs, rt	Set if equal	**xor** rd, rs, rt **sltiu** rd, rd, 1
sge rd, rs, rt	Set if greater or equal	**slt** rd, rs, rt **xori** rd, rd, 1

Pseudo Instructions

Instruction	Description	Instruction (s)
sgeu rd, rs, rt	Set if greater than or equal unsigned	**sltu** rd, rs, rt **xori** rd, rd, 1
sgt rd, rs, rt	Set if greater than	**slt** rd, rt, rs

Expression

Expression: x + y + z × (w + 3)

Expression 1: ((w + 3) × z) + y) + x

w	$s0
x	$s1
y	$s2
z	$s3

Need only one temporary register

```
addiu   $t0, $s0, 3
mul     $t0, $t0, $s3
addu    $t0, $t0, $s2
addu    $t0, $t0, $s1
```

Expression

Expression: x + y + z × (w + 3)

Expression 1: (x + y) + ((w + 3) × z)

w	$a0
x	$a1
y	$a2
z	$a3

Need two temporary registers

```
addu    $t0, $s1, $s2
addiu   $t1, $s0, 3
mul     $t1, $t1, $s3
add     $t0, $t0, $t1
```

Conditional Expression

```
if ((x - y) < 3)
    x = x + y;
else
    x = x - y;
```

```
        subu    $t0, $s0, $s1       # x - y
        slti    $t0, $t0, 3         # (x-y)<3
        beq     $t0, $zero, ELSE
        addu    $s0, $s0, $s1       # x += y
        b       DONE
ELSE:   addu    $s0, $s0, $s1       # x -= y
DONE:
```

While Loop

```
sum = 0;
cnt = 0;
while (cnt != 10) {
    sum += cnt;
    cnt += 1;
}
```

```
        move   $s0, $zero          # sum = 0
        move   $s1, $zero          # cnt = 0
        li     $t0, 10
        b      TEST                # test first
BODY:   addu   $s0, $s0, $s1       # sum += cnt
        addiu  $s1, $s1, 1         # cnt += 1
TEST:   bne    $s1, $t0, BODY      # cnt != 10?
```

For Loop

```
sum = 0;
for (i = 0; i < 100; i++)
    sum += A[i];
```

```
        move   $s0, $zero          # sum = 0
        move   $s1, $zero          # i = 0
        li     $t0, 400
        b      TEST                # test first
BODY:   lw     $t1, A ($t0)        # load A[i]
        add    $s0, $s0, $t1       # sum += A[i]
        addiu  $s1, $s1, 4         # i += 4
TEST:   bne    $s1, $t0, BODY      # i < 400?
```

Summary

- Topics covered
  - MIPS integer / floating point general purpose registers
  - MIPS instruction format
  - MIPS integer instructions
  - MIPS floating point instructions
  - How to convert high-level programming constructs to MIPS assembly program