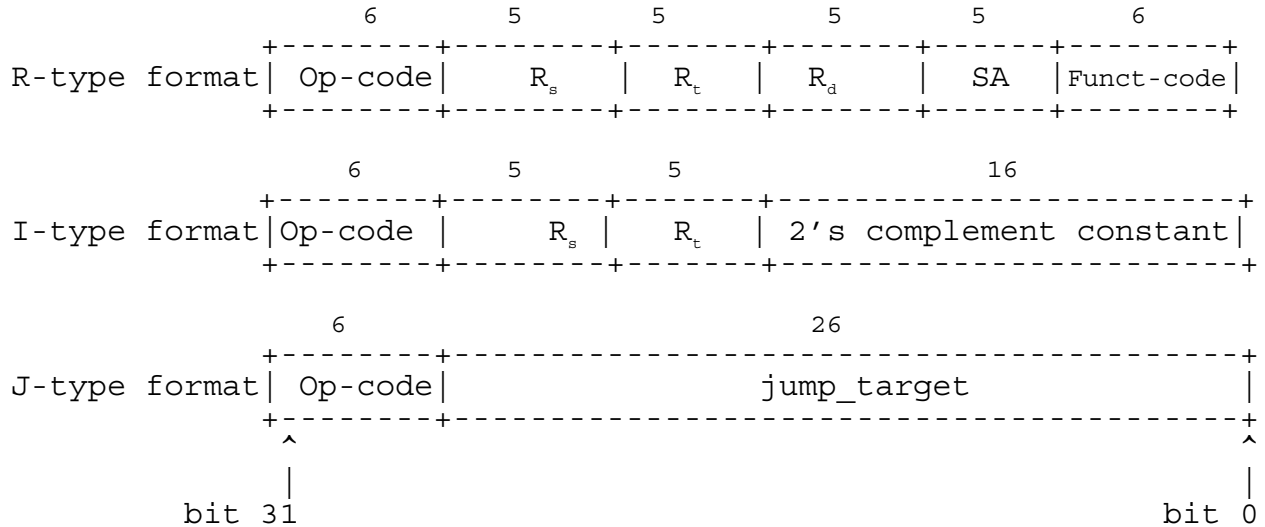


MIPS Instructions

Note: You can have this handout on both exams.

Instruction Formats:

Instruction formats: all 32 bits wide (one word):

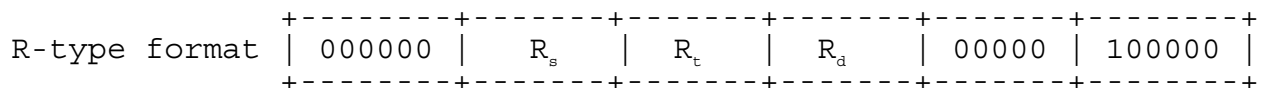


Instructions and their formats

General notes:

- a. R_s , R_t , and R_d specify general purpose registers
- b. Square brackets ([]) indicate "the contents of"
- c. [PC] specifies the address of the instruction in execution
- d. **I** specifies part of instruction and its subscripts indicate bit positions of sub-fields
- e. || indicates concatenation of bit fields
- f. Superscripts indicate repetition of a binary value
- g. $M\{i\}$ is a value (contents) of the word beginning at the memory address i
- h. $m\{i\}$ is a value (contents) of the byte at the memory address i
- i. all integers are in 2's complement representation if not indicated as unsigned

1. addition with overflow: **add** instruction



Effects of the instruction: $R_d \leftarrow [R_s] + [R_t]$; $PC \leftarrow [PC] + 4$
 (If overflow then exception processing)

Assembly format: **add** R_d, R_s, R_t

2. *add without overflow*: **addu** instruction

Identical as **add** instruction, except:

- funct=33_{dec}
- overflow ignored

3. *subtract with overflow*: **sub** instruction

```
R-type format | 000000 | Rs | Rt | Rd | 00000 | 100010 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $R_d \leftarrow [R_s] - [R_t]$; $PC \leftarrow [PC] + 4$
(If overflow then exception processing)

Assembly format: **sub** R_d, R_s, R_t

4. *subtract without overflow*: **subu** instruction

Identical as **sub** instruction, except:

- funct=35_{dec}
- overflow ignored

5. *multiply*: **mul** instruction

```
R-type format | 000000 | Rs | Rt | 00000 | 00000 | 011000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: Hi | Lo $\leftarrow [R_s] * [R_t]$; $PC \leftarrow [PC] + 4$

Assembly format: **mult** R_s, R_t

6. *unsigned multiply*: **mulu** instruction

Identical as **mut** instruction, except:

- funct = 25_{dec}
- contents of R_s and R_t are considered as unsigned integers

7. *divide*: **div** instruction

```
R-type format | 000000 | Rs | Rt | 00000 | 00000 | 011010 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: Lo $\leftarrow [R_s] / [R_t]$; Hi $\leftarrow [R_s] \bmod [R_t]$
 $PC \leftarrow [PC] + 4$

Assembly format: **div** R_s, R_t

8. *unsigned divide*: **divu** instruction

Identical as **div** instruction, except:

- funct = 27_{dec}
- contents of R_s and R_t are considered as unsigned integers

9. *set less than*: **slt** instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 |  Rs  |  Rt  |  Rd  | 00000 | 101010 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction:

if $[R_s] < [R_t]$ then $R_d \leftarrow 0^{31} || 1$ else $R_d \leftarrow 0^{32}$; $PC \leftarrow [PC] + 4$

Assembly format: **slt R_d,R_s,R_t**

10. *set less than unsigned*: **sltu** instruction

Identical as **slt** instruction, except:

- funct = 43_{dec}
- contents of R_s and R_t are considered as unsigned integers.

11. *logical and*: **and** instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 |  Rs  |  Rt  |  Rd  | 00000 | 100100 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $R_d \leftarrow [R_s] \text{ AND } [R_t]$; $PC \leftarrow [PC] + 4$

Assembly format: **and R_d,R_s,R_t**

12 - 14. *logical or, nor & exclusive or*: **or**, **nor**, & **xor** instructions

Identical as **and** instruction, except:

- funct=37_{dec} for **or** instruction
- funct=39_{dec} for **nor** instruction
- funct=40_{dec} for **xor** instruction
- appropriate logical function performed instead of logical **and**

15. *addition immediate with overflow*: **addi** instruction

```
+-----+-----+-----+-----+-----+-----+
I-type format: | 001000 |  Rs  |  Rt  |           immediate           |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction:

$R_t \leftarrow [R_s] + ([I_{15}]^{16} || [I_{15..0}])$; $PC \leftarrow [PC] + 4$
(If overflow then exception processing)

Assembly format: **addi R_t,R_s,immediate**

16. *addition immediate without overflow*: **addiu** instruction

Identical as **addi** instruction, except:

- op-code=9_{dec}
- overflow ignored

17. *set less than immediate*: **slti** instruction

```
+-----+-----+-----+-----+
I-type format: | 001010 |  Rs  |  Rt  |      immediate      |
+-----+-----+-----+-----+
```

Effects of the instruction:

if $[R_s] < ([I_{15}]^{16} || [I_{15..0}])$ then $R_t \leftarrow 0^{31} || 1$ else $R_t \leftarrow 0^{32}$
PC $\leftarrow [PC] + 4$

Assembly format: **slti R_t,R_s,immediate**

18. *set less than immediate unsigned*: **sltiu** instruction

Identical as **slti** instruction, except:

- op-code = 11_{dec}
- contents in the comparison are considered as unsigned integers.

19. *logical and immediate*: **andi** instruction

```
+-----+-----+-----+-----+
I-type format: | 001100 |  Rs  |  Rt  |      immediate      |
+-----+-----+-----+-----+
```

Effects of the instruction: $R_t \leftarrow [R_s] \text{ AND } (0^{16} || [I_{15..0}]);$
PC $\leftarrow [PC] + 4$

Assembly format: **andi R_t,R_s,immediate**

20-21. *logical or immediate & xor immediate*: **ori**, & **xori** instr.

Identical as **andi** instruction, except:

- op-code=13_{dec} for **ori** instruction
- op-code=14_{dec} for **xori** instruction
- appropriate logical function performed instead of logical **and**

22. *load word*: **lw** instruction

```
+-----+-----+-----+-----+
I-type format: | 100011 | Rs  |  Rt  |      offset      |
+-----+-----+-----+-----+
```

Effects of the instruction: $R_t \leftarrow M\{[R_s] + [I_{15}]^{16} || [I_{15..0}]\}$
PC $\leftarrow [PC] + 4$

(If an illegal memory address then exception processing)

Assembly format: **lw R_t,offset(R_s)**

23. *store word*: **sw** instruction

```
+-----+-----+-----+-----+
I-type format: | 101011 | Rs  |  Rt  |      offset      |
+-----+-----+-----+-----+
```

Effects of the instruction: $M\{[R_s] + [I_{15}]^{16} || [I_{15..0}]\} \leftarrow [R_t]$
PC $\leftarrow [PC] + 4$

(If an illegal memory address then exception processing)

Assembly format: **sw R_t,offset(R_s)**

24. load unsigned byte: `lbu` instruction

I-type format:

	+-----+	+-----+	+-----+	+-----+	
I-type format:	100100	R _s	R _t	offset	
	+-----+	+-----+	+-----+	+-----+	

Effects of the instruction: $R_t \leftarrow 0^{24} \parallel m\{[R_s] + [I_{15}]^{16} \parallel [I_{15..0}]\}$
 $PC \leftarrow [PC] + 4$

(If an illegal memory address then exception processing)

Assembly format: **`lbu Rt,offset(Rs)`**

25. load byte: `lb` instruction

Identical as `lbu` instruction, except:

- leftmost 24 bits of R_t are loaded by a value of leftmost bit of the byte instead of zeros
- op-code = 32_{dec}

26. store byte: `sb` instruction

I-type format:

	+-----+	+-----+	+-----+	+-----+	
I-type format:	101000	R _s	R _t	offset	
	+-----+	+-----+	+-----+	+-----+	

Effects of the instruction: $m\{[R_s] + [I_{15}]^{16} \parallel [I_{15..0}]\} \leftarrow [R_t]_{7..0}$
 $PC \leftarrow [PC] + 4$

(If an illegal memory address then exception processing)

Assembly format: **`sb Rt,offset(Rs)`**

27. load upper immediate: `lui` instruction

I-type format:

	+-----+	+-----+	+-----+	+-----+	
I-type format:	001111	00000	R _t	immediate	
	+-----+	+-----+	+-----+	+-----+	

Effects of the instruction: $R_t \leftarrow [I_{15..0}] \parallel 0^{16}$; $PC \leftarrow [PC] + 4$

Assembly format: **`lui Rt,immediate`**

28. branch on equal: `beq` instruction

I-type format:

	+-----+	+-----+	+-----+	+-----+	
I-type format:	000100	R _s	R _t	offset	
	+-----+	+-----+	+-----+	+-----+	

Effects of the instruction:

if $[R_s] = [R_t]$ then $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \parallel [I_{15..0}] \parallel 0^2)$
(i.e. $PC \leftarrow [PC] + 4 + 4 \cdot \text{offset}$)

else $PC \leftarrow [PC] + 4$

Assembly format: **`beq Rs,Rt,offset`**

29. *branch on not equal*: **bne** instruction

```
+-----+-----+-----+-----+
I-type format: | 000101 | Rs   | Rt   |          offset          |
+-----+-----+-----+-----+
```

Effects of the instruction:

```
if [Rs] <> [Rt] then PC <-- [PC] + 4 + ([I15]14 || [I15..0] || 02)
    else PC <-- [PC] + 4
```

Assembly format: **bne R_s,R_t,offset**

30. *branch on less than or equal zero*: **blez** instruction

```
+-----+-----+-----+-----+
I-type format: | 000110 | Rs   | 00000 |          offset          |
+-----+-----+-----+-----+
```

Effects of the instruction:

```
if [Rs] ≤ 0 then PC <-- [PC] + 4 + ([I15]14 || [I15..0] || 02)
    else PC <-- [PC] + 4
```

Assembly format: **blez R_s,offset**

31. *branch on greater than zero*: **bgtz** instruction

```
+-----+-----+-----+-----+
I-type format: | 000111 | Rs   | 00000 |          offset          |
+-----+-----+-----+-----+
```

Effects of the instruction:

```
if [Rs] > 0 then PC <-- [PC] + 4 + ([I15]14 || [I15..0] || 02)
    else PC <-- [PC] + 4
```

Assembly format: **bgtz R_s,offset**

32. *branch on less than zero*: **bltz** instruction

```
+-----+-----+-----+-----+
I-type format: | 000001 | Rs   | 00000 |          offset          |
+-----+-----+-----+-----+
```

Effects of the instruction:

```
if [Rs] < 0 then PC <-- [PC] + 4 + ([I15]14 || [I15..0] || 02)
    else PC <-- [PC] + 4
```

Assembly format: **bltz R_s,offset**

33. *jump*: **j** instruction

```
+-----+-----+-----+-----+
J-type format|000010 |          jump_target          |
+-----+-----+-----+-----+
```

Effects of the instruction: PC <-- [PC_{31..28}] || [I_{25..0}] || 0²

Assembly format: **j jump_target**

34. jump and link: jal instruction

```
+-----+-----+-----+-----+
J-type format|000011 |                jump_target                |
+-----+-----+-----+-----+
```

Effects of the instruction: $R_{31} \leftarrow [PC] + 4$
 $PC \leftarrow [PC_{31..28}] || [I_{25..0}] || 0^2$
Assembly format: **jal jump_target**

35. jump register: jr instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 |  Rs | 00000 | 00000 | 00000 | 001000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $PC \leftarrow [R_s]$
Assembly format: **jr R_s**

36. jump and link register: jalr instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 |  Rs | 00000 |  Rd  | 00000 | 001001 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $R_d \leftarrow [PC] + 4$; $PC \leftarrow [R_s]$
Assembly format: **jalr R_d, R_s**

37. no operation: nop instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 | 00000 | 00000 | 00000 | 00000 | 000000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $PC \leftarrow [PC] + 4$
Assembly format: **nop** (= **sll R₀, 0** shift logical left 0)

38. move from Hi: mfhi instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 | 00000 | 00000 |  Rd  | 00000 | 010000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $R_d \leftarrow [Hi]$; $PC \leftarrow [PC] + 4$
Assembly format: **mfhi R_d**

39. move from Lo: mflo instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 000000 | 00000 | 00000 |  Rd  | 00000 | 010010 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: $R_d \leftarrow [Lo]$; $PC \leftarrow [PC] + 4$
Assembly format: **mflo R_d**

Exception Handling

When a condition for any exception (overflow, illegal op-code, division by zero, etc.) occurs the following hardware exception processing is performed:

```
EPC <-- [PC]
Cause_Reg <--  $\left\{ \begin{array}{l} 0^{28} \quad || \quad 1010 \quad \text{if illegal op-code (10)} \\ 0^{28} \quad || \quad 1100 \quad \text{if overflow (12)} \\ 0^{29} \quad || \quad 100 \quad \text{if illegal memory address (4)} \\ \dots\dots\dots \end{array} \right.$  etc.
PC <-- 80000180hex
```

40. move from EPC: **mfepc** instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 010000 | 00000 | Rt | 01110 | 00000 | 000000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: R_d <-- [EPC]; PC <-- [PC] + 4
Assembly format: **mfepc** R_t (This is mfc0 Rt,CP0reg14)

41. move from Cause_Reg: **mfco** instruction

```
+-----+-----+-----+-----+-----+-----+
R-type format | 010000 | 00000 | Rd | 01101 | 00000 | 000000 |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: R_d <-- [Cause_Reg]; PC <-- [PC] + 4
Assembly format: **mfco** R_t (This is mfc0 Rt,CP0reg13)

Floating Point Instructions

42. load word into co-processor 1: **lwcl** instruction

```
+-----+-----+-----+-----+-----+-----+
I-type format: | 110001 | Rs | ft | offset |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: f_t <-- M{ [R_s] + [I₁₅]¹⁶ || [I_{15..0}] }
PC <-- [PC] + 4
Assembly format: **lwcl** f_t,offset(R_s)

43. store word from co-processor 1: **swcl** instruction

```
+-----+-----+-----+-----+-----+-----+
I-type format: | 111001 | Rs | ft | offset |
+-----+-----+-----+-----+-----+-----+
```

Effects of the instruction: M{ [R_s] + [I₁₅]¹⁶ || [I_{15..0}] } <-- [f_t]
PC <-- [PC] + 4
Assembly format: **swcl** f_t,offset(R_s)

44. *addition single precision*: **add.s** instruction

R-type format	010001	00000	f_t	f_s	f_d	000000	
---------------	--------	-------	-------	-------	-------	--------	--

Effects of the instruction: $f_d \leftarrow [f_s] + [f_t]$; $PC \leftarrow [PC] + 4$
(If overflow then exception processing)

Assembly format: **add.s** R_d, R_s, R_t

45. *addition double precision*: **add.d** instruction

R-type format	010001	00001	f_t	f_s	f_d	0000000	
---------------	--------	-------	-------	-------	-------	---------	--

Effects of the instruction: $f_d \leftarrow [f_s] + [f_t]$; $PC \leftarrow [PC] + 4$
(If overflow then exception processing)

Assembly format: **add.d** f_d, f_s, f_t

45. *subtract single precision*: **sub.s** instruction

Similar as add.s but with funct=1

46. *subtract double precision*: **sub.d** instruction

Similar as add.d but with funct=1

47. *multiply single precision*: **mul.s** instruction

Similar as add.s but with funct=2

48. *multiply double precision*: **mul.d** instruction

Similar as add.d but with funct=2

49. *divide single precision*: **div.s** instruction

Similar as add.s but with funct=3

50. *divide double precision*: **div.d** instruction

Similar as add.d but with funct=3