CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #9

# MIPS

## Part III: Instruction Formats and Encoding

NUS National University of Singapore | School of Computing

# Lecture #9: MIPS Part 3: Instruction Formats

# Lecture #9: MIPS Part 3: Instruction Formats

# 1. Overview and Motivation

- **Recap:** Assembly instructions will be translated to **machine code** for actual execution
  - This section shows how to translate MIPS assembly code into binary patterns
- Explains some of the "strange facts" from earlier:
  - Why is ***immediate*** limited to 16 bits?
  - Why is ***shift*** amount only 5 bits?
  - etc.
- Prepare us to "build" a MIPS processor in later lectures!

# 2. MIPS Encoding: Basics

- Each MIPS instruction has a **fixed-length** of 32 bits

  ➔ All relevant information for an operation must be encoded with these bits!

- Additional challenge:
  - To reduce the complexity of processor design, the instruction encodings should be as regular as possible

    ➔ Small number of formats, i.e. as few variations as possible

# 3. MIPS Instruction Classification

- Instructions are classified according to their operands:
  ➔ Instructions with same operand types have same encoding

**R-format** (Register format: `op $r1, $r2, $r3`)

- Instructions which use 2 source registers and 1 destination register
- `e.g. add, sub, and, or, nor, slt, etc`
- **Special cases:** `srl, sll`, etc.

**I-format** (Immediate format: `op $r1, $r2, Immd`)

- Instructions which use 1 source register, 1 immediate value and 1 destination register
- `e.g. addi, andi, ori, slti, lw, sw, beq, bne`, etc.

**J-format** (Jump format: `op Immd`)

- `j` instruction uses only one immediate value

# 4. MIPS Registers (Recap)

- For simplicity, register numbers (`$0`, `$1`, …, `$31`) will be used in examples here instead of register names

| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | Constant value 0 |
| $v0-$v1 | 2-3 | Values for results and expression evaluation |
| $a0-$a3 | 4-7 | Arguments |
| $t0-$t7 | 8-15 | Temporaries |
| $s0-$s7 | 16-23 | Program variables |

| Name | Register number | Usage |
|---|---|---|
| $t8-$t9 | 24-25 | More temporaries |
| $gp | 28 | Global pointer |
| $sp | 29 | Stack pointer |
| $fp | 30 | Frame pointer |
| $ra | 31 | Return address |

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

# 5. R-Format (1/2)

- Define fields with the following number of bits each:
  - 6 + 5 + 5 + 5 + 5 + 6 = 32 bits

| | | | | | |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

- Each field has a name:

| | | | | | |
|---|---|---|---|---|---|
| `opcode` | `rs` | `rt` | `rd` | `shamt` | `funct` |

- Each field is an independent 5- or 6-bit unsigned integer
  - A 5-bit field can represent any number 0 – 31
  - A 6-bit field can represent any number 0 – 63

# 5. R-Format (2/2)

| Fields | Meaning |
|---|---|
| **`opcode`** | - Partially specifies the instruction<br>- Equal to 0 for all R-Format instructions |
| **`funct`** | - Combined with opcode exactly specifies the instruction |
| **`rs`** (Source Register) | - Specify register containing first operand |
| **`rt`** (Target Register) | - Specify register containing second operand |
| **`rd`** (Destination Register) | - Specify register which will receive result of computation |
| **`shamt`** | - Amount a shift instruction will shift by<br>- 5 bits (i.e. 0 to 31)<br>- Set to 0 in all non-shift instructions |

# 5.1 R-Format: Example (1/3)

**MIPS instruction**

```
add  $8, $9, $10
```

| R-Format Fields | Value | Remarks |
|---|---|---|
| `opcode` | 0 | (textbook pg 94 - 101) |
| `funct` | 32 | (textbook pg 94 - 101) |
| `rd` | 8 | (destination register) |
| `rs` | 9 | (first operand) |
| `rt` | 10 | (second operand) |
| `shamt` | 0 | (not a shift instruction) |

# 5.1 R-Format: Example (2/3)

**MIPS instruction**

```
add $8, $9, $10
```

Note the ordering of the 3 registers

Field representation in decimal:

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 9 | 10 | 8 | 0 | 32 |

Field representation in binary:

| | | | | | |
|---|---|---|---|---|---|
| 000000 | 01001 | 01010 | 01000 | 00000 | 100000 |

Split into 4-bit groups for hexadecimal conversion:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0000 | 0001 | 0010 | 1010 | 0100 | 0000 | 0010 | 0000 |
| $0_{16}$ | $1_{16}$ | $2_{16}$ | $A_{16}$ | $4_{16}$ | $0_{16}$ | $2_{16}$ | $0_{16}$ |

# 5.1 R-Format: Example (3/3)

**MIPS instruction**

```
sll  $8, $9, 4
```

Note the placement of the source register

Field representation in decimal:

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 0 | 9 | 8 | 4 | 0 |

Field representation in binary:

| | | | | | |
|---|---|---|---|---|---|
| 000000 | 00000 | 01001 | 01000 | 00100 | 000000 |

Split into 4-bit groups for hexadecimal conversion:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0000 | 0000 | 0000 | 1001 | 0100 | 0001 | 0000 | 0000 |
| $0_{16}$ | $0_{16}$ | $0_{16}$ | $9_{16}$ | $4_{16}$ | $1_{16}$ | $0_{16}$ | $0_{16}$ |

# 5.2 Try It Yourself #1

**MIPS instruction**

```
add  $10, $7, $5
```

Field representation in decimal:

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 7 | 5 | 10 | 0 | 32 |

Field representation in binary:

| | | | | | |
|---|---|---|---|---|---|
| 000000 | 00111 | 00101 | 01010 | 00000 | 100000 |

Hexadecimal representation of instruction:

$0\ 0\ E\ 5\ 5\ 0\ 2\ 0_{16}$

# 6. I-format (1/4)

- What about instructions with immediate values?
  - 5-bit `shamt` field can only represent **0 to 31**
  - Immediates may be much larger than this
    - e.g. `lw`, `sw` instructions require bigger offset

- **Compromise:** Define a new instruction format partially consistent with R-format:
  - If instruction has immediate, then it uses at most 2 registers

# 6. I-format (2/4)

- Define fields with the following number of bits each:
  - 6 + 5 + 5 + 16 = 32 bits

| 6 | 5 | 5 | 16 |
|---|---|---|---|

- Again, each field has a name:

| opcode | rs | rt | immediate |
|---|---|---|---|

- Only one field is inconsistent with R-format.
  - **`opcode`**, **`rs`**, and **`rt`** are still in the same locations.

# 6. I-format (3/4)

- **`opcode`**
  - Since there is no **`funct`** field, **`opcode`** uniquely specifies an instruction
- **`rs`**
  - specifies the source register operand (if any)
- **`rt`**
  - specifies register to receive result
  - **note the difference from R-format instructions**

- **Continue on next slide……**

# 6. I-format (4/4)

- **`immediate`**:
  - Treated as a ***signed integer***
  - 16 bits → can be used to represent a constant up to $2^{16}$ different values
  - Large enough to handle:
    - The offset in a typical **`lw`** or **`sw`**
    - Most of the values used in the **`addi`**, **`subi`**, **`slti`** instructions

# 6.1 I-format: Example (1/2)

**MIPS instruction**

```
addi  $21, $22, -50
```

| I-Format Fields | Value | Remarks |
|---|---|---|
| `opcode` | 8 | (textbook pg 94 - 101) |
| `rs` | 22 | (the only source register) |
| `rt` | 21 | (target register) |
| `immediate` | -50 | (in base 10) |

# 6.1 I-format: Example (2/2)

| MIPS instruction |
|---|
| `addi $21, $22, -50` |

Field representation in decimal:

| | | | |
|---|---|---|---|
| 8 | 22 | 21 | -50 |

Field representation in binary:

| | | | |
|---|---|---|---|
| 001000 | 10110 | 10101 | 1111111111001110 |

Hexadecimal representation of instruction:

$\mathbf{2\ 2\ D\ 5\ F\ F\ C\ E}_{16}$

# 6.2 Try It Yourself #2

**MIPS instruction**

```
lw   $9, 12($8)
```

Field representation in decimal:

| opcode | rs | rt | immediate |
|---|---|---|---|
| 35 | 8 | 9 | 12 |

Field representation in binary:

| | | | |
|---|---|---|---|
| 100011 | 01000 | 01001 | 0000000000001100 |

Hexadecimal representation of instruction:

$8\ D\ 0\ 9\ 0\ 0\ 0\ C_{16}$

# 6.3 Instruction Address: Overview

- As instructions are stored in memory, they too have addresses
  - Control flow instructions uses these addresses
  - E.g. **`beq`**, **`bne`**, **`j`**
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- **P**rogram **C**ounter (**PC**)
  - A special register that keeps address of instruction being executed in the processor

# 6.4 Branch: PC-Relative Addressing (1/5)

- Use I-Format

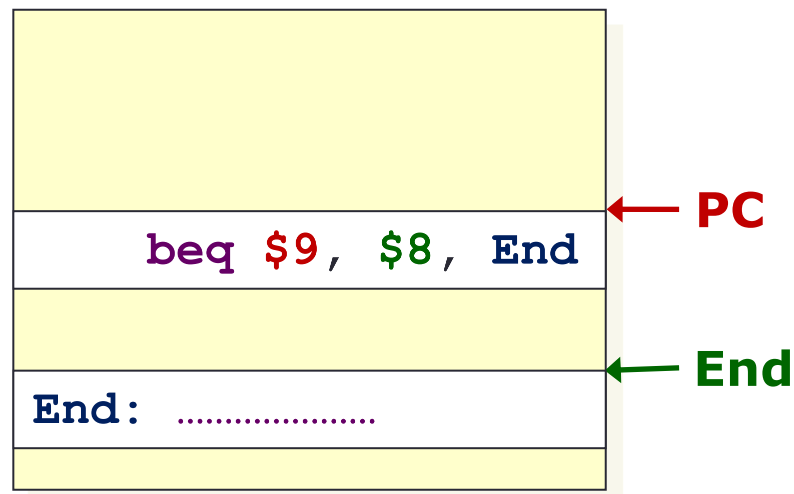
| opcode | rs | rt | immediate |
|---|---|---|---|

- `opcode` specifies `beq`, `bne`
- `rs` and `rt` specify registers to compare
- What can `immediate` specify?
  - `Immediate` is only 16 bits
  - Memory address is 32 bits
  - → `immediate` is not enough to specify the entire target address!

# 6.4 Branch: PC-Relative Addressing (2/5)

- How do we usually use branches?
  - **Answer:** **`if-else`**, **`while`**, **`for`**
  - Loops are generally **small**:
    - Typically up to 50 instructions
  - Unconditional jumps are done using jump instructions (`j`), not the branches

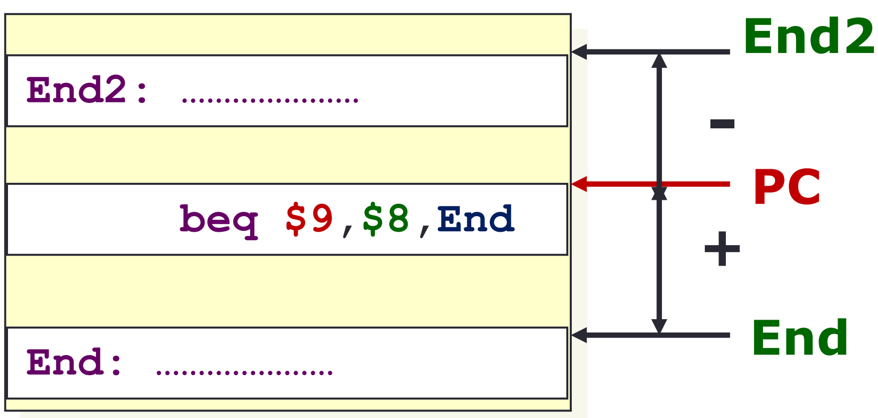
- **Conclusion:** A branch often changes **PC** by a small amount



# 6.4 Branch: PC-Relative Addressing (3/5)

- **Solution:**
  - Specify target address **relative to the PC**
- Target address is generated as:
  - PC + the 16-bit **`immediate`** field
  - The **`immediate`** field is a signed two's complement integer

➔ Can branch to ± $2^{15}$ bytes from the PC:
  - Should be enough to cover most loops



# 6.4 Branch: PC-Relative Addressing (4/5)

- Can the branch target range be enlarged?
- **Observation:** Instructions are word-aligned
  - Number of bytes to add to the PC will always be a multiple of 4.

  ➔ Interpret the `immediate` as number of words, i.e. automatically multiplied by $4_{10}$ $(100_2)$

➔ Can branch to $\pm 2^{15}$ **words** from the PC

  - i.e. $\pm 2^{17}$ bytes from the **PC**
  - We can now branch 4 times farther!

# 6.4 Branch: PC-Relative Addressing (5/5)

- Branch calculation:

If the branch is **not taken**:

**PC = PC + 4**

(**PC + 4** is address of next instruction)

If the branch is **taken**:

**PC = (PC + 4) + (`immediate` × 4)**

- Observations:
  - **`immediate`** field specifies the number of words to jump, which is the same as the number of instructions to "skip over"
  - **`immediate`** field can be positive or negative
  - Due to hardware design, add **`immediate`** to (PC+4), not to PC (more in later topic)

# 6.5 Branch: Example (1/3)

```
Loop:  beq  $9, $0, End   # rlt addr: 0
       add  $8, $8, $10   # rlt addr: 4
       addi $9, $9, -1    # rlt addr: 8
       j    Loop          # rlt addr: 12
End:                      # rlt addr: 16
```

- **beq** is an I-Format instruction →

| I-Format Fields | Value | Remarks |
|---|---|---|
| opcode | 4 | |
| rs | 9 | (first operand) |
| rt | 0 | (second operand) |
| immediate | ??? | (in base 10) |

# 6.5 Branch: Example (2/3)

```
Loop:  beq  $9, $0, End   # rlt addr: 0
       add  $8, $8, $10   # rlt addr: 4
       addi $9, $9, -1    # rlt addr: 8
       j    Loop          # rlt addr: 12
End:                      # rlt addr: 16
```

- `immediate` field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch
  - In `beq` case, **`immediate` = 3**
  - **End = (PC + 4) + (immediate × 4)**

# 6.5 Branch: Example (3/3)

```
Loop:  beq  $9, $0, End    # rlt addr: 0
       add  $8, $8, $10    # rlt addr: 4
       addi $9, $9, -1     # rlt addr: 8
       j    Loop           # rlt addr: 12
End:                       # rlt addr: 16
```

Field representation in decimal:

| opcode | rs | rt | immediate |
|---|---|---|---|
| 4 | 9 | 0 | 3 |

Field representation in binary:

| | | | |
|---|---|---|---|
| 000100 | 01001 | 00000 | 0000000000000011 |

# 6.6 Try It Yourself #3

```
Loop:   beq  $9, $0, End   # rlt addr: 0
        add  $8, $8, $10   # rlt addr: 4
        addi $9, $9, -1    # rlt addr: 8
        beq  $0, $0, Loop  # rlt addr: 12
End:                       # rlt addr: 16
```

- What would be the **`immediate`** value for the second **`beq`** instruction?

Answer: **– 4**

# 7. J-Format (1/5)

- For branches, PC-relative addressing was used:
  - Because we do not need to branch too far
- For general jumps (`j`):
  - We may jump to anywhere in memory!
- The ideal case is to specify a 32-bit memory address to jump to
  - Unfortunately, we can't (☹ why?)

# 7. J-Format (2/5)

- Define fields of the following number of bits each:

| 6 bits | 26 bits |
|---|---|

- As usual, each field has a name:

| opcode | target address |
|---|---|

- Keep **opcode** field identical to R-format and I-format for consistency
- Combine all other fields to make room for larger target address

# 7. J-Format (3/5)

- We can only specify 26 bits of 32-bit address
- **Optimization:**
  - Just like with branches, jumps will only jump to word-aligned addresses, so last two bits are always `00`
  - So, let's assume the address ends with '`00`' and leave them out

➔ Now we can specify **28 bits** of 32-bit address

# 7. J-Format (4/5)

- Where do we get the other 4 bits?
  - MIPS choose to take the **4 most significant bits from PC+4** (the next instruction after the jump instruction)

  ➔ This means that we cannot jump to anywhere in memory, but it should be sufficient ***most of the time***
- Question:
  - What is the **maximum jump range?** 256MB boundary
- Special instruction if the program straddles 256MB boundary
  - Look up `jr` instruction if you are interested
  - Target address is specified through a register

# 7. J-Format (5/5)

- **Summary:** Given a **Jump** instruction

| 32bit PC | opcode | target address |
| --- | --- | --- |
| 1010......... | 000010 | 00001111000011110000111100 |

Jumps To

$$\underbrace{1010}_{\text{Most significant 4bits of PC+4}}\ \underbrace{00001111000011110000111100}_{\text{26bits Target address specified in instruction}}\ \underbrace{00}_{\text{Default 2bit "00" for word address}}$$

# 7.1 J-Format: Example

```
Loop:   beq  $9, $0, End   # addr: 8   ← jump target
        add  $8, $8, $10   # addr: 12
        addi $9, $9, -1    # addr: 16
        j    Loop          # addr: 20  ← PC
End:                       # addr: 24
```

| | |
|---|---|
| PC | 0000..........................10100 |
| Address to jump to | 0000..........................01000 |

immediate = **2** (26 bits)

Check your understanding by constructing the new PC value

| opcode | target address |
|---|---|
| 000010 | 00000000000000000000000010 |

# 7.2 Branching Far Way

- Given the instruction

```
beq $s0, $s1, L1
```

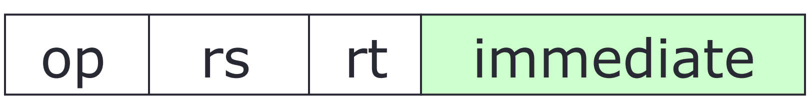
  Assume that the address `L1` is farther away from the PC than can be supported by `beq` and `bne` instructions

- **Challenge:**
  - Construct an equivalent code sequence with the help of unconditional (`j`) and conditional branch (`beq, bne`) instructions to accomplish this far away branching

# 8. Addressing Modes (1/3)

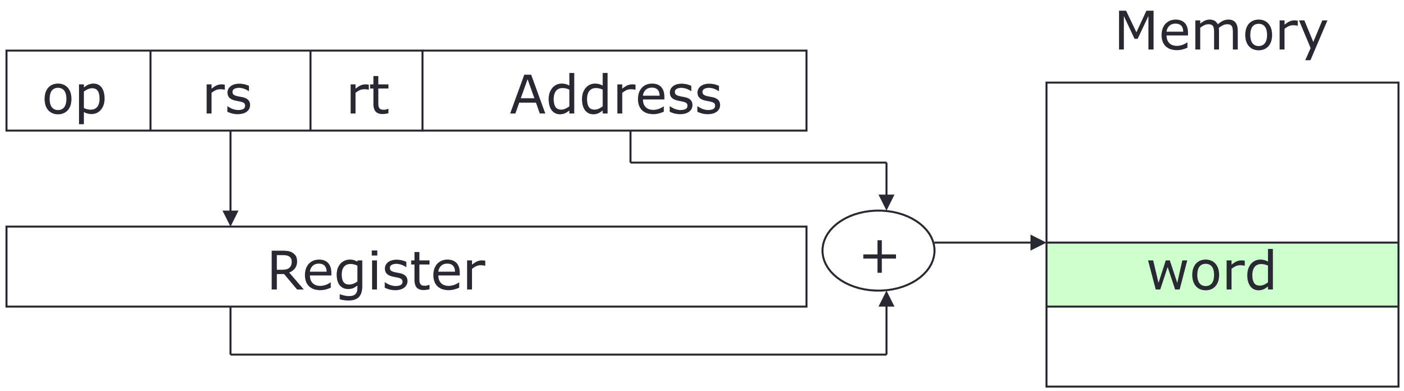
- **Register addressing**: operand is a register


- **Immediate addressing**: operand is a constant within the instruction itself (`addi`, `andi`, `ori`, `slti`)



# 8. Addressing Modes (2/3)

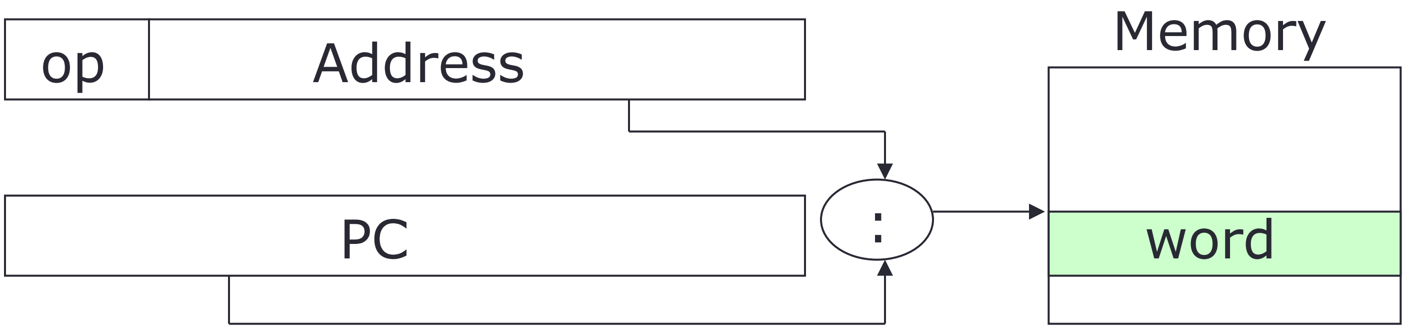
- **Base addressing (displacement addressing)**: operand is at the memory location whose address is sum of a register and a constant in the instruction (`lw`, `sw`)



# 8. Addressing Modes (3/3)

- **PC-relative addressing**: address is sum of PC and constant in the instruction (`beq`, `bne`)


- **Pseudo-direct addressing**: 26-bit of instruction concatenated with upper 4-bits of PC (`j`)



# Summary (1/2)

- MIPS Instruction:
  32 bits representing a single instruction

| | | | | | | |
|---|---|---|---|---|---|---|
| **R** | `opcode` | `rs` | `rt` | `rd` | `shamt` | `funct` |
| **I** | `opcode` | `rs` | `rt` | `immediate` | | |
| **J** | `opcode` | `target address` | | | | |

  - Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
  - Branches use PC-relative addressing; jumps use pseudo-direct addressing
  - Shifts use R-format, but other immediate instructions (**`addi`**, **`andi`**, **`ori`**) use I-format

# Summary (2/2)

| MIPS assembly language | | | | |
|---|---|---|---|---|
| **Category** | **Instruction** | **Example** | **Meaning** | **Comments** |
| Arithmetic | add | `add $s1, $s2, $s3` | `$s1 = $s2 + $s3` | Three operands; data in registers |
| | subtract | `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3` | Three operands; data in registers |
| | add immediate | `addi $s1, $s2, 100` | `$s1 = $s2 + 100` | Used to add constants |
| Data transfer | load word | `lw $s1, 100($s2)` | `$s1` = Memory[`$s2` + 100 | Word from memory to register |
| | store word | `sw $s1, 100($s2)` | Memory[`$s2` + 100] = $s1 | Word from register to memory |
| | load byte | `lb $s1, 100($s2)` | `$s1` = Memory[`$s2` + 100 | Byte from memory to register |
| | store byte | `sb $s1, 100($s2)` | Memory[`$s2` + 100] = $s1 | Byte from register to memory |
| | load upper immediate | `lui $s1, 100` | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| 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 less than immediate | `slti $s1, $s2, 100` | if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0 | Compare less than constant |
| 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 |

# End of File