ELT3047 Computer Architecture

Lecture 4: MIPS ISA (1)

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

Last lecture review

- ISA design is hard
  - Adhere to 4 qualitative principles
  - Applying quantitative method
- Five aspects of ISA design
  - **Data Storage** choices: GPR (load/store, register-memory), Stack, Register-memory, Accumulator.
  - Common **addressing modes**: displacement, immediate, register indirect
  - Most important **operations** are simple instructions (96% of the instructions executed) → *make the common case fast*.
  - **Instruction encoding**: performance vs code size trade-off (fixed- vs variable-length)
  - To support the **compiler performance**: at least 16 (preferably 32) GPR's, aim for a minimalist instruction set, & ensure all addressing modes apply to all data transfer instructions.
- **Today's lecture**: Introduction to MIPS ISA
  - Showing how it follows previously covered design principles.

Overview

- Development
  - First developed at Stanford by Hennessey et al.; later acquired by MIPS Technologies.
  - By the late 2010s, MIPS machines have a large share of embedded core market (automotive, router & modems, microcontrollers).
  - Ceased 2021, moved to RISC-V.
- Why study MIPS?
  - Good architectural model for study: elegant and easy to understand
  - Typical of many modern ISAs
- What will be covered?
  - Application of ISA design principles in 5 aspects covered in week 3
  - Illustrations of SW-HW interface via assembly language

1. Data Storage

2. Memory Addressing Modes

3. Operations in the Instruction Set

4. Encoding the Instruction Set

5. The role of compilers

MIPS storage model

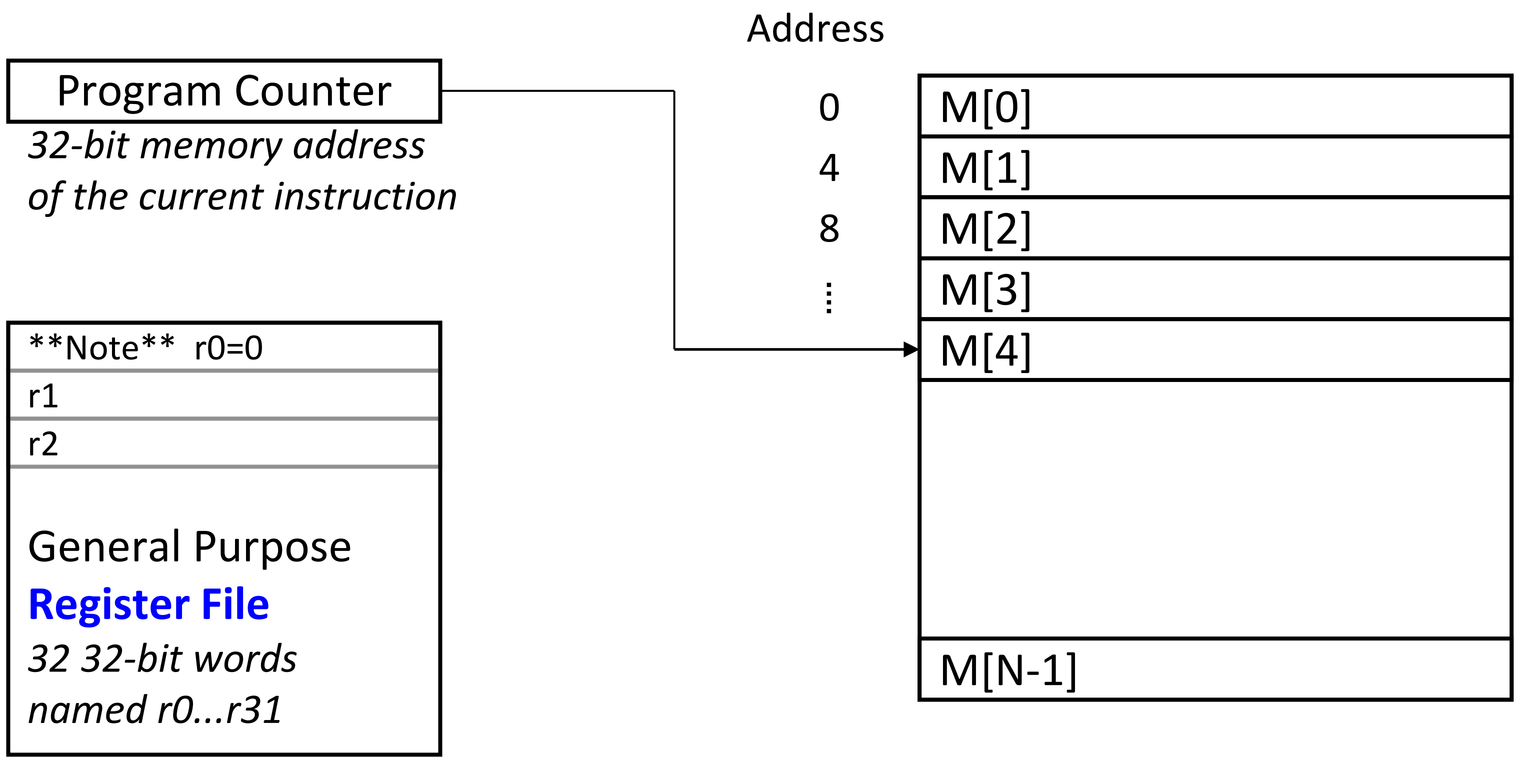
- General-Purpose Register (GPR) with Load/Store design
  - Recap: what are the trade-off? E.g. Stack/Accumulator vs GPR, Load/Store vs Memory-Memory

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

- Quantitative design
  - How many GPR & trade-off? **32**
  - What is the GPR width & trade-off? **32 bit**

5. The role of compilers

MIPS storage model summary



Memory

Alignment is required

Byte addressable

32-bit address (4 GB)

Support both endianness's

1. Data Storage

2. Memory Addressing Modes

3. Operations in the Instruction Set

4. Encoding the Instruction Set

5. The role of compilers

Addressing mode

- Recap:

MIPS uses only the first 3 modes

Addressing mode	*Example*	*Meaning*
Register	Add R4,R3	R4 ← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2-d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

- More modes trade off:
  - ✓ Better support programming constructs (arrays, pointer-based accesses) → reduced number of instructions and code size
  - ✗ More work for the compiler
  - ✗ More complicated HW implementation

1. Data Storage

2. Memory Addressing Modes

3. Operations in the Instruction Set

4. Encoding the Instruction Set

5. The role of compilers

Operations in the instruction set

- MIPS is a RISC ISA
  - vs CISC trade off?
- Operations studied in **this course**
  - Just a subset of real MIPS, sufficient for later implementation process.

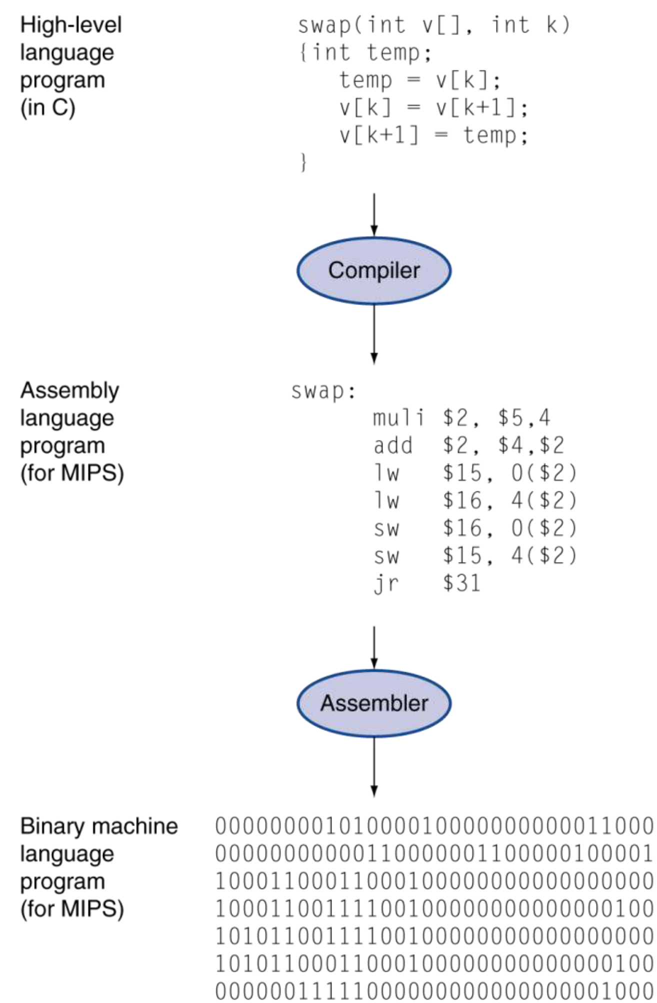
Operator type	**Examples**
Arithmetic and Logical	Integer arithmetic and logical operations: add, and, subtract, or
Data Transfer	Load/Store (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, traps

- Design principles #2, #3: *“smaller is faster”, “make the common case fast”*

5. The role of compilers

Recap: MIPS Assembly Language

- ❑ Architectural representative
  - ➢ Interface btw HLL and machine code
  - ➢ Human-readable format of instructions
- ❑ Each instruction executes a simple command
  - ➢ Usually has a counterpart in high level programming languages like C, Java
- ❑ Each line of assembly code contains at most 1 instruction
  - ➢ **Mnemonic:** operation to perform
  - ➢ **Operands**: ***source*** - on which the operation is performed; ***destination*** - to which the result is written
  - ➢ **# (hex-sign)** is used for comments



1. Data Storage

2. Memory Addressing Modes

3. Operations in the Instruction Set

4. Encoding the Instruction Set

5. The role of compilers

MIPS instruction encoding

- Fixed instruction length = 32 bit
  - Trade-off vs variable length?
- Use a rigid format for instructions in the same class
  - **Design principles #1**: "*simplicity favors regularity*" - regularity makes hardware implementation simpler → higher performance at lower cost.
- MIPS arithmetic instructions
  - Encoding: (**R-type**)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

  - Semantics for **`add $16, $17, $18`** & **`sub $8, $11, $13`** instructions:

`[$16]←[$17]+[$18]`	000000	10001	10010	10000	00000	**100000**
`[$8]←[$11]-[$13]`	000000	01011	01101	01000	00000	**100010**

  - Syntax is **rigid**: 1 operator, 3 operands in 3 registers following a fixed order, note the difference in the operand order between machine code & assembly.

Support for constant operands?

- ISA designers often receive many requests for additional instructions that, in theory, will make the ISA better in some way.
  - Apply quantitative approach to judge the tradeoffs between cost and benefits.
- **Example**: many programs use small constants frequently → should we support them in ALU instructions?
  - **Trade-off**: saves registers, instructions (makes programs shorter) but requires more complex control & datapath logic for additional opcodes.
  - **Quantitative analysis**: simulate the impact of the ISA augmented with this feature by running benchmark programs.
    - >50% of executed arithmetic instructions (e.g. loop increments, scaling indices)
    - >80% of executed compare instructions (e.g. loop termination condition)
    - >25% of executed load instructions (e.g. offsets into data structures)
  - **Conclusion**: constant operands = common case → *make it fast!*

Instructions with immediate operands

- Encoding: (I-type)

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

  - Keep the format as similar as possible to that of the R-type: same bits correspond to the same meaning - op, rs, rt occupy the same location.
  - The constant ranges from [-2^{15} to $2^{15}-1$].
- Semantics:
  - `[rt] ← op{[rs], sign-extend(imm)}`
  - Pseudo-instruction: move = add instruction with **zero** immediate
  - register zero ($0 or $zero) is hardwired to 0 ("*make the common case fast*")
- Assembly instructions use same mnemonics, but with an "i" suffix to indicate the second operand is a constant, e.g. `addi`
  - Why don't we need `subi`? ("*smaller is faster*")

Logical Operations

- Arithmetic instructions view the content of a register as a single quantity (signed or unsigned integer)
- **New perspective**:
  - View register as 32 raw bits rather than as a single 32-bit number → possible to operate on individual bits or bytes within a word
  - Share the same encoding with arithmetic instructions (R- & I- types).

Logical operation	C operator	Java operator	MIPS operator
Shift Left	`<<`	`<<`	`sll`
Shift right	`>>`	`>>, >>>`	`srl`
Bitwise AND	`&`	`&`	`and, andi`
Bitwise OR	`|`	`|`	`or, ori`
Bitwise NOT*	`~`	`~`	`nor`
Bitwise XOR	`^`	`^`	`xor, xori`

Logical Operations: Shifting (1/2)

Opcode: `sll` (**s**hift **l**eft **l**ogical)

Move all the bits in a word to the left by a number of positions; fill the emptied positions with zeroes.

- E.g. Shift bits in **`$s0`** to the left by 4 positions

```
$s0   1011 1000 0000 0000 0000 0000 0000 1001

        sll $t2, $s0, 4   # $t2 = $s0<<4

$t2   1000 0000 0000 0000 0000 0000 1001 0000
```

Logical Operations: Shifting (2/2)

Opcode: `srl` (shift right logical)

Shifts right and fills emptied positions with zeroes.

- What is the equivalent math operations for shifting left/right n bits? Answer: Multiply/divide by 2^n
- Shifting is **faster** than multiplication/division → good compiler translates such operations into shift instructions

C Statement	MIPS Assembly Code
`a = a * 8;`	`sll $s0, $s0, 3`

Logical Operations: Bitwise AND

Opcode: `and` (bitwise AND)

Bitwise operation that leaves a 1 only if both the bits of the operands are 1

- E.g.: `and $t0, $t1, $t2`

	`$t1`	`0110 0011 0010 1111 0000 1101 0101 1001`
mask	`$t2`	`0000 0000 0000 0000 0011 1100 0000 0000`
	`$t0`	`0000 0000 0000 0000 0000 1100 0000 0000`

- **`and`** can be used for **masking** operation:
  - Place 0s into the positions to be ignored → bits will turn into 0s
  - Place 1s for interested positions → bits will remain the same as the original.

Exercise: Bitwise AND

- We are interested in the last 12 bits of the word in register `$t1`. Result to be stored in `$t0`.
  - Q: What's the mask to use?

`$t1`	0000 1001 1100 0011 0101 1101 1001 1100
`mask`	**0000 0000 0000 0000 0000 1111 1111 1111**
`$t0`	**0000 0000 0000 0000 0000 1101 1001 1100**

Notes:
The `and` instruction has an immediate version, `andi`

Logical Operations: Bitwise OR

Opcode: `or` (bitwise OR)

Bitwise operation that that places a 1 in the result if either operand bit is 1

Example: `or $t0, $t1, $t2`

- The `or` instruction has an immediate version `ori`
- Can be used to force certain bits to 1s
- E.g.: `ori $t0, $t1, 0xFFF`

$t1	0000 1001 1100 0011 0101 1101 1001 1100
0xFFF	0000 0000 0000 0000 0000 1111 1111 1111
$t0	0000 1001 1100 0011 0101 1111 1111 1111

Logical Operations: Bitwise NOR

- **Strange fact 1:**
  - There is no **`not`** instruction in MIPS to toggle the bits (1 → 0, 0 → 1)
- However, a **`nor`** instruction is provided:

Opcode:	`nor`	(bitwise nor)
Example:	`nor $t0, $t1, $t2`	

- Question: How do we get a `not` operation?

```
nor $t0, $t0, $zero
```

- Question: Why do you think is the reason for not providing a `not` instruction?

Design principles #3: smaller is faster.

Logical Operations: Bitwise XOR

Opcode:	`xor` (bitwise xor)
Example:	`xor $t0, $t1, $t2`

- Question: Can we also get **`not`** operation from `xor`?

  Yes, let `$t2` contain all 1s:
  `xor $t0, $t0, $t2`

- **Strange Fact 2**:
  - There is no **`nori`**, but there is **`xori`** in MIPS
  - Why?

Data transfer instructions

- **Encoding:**

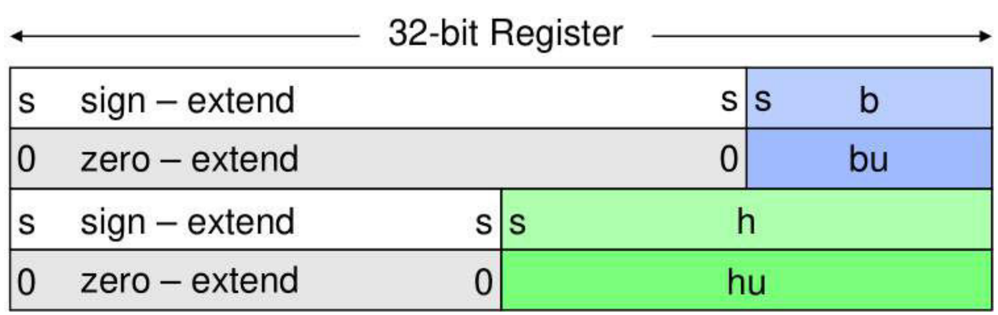
op	rs	rt	imm
6 bits	5 bits	5 bits	16-bit signed constant

- **Semantics**
  - `lw rt, imm(rs)     # rt ← Mem[rs+imm], load a word from Memory`
  - `sw rt, imm(rs)     # rt → Mem[rs+imm], store a word in Memory`
  - In load-store architecture, `lw` and `sw` are the **only** mechanism for accessing memory values.
  - Memory address = **base** (stored in register) + **offset** (stored in instructions): **base addressing mode**, a special case of **displacement addressing.**

- **Two variations of base addressing:**
  - If rs = $zero = 0 then address = imm (*absolute addressing*)
  - If imm = 0 then address = rs (*register indirect addressing*)

Data transfer instructions (cont.)

- Register optimization is important: only keep most frequently used values in the register file ("*make the common case fast*").
  - The process of putting less frequently used variables (or those needed later) into memory is called ***spilling*** registers.
- MIPS processor supports load & store instructions for bytes and halfwords ("*make the common case fast*").
  - Load expands a memory data to fit into a 32-bit register: `lb` = load byte, `lbu` = load byte unsigned, `lh` = load half, `lhu` = load half unsigned
  - Store reduces a 32-bit register to fit in memory: `sb` = store byte, `sh` = store halfword. Why don't we need `sbu` and `shu`?



A load byte example

- Suppose `$t0` initially contains 0x23456789. After the following code runs, what is the value of `$s0`

```
sw $t0, 0($0)
lb $s0, 1($0)
```

  a. in a big-endian system?

  b. in a little-endian system?

- **Solution**:

  a. 0x00000045

  b. 0x00000067

Big-Endian

Byte Address	0	1	2	3
Data Value	23	45	67	89
	MSB			LSB

Word Address: 0

Little-Endian

3	2	1	0	Byte Address
23	45	67	89	Data Value
MSB			LSB	

32-bit constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the *occasional* 32-bit constant, use `lui $rt, constant`
  - **L**oad **u**pper **i**mmediate: copies 16-bit constant to left 16 bits of `$rt` & clears right 16 bits of `$rt` to 0
  - Subsequent instruction specifies the lower 16 bits of the constant.
- **Example**:
  - What is the MIPS assembly code to load this 32-bit constant into `$s0`?

0000 0000 0111 1101 0000 1001 0000 0000

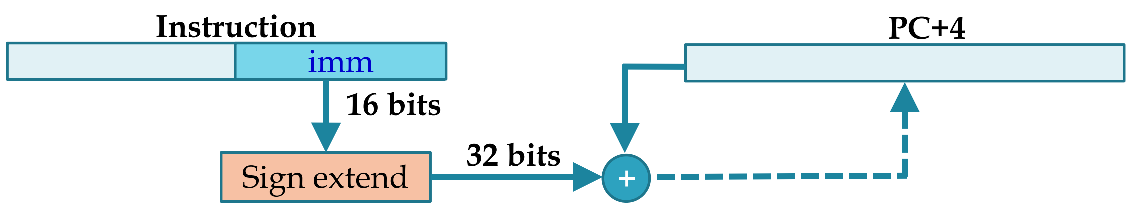
`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

(Conditional) Branch Instructions

- Assembly (e.g., branch if equal).
  - `BEQ rs rt target # assembly uses a` **`symbolic`** `target address`
- Encoding

op	rs	rt	imm
6 bits	5 bits	5 bits	16-bit signed ***instruction*** offset

- Semantics
  - ```
    target = PC + 4 + sign-extend(imm) x 4;
    if [rs] == [rt] then PC ← target else PC ← PC + 4;
    ```



  - This addressing mode is called **PC-relative**. How far can you jump?
- Variations: `BEQ, BNE, BLEZ, BGTZ`
  - Why isn't there a `BLE` or `BGT` instruction?
    ```

Branch encoding example

```
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 ➔

# of instructions to add to (or subtract from) the PC, starting at the instruction **following** the branch

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

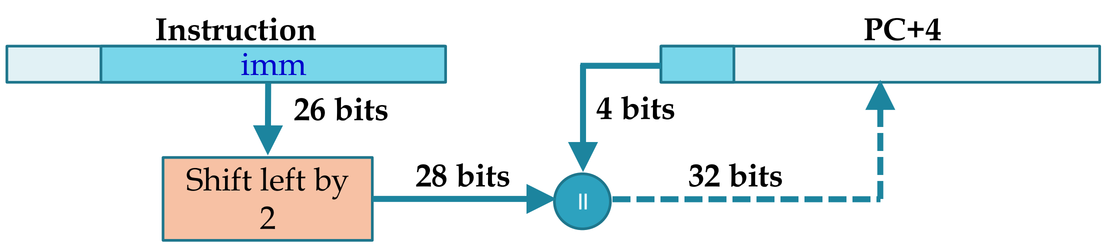
- In this example, immediate = 3
- End address = PC + 4 + (immediate $\times$ 4)

(Unconditional) Jump instruction

- Assembly.
  - `J target # assembly uses a` **symbolic** `target address`
- Encoding

op	imm
6 bits	26-bit signed ***instruction*** offset

- Semantics
  - `target = (PC+4)[31:28] x`2^{28}` || zero-extend(imm)x4;`
    `PC ← target;`



  - This addressing mode is called **Pseudodirect**. How far can you jump?
- Variations: `jal, jr`

IF statement (1/2)

C Statement to translate	Variables Mapping
`if (i == j)` `    f = g + h;`	f → \$s0 g → \$s1 h → \$s2 i → \$s3 j → \$s4

```
        beq $s3, $s4, L1
        j   Exit
L1:     add $s0, $s1, $s2
Exit:
```

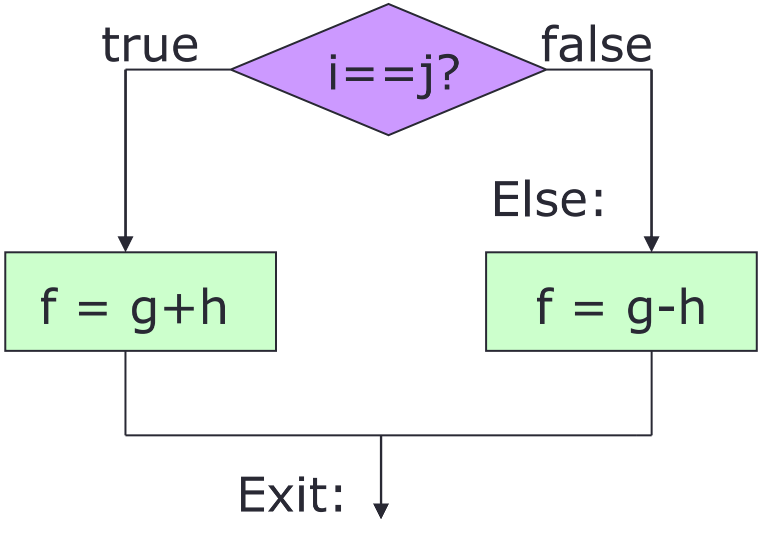
```
        bne $s3, $s4, Exit
        add $s0, $s1, $s2
Exit:
```

- Two equivalent translations:
  - The one on the right is more efficient
- Common technique: **Invert the condition** for shorter code

IF statement (2/2)

C Statement to translate	Variables Mapping

`if (i == j)`
`    f = g + h;`
`else`
`    f = g - h;`
	f ➔ \$s0
	g ➔ \$s1
	h ➔ \$s2
	i ➔ \$s3
	j ➔ \$s4




```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```

- Question: Rewrite with **`beq`**?

Loops (1/2)

Key concept:

Any form of loop can be written in assembly with the help of conditional branches and jumps.

- C while-loop:

```
while (j == k)
    i = i + 1;
```

- Rewritten with goto

```
Loop:  if (j != k)
           goto Exit;
       i = i+1;
       goto Loop;

Exit:
```

Loops (2/2)

C Statement to translate	Variables Mapping

`Loop: if (j != k)`
`goto Exit;`
`i = i+1;`
`goto Loop;`
`Exit:`
	i → \$s3
	j → \$s4
	k → \$s5

```
Loop: bne  $s4, $s5, Exit  # if (j!= k) Exit
      addi $s3, $s3, 1
      j    Loop            # repeat loop
Exit:
```

Inequalities

- There is no real branch-if-less-than instruction in MIPS
  - Use `slt` (set on less than) or `slti`

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
      $t0 = 1;
else
      $t0 = 0;
```

to build a "`blt $s1, $s2, L`" instruction:

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

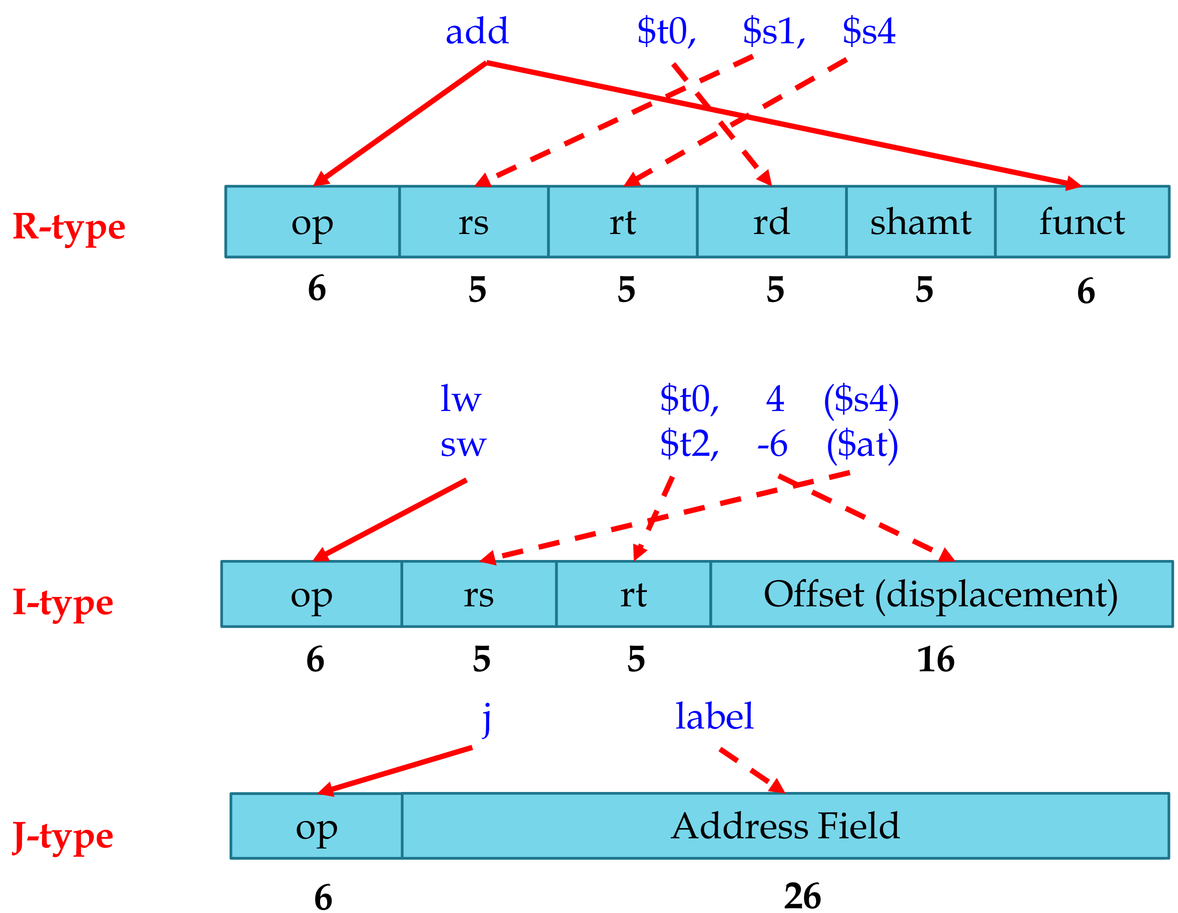
==

```
if ($s1 < $s2)
      goto L;
```

- This is another example of **pseudo-instruction**:
  - Assembler translates pseudo-instruction `blt` instruction into the (two) equivalent MIPS instructions

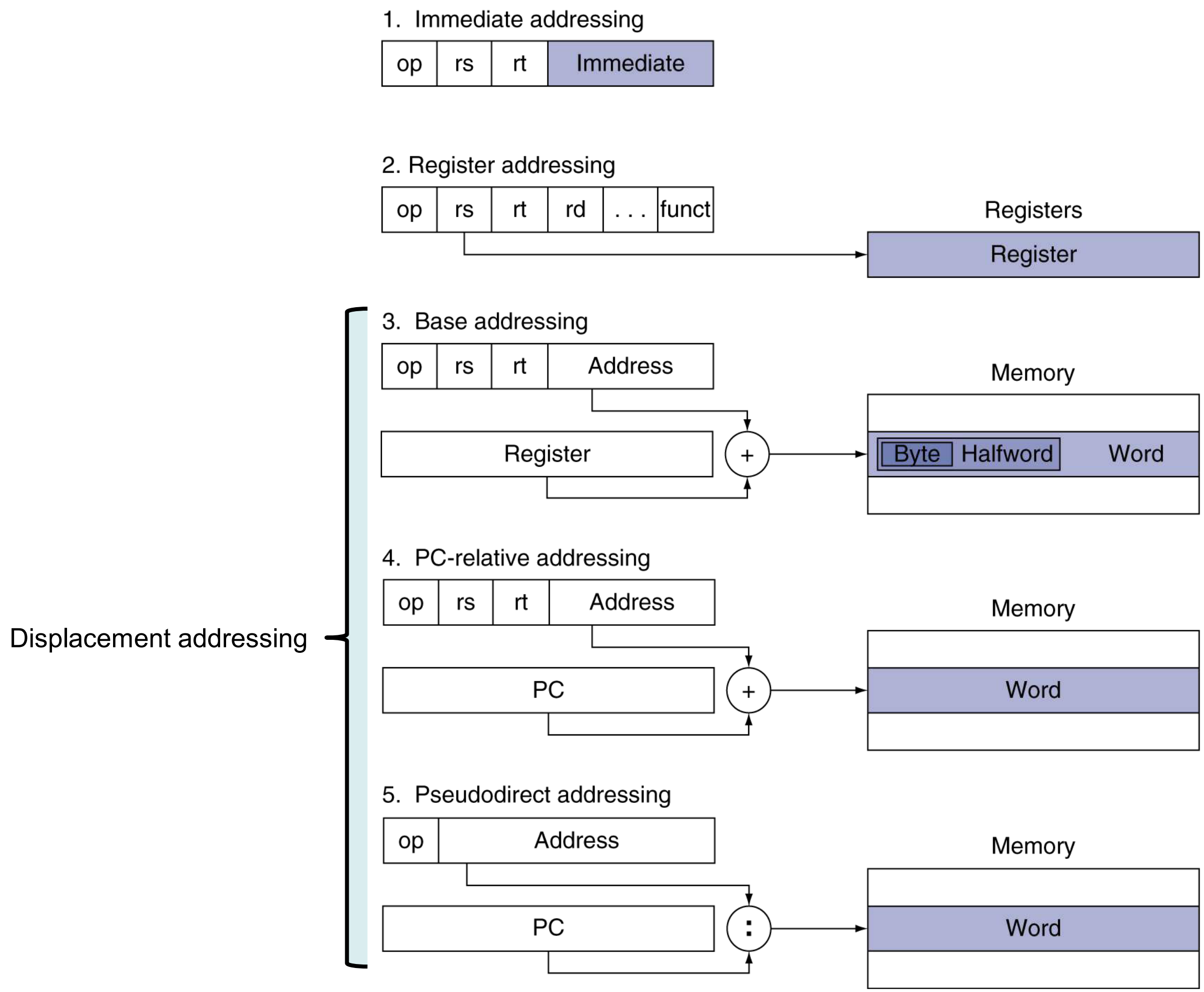
Summary (1)

- Three MIPS instruction types



Summary (2)

- MIPS addressing modes



Summary (3)

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