5 - MIPS Instruction

5.1 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 “target facts” from earlier:
  - Why is `immediate` limited to 16 bits
  - Why is `shift` amount only 5 bits

5.2 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

> 每条 MIPS 指令都有固定的32位长度。
>
> - 所有操作必须在这32位内编码。这意味着每条指令，无论其功能如何，都被设计为具有相同的长度。

5.3 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

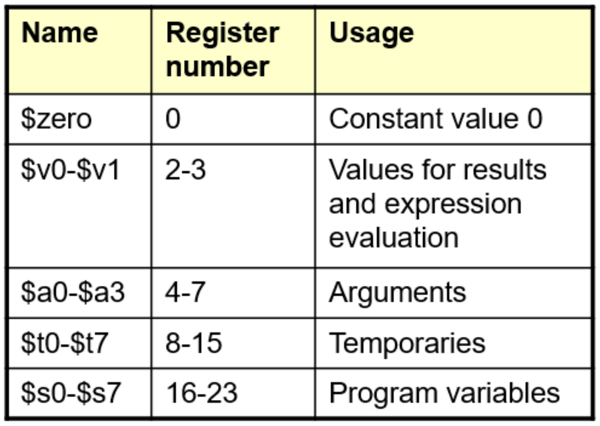
> J-Format (Jump Format: `op Immd`)

> - `j` instruction uses only one immediate value

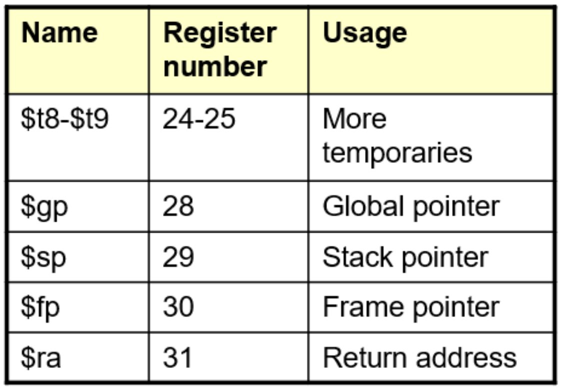
> 1. **按操作数分类的指令**:
>   - 同种类型的操作数的指令具有相同的编码格式。这意味着，具有相似操作数结构的指令会按照相同的模式进行编码。
> 2. **R-Format（寄存器格式）**:
>   - 这类指令使用两个源寄存器和一个目标寄存器。
>   - 示例：`add`，`sub`，`and`，`or`，`nor`，`slt` 等都是 R-格式的指令。
>   - 特殊情况：像 `srl` 和 `sll` 这样的位移指令也是 R-格式的指令，但它们有些特别，因为它们可能涉及到一个立即数值（位移的数量）。
> 3. **I-Format（立即数格式）**:
>   - 这类指令使用一个源寄存器，一个立即数值（通常是一个具体的数值，而不是另一个寄存器的值），以及一个目标寄存器。
>   - 示例：`addi`，`andi`，`ori`，`slti`（这些都是与常数值进行操作的算术和逻辑指令），以及 `lw`，`sw`（加载和存储指令），`beq`，`bne`（分支指令）等都是 I-格式的指令。
> 4. **J-Format（跳转格式）**:
>   - 这类指令主要与跳转操作有关。
>   - `j` 指令就是一个示例，它只使用一个立即数值来表示跳转的目标地址。

5.4 MIPS Registers

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



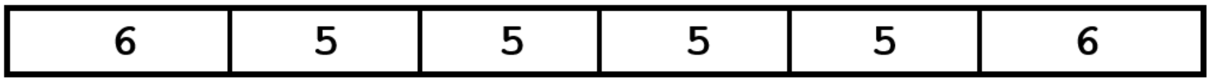
Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables



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

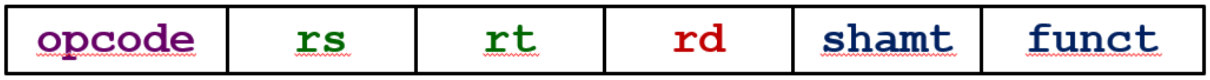
5.5 R-Format

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



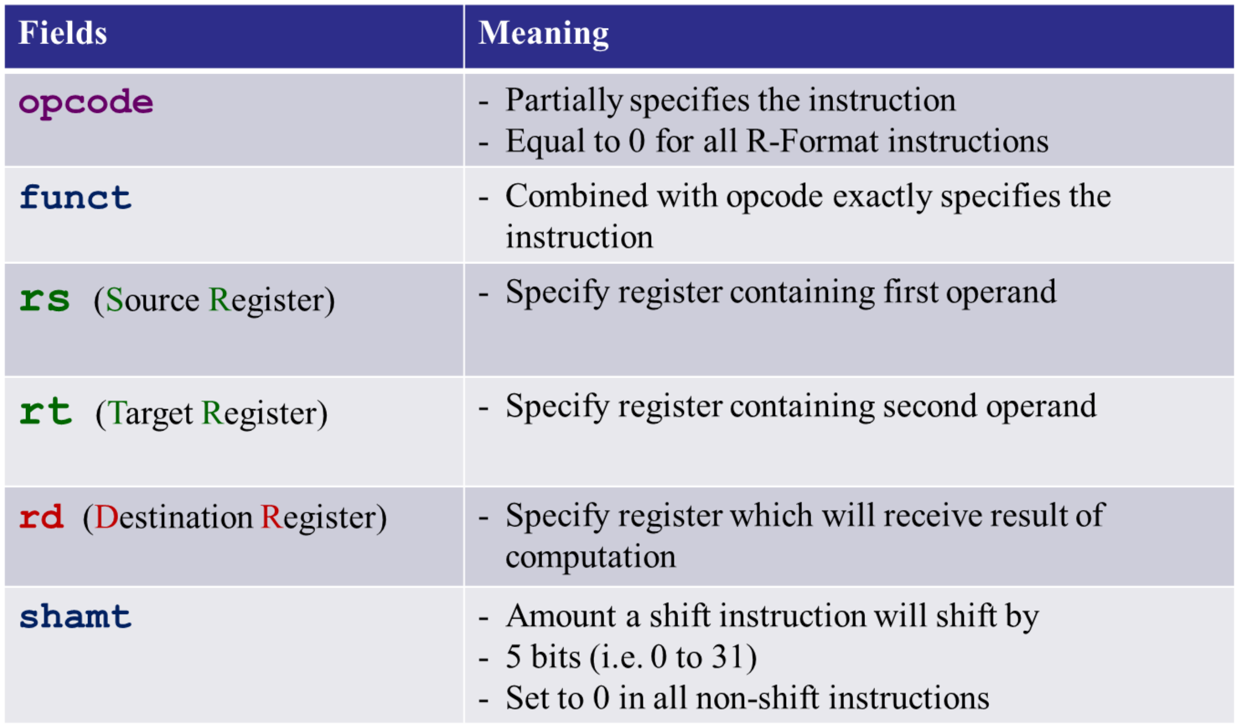
6	5	5	5	5	6

- Each field has a name:



opcode	rs	rt	rd	shamt	funct

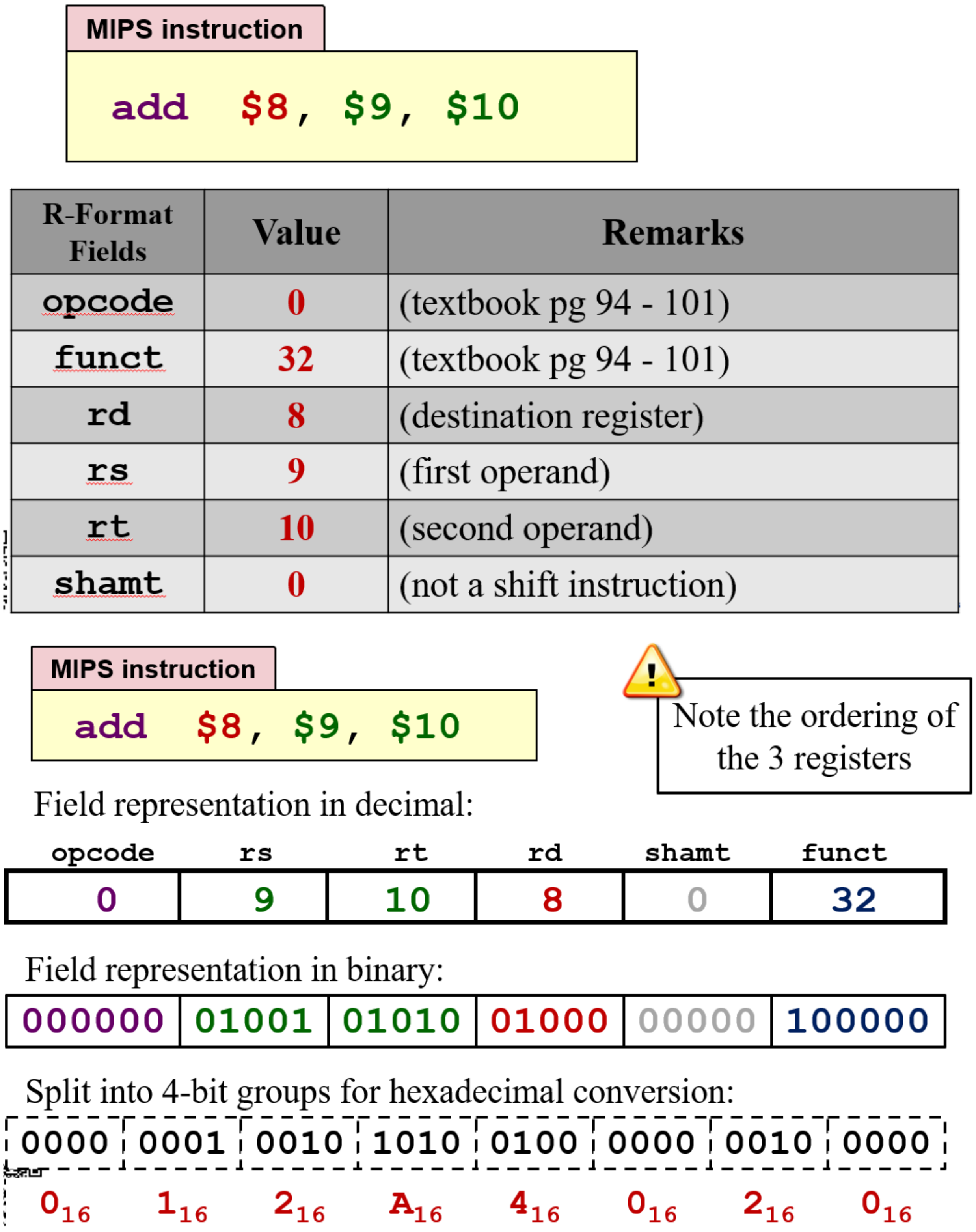
- Each field is an independent 5 or 6 bits unsigned integer
    - A 5-bit field can represent any number 0-31
    - A 6-bit field can represent any number 0-63



Fields	Meaning
opcode	- Partially specifies the instruction - 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 - 5 bits (i.e. 0 to 31) - Set to 0 in all non-shift instructions

> MIPS R-Format 指令用于表示那些涉及两个源寄存器和一个目的地寄存器的指令，如加法、减法和位逻辑操作等。这些字段将 32 位的指令分解为特定的部分，每个部分有其特定的功能和意义。
>
> 以下是对每个字段的解释：
>
> 1. **opcode**：操作码字段，用于部分指定指令的类型。对于所有的 R-Format 指令，opcode 都设置为 0。
> 2. **funct**：函数代码字段，与 opcode 一起合并，可以准确地指定指令的类型（例如，加法、减法等）。
> 3. **rs (Source Register)**：源寄存器字段，指定第一个操作数的寄存器。
> 4. **rt (Target Register)**：目标寄存器字段，指定第二个操作数的寄存器。
> 5. **rd (Destination Register)**：目的地寄存器字段，指定存放操作结果的寄存器。
> 6. **shamt**：位移量字段，用于指定位移指令（如 srl 和 sll）的位移量。此字段有 5 位，因此可以表示从 0 到 31 的值。对于非位移指令，此字段设置为 0。
>
> 总体来说，这些字段组合在一起，形成了一个 32 位的 R-Format 指令，允许处理器知道它需要执行什么操作以及操作的操作数是哪些。这些字段是指令编码的基础，使得处理器能够快速地解码和执行这些指令。

5.5.1 R-Format: Example



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)

MIPS instruction

`add $8, $9, $10`

Note the ordering of the 3 registers

Field representation in decimal:

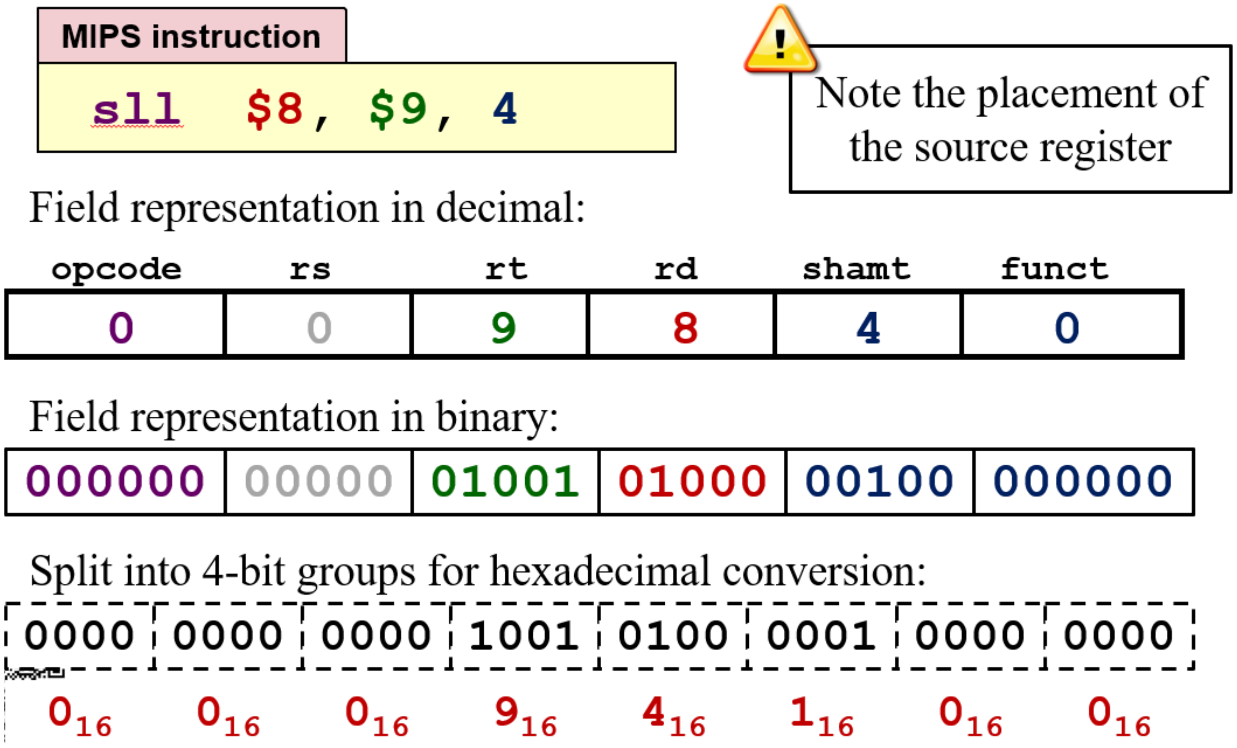
opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	32

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}



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}

5.6 I-Format

- 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
- Define fields with the following number of bits each:

6	5	5	16

opcode	rs	rt	immediate

- I-Format has no `funct` field, so the `opcode` uniquely specifies an instruction
- `rs` field specifies the source register operand
- `rt` field specifies the register to receive the result
- Immediate treated as a **single integer**

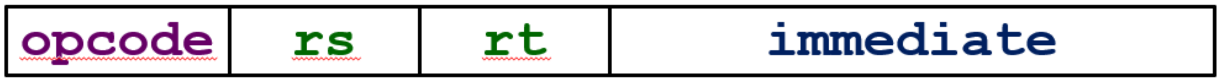
5.6.1 Instruction Address: Overview

- As instructions are stored in memory, they too have addresses
    - Conrtol flow instructions uses these addresses
- As instructions are 32-bit long, instruction addresses are word-aligned as well
- Program Counter(PC) is a special register that keeps address of instruction being executed in the processor

> 在介绍I-format指令时引入program counter (PC) 是因为一些I-format指令，特别是那些涉及分支和跳转的指令，使用立即数值作为基于当前PC值的偏移。这样的偏移方式允许指令跳转到程序内的特定位置。

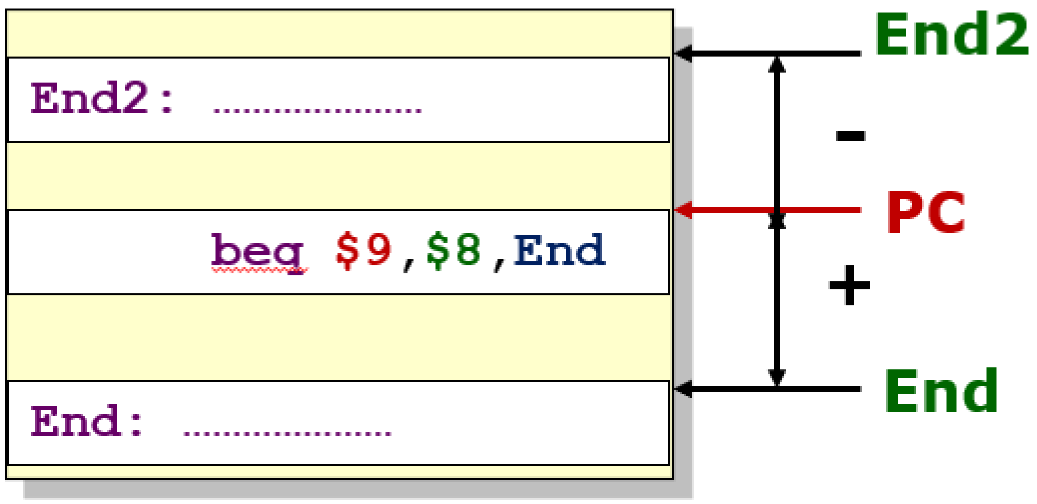
5.6.2 Branch: PC-Relative Addressing

- Use I-Format



opcode	rs	rt	immediate

- `opcode` specifies `beq` , `bne`
- `rs` and `rt` specify registers to compare
- `immediate` is only 16 bits, the memory address is 32 bits. `immediate` is not enough to specify the entire target address
- Solution: Specify target address relative to the PC
- Target address is generated as :
    - PC + 16-bit `immediate` field
    - The `immediate` field is a signed two's complement integer



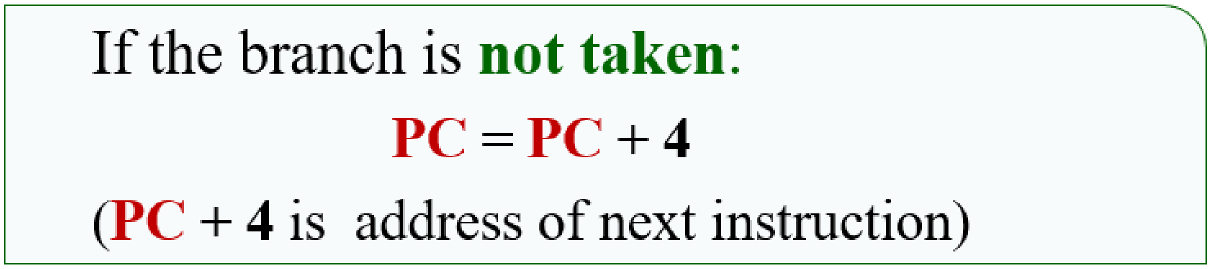
> **相对寻址**：许多I-format指令，如 `beq` (branch if equal) 和 `bne` (branch if not equal)，使用立即字段作为跳转偏移量。这些偏移量是相对于当前的PC值的。通过这种方法，指令可以确定跳转到程序中的哪个位置。
>
> 1. **`immediate` is only 16 bits**：在I-format指令中，

> 立即数字段（immediate field）的大小是固定的16位。这意味着它可以直接表示的值的范围是从0到65535（如果被视为无符号数）或从-32768到32767（如果被视为有符号数）。
>
> 1. **the memory address is 32 bits**：在MIPS架构中，完整的内存地址是32位的。这意味着系统可以直接访问的内存地址范围是从0到4GB。
> 2. **`immediate` is not enough to specify the entire target address**：由于立即数字段只有16位，它不能直接表示32位的完整内存地址。换句话说，一个16位的值不足以覆盖整个4GB的内存空间。
>
> 这有什么意义呢?
>
> 当使用I-format指令（如加载和存储指令）访问内存时，通常使用一个基址寄存器和一个16位的偏移量（即立即数）来确定目标地址。例如，在 `lw $t0, 4($t1)` 这样的指令中， `$t1` 包含一个基地址，而4是一个16位的偏移量。最终的内存访问地址是 `$t1` 中的值加上4。
>
> 此外，对于跳转和分支指令，16位的立即数经常被解释为相对于当前指令地址（即程序计数器，PC）的偏移量。这允许指令跳转到相对近的位置，而不需要指定完整的32位地址。
>
> 综上所述，尽管16位的立即数不能直接指定完整的32位地址，但通过与其他值（如基址寄存器或当前PC值）结合，可以有效地指定或计算目标地址。

Branches

- We usually use branches by `if-else` , `while` , `for`
- Loops are generally small:
  - Typically up to 50 instructions
- Unconditional jumps are done using jump instruction `j` , not the branches
- Conclusion: A branch often changes PC by a small amount
- 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 (Since each MIPS instruction is 32-bit)
  - Interpret the `immediate` as a 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

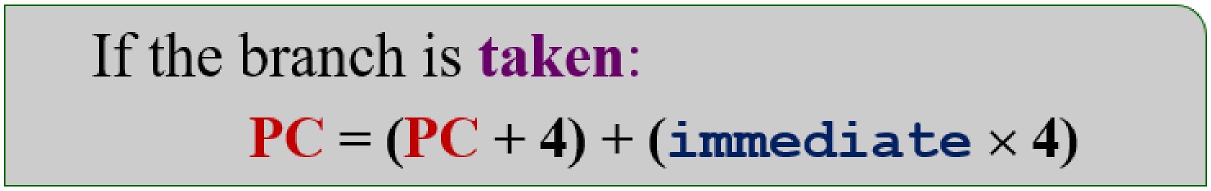
Branch Calculation



If the branch is **not taken:**

$$PC = PC + 4$$

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



If the branch is **taken:**

$$PC = (PC + 4) + (\texttt{immediate} \times 4)$$

- `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

> 即如果没有进入branch，则跳转到下一个指令，由于每个MIPS指令是32 bits，所以在PC上加4字节
>
> 如果进入了branch，则跳转到一个新的地址来执行指令，这个新的地址是基于当前PC值，偏移量（即下一条指令的位置PC+4），以及由分支指令中的立即字段(`immediate`)给出的额外偏移量计算出来的
>
> 立即数字段(`immediate`)被解释为word的数量，而每个word有4字节，所以被乘以4

5.7 J-Format

- For branches, PC-relative addressing was used, because we do not need to branch too far
- For general jump `j` , we may jump to anywhere in memory
- The ideal case is to specify a 32-bit memory address to jump to.

> J-format 是 MIPS 指令集中用于跳转（jump）指令的格式。虽然理论上我们确实希望直接指明一个32位的内存地址以进行跳转，但在实际的指令编码中，由于指令的长度也是32位，不可能在单一的指令中同时包含操作码、其他字段和一个完整的32位地址。
>
> 考虑到这个限制，J-format 被设计为只包含一个26位的跳转目标地址字段。这意味着我们不能直接指定整个32位的跳转目标地址。但是，由于所有指令都是字对齐的，这意味着指令的地址的最后两位总是00（因为指令长度为4字节，或32位）。这为我们提供了一个小小的优势：我们只需要指定高26位的地址，而低2位则可以简单地置为0。
>
> 此外，J-format 的跳转是基于绝对地址的，而不是相对于当前 PC 的偏移。这意味着 J-format 跳转指令可以跳转到内存中的任何位置，只要这个位置在那26位地址范围内。
>
> 总的来说，虽然我们希望能够直接指明一个32位的地址进行跳转，但由于指令长度的限制和字对齐的特性，J-format 只提供了一个26位的地址字段来进行跳转。

- Define fields of the following number of bits each

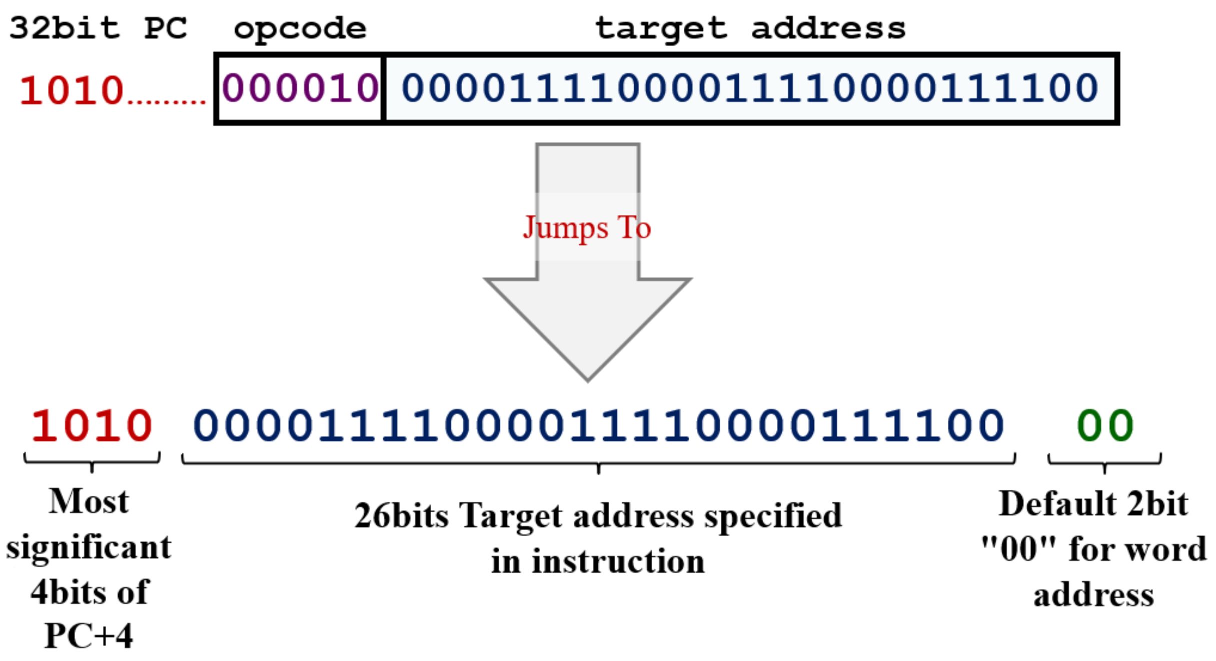
6 bits	26 bits

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
- We can only specify 26-bits of 32 bits 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
- Where do we get the other 4 bits?
  - MIPS choose to take the 4 most significant bits from PC+4
  - This means that we cannot jump to anywhere in memory, but it should be sufficient most of the time
  - The maximum jump range is **256MB**
- Special instruction if the program straddles 256MB boundary
  - `jr` : use `load` instruction to load full 32-bit memory address to register, then use `jr` instruction to jump
  - Target address is specified through a register

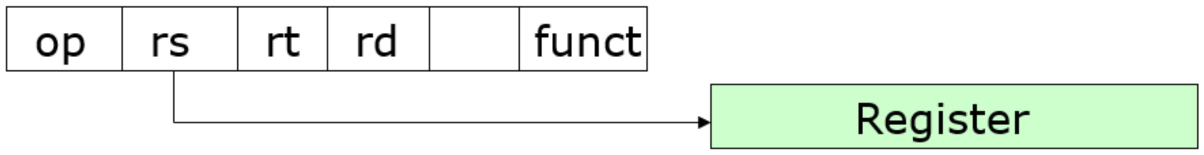
> 在MIPS中，使用J-format的 `j` 指令时，我们确实只有26位来指定跳转地址。这26位的地址是如何转化为完整的32位地址的呢？它是通过将这26位的跳转字段左移2位（因为每个指令是4字节或32位）来实现的，然后再与当前指令地址的高4位进行组合，从而生成完整的32位跳转地址。
>
> 因此，这26位地址可以指定 2^{26} 不同的跳转目标，也就是 67,108,864 个可能的目标。由于每个指令都是4字节，所以跳转范围是 $2^{26} * 4$ 字节，也就是 268,435,456 字节或 256 MB。

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

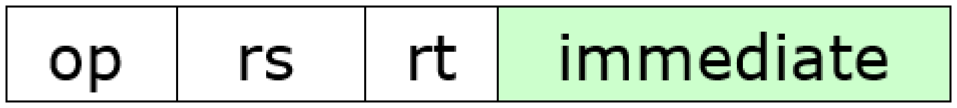


5.8 Addressing Modes

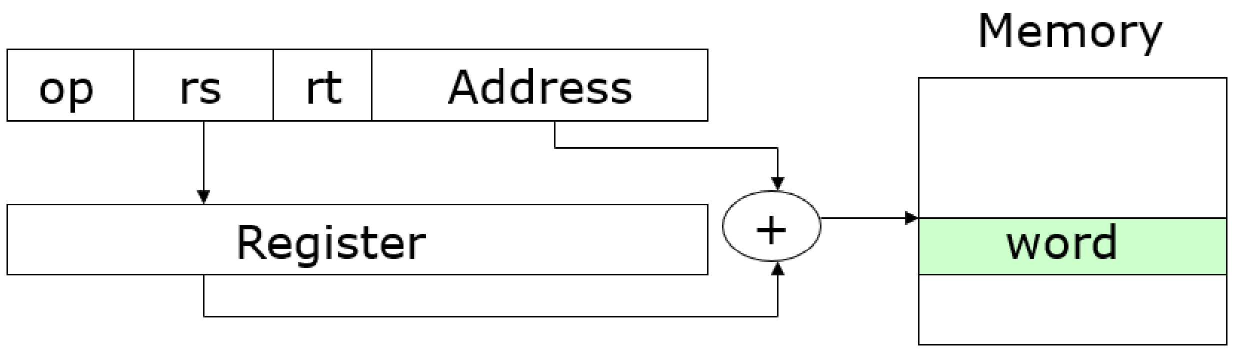
- **Register addressing**: operand is a register



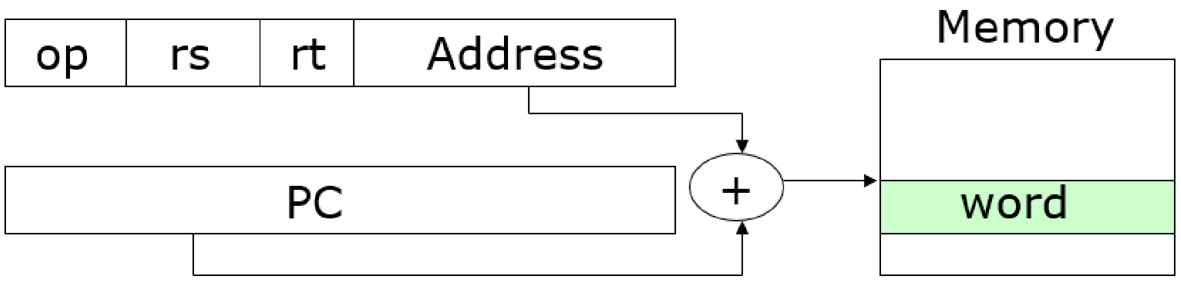
- Immediate addressing: operand is a constant within the instruction itself (`addi` , `andi` , `ori` , `slti`)



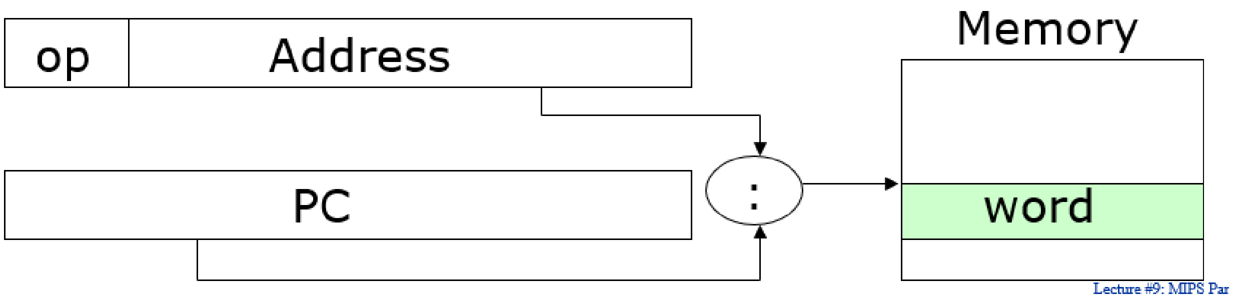
- Base addressing (displacement addressing): operand is at the memory location whose address is sum of a register and a constant in the instruction (`lw` , `sw`)



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

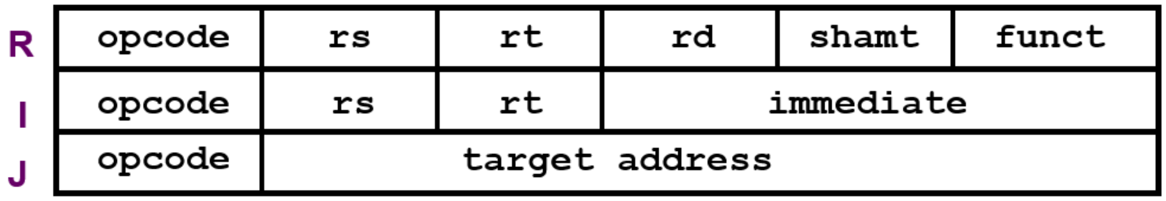


- Pseudo-direct addressing: 26-bit of instruction concatenated with upper 4-bits of PC (`j`)



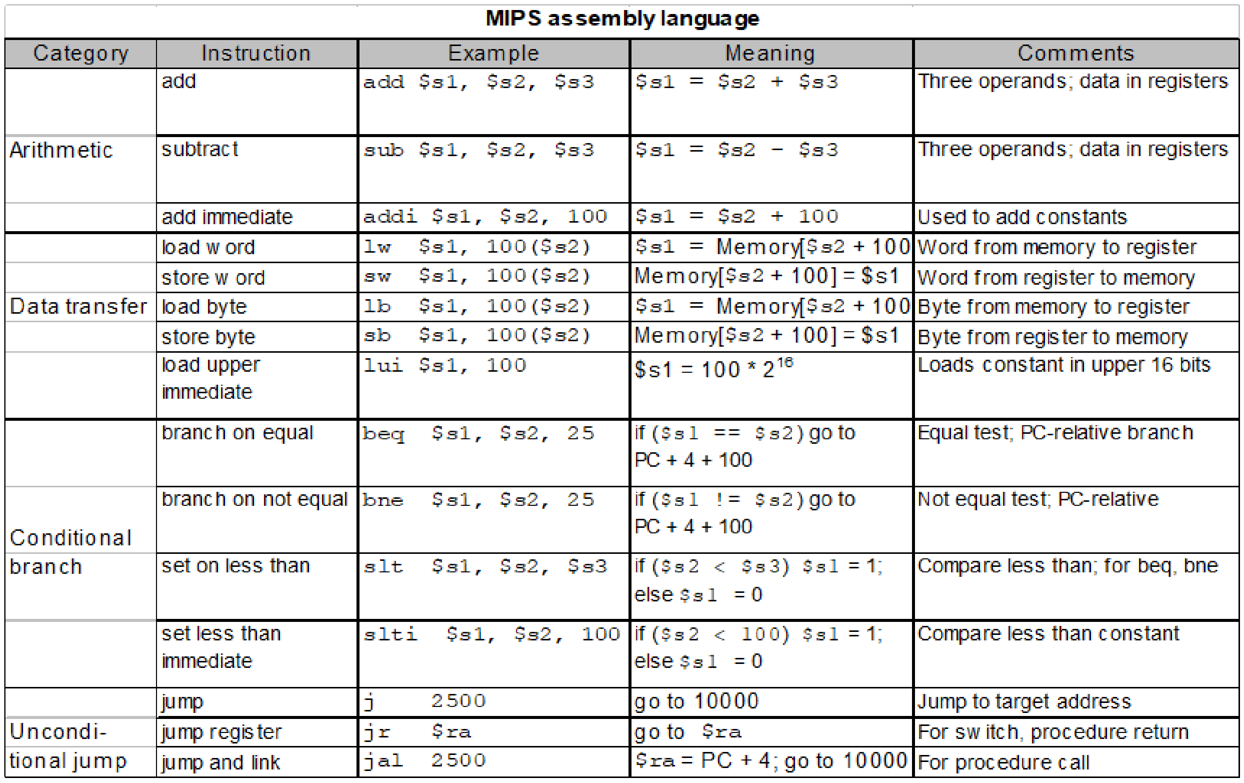
5.9 Summary

- MIPS instruction:
  - 32 bits representing a single instruction, for each format (R, I, J) the fields included are different



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



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	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
	load w ord	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100	Word from memory to register
	store w ord	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
Data transfer	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
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
branch	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
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For sw itch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call