## 5. Addressing Modes

## MIPS Addressing Modes

- Addressing takes care of where to find
  - data
  - instruction

- We have seen, so far three addressing modes of MIPS (to find data):
  1. Immediate addressing: provides fast access of small **constants**

     e.g. `addi $t0, $t0, 1023`
  2. Register addressing: the operand is available in a register

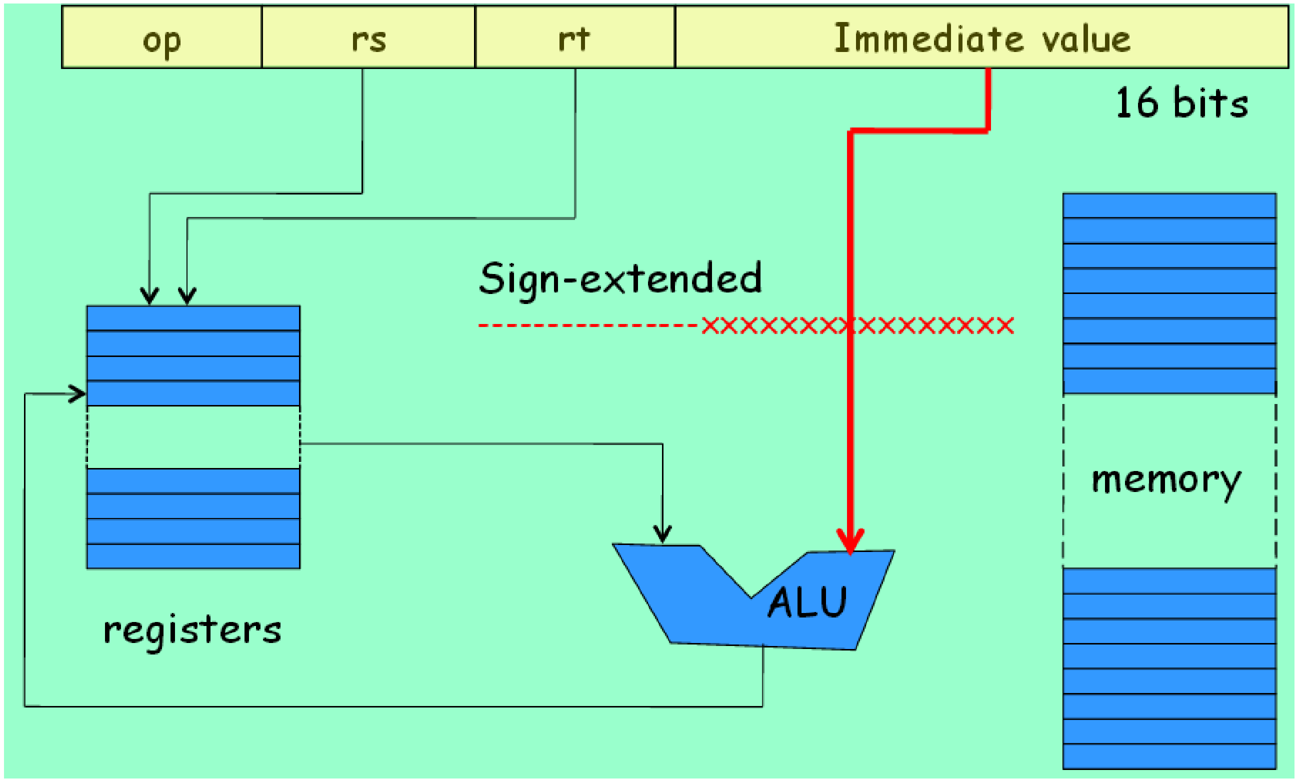
     e.g. `add $t0, $t0, $t1`
  3. Base addressing: the operand is the sum of a (**base**) register and a **displacement**

     e.g. `lw $t0, 1024($t1)`

- MIPS architecture provides two more ways of addressing (to find instruction)

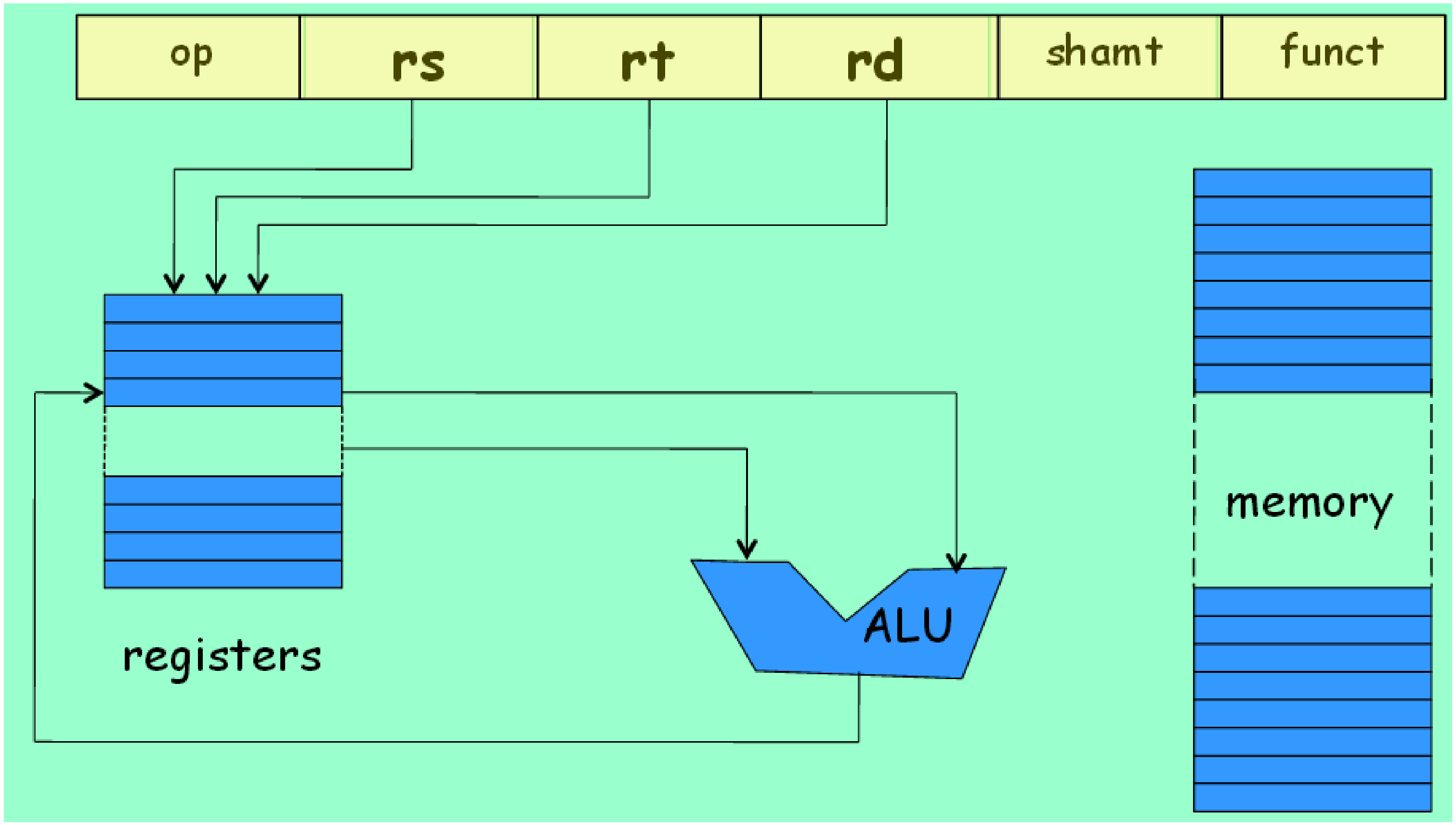
## Immediate Addressing

- One operand is embedded inside the encoded instruction
- 16-bit immediate is a two's complement number
  - $-2^{15} <= \text{value} <= 2^{15}-1$
- Example: `addi` or similar



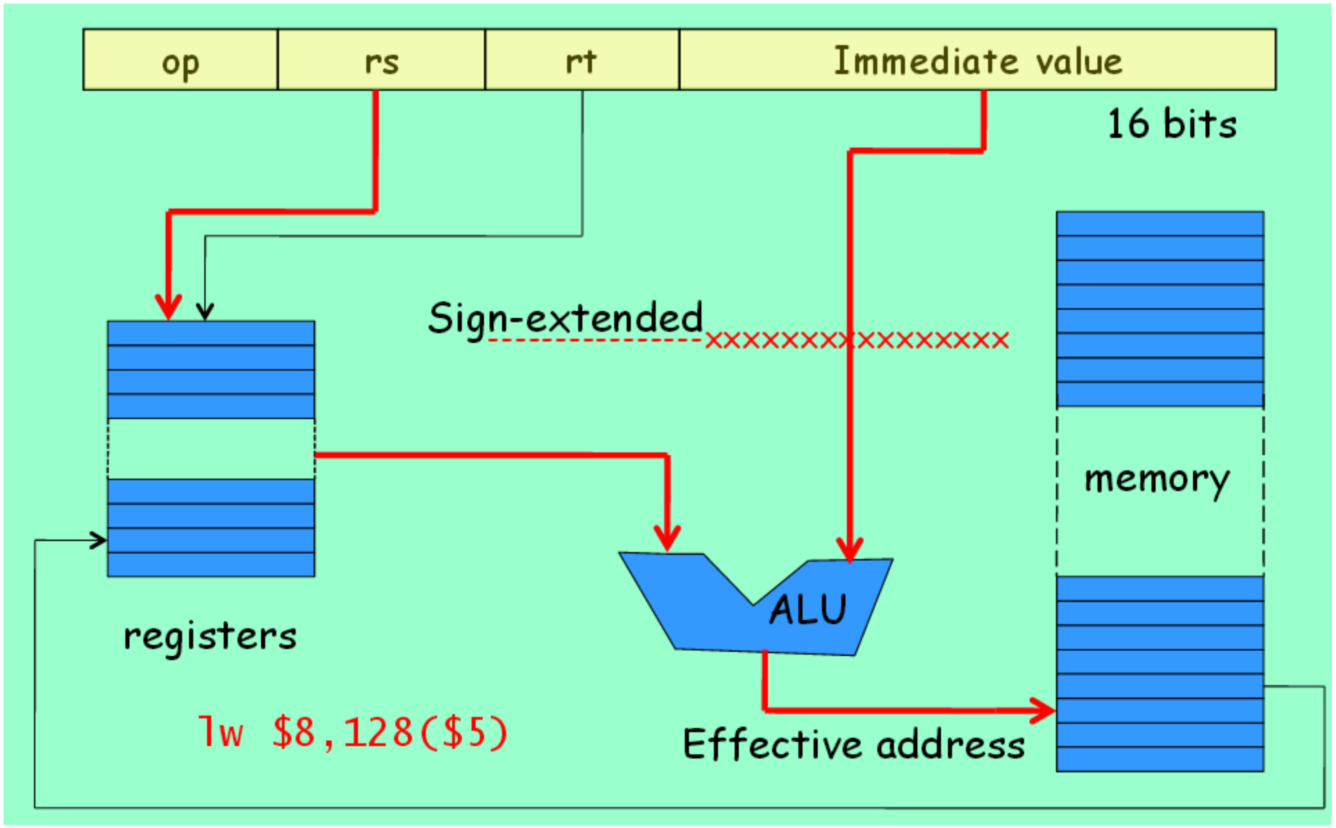
## Register Addressing

- The operands are in registers
- Takes n bits to address $2^n$ registers
- Example: `add`, `sub,` `and,` `sll` or similar



## Base Addressing

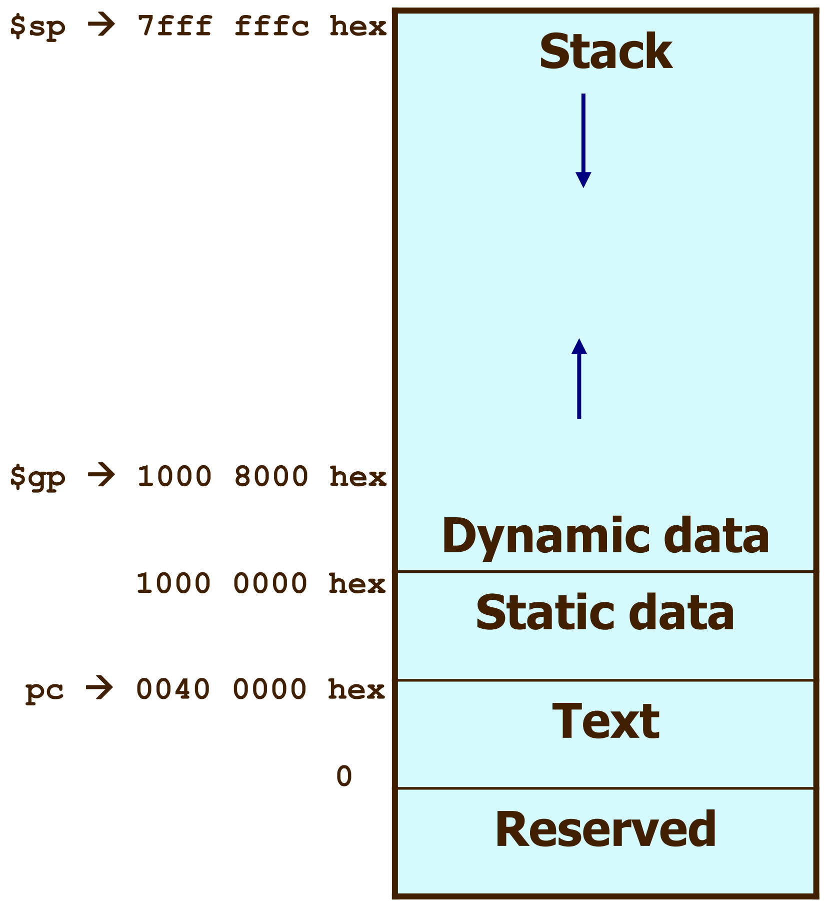
- One operand is in main memory
- Its address is the sum of the immediate and the value in register **`$rs`**
- 16-bit immediate is a two's complement number
- Example: **`lw $s1, 16($s0)`**



## Where is the next instruction?

- Address of current instruction is in PC
- Sequential execution
  - Address of the next instruction is PC+4
- Conditional branch? Un-conditional branch?

- Re-visit memory space
  - Text segment starts 0x00400000
  - Each instruction occupies 4 bytes (1 word)
  - Last 2 digits of instruction address is always 00 (we can make it implicit and use 'word address')



## Addressing in Conditional Branches

We know that

- Conditional branch instructions (e.g., `beq`, `bne`) use I-format
- I-format can only specify 16-bit addresses

How to branch?

- PC-relative addressing
  - A branch offset is added to (PC+4) to obtain address to branch to
    - Branch offset is described in number of **words**.
  - Branching within $2^{15}$ words before or after the current instruction is possible
  - This is good enough since conditional branches tend to branch to a nearby instruction

Notes:

While an instruction is being executed, the PC always points to the current instruction, i.e., address of current instruction

# Example: Branch Offset in Machine Language

| Address | Instruction |
|---|---|
| 40000008 | `addi $s0, $s0, 1` |
| 4000000C | `beq  $zero, $s0, label` |
| 40000010 | `addi $s0, $s0, 1` |
| 40000014 | `addi $s0, $s0, 1` |
| 40000018 | `label:  addi $s0, $s0, 1` |
| 4000001C | `addi $s0, $s0, 1` |
| 40000020 | `etc…` |

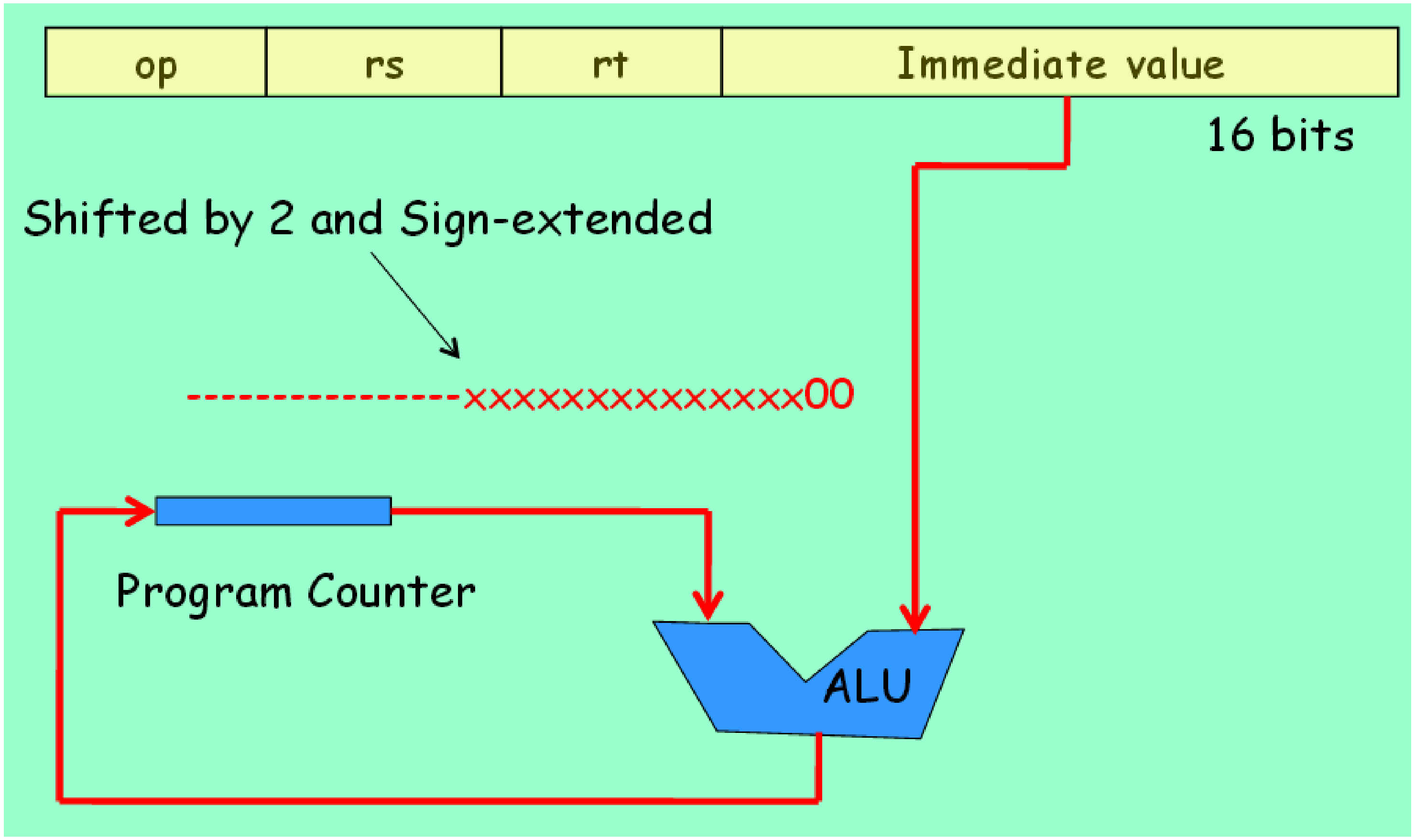
- Machine code to `beq` is `0x1005002`, which means 2 instructions from the next instruction

| | |
|---|---|
| PC = | `0x4000000C` |
| PC+4 = | `0x40000010` |
| Add 4*2 = | `0x00000008` |
| Target = | `0x40000018` |

| op | rs | rt | const or address |
|---|---|---|---|
| 00010 | 00000 | 00101 | 0000000000000010 |

## PC-relative Addressing

- The value in the immediate field is interpreted as an offset of the next instruction (PC+4 of current instruction)


op
rs
rt
Immediate value
16 bits
Shifted by 2 and Sign-extended
----------------xxxxxxxxxxxxxx00
Program Counter
ALU

# J-type Instruction Format

**J-type** or **J-format**

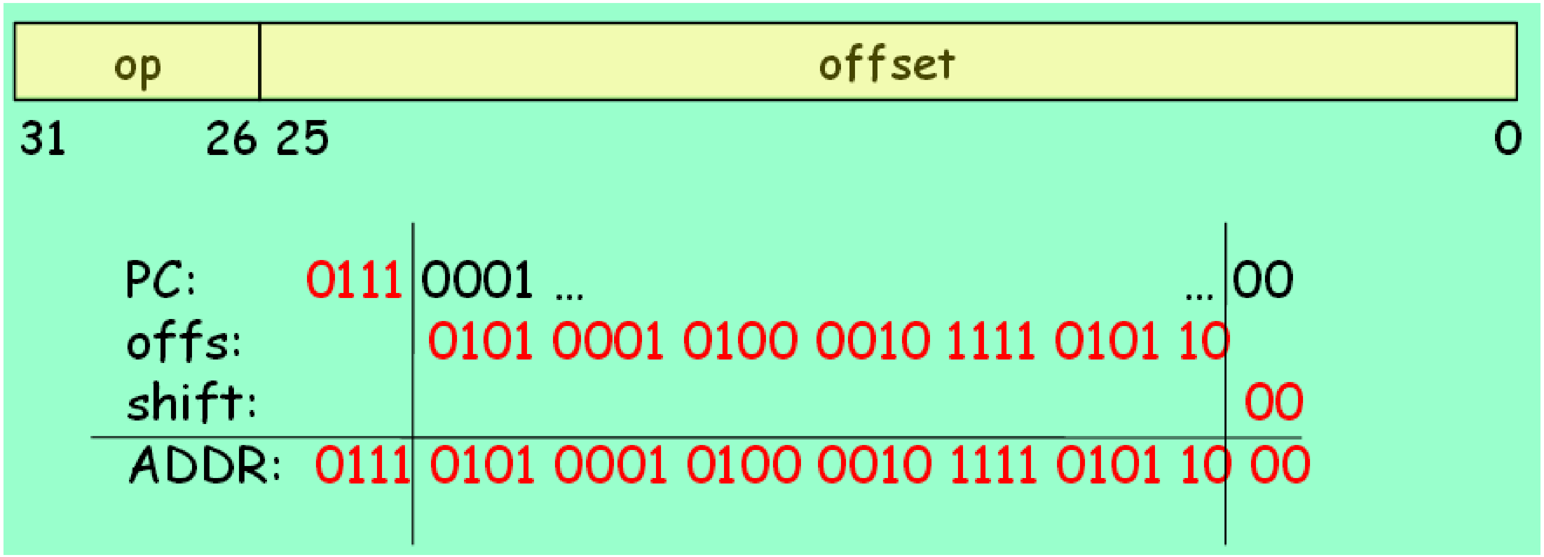
| op | address |
|---|---|
| 6 bits | 26 bits |

- Used by instructions such as `j` ('jump') and `jal` ('jump and link')
  - e.g. `j   L1        # go to instruction labeled L1`

## Addressing in Un-conditional Jumps

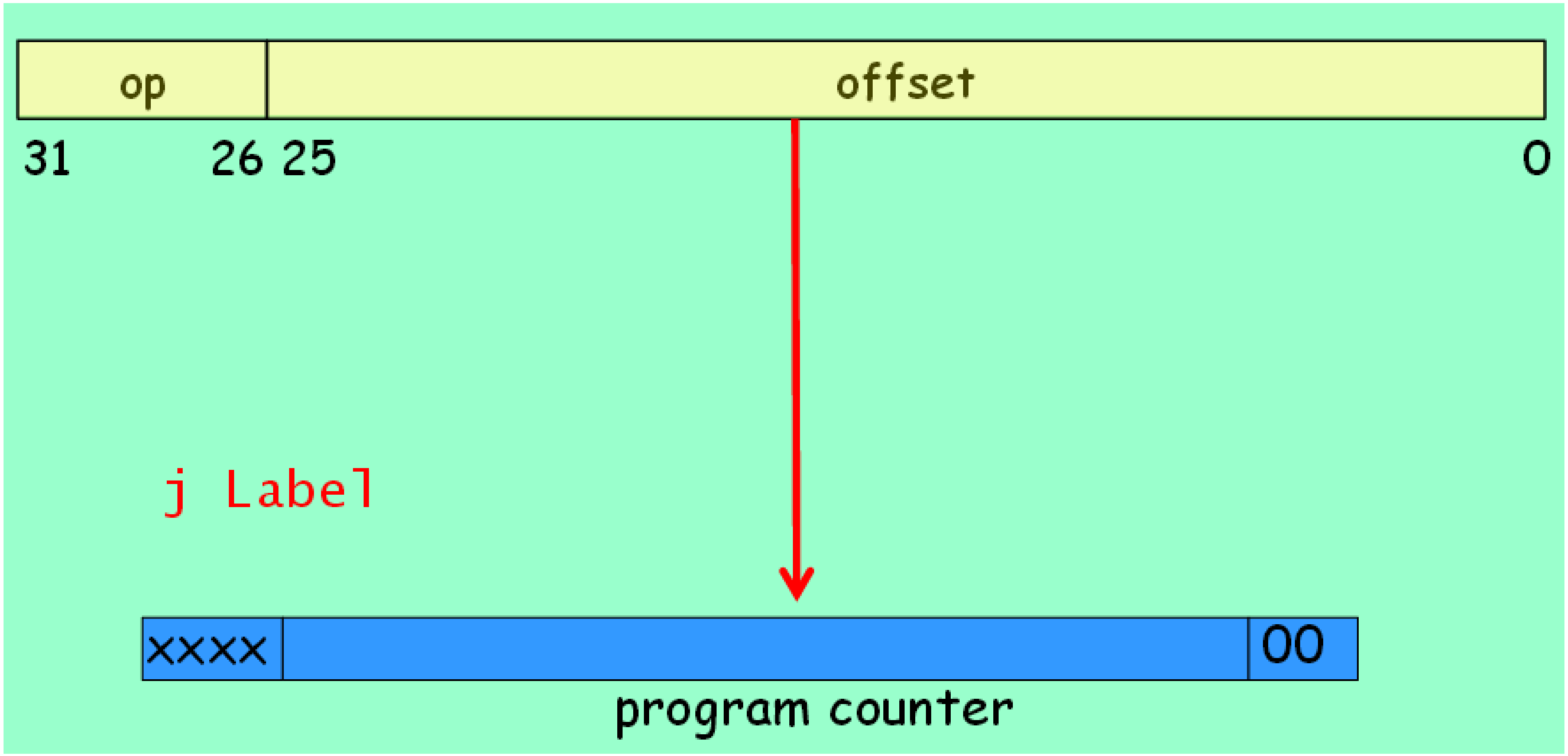
- **Direct Addressing:** the address is 'the immediate'. 32-bit address cannot be embedded in a 32-bit instruction
- **Psuedo-direct Addressing:** 26 bits of the address is embedded as the immediate

**Example:** `j Label`



op
offset
31
26 25
0
PC: 0111 0001 ... ... 00
offs: 0101 0001 0100 0010 1111 0101 10
shift: 00
ADDR: 0111 0101 0001 0100 0010 1111 0101 10 00

## Pseudo-direct Addressing (2)



## Branching Far Away

- From 16-bit word address to 26-bit word address:
  - replace this

```
        beq $s0, $s1, L1   # L1 = 16-bit address
```

  - with this

```
        bne $s0, $s1, L2   # L2 = 16-bit address
        j   L1             # L1 = 26-bit address
    L2:
```

**Attention:**

The tradeoff is longer program execution time

- i.e. need to execute two instructions rather than just one

## Stretching the Maximum Possible Distance

Because,

- ❑ All MIPS instructions are 4 bytes long

So,

- ❑ A branch target or offset can refer to number of words instead of number of bytes

⇒ essentially stretch the maximum possible branching distance by 4x

**Questions**:

- ❑ What is the range a 'j' and 'jal' can jump to?
  - ❍ Within 256MB
- ❑ What if we want to jump beyond 256MB?

## Stretching the Maximum Possible Distance

| | |
|---|---|
| 0x0 0000000 | |
| … | |
| 0x0 FFFFFFF | |
| 0x1 0000000 | |
| … | |
| 0x1 FFFFFFF | |
| 0x2 0000000 | J L1 |
| … | |
| 0x2 FFFFFFF | L1: … |
| … | |
| 0x7 0000000 | |
| … | |
| 0x7 FFFFFFF | |

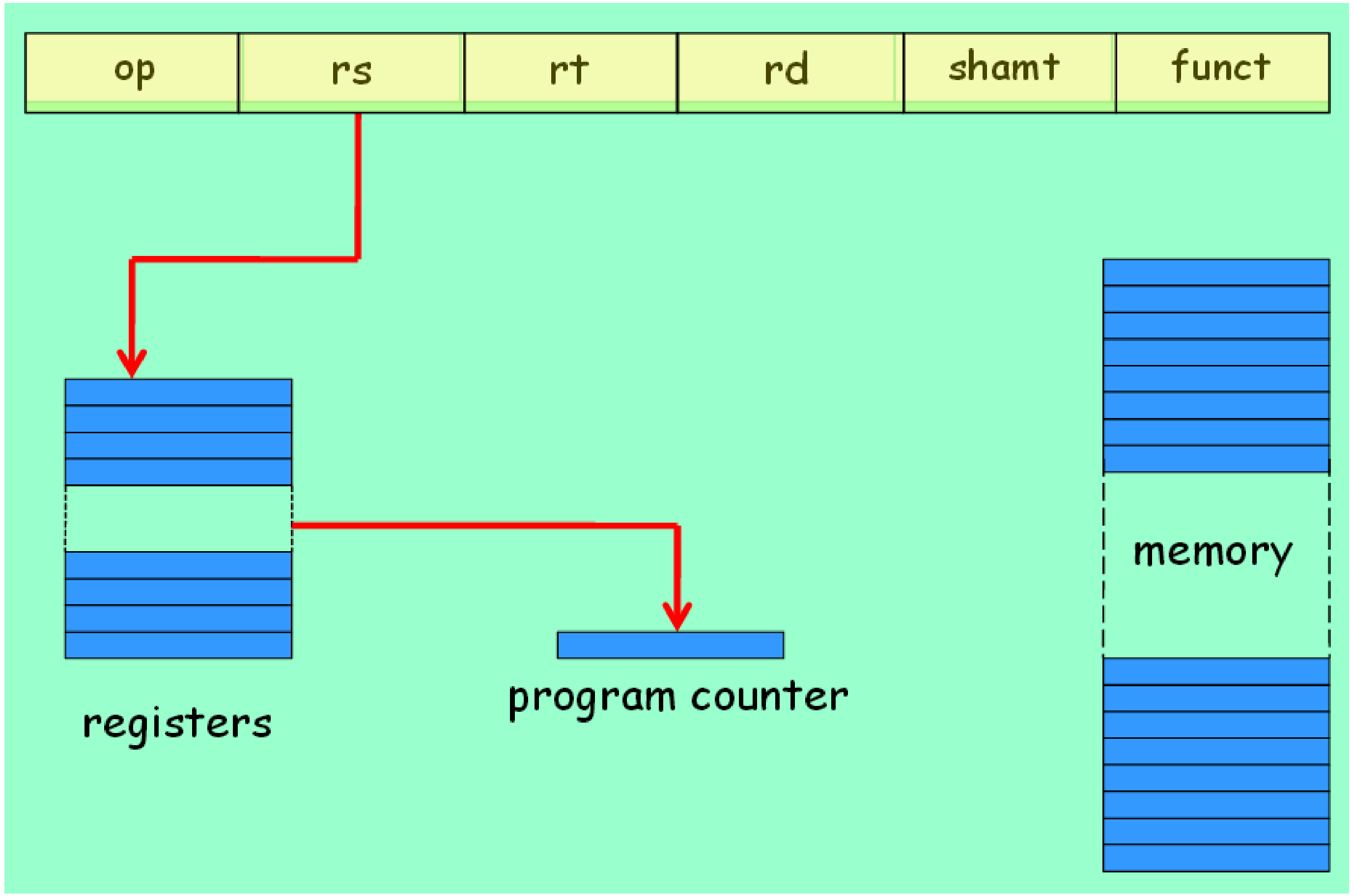
## Stretching the Maximum Possible Distance

| | |
|---|---|
| 0x0 0000000 | |
| … | |
| 0x0 FFFFFFF | |
| 0x1 0000000 | J L1 |
| … | |
| 0x1 FFFFFFF | |
| 0x2 0000000 | L1: … |
| … | |
| 0x2 FFFFFFF | |
| … | |
| 0x7 0000000 | |
| … | |
| 0x7 FFFFFFF | |

What if the Jump target is more than 256 MB away?

## Jump Register

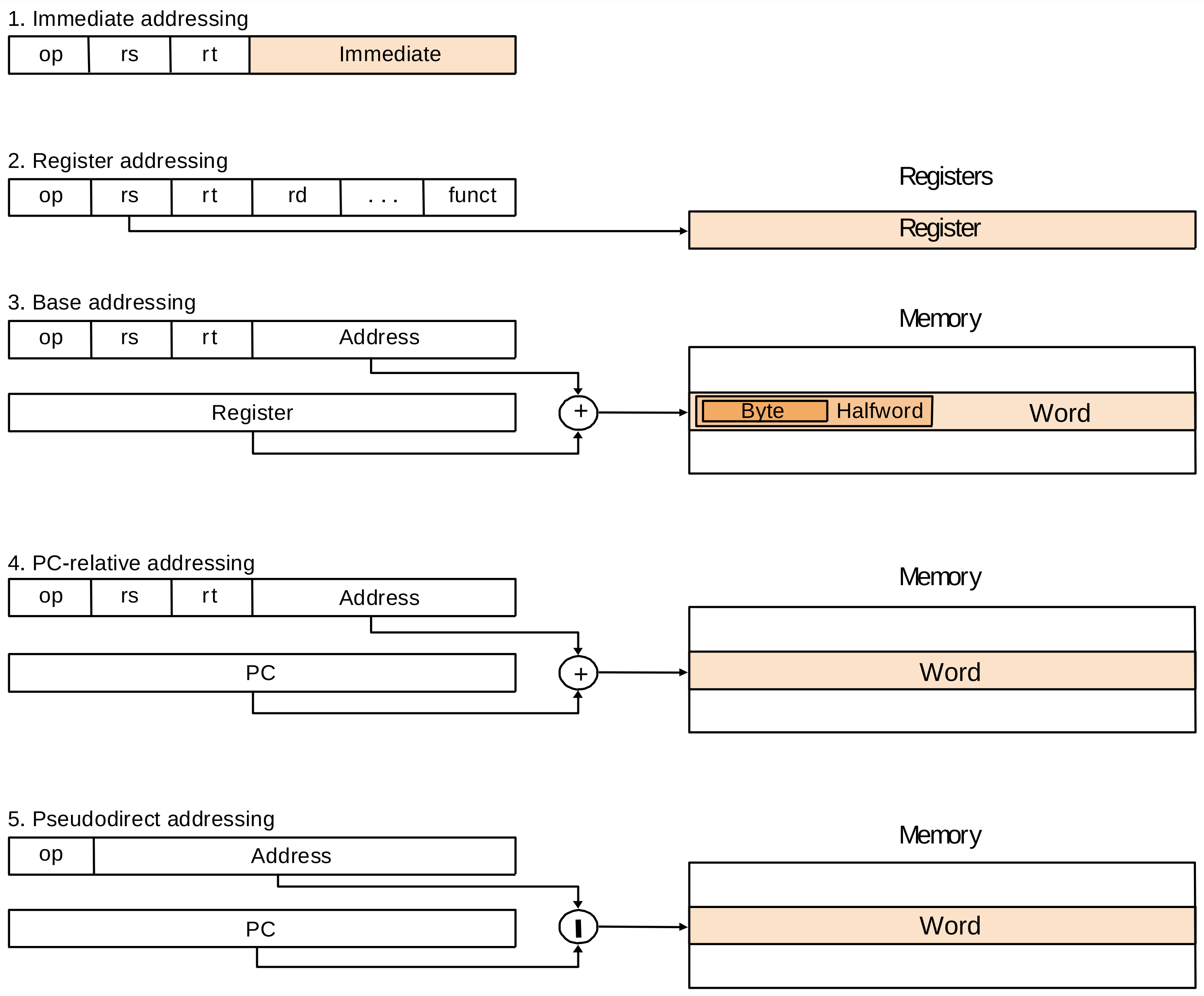
- Set the register content as the target address
- Then simply `jr`



## Summary of MIPS Addressing Modes

1. **Immediate addressing**
   - The operand is a constant within the instruction itself
2. **Register addressing**
   - The operand is a register
3. **Base addressing** or **displacement addressing**
   - The operand is at the memory location with address
     = (register) + constant
4. **PC-relative addressing**
   - The address is = (PC) + 4 + constant
5. **Pseudodirect addressing**
   - The jump address is a constant in the instruction concatenated with the upper 4 bits of the PC

## Summary of MIPS Addressing Modes (cont'd)



## Summary of MIPS Architecture Revealed So Far

- **MIPS operands**:
  - 32 registers (32 bits each)
  - $2^{30}$ memory word locations (32 bits each)

- **MIPS instructions**:
  - Arithmetic: `add`, `sub`, `addi`
  - Data transfer: `lw`, `sw`, `lb`, `sb`, `lui`
    - `lb` and `sb` are similar to `lw` and `sw`, but for transferring bytes instead of words
  - Conditional branch: `beq`, `bne`, `slt`, `slti`
  - Unconditional jump: `j`, `jr`, `jal`

- **MIPS instruction formats**:
  - R-format, I-format, J-format

## 6. Other Issues (optional)

## Alternatives to the MIPS Approach

- MIPS is an example of **RISC**
  - **Reduced Instruction Set Computer**
  - Each instruction does one simple thing
  - Most existing processors are RISC since it is more promising

- Another approach is **CISC**
  - **Complex Instruction Set Computer**
  - One instruction may do multiple things, e.g. Intel's instruction set

| | **RISC** | **CISC** |
|---|---|---|
| Number of instructions in a program | (–) more | (+) less |
| Time to execute the program | (+) usually less | (–) usually more |
| Hardware design | (+) simple | (–) complex |

(+) means advantage, (–) means disadvantage

## Comparing RISC and CISC in Details

| RISC | CISC |
|---|---|
| Through quantitative measurements, choose only the most useful instructions and addressing modes. | Choose instructions and addressing modes that make the translation of high-level languages to assembly language simpler. |
| With few instructions and addressing modes, we can directly execute them in hardware. | Since we can have many instructions and addressing modes, we need a **microcode** (or **microprogrammed control**) to execute them in hardware. |
| A lot of chip space can be left for a large number of registers and cache memory. | We can have only few registers and small cache memory. |
| Compilers are more difficult to write. | Compilers are easier to write. |
| Assembly language programs are more difficult to write. | Assembly language programs are easier to write. |

## Pseudoinstructions

**Pseudoinstructions**

- Assembly language instructions that do not have corresponding machine instructions (i.e., they need not be implemented directly in hardware)

Why Pseudoinstructions?

- Their appearance in assembly language simplifies programming and translation, giving MIPS a richer set of assembly language instructions than those implemented by the hardware.

Cost of supporting Pesudoinstructions

- The only cost is reserving one register, **`$at`**, for use by the assembler

## Examples of Pseudoinstructions

- `move`:
  - `move $t0, $t1            # $t0 gets value of $t1`
  - The assembler converts this pseudoinstruction into the machine language equivalent of the following instruction:

    ```
    add  $t0, $zero, $t1      # $t0 gets 0 + value of $t1
    ```

- Others:
  - `blt` (‘branch on less than’)
  - `ble` (‘branch on less than or equal’)
  - `bgt` (‘branch on greater than’)
  - `bge` (‘branch on greater than or equal’)

## Key Concepts to Remember

- The stored-program concept underlies today's digital computers
- An instruction specifies an operation and its corresponding operand(s)
- All MIPS instructions are 32 bits in length
    - To simplify the instruction set architecture
    - But, multiple instruction formats are supported
- Registers are fast temporary storage inside the processor
- Four design principles for ISA
    - Simplicity favors regularity
    - Smaller is faster
    - Make common case fast
    - Good design demands good compromises

## Key Concepts to Remember (cont'd)

- Program counter is a special register
  - Pointing to the current instruction to be fetched and executed
- Branch/jump instructions often require branch address calculation
- MIPS supports different addressing modes
  1. Register
  2. Displacement
  3. Immediate
  4. PC-relative
  5. Pseudodirect
- Pseudoinstructions extend the MIPS instruction set
  - To facilitate program development
- RISC and CISC are two very different design philosophies