Jump instructions

Instruction			Semantics	
`j`	`offset`	#	`PC <-- PC`$_{31\text{-}28}$ `:: IR`$_{25\text{-}0}$ `:: 00`	jump to target address
`jr`	`$rs`	#	`PC <-- R[s]`	jump to address in $rs
`jal`	`offset`	#	`R[31] = PC + 4`	save return address in $ra
		#	`PC <-- PC`$_{31\text{-}28}$ `:: IR`$_{25\text{-}0}$ `:: 00`	jump to target address
`jalr`	`$rs`	#	`R[31] = PC + 4`	save return address in $ra
		#	`PC <-- R[s]`	jump to address in $rs

Formats

- What's different about `j, jal`? No regs specified.
  - offset: usually 16-bit 2C immediate, except j and jal 26-bit UB (J-type)
- Note that type refers to FORMAT, not function, so branch instrs are not J-type
- What type are `jr` and `jalr`?
- Since registers have to be specified, `jr` and `jalr` are R-type

Addressing: pseudo-direct

- doesn't specify 32-bit address directly
- allows you to access how much of all possible word-aligned addresses?

Jumping to arbitrary word-aligned addresses

- `j` and `jal` still don't allow you to access all possible word-aligned addresses
- `jr` and `jalr` refer to a register, which specifies the entire 32 bit instruction

Instruction formats: J-type

J-type: jump

```
j       target  # jump to target address
```

b_{31-26} b_{25-0}

opcode target

semantics:

PC <- PC_{31-28} :: IR_{25-0} :: 00

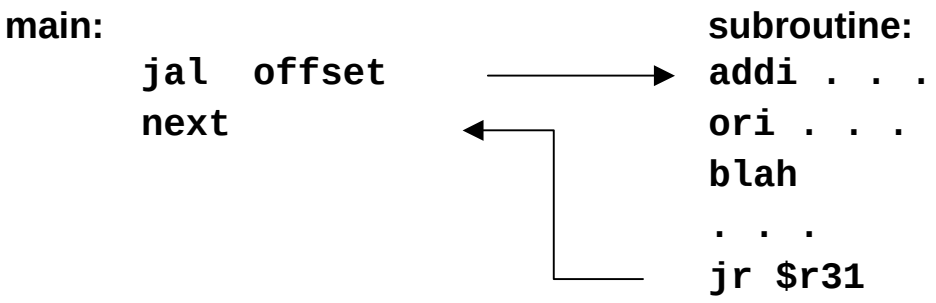
update the PC by using:

- upper 4 bits of the program counter
- 26 bits of the target (lower 26 bits of instruction register)
- two 0's

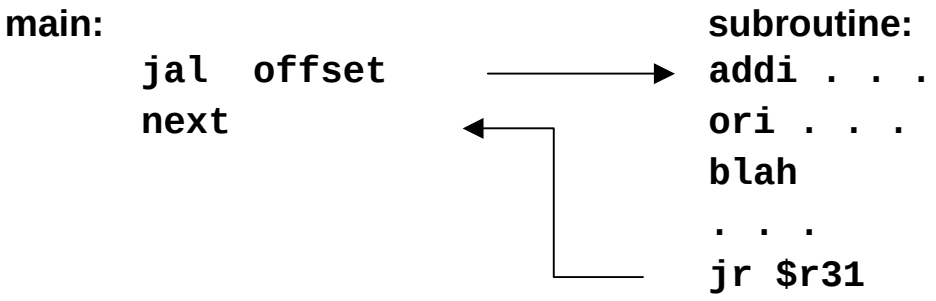
(creates a 32-bit address)

Why 2 0's?

Jump: subroutine calls



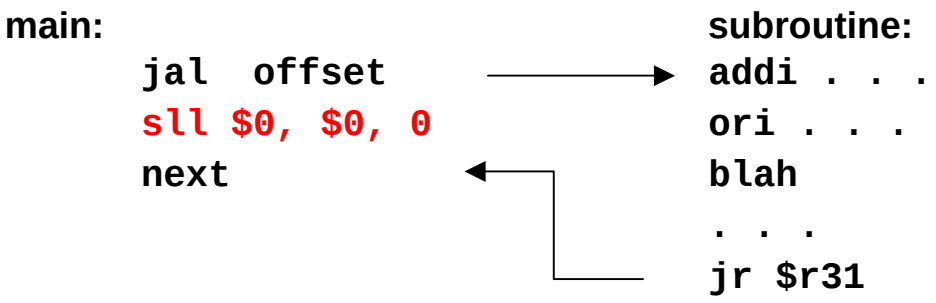
Jump: subroutine calls



What happens if a `jal` call is made while in a subroutine?

- **return address overwritten with a new return address**
- **must place the return address onto the stack (memory)**

Jump: subroutine calls



What happens if a `jal` call is made while in a subroutine?

- **return address overwritten with a new return address**
- **must place the return address onto the stack (memory)**

It turns out that the return address is PC + 8, not PC + 4

- **jump instructions have to be followed by a branch delay slot instruction**
- **purpose of avoiding stalling in pipelines**

Labels

Branch Instructions and Labels

- **Do we want to have to compute the value of the offset for a branch instruction? No.**
  - **too much work**
  - **subject to error**
  - **may need to change if instructions are added or removed**
- **better way:**

```
    beq  $r1, $r2, L1     # (0) If R[1] == R[2] goto L1
    addi $r1, $r1, 1      # (1) R[1]++
    addi $r2, $r2, 1      # (2) R[2]++
L1: add  $r1, $r1, $r2    # (3) R[1] = R[1] + R[2]
```

- **L1 is a label referring to the address of the instruction where it is located**
- **The assembler computes the offset**
  - **target instruction - (branch instruction + 1).**
- **The same rule applies even if you branch backwards**
- **Why compute branch instruction + 1?**
  - **Once the instruction is fetched, PC is incremented to PC + 4,**
    - **next instruction in memory**
- **This kind of address computation is called PC-relative**

Jump Instructions and Labels

- **Jump instructions also jump to labels (at least, `j` and `jal`)**
- **Assembler figures out the appropriate addresses using pseudo-direct addressing.**
- **Disadvantage compared to PC-relative addressing: It's harder to relocate the code**

Instruction Types

- **Arithmetic** 
- **Logical** 
- **Data Transfer** 
- **Compare/Branch** 
- **Jump**