# MIPS control flow instructions:

## *Jumps, Branches, and Loops*

PLEASE
PICK UP
HANDOUT

# Today's lecture

- **Control Flow**
  - Programmatically updating the program counter (PC)
- **Jumps**
  - Unconditional control flow
  - How is it implemented?
- **Branches**
  - Loops
  - How implemented?
- **Jump Register**
  - Unlimited range jumps
  - How implemented?

# Control Flow

iclicker.

- **So far, only considered sequences of arithmetic instructions**

```
mul     $14, $13, $20      0x400000
addi    $14, $14, 4        0x400004
sub     $15, $14, $15      0x400008
```

- **These are executed one after another**
  - Stored sequentially in memory
  - Program counter is incremented by 4 each cycle.

**a) 0x400010    b) 0x400012    c) 0x40000b    d) 0x40000c**

# Control Flow in high-level languages

- **In high-level languages, we can:**
  - Repeat statements with loops

```
for (int i = 0 ; i < N ; i ++) {
    sum += i;
}
```

  - Selectively execute statements with if/then/else

```
if (x < 0) {
    x = -x;
}
```

- **Need ways to control which instruction is executed next.**

# Unconditional Jumps

- **The simplest control flow instruction is jump:**

  always happen

  - Unconditional control flow transfer
    - always taken, much like a goto statement in C

```
j target_label
```

- **Uses a "label" to tell where in the code to jump to:**
- **Example:**

```
Loop:        j Loop
```

- **What does this code do?** infinite loop

# Encoding Jumps

- **To encode jumps we use the J-type instruction format:**

31 26

only use

| op | address 00 |
|---|---|
| 6 bits | 26 bits |

- **This format provides a very long immediate**
  - But, not quite long enough to specify a whole 32-bit PC
  - Where do the other 6 bits come from? divide by 4
    - Last two bits are always 00, because PC value is always word aligned
    - 4 most significant bits come from existing PC value.

$6-2=4$

# Example encoding

- **The infinite loop:**

```
Loop:      j Loop
```

- **After assigning instructions to memory addresses**

```
0x400024:      j 0x400024
```

0x400024 = 0000 0000 0100 0000 0000 0000 0010 0100

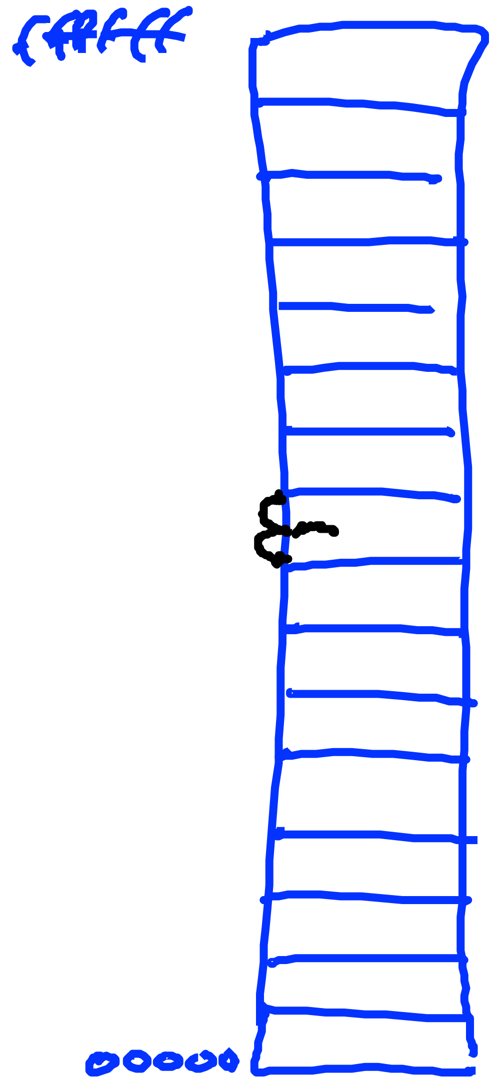
0 2

| op | address |
|---|---|
| 00 0010 | 00 0001 0000 0000 0000 0000 1001 |

Jump opcode 2

# Limitations

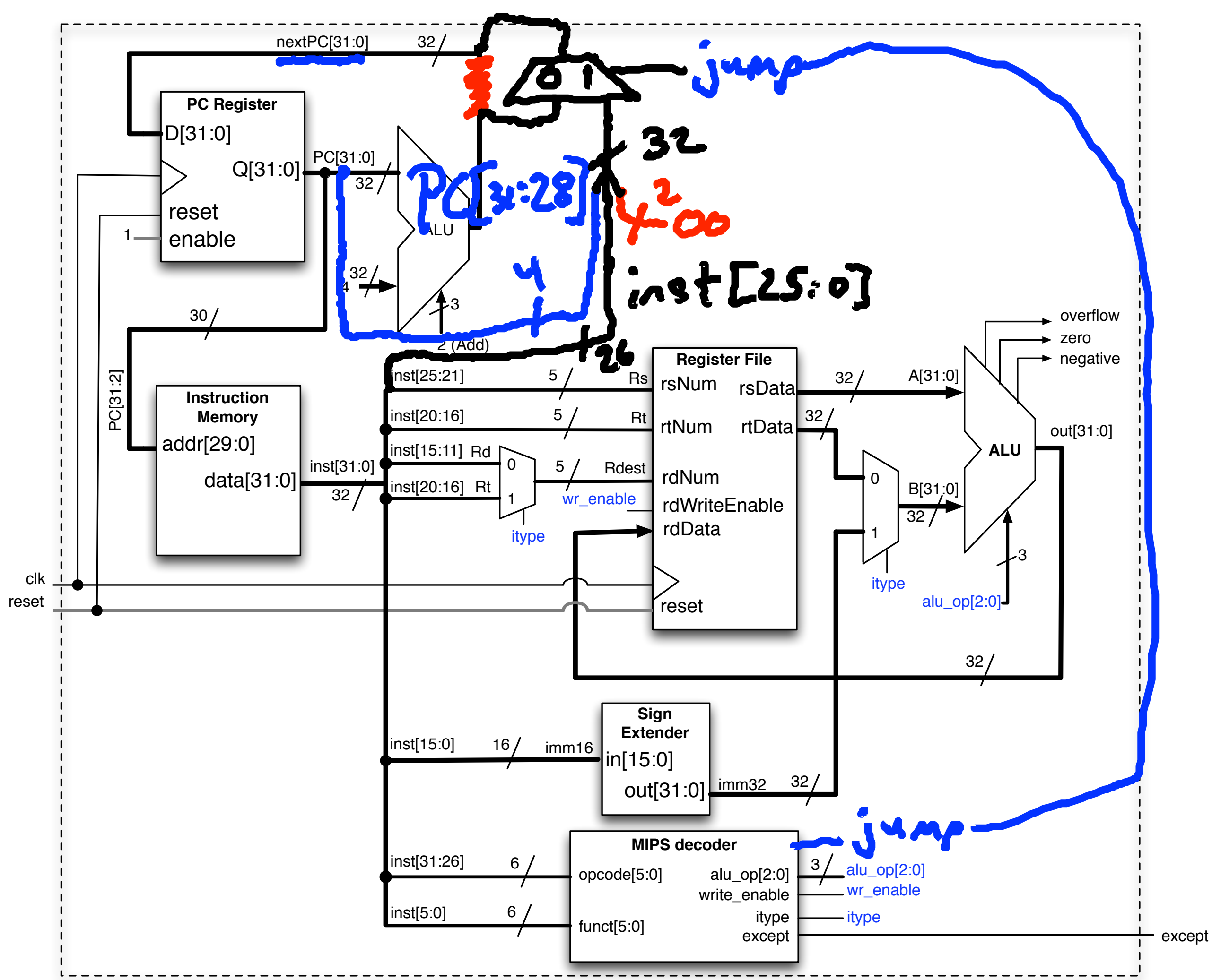
- **Top 4 bits coming from current PC means:**
  - Memory is cut into 16 regions
  - Can only jump within current region with j instruction.

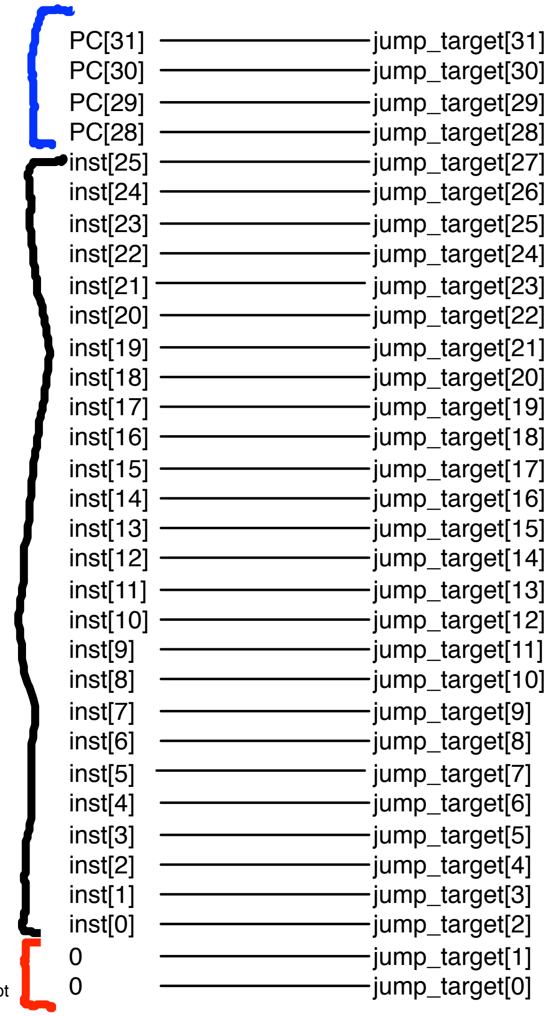


- **A 26-bit address field lets you jump to any address from 0 to $2^{28}$.**
  - your Lab solutions had better be smaller than 256MB

# Implementing Jumps

4+26+2=32



# Conditional Branches

- **For our loops to exit, we need conditional control flow.**

Sometimes Jump: somtimes not

```
beq rs, rt, target_label
```

- **Branch if EQual (BEQ):** Equal → Jump
  - If `(R[rs] == R[rt])`, then branch to `target_label`
  - Otherwise execute next instruction

- **Also, Branch if Not Equal (BNE):**
  - Same, but branch when `(R[rs] != R[rt])`

# Using beq/bne to implement loops:

- How could we use branches to implement the following?

```
int sum = 0, i = 0;
do {
   sum += i;
   i++
} while (i != 10)
```

2 3

true

```
      addi $4, $0, 10
      addi $2, $0, 0
      addi $3, $0, 0
loop: add  $2, $2, $3
      addi $3, $3, 1
      bne  $3, $4, loop
```

sum += i ⇒ sum = sum + i;

i+1 → i = i + 1;

# Using beq/bne and j to implement loops:

- Let's implement the for version of the loop?

For loop

$2  $3  $7

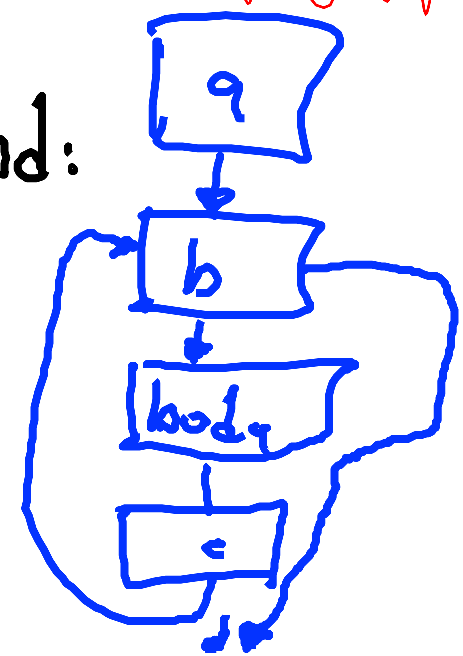
```
int sum = 0;
for (int i = 0 ; i != x ; i++)
   sum += i;
}
```

```
        a →  addi  $2, $0, 0
        a →  addi  $3, $0, 0
loop:   b →  beq   $3, $7, end
             add   $2, $2, $3
        c →  addi  $3, $3, 1
             j     loop
end:
```

if不相等执行 直到相等

for loop条件符合执行的code

for (A ; B ; c) {
    body
}
d



# Using beq/bne to implement if/then:

$2

- **How could we use branches to implement the following?**

```
if (x == 0) {
    x = 1;
}
```

```
      bne  $2, $0, skip
      add  $2, $0, 1
skip:
```

- *Hint: Sometimes it's easier to invert the original condition.*
  - *Change "continue if x == 0" to "skip if x != 0".*

# Encoding Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

$rs $rt

3 -3

```
      beq   $1, $0, L
      add   $1, $3, $0
      add   $2, $3, $3
      j     Somewhere
L:    add   $2, $3, $3
```

- Since the target L is 3 *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:

I-type

| | | | |
|---|---|---|---|
| 00 0100 | 00001 | 00000 | 0000 0000 0000 0011 |
| op | rs | rt | address |

*SPIM's encoding of branch offsets is off by one, so its code would contain an address of 4. (But it has a compensating error when it executes branches.)*

# Larger branch constants

- **Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away**
  - branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- **If you do need to branch further, you can use a jump with a branch. For example, if "Far" is far away, then the effect of:**

```
        beq  $s0, $s1, Far
        ...
```

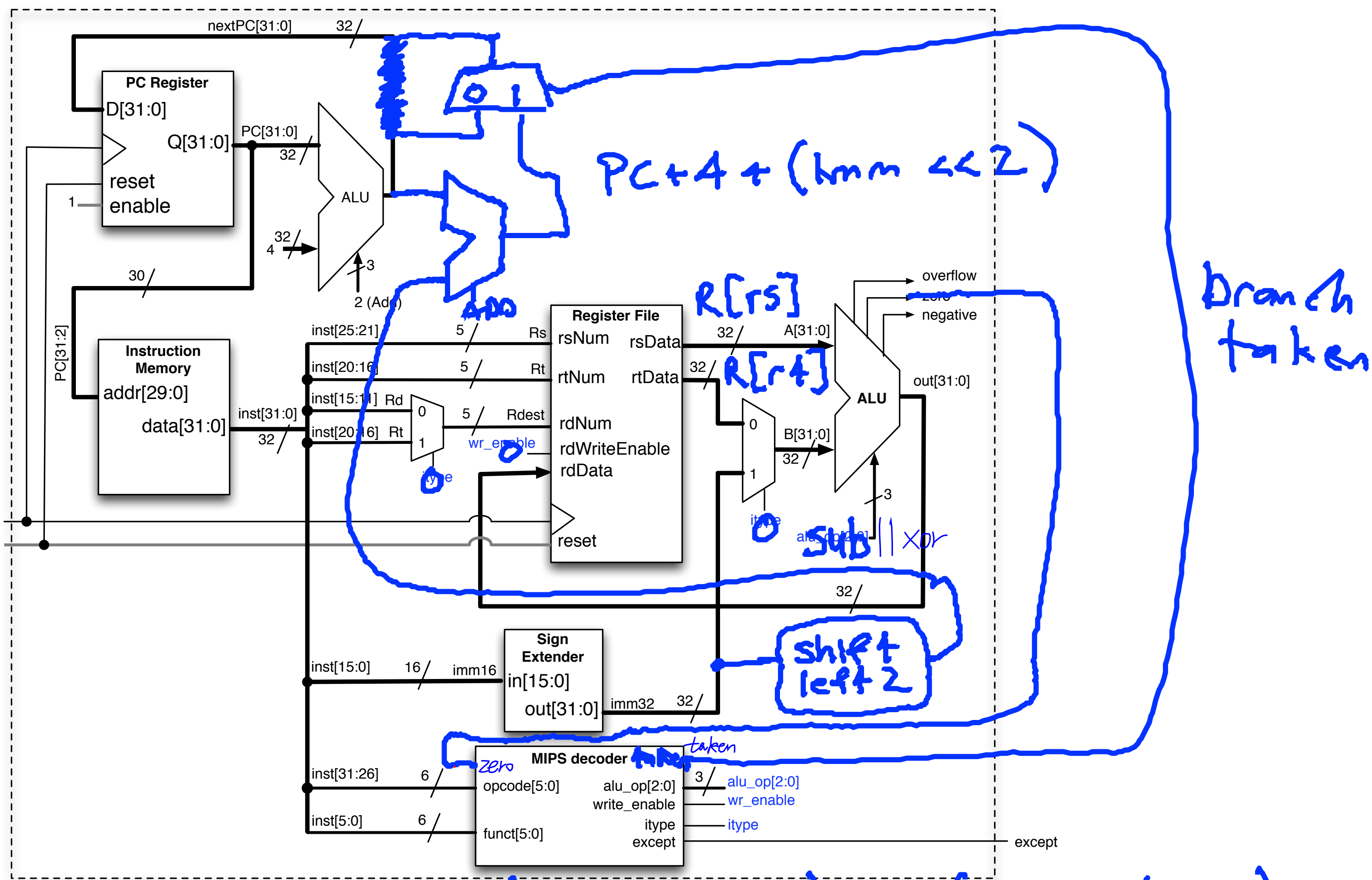
**can be simulated with the following actual code.**

```
        bne  $s0, $s1, Next
        j    Far
Next:   ...
```

- **The MIPS designers have taken care of the common case first.**

# Implementing Branches

be




taken = (beq & zero) or (bne & !zero)

# Jump Register

- **`j` instructions allow you to jump within a 256MB range**
  - What if you want to go outside that range

```
jr $3
```

- **Jump Register (JR)**
  - Copy the 32-bit contents of a register into the PC.

```
PC=R[rs]
```

  - That value better be word aligned (i.e., divisible by 4)

- **We'll see how this is used later.**

# Encoding Jump Register

- **Jump register only needs 1 register specifier**
- **Use R-type encoding, because it is cheapest opcode-wise.**

`jr $rs`

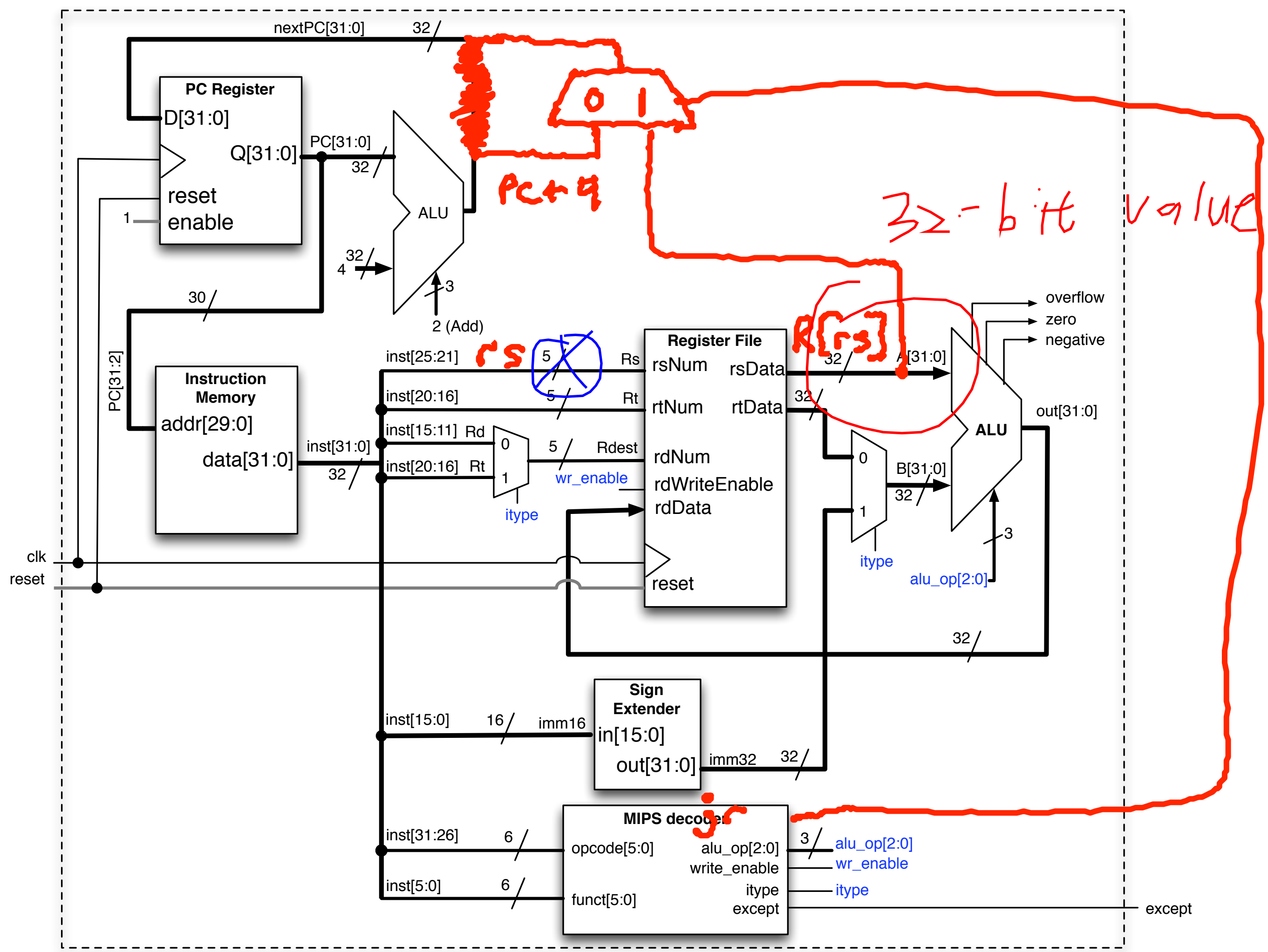
| op | rs | rt | rd | shamt | func |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- **Example:**

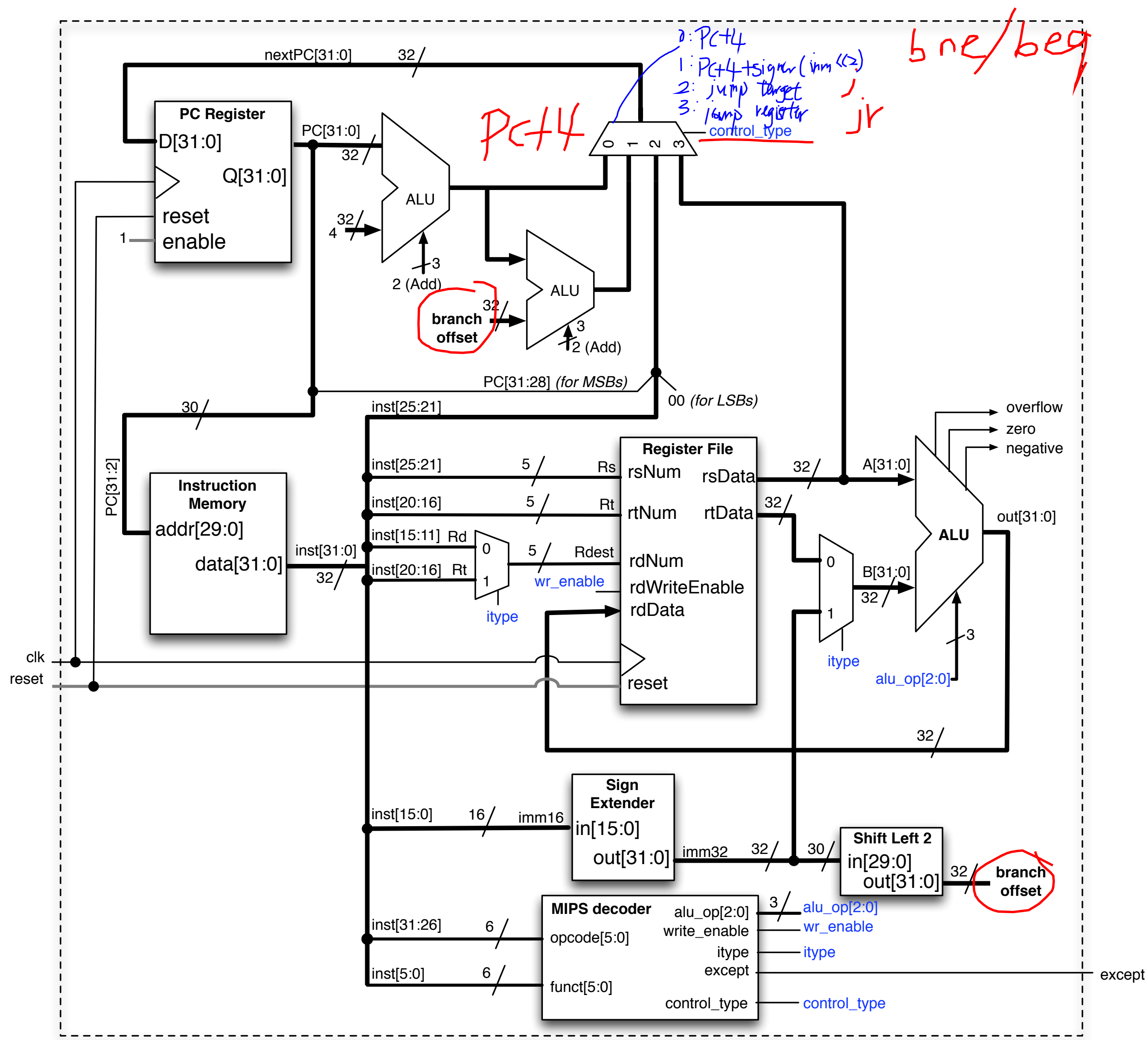
`jr $3`

| | | | | | |
|---|---|---|---|---|---|
| 00 0000 | 000 11 | 00000 | 00000 | 00000 | 00 1000 |

# Implementing Jump Register



# Control Implemented



# How do we put a 32-bit value into a register?

- **I-type instructions can do a 16-bit immediate:**
  - Many instructions are useful for setting the low 16b, *e.g.,*

```
addi  $12, $0, 0xbeef   # $12 = 0x0000beef
```

- **Would be useful to be able to set the top 16b.**

- **MIPS provides the Load Upper Immediate (`lui`) instruction**
  - lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.

```
lui $12, 0xdead   # $12 = 0xdead0000
```

# Load Upper Immediate

```
lui $12, 0x3D
ori $12, $12, 0x900
```

a) 0x0000093d
b) 0x0003d900
c) 0x003d0900
d) 0x0009003d
e) 0x0900003d

- **This illustrates the principle of making the common case fast.**
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.

# LUI is an I-type instruction

| | | | |
|---|---|---|---|
| 00 1111 | 00000 | | |
| op | rs | rt | imm |

- **R[rt] = {imm, 16'b0}**

```
lui $12, 0xdead    # $12 = 0xdead0000
```