MIPS control flow instructions:

Jumps, Branches, and Loops

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

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

```
mul     $14, $13, $20
addi    $14, $14, 4
sub     $15, $14, $15
```

- **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:**
  - 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?**

Encoding Jumps

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

op	address
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?
    - Last two bits are always 00, because PC value is always word aligned
    - 4 most significant bits come from existing PC value.

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
```

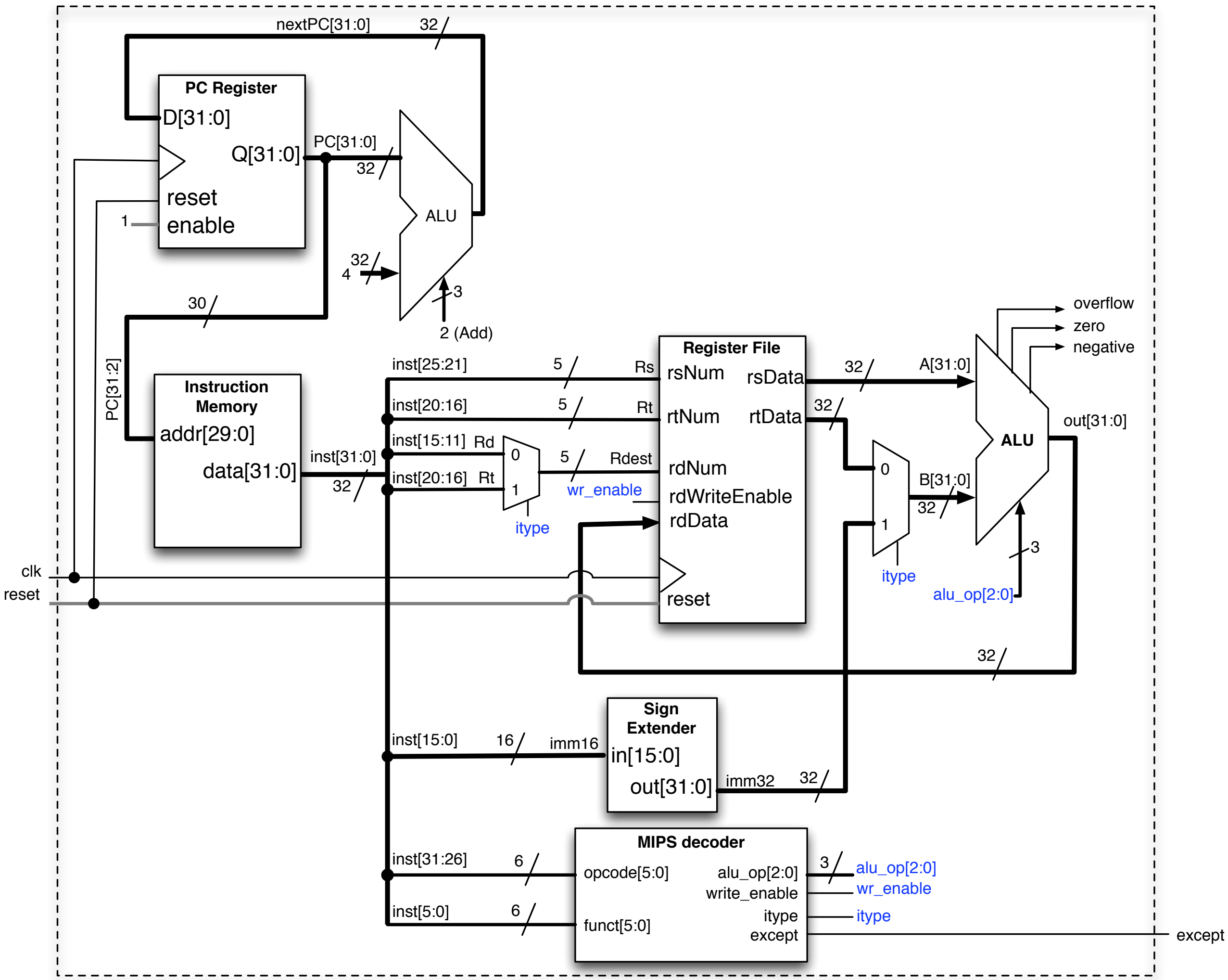
op	address

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



PC[31]	jump_target[31]
PC[30]	jump_target[30]
PC[29]	jump_target[29]
PC[28]	jump_target[28]
inst[25]	jump_target[27]
inst[24]	jump_target[26]
inst[23]	jump_target[25]
inst[22]	jump_target[24]
inst[21]	jump_target[23]
inst[20]	jump_target[22]
inst[19]	jump_target[21]
inst[18]	jump_target[20]
inst[17]	jump_target[19]
inst[16]	jump_target[18]
inst[15]	jump_target[17]
inst[14]	jump_target[16]
inst[13]	jump_target[15]
inst[12]	jump_target[14]
inst[11]	jump_target[13]
inst[10]	jump_target[12]
inst[9]	jump_target[11]
inst[8]	jump_target[10]
inst[7]	jump_target[9]
inst[6]	jump_target[8]
inst[5]	jump_target[7]
inst[4]	jump_target[6]
inst[3]	jump_target[5]
inst[2]	jump_target[4]
inst[1]	jump_target[3]
inst[0]	jump_target[2]
0	jump_target[1]
0	jump_target[0]

Conditional Branches

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

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

- **Branch if EQual (BEQ):**
  - 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)
```

Using beq/bne and j to implement loops:

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

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

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.

```
        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:

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:**

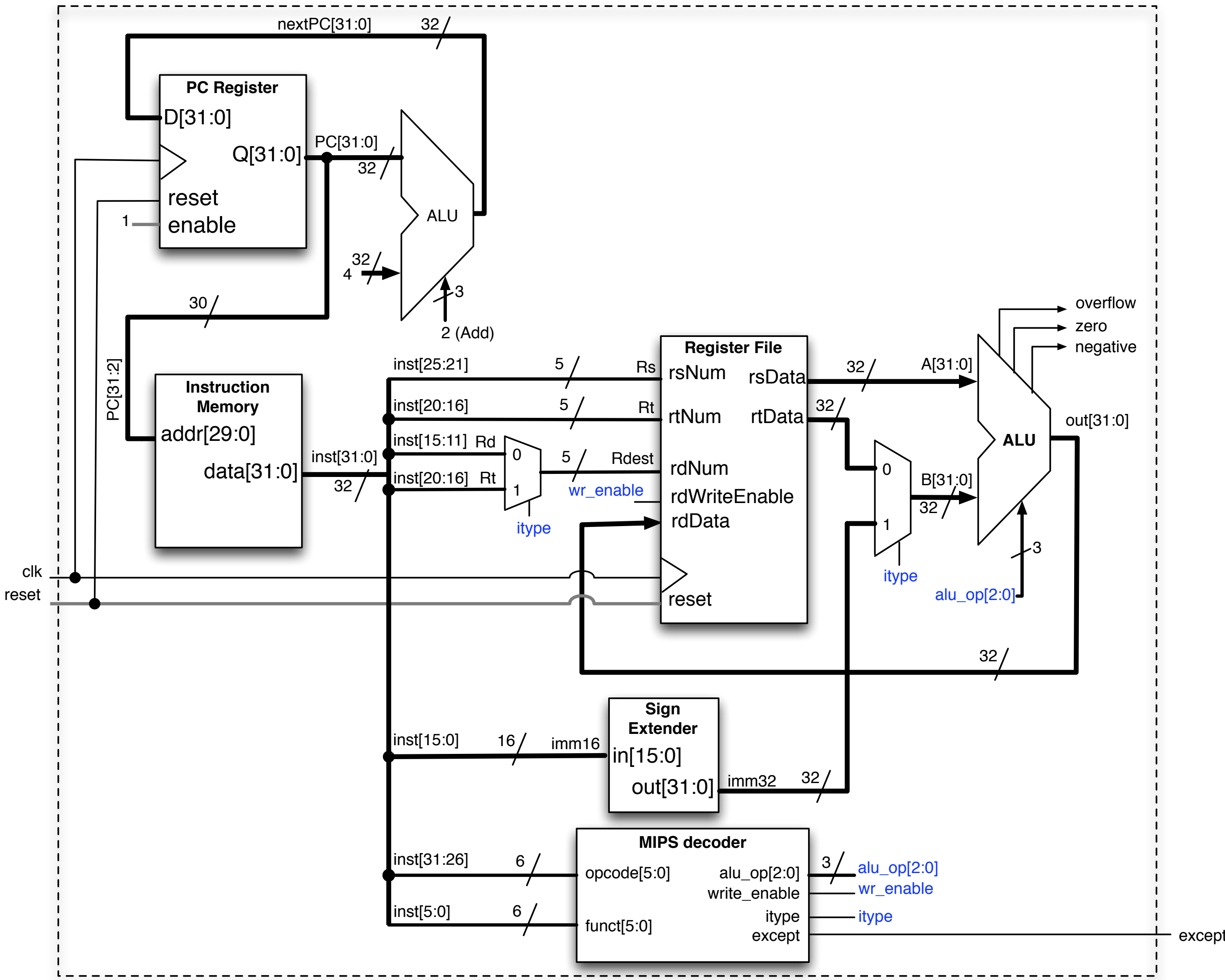
```
        beq  $s0, $s1, Far
        ...
```

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



Jump Register

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

```
jr $3
```

- **Jump Register (JR)**
  - Put any 32-bit address into a register.
    - Make sure it is word aligned (i.e., divisible by 4)
  - That value is copied to the PC.
- **We'll see how this is used later.**

Encoding Jump Register

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

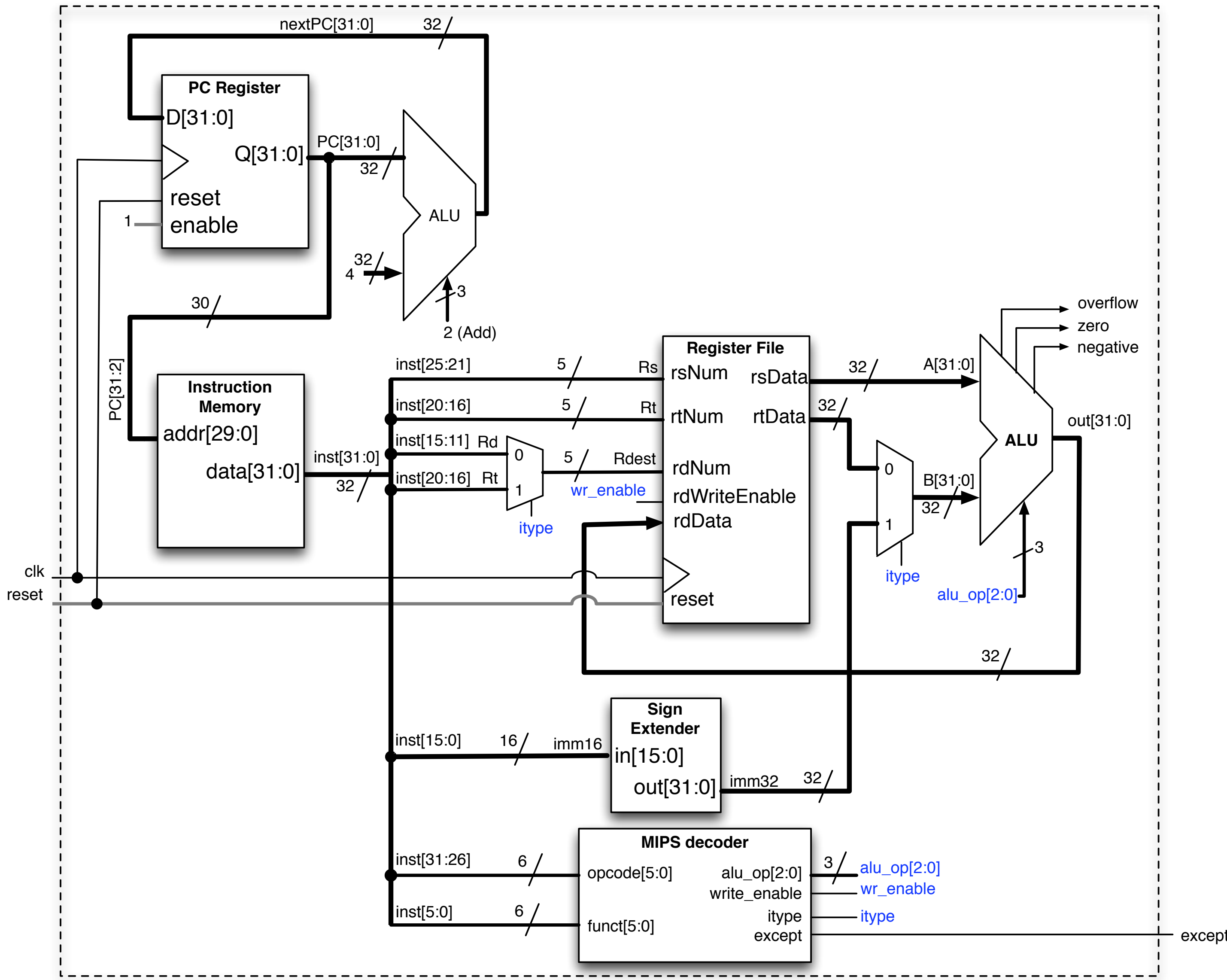
`jr $rs`

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

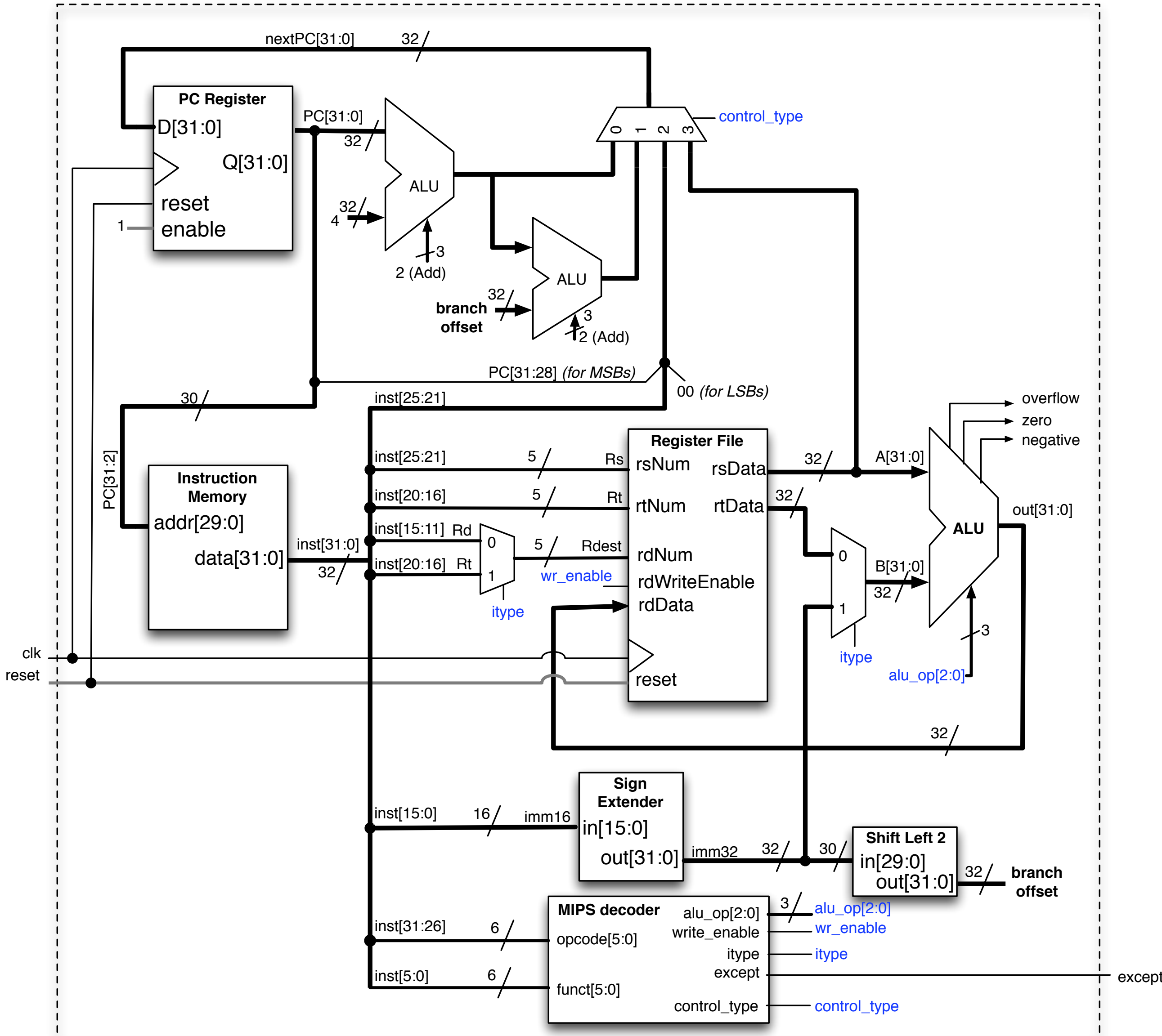
- Example:

`jr $3`

Implementing Jump Register



Control Implemented



Overflow

- **In which circumstance can overflow <u>not</u> occur?**
  - A: subtracting a positive number from a negative number
  - B: subtracting a negative number from zero
  - C: adding two negative numbers
  - D: subtracting a negative number from a positive number
  - E: subtracting a negative number from a negative number