MIPS control flow instructions: Jumps, Branches, and Loops

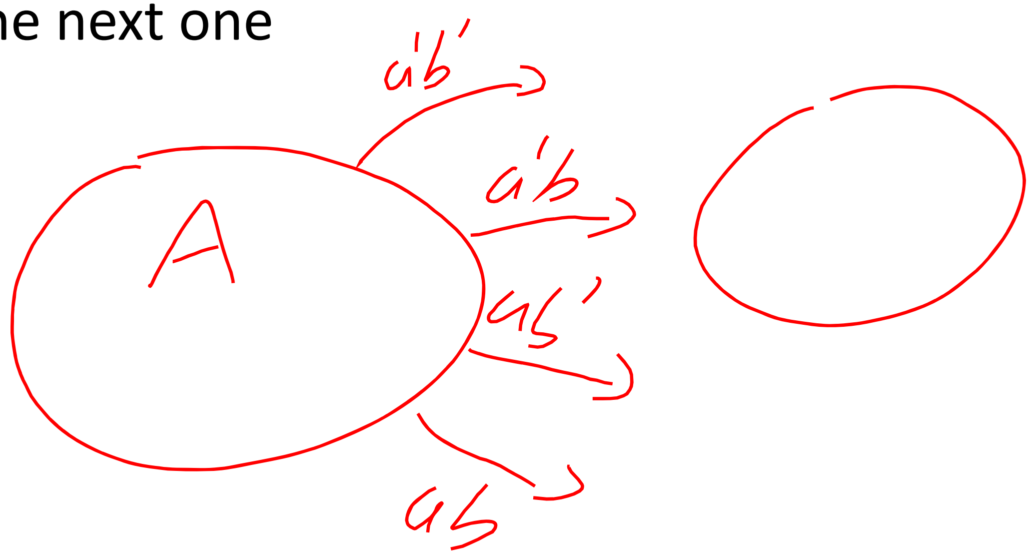
Sign up for exam 2

Pick up handout

+ bring back FRIDAY

#1 tip for exam 2

- Work one state at a time
- Analyze ALL POSSIBLE input combinations for each state before analyzing the next one



Today's lecture

- Control Flow
  - Programmatically updating the program counter (PC)
- Jumps
  - Unconditional control flow
  - How is it implemented?
- Branches
  - Loops
  - If/then/else
  - How implemented?

Sequential lines of code are executed by "incrementing" the Program Counter

PC → 0x00400004	mul	$14, $13, $20
8	addi	$14, $14, 4
C	sub	$15, $14, $15
10	xor	$12, $15, $8

- Where is instruction XOR located?

a) 0x00400007 b) 0x00400008

c) 0x00400010 d) 0x00400016

We use control flow in high level languages to implement conditionals and loops

Loops

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

Conditionals

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

How do we implement these in MIPS assembly?

An unconditional jumps always transfer control (like a goto statement in C)

- Use a “label” to tell where in the code to jump to:

jump

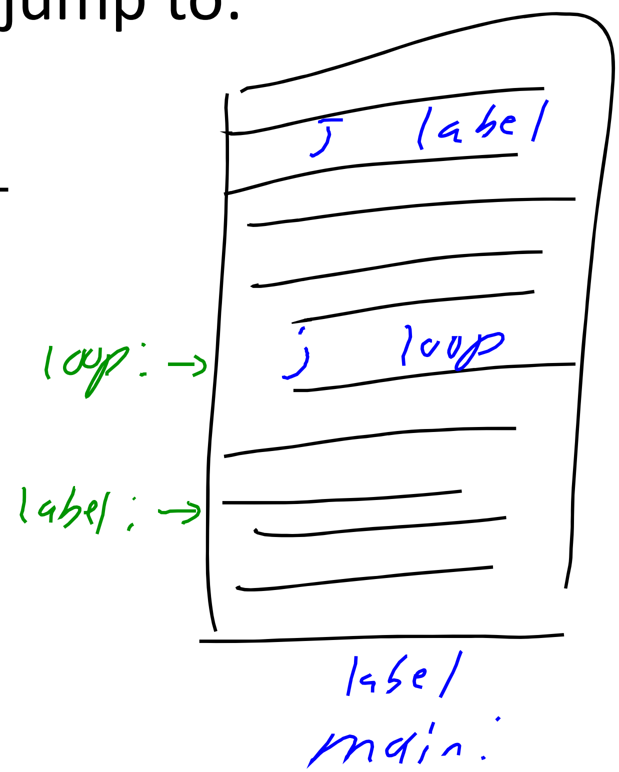
```
j target_label
```

address

- Example:

```
Loop:    j Loop
```

- What does this code do?



Jumps are encoded with J-type instructions

00 0010

op	address
6 bits	26 bits

0000 conditional 00

PC [31:28] 28 bits

Address	Data
0x00000000	
0x00000001	
0x00000002	
0x00000003	
0x00000004	j label
0x00000005	
…	
…	

32 bits

- jump Where do the other 6 bits come from?
  - unconditional Last two bits are always 00, because PC is word aligned
  - 4 most significant bits come from existing PC value.

Example encoding: The infinite loop

Loop Loop

0x00400024: j 0x00400024

PC+4[31:28]

26

0x00400024	0000 0000 0100 0000 0000 0000 0010 0100

32

0 1 0 0 0 0 9

op	address
00 0010	00 0001 0000 0000 0000 0000 1001

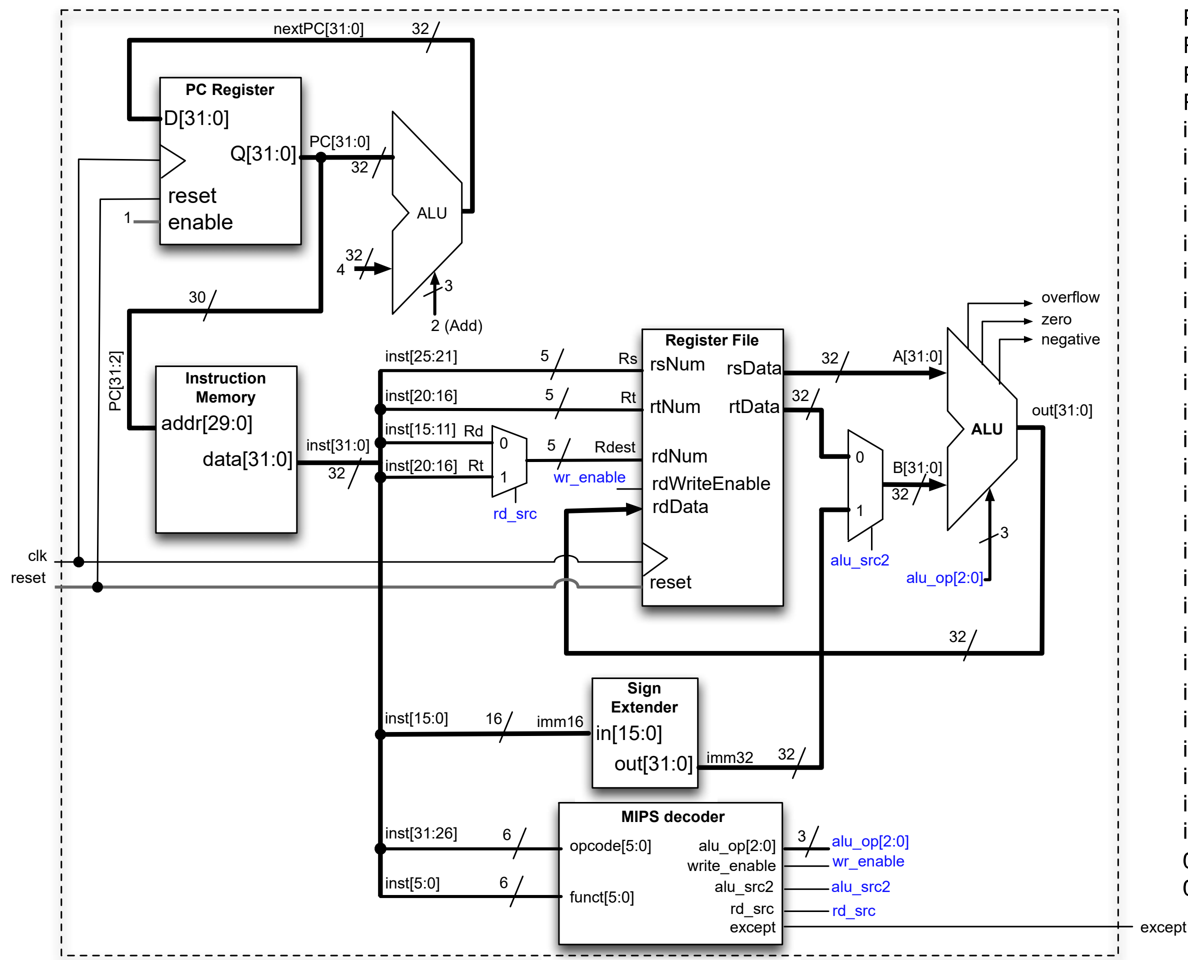
6

26 — Inst[25:0]

Jump instructions can only stay within 1 of 16 regions

- 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

Implement Jump



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]

What should wr_enable be?

a) 0

b) 1

c) don't care

Branches provide conditional **control** flow

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

source source label address

- Branch if EQual (BEQ):
  - If `(R[rs] == R[rt])`, then branch to `target_label`
  - Otherwise execute next instruction (PC+4)

```
bne rs, rt, target_label
```

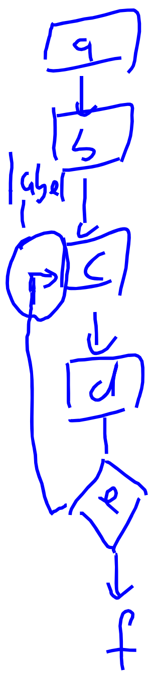
- Branch if Not Equal (BNE):
  - Branch when `(R[rs] != R[rt])`

1) where to edit

2) Add muxes

sum = sum + i (rd = sum, rs = sum, rt = i)

Implement the C code in MIPS assembly




```
int sum = 0;      // a   ($2)
int i = 0;        // b   ($3)

do {              // c
  sum += i;       // label
  i ++;           // d
} while (i != 10) // e
                  // f
```

Assembly:

```
      addi  $4, $0, 10     # temp = 10+0
a     addi  $2, $0, 0      # sum = 0
b     addi  $3, $0, 0      # i = 0
c  do:
      add   $2, $2, $3     # sum += i
      addi  $3, $3, 1      # i++
d     b___  $3, $4, do     # while (i != 10)
```

iclicker

A) eq
B) ne

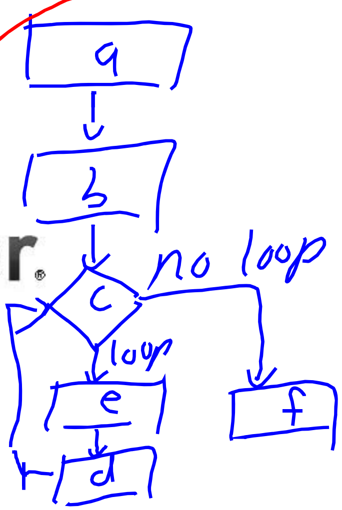
Implement the C code in MIPS assembly

```
         $2
  d      ↓
int sum = 0;

             b  $3            c   $4       d
for (int i = 0 ; i != x ; i ++) {

           e          ?
   sum += i;

}
f
```



iclicker

A) eq
B) ne

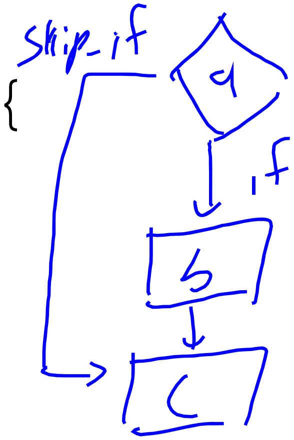
Assembly:

```
          addi  $2, $0, 0          #a
          addi  $3, $0, 0          #b
loop:     b___  $3, $4, end_loop   #c
          add   $2, $2, $3         #e
          addi  $3, $3, 1          #d
          j     loop
end_loop:                          #f
```

Implement the C code in MIPS assembly

$2

```
a        !=
if (x ~~==~~ 0) {
  b x = 1;
}
c
```



Assembly: b ____ $2, $0, skip_if #a

addi $2, $0, 1 #b

skip_if: #c

Hint: Sometimes it's easier to invert the original condition.

Change "continue if x < 0" to "skip if x >= 0".

A) eq

B) ne

The address in branch is an *offset* from PC+4 to the target address

Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}

```
PC ->     beq $1, $0, L
PC+4 ->   add $1, $3, $0
          add $2, $3, $3
          j   Somewhere
PC+16 L:  add $2, $3, $3
```

a) 1
b) 2
c) 3
d) 4

Branch Address

- What value should be stored in the address of the beq instruction?

op	rs	rt	address	
000100	00001	00000	0000 0000 0000 0011	00

← # of instructions

Architecture Design: Make the common Case fast

- Most branches go to targets less than 32,767 instructions away

- Slowly simulate branches that are farther than 32,767 (i.e., Far) instructions away

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

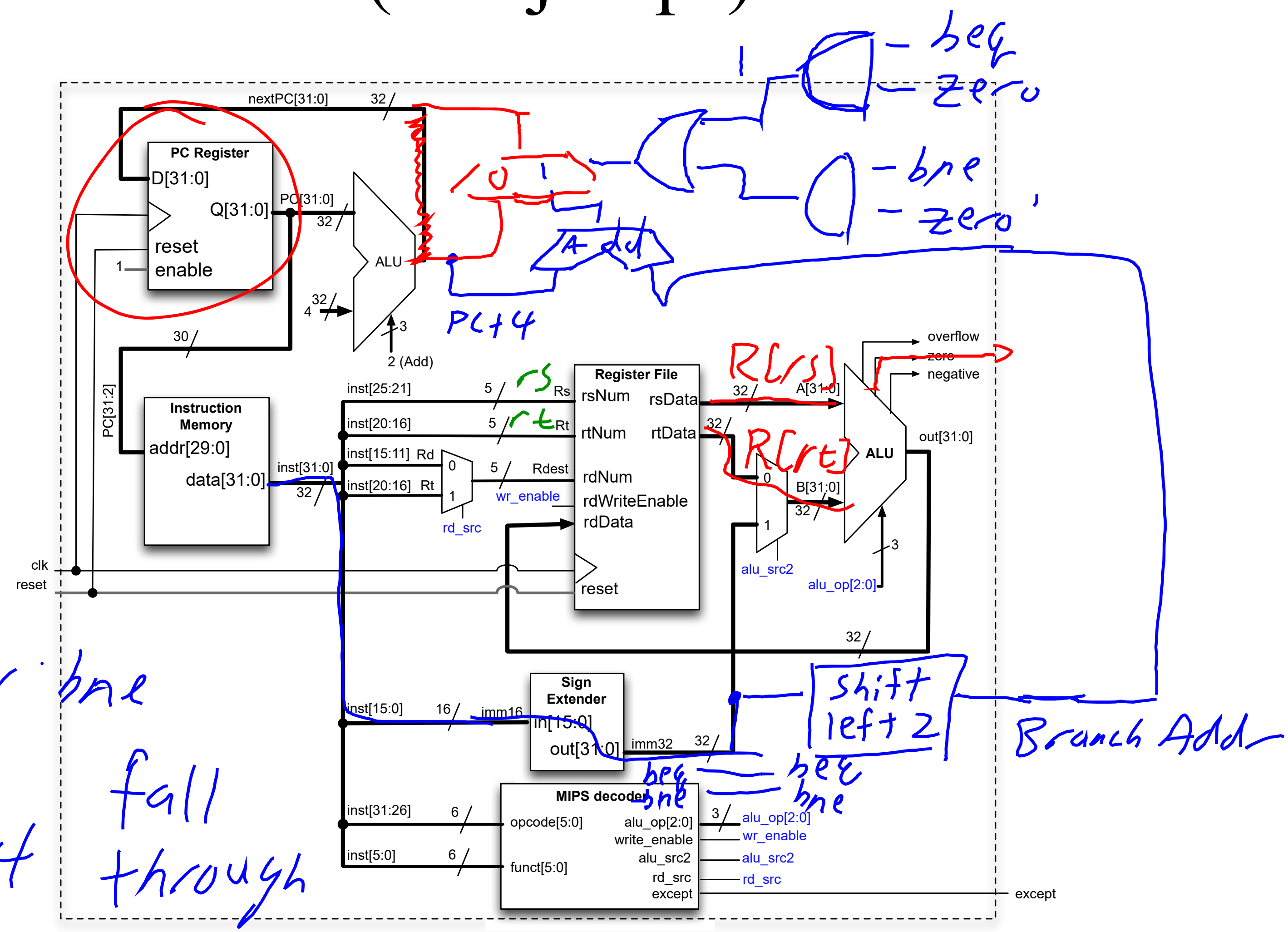
```
        bne    $s0, $s1, Next
      j   Far
Next: ...
```

Implement Branches (w/o jumps)

iclicker.

What should we set alu_op to for beq and bne?

a) ADD
b) SUB
c) AND
d) OR
e) NOR



beq or bne
↓
PC+4

fall through

Use Jump Register (JR) to jump beyond 256MB

2

jr rs

PC=R[rs]

- rs acts as a pointer to a pointer

rs	R[rs]
$1	0xE1831525
$2	0x10105603
$3	0x49318461
$4	0xA1891028
…	…
…	…

PC

Address	Data
0x00000000	
0x0	…
0x10000000	
0x1	…
0x20000000	
0x2	…
0x30000000	
0x3	…
0x40000000	
0x4	…
…	…
…	…
…	…
0xF0000000	
0xF	…

Which rs could be used correctly in JR?

A) $1 B) $2 C) $3 D) $4 E) Any

Jump register is R-type but only needs 1 register specifier

`jr $rs`

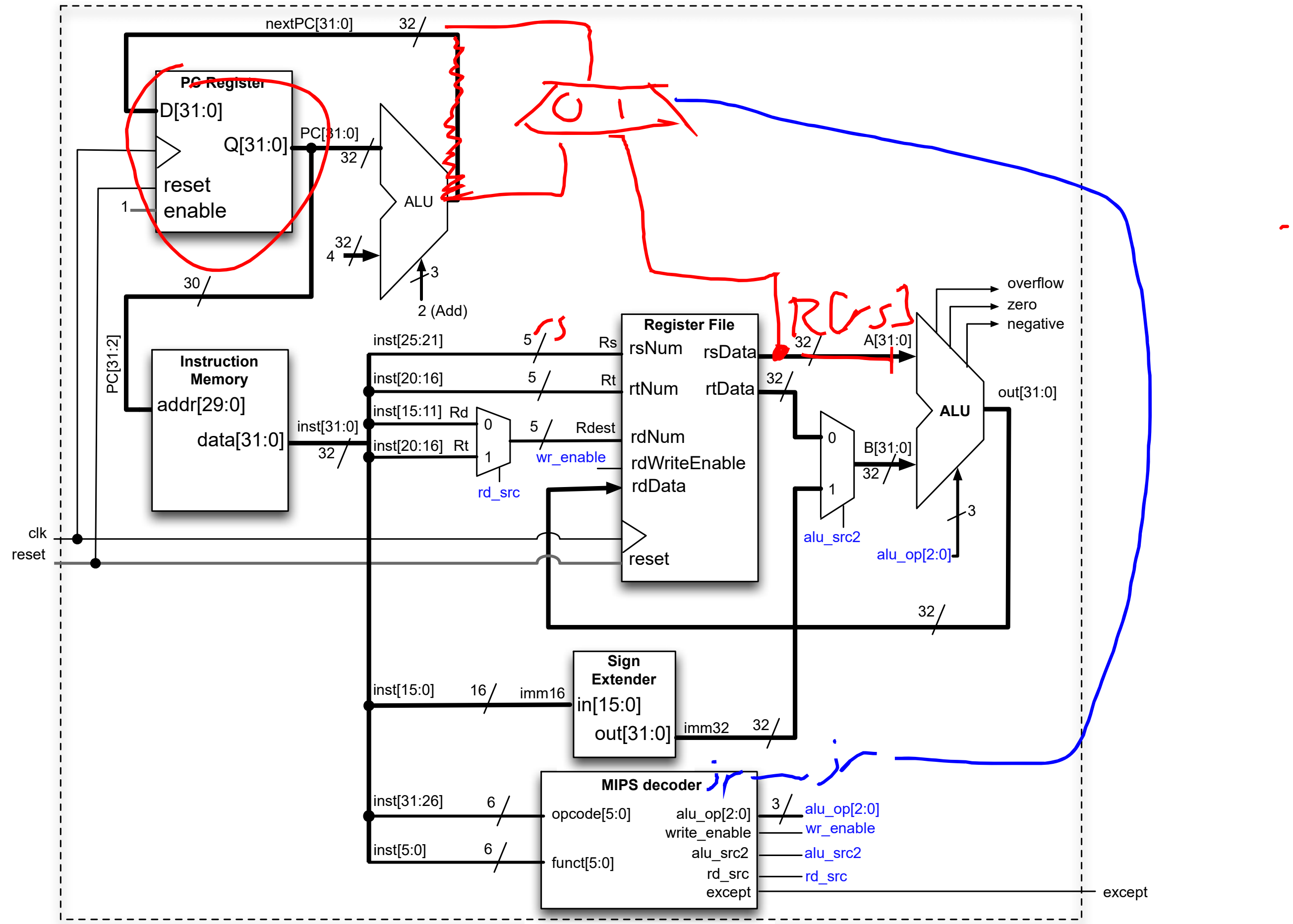
op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Example:

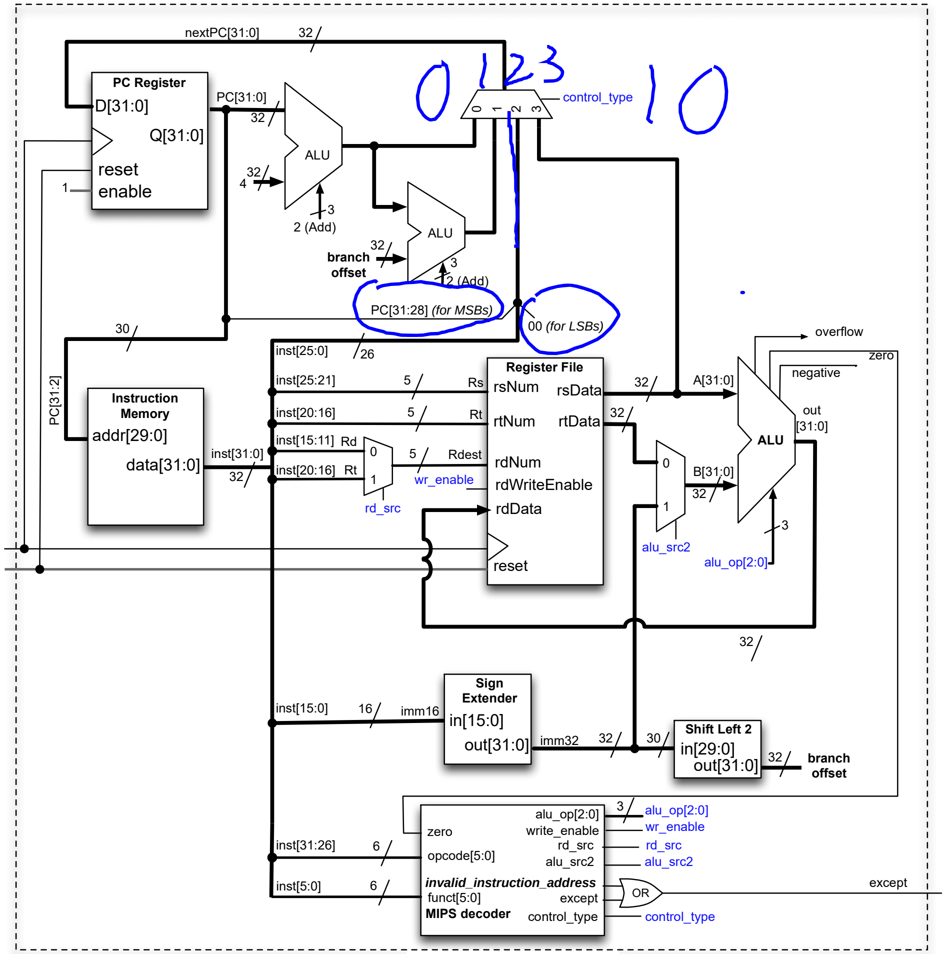
`jr $3`

000000	00011				001000

Implementing Jump Register



Control Implemented



Which type of branch is taken when control_type = 10

a) No branch taken

b) Taken branch

c) j

d) jr

Architecture Design: Make the common Case fast

- To use JR we need to set all 32 bits in a register, but we do not have an instruction to do this directly.

- Most of the time, 16-bit constants are enough.

- It's still possible to load 32-bit constants, but at the cost of multiple instructions and temporary registers.

Use two instructions `ori` and `lui` to construct 32-bit addresses

- `ori` can set the lower 16 bits

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

- Load Upper Immediate (`lui`) can set the upper 16 bits
  - 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
```

0

lui is an I-type instruction

op	rs	rt	imm
00 1111	X	01100	

- R[rt] = {imm, 16’b0}

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

iclicker

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

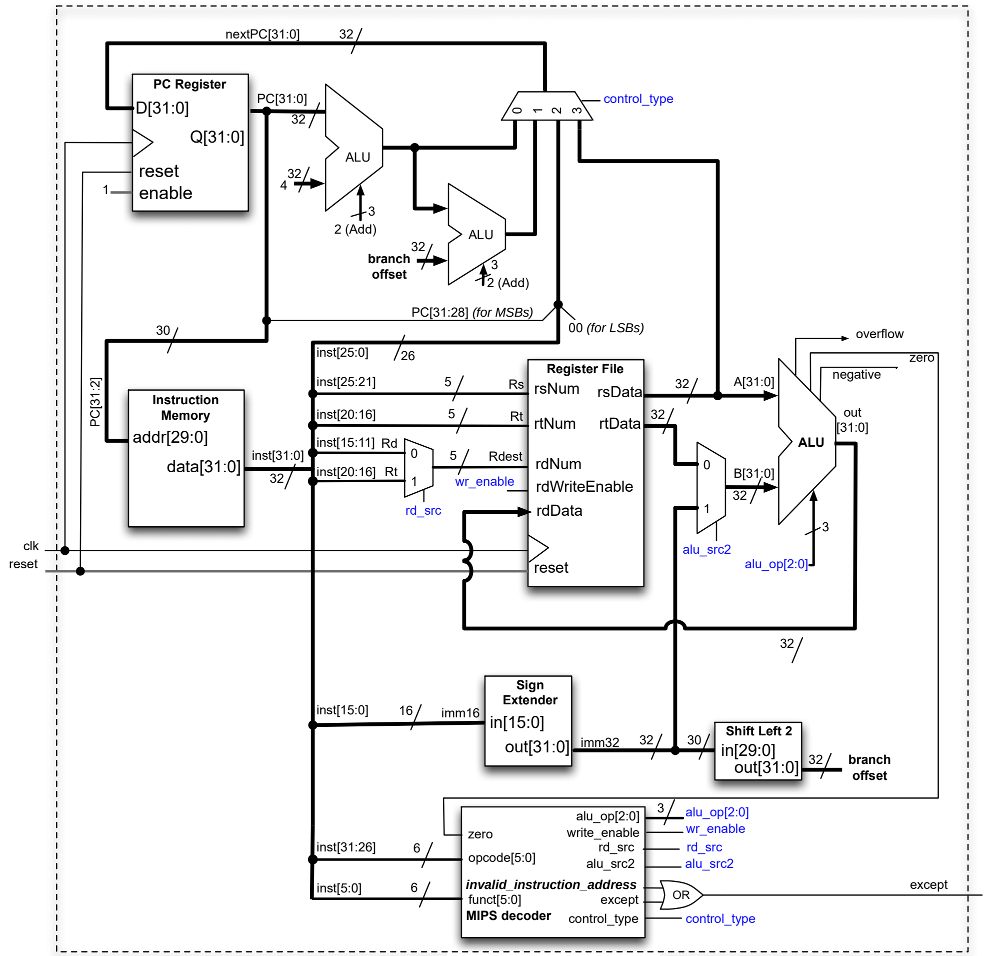
```
              $0
ori $12, $12, 0x900
lui $12, 0x3D
```

These two code snippets will store the same value in Register 12.

A) True

B) False

lui Implemented



Value for alu_src2?

rd_src?

a) 0

b) 1

c) x

Implement the C code in MIPS assembly

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

Assembly:

iclicker

A) eq

B) ne

Set if Less Than (slt) sets a register to a Boolean (1 or 0) based on a comparison.

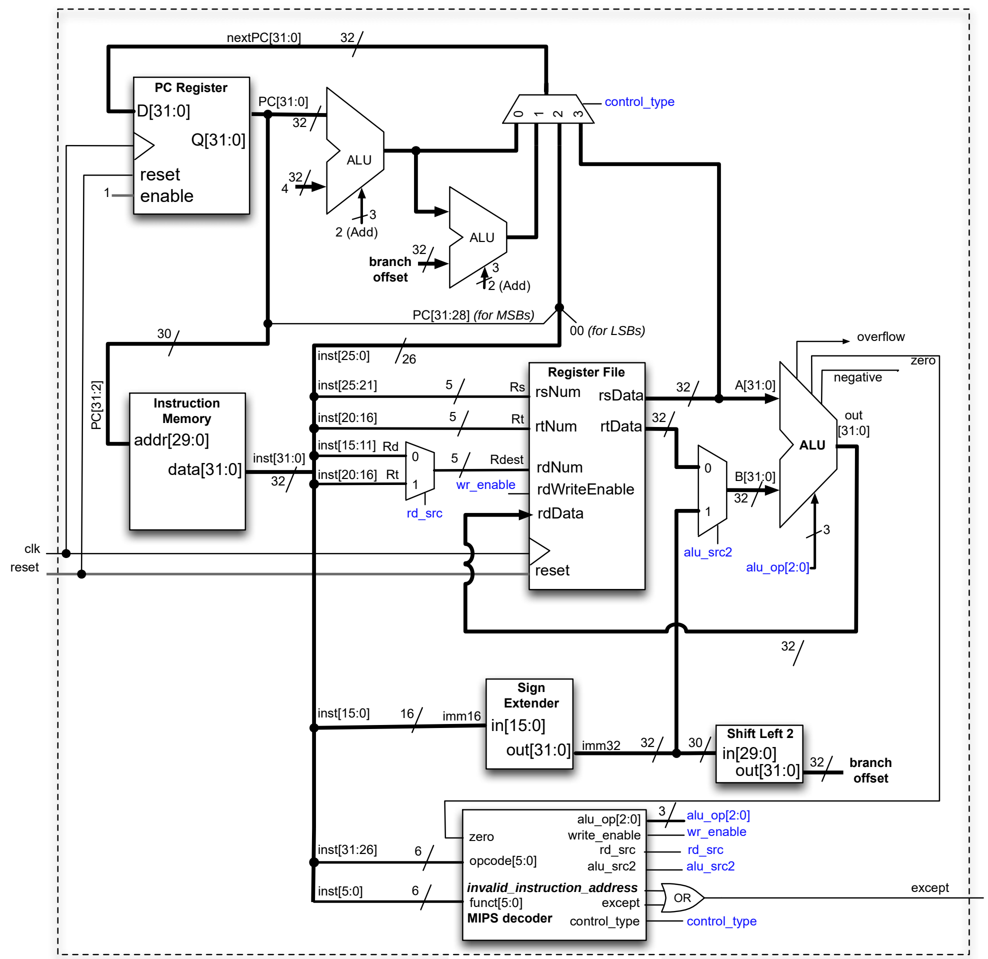
```
slt rd, rs, rt # R[rd] = (R[rs]<R[rt]) ? 1 : 0
```

op	rs	rt	rd	shamt	func

```
slti rt, rs, imm # R[rt] = (R[rs]<imm) ? 1 : 0
```

op	rs	rt	imm

`slt` and `slti` Implemented



Full Machine Datapath (so far)