MIPS control flow instructions: Jumps, Branches, and Loops

BE SURE TO SIGN UP FOR EXAM 2 (only 1 question)

PICK UP HANDOUT, BRING BACK ON FRIDAY

Quick Policy Reminder

Penalty for involvement in Plagiarism

- All parties involved will receive a 0 on that assignment or exam and their final course grade reduced by one letter grade (e.g., A->B, B->C, etc.). A second offense will result in a failing grade for the class.

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

```
0x00400004      mul     $14, $13, $20
                addi    $14, $14, 4
                sub     $15, $14, $15
                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

Repetition via Loops

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

Selective execution via Conditionals

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

How do we implement these in MIPS assembly?

An unconditional jump always transfers control (like a goto statement in C)

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

opcode label

```
j target_label
```

- Example:

```
Loop:   j Loop
```

label

- What does this code do? Infinite

Jumps use the J-type encoding

ump

$26 < 32$

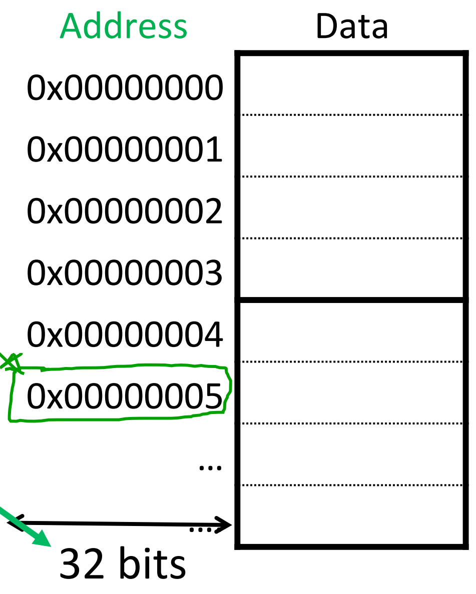
op	address
6 bits	26 bits

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

PC[1:0] = 2'b00;

jump PC[31:28] = PC[31:28]

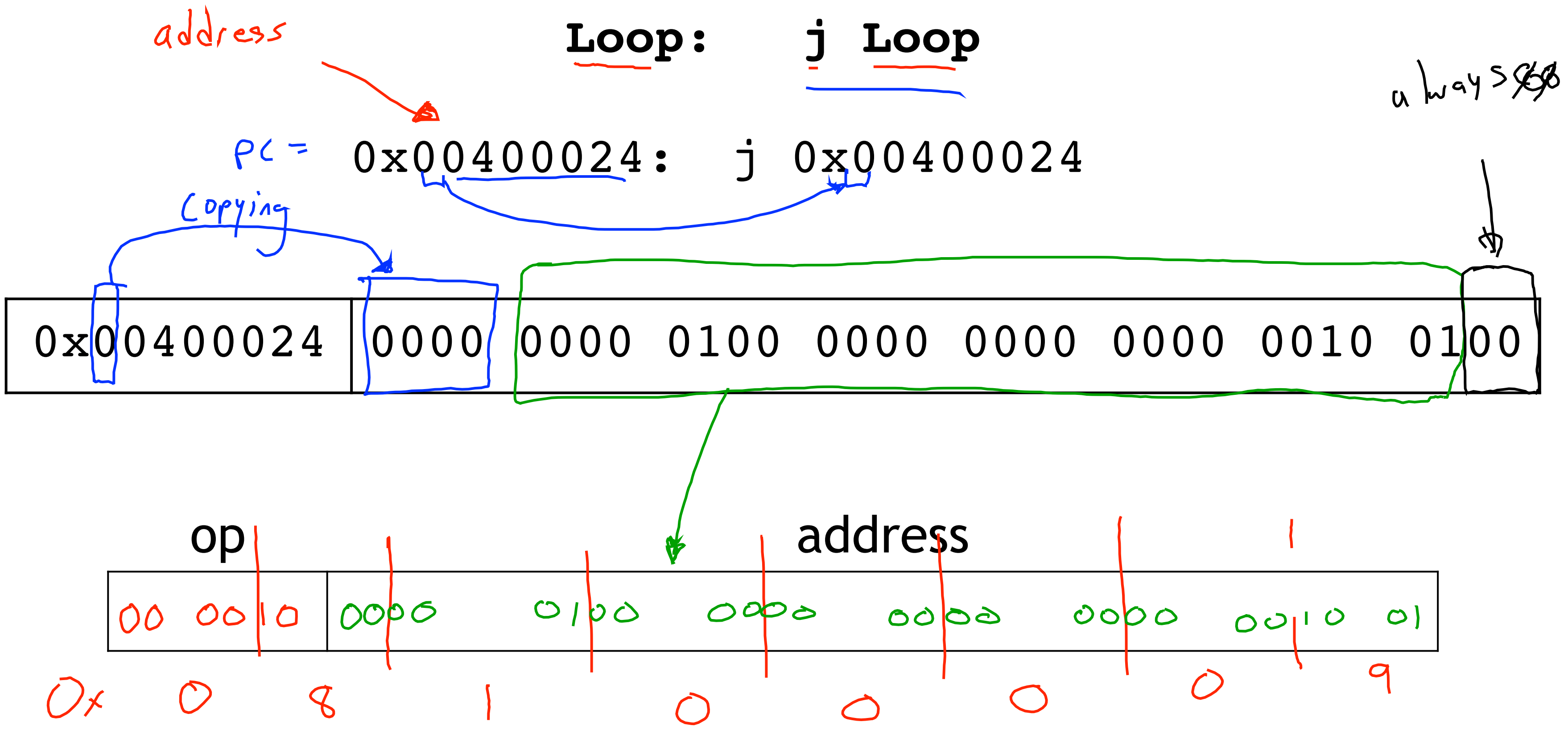
32b



Example encoding: The infinite loop

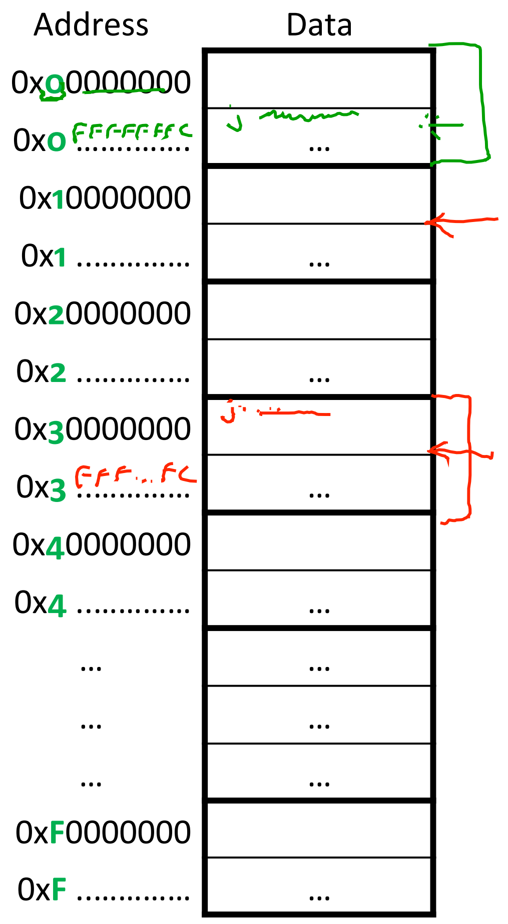
```
Loop:    j Loop

0x00400024:   j 0x00400024
```

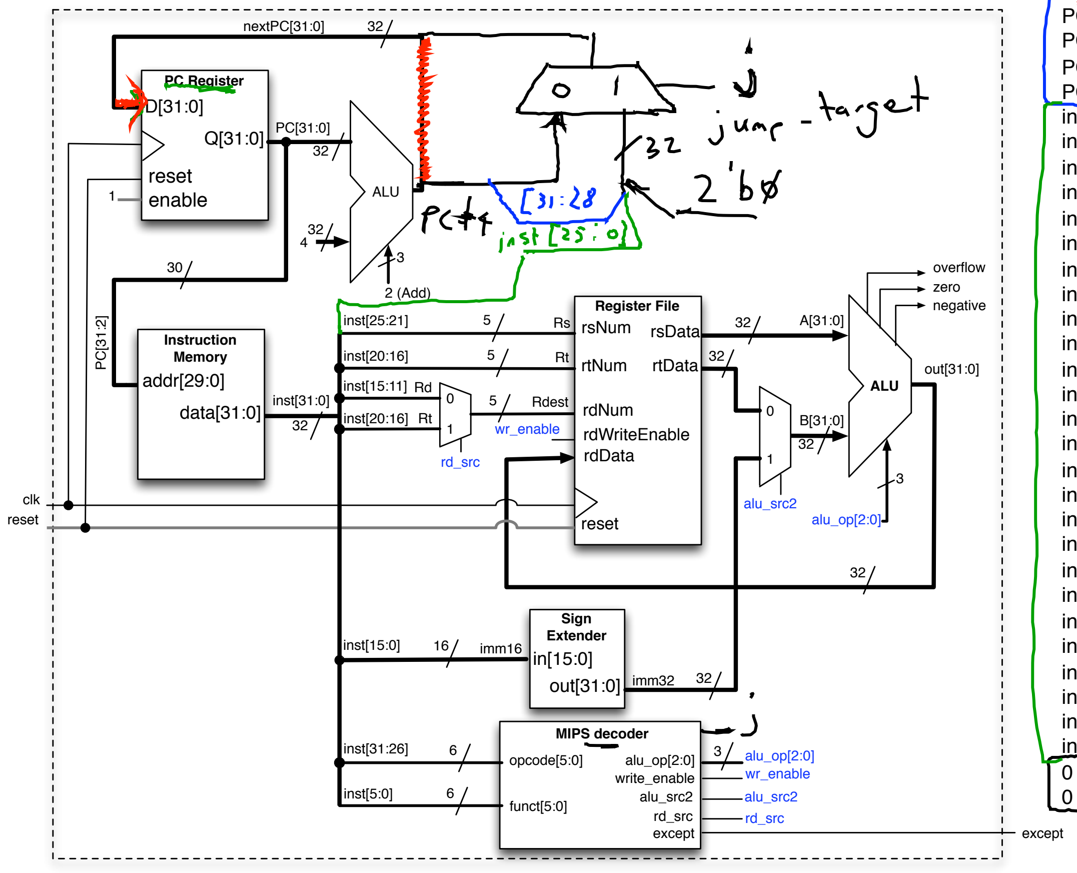


Jump instructions can only move within 1 of 16 regions

- 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]

PC+4

iclicker

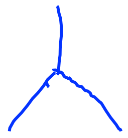
What should wr_enable be?

a) 0

b) 1

c) don't care

Branches provide 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 (PC+4)

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

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

Implement the C code in MIPS assembly

$2

```
int sum = 0;
int i = 0;
```

$3

$2 $3

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

$10

iclicker

A) eq
B) ne

Assembly:

```
add  $10, $0, 10
add  $2, $0, $0  # sum=0
add  $3, $0, 0   # i=0
do:
add  $2, $2, $3
add  $3, $3, 1   # i++
bne  $3, $10, do
```

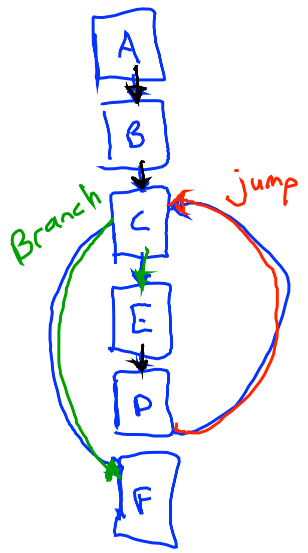
```
addi $7, $0, 0xdeadbeef
R[7] = 0xbeef
add  $7, $0, 0xdeadbeef
R[7] = 0xdeadbeef
```

Implement the C code in MIPS assembly

```
           $2
A
int sum = 0;
                     $3                       $4
     B                         C            D
for (int i = 0 ; i != x ; i ++) {
  E
  sum += i;

}
F
```

```
Assembly:      add  $2, $0, $0     ]A
               add  $3, $0, $0     ]B
     loop:     beq  $3, $4, done   ]C
               add  $2, $2, $3     ]E
               add  $3, $3, 1      ]D
               j    loop
     done:                         ]F
```

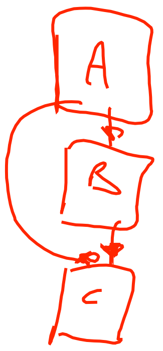


A) eq
B) ne

Implement the C code in MIPS assembly

$2

```
if (x == 0) {     A
    x = 1;        B
}
                  C
```



Assembly:

```
bne   $2, $0, skip
add   $2, $0, 1     # x=1
skip:
```

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

iclicker

A) eq
B) ne

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

address << 2

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

```
0x1004000   beq $1, $0, L
4           add $1, $3, $0
8           add $2, $3, $3
c           j   Somewhere
0x1004010   L: add $2, $3, $3
```

things skipped

# of instructs to skip

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

1004000
+4
+c
1004010

iclicker

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

imm

op	rs	rt	address			
00 0100	000 01	00000	0000	0000	0000	0011

srcs

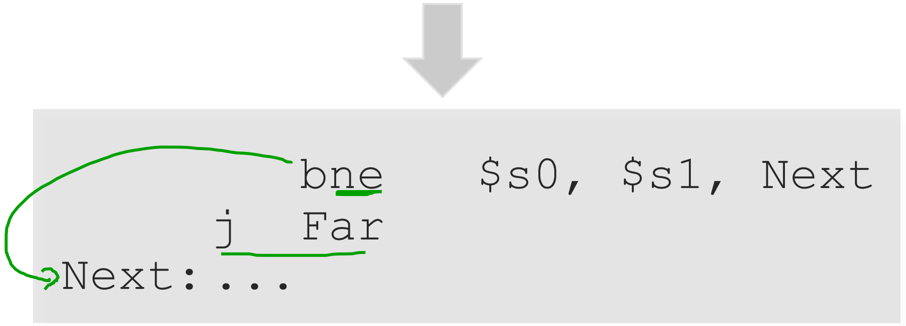
16 -2^{15} .. $+2^{15}$

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
...
```

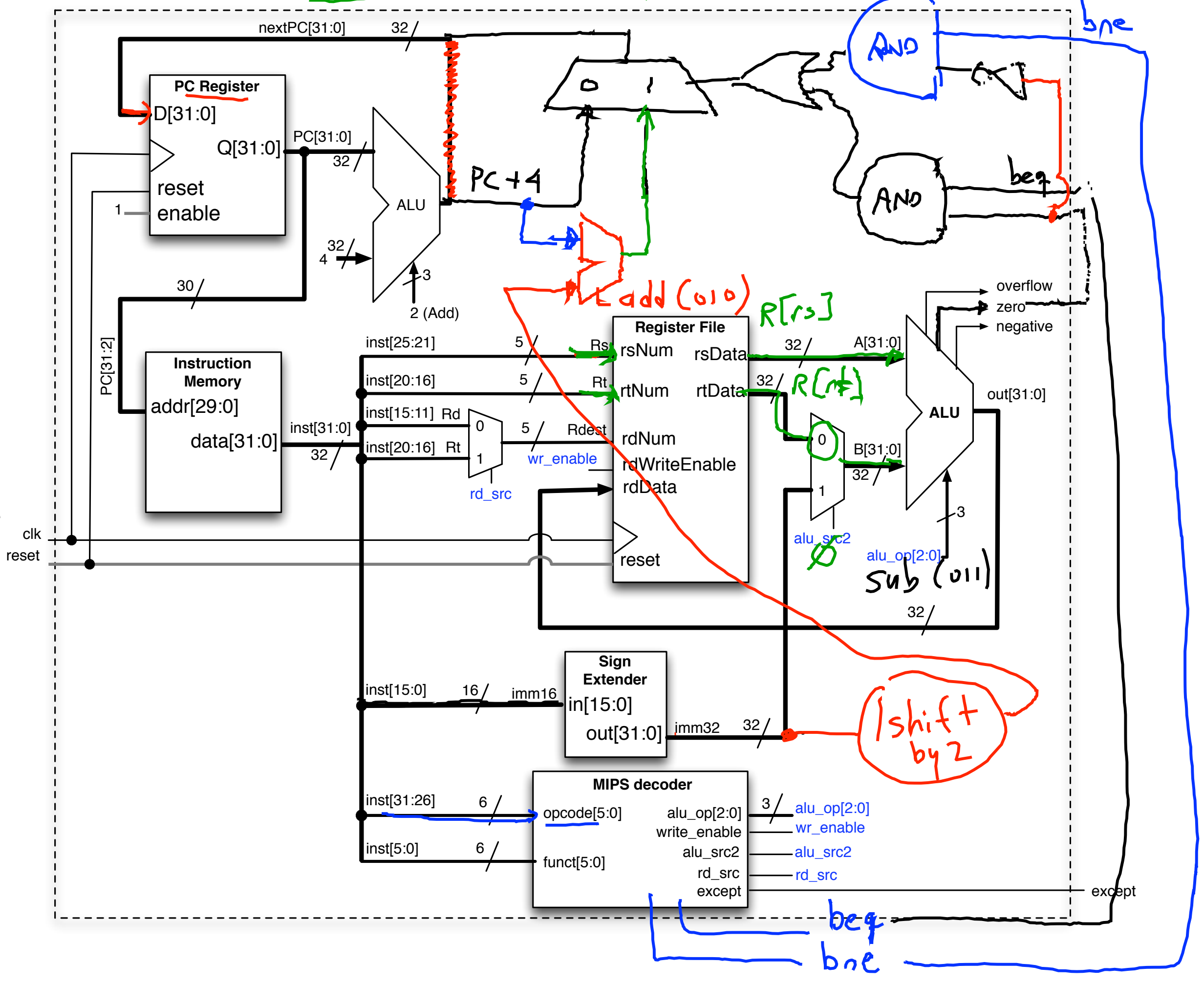
```
         bne    $s0, $s1, Next
      j   Far
Next: ...
```



Implement Branches (w/o jumps)

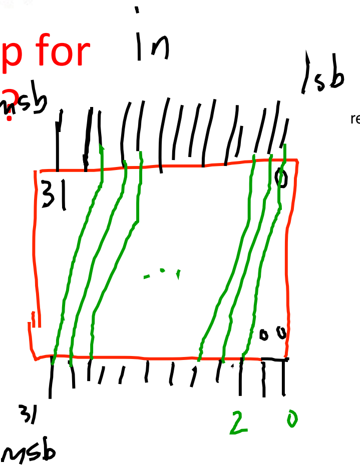
beq
bne

nextPC = PC + 4 + imm * 4



iclicker.

Which alu_op for beq and bne?



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

Use Jump Register (JR) to jump beyond 256MB

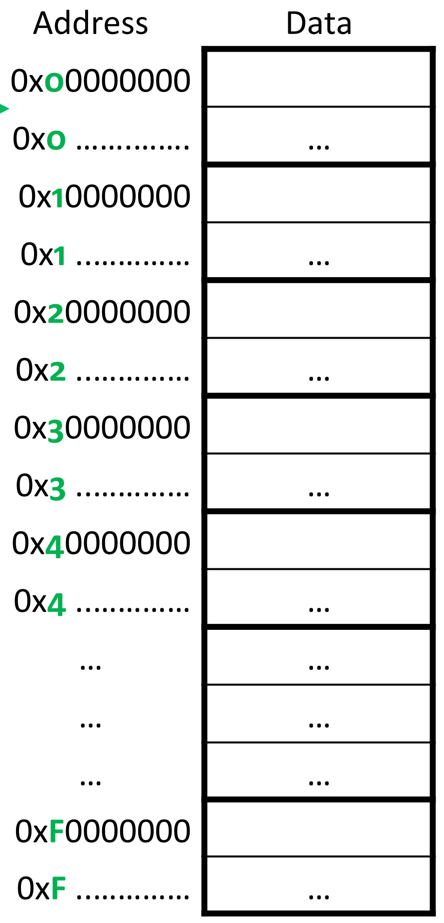
```
jr  rs
```

jr $2

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

- rs acts as a pointer to a pointer

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



iclicker

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

1 reg spec

`jr $rs`

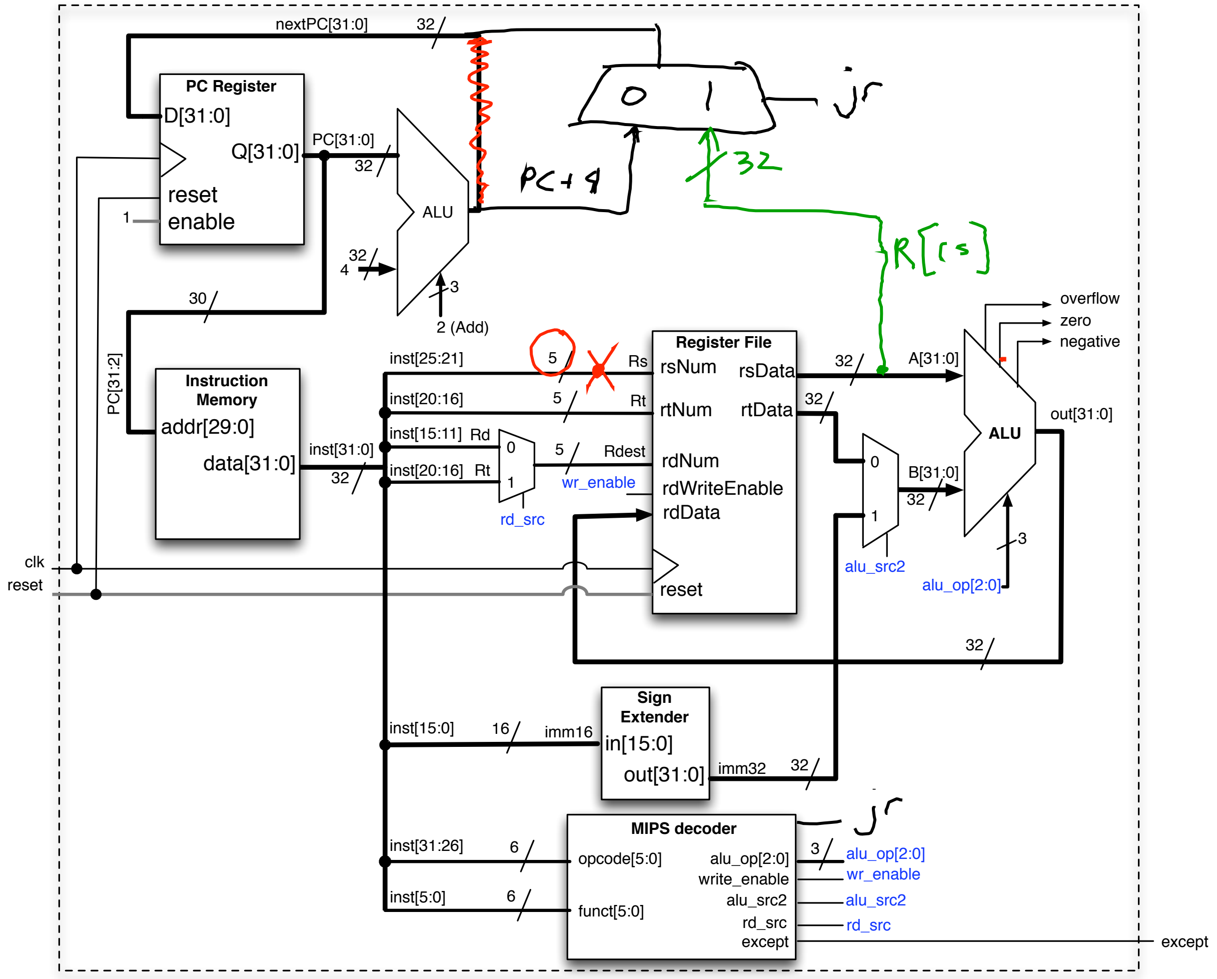
op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Example:

`jr $3`

00 0000	000 11	00000	00000	00000	00 1000

Implementing Jump Register



Control Implemented

iclicker

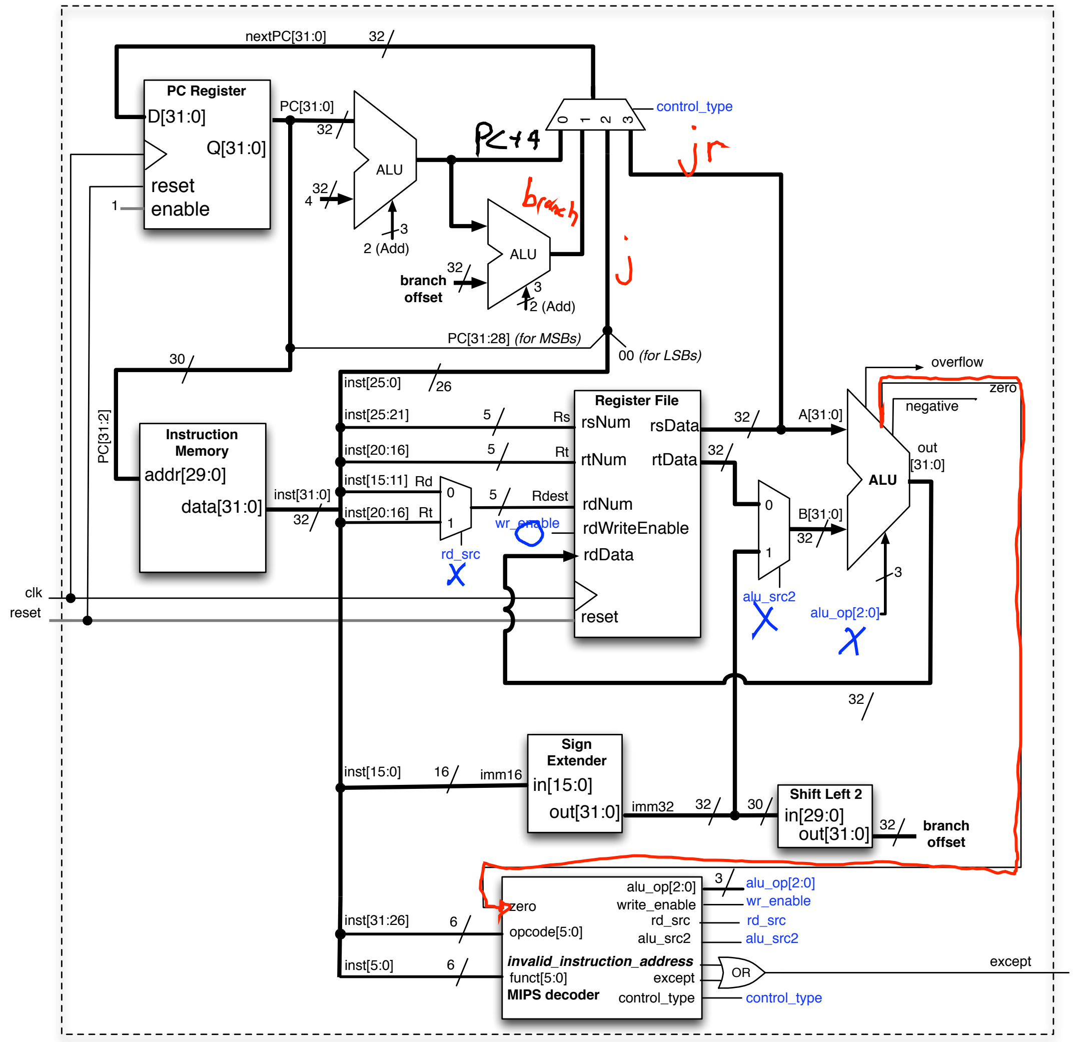
Which type of branch is taken when control_type = 10

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 `lui`, `ori` sequence to construct 32-bit addresses

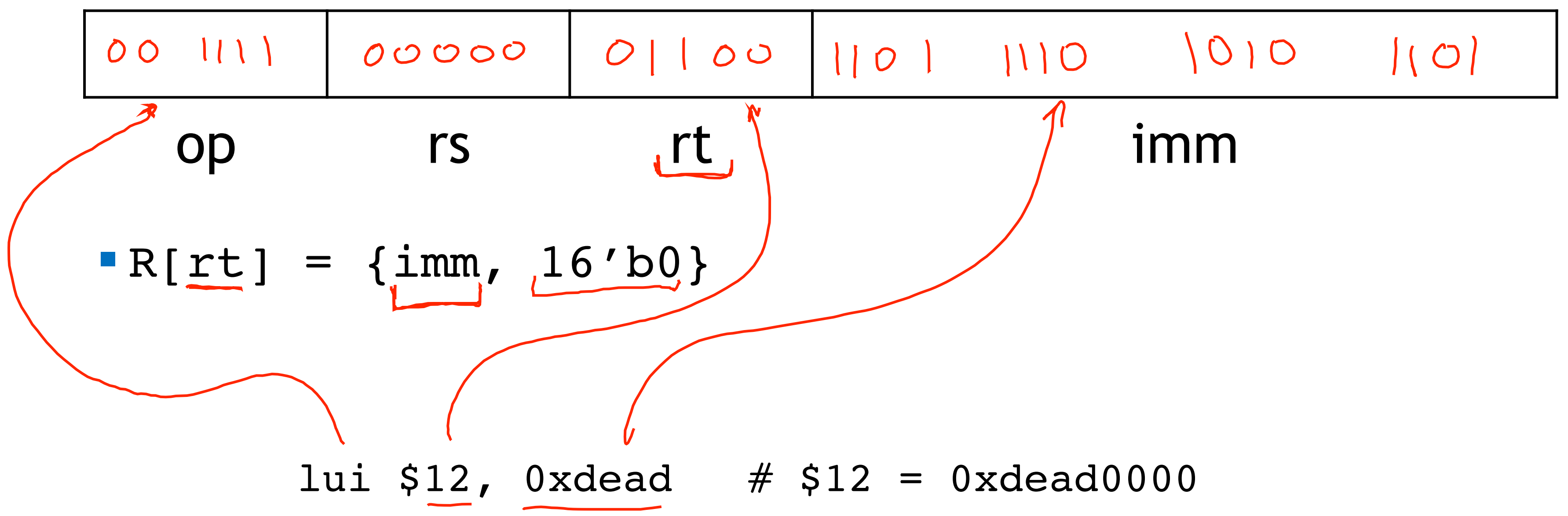
- **`ori`** can set the lower 16 bits

```
    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
```

`lui` is an I-type instruction



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

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

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

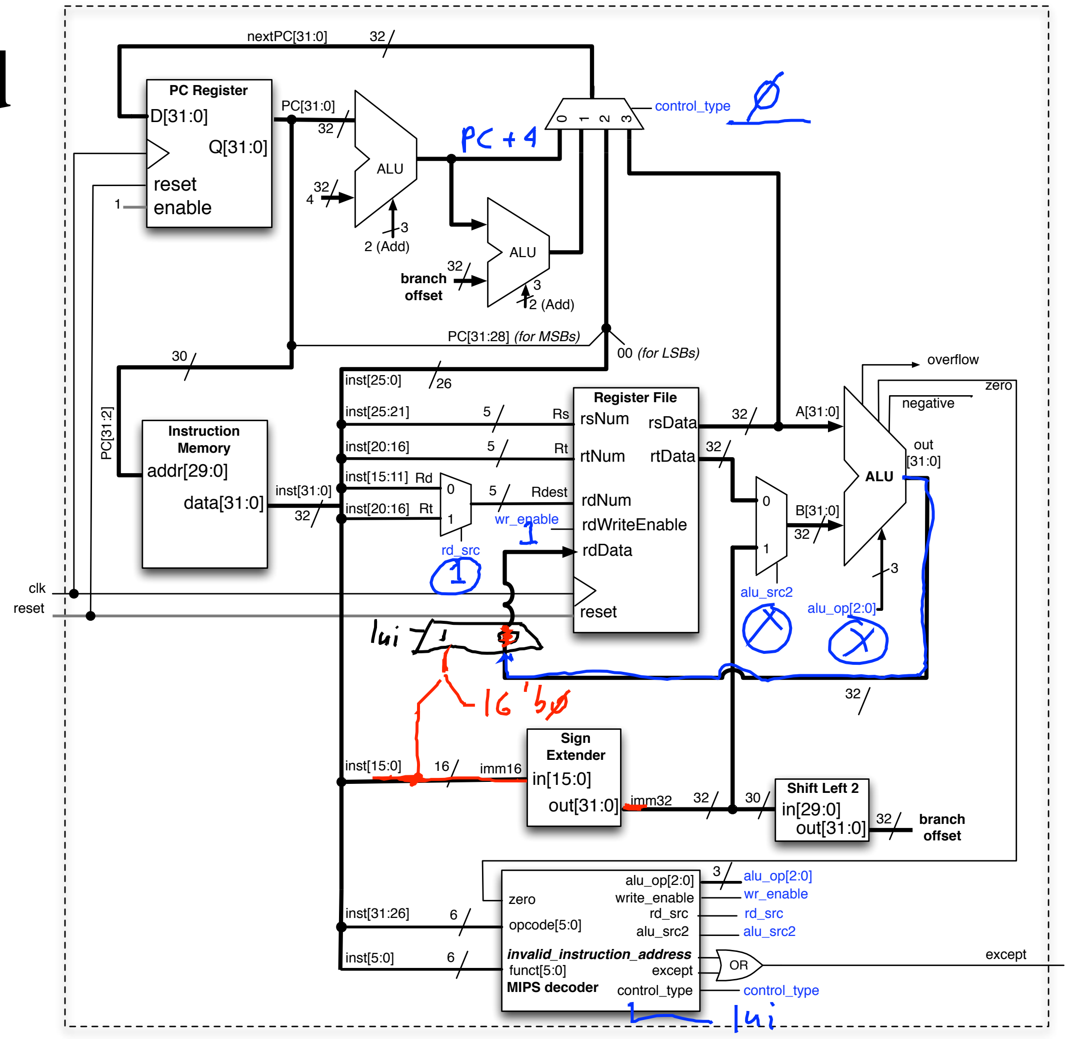
✗
```
ori $12, $12, 0x900
lui $12, 0x3D
```

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

x = -2 $2

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

Assembly:

dest   1 if x<0
-    -2

```
slt   $3, $2, $0     src
beq   $3, $0, skip
→ sub   $2, $0, $2
skip:
```

BNE
BEQ

iclicker

A) eq
B) ne

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

32'b1
32'bø

expression true false

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

r type

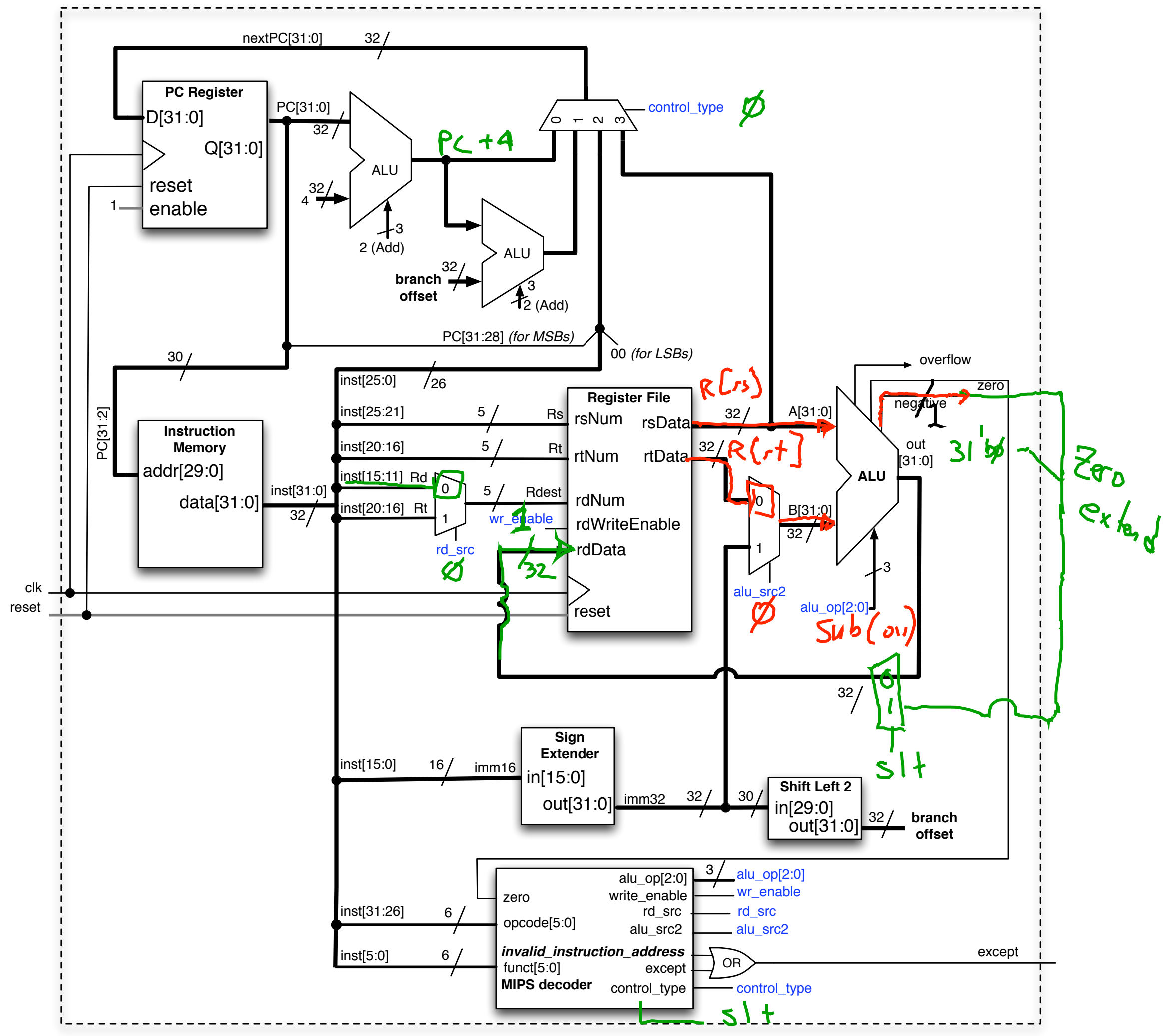
op	rs	rt	rd	shamt	func

src

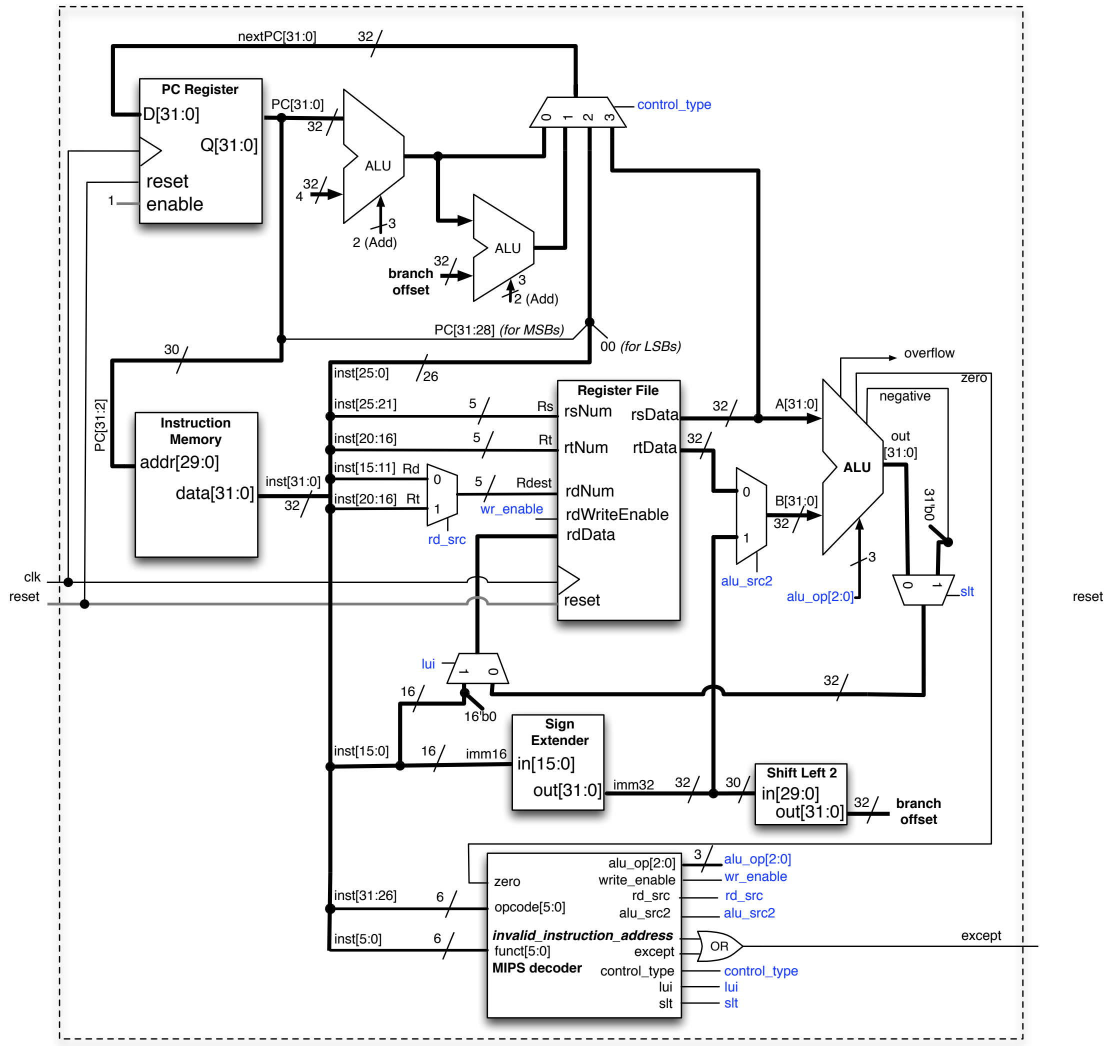
```
slti rt, rs, imm # R[rt] = (R[rs] < imm) ? 1 : 0
```

op	rs	rt	imm

`slt` and `slti` Implemented



Full Machine Datapath (so far)



nextPC[31:0]
32
PC Register
D[31:0]
Q[31:0]
reset
enable
1
PC[31:0]
32
ALU
32
4
3
2 (Add)
0 1 2 3
control_type
ALU
32
branch offset
3
2 (Add)
PC[31:28] (for MSBs)
00 (for LSBs)
30
PC[31:2]
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
32
inst[25:0]
26
inst[25:21]
5
Rs
inst[20:16]
5
Rt
inst[15:11] Rd
inst[20:16] Rt
0
1
rd_src
5
Rdest
wr_enable
Register File
rsNum
rsData
rtNum
rtData
rdNum
rdWriteEnable
rdData
reset
32
32
A[31:0]
0
1
B[31:0]
32
alu_src2
ALU
3
alu_op[2:0]
overflow
zero
negative
out [31:0]
31'b0
0 1
slt
clk
reset
reset
lui
1 0
16
16'b0
32
inst[15:0]
16
imm16
Sign Extender
in[15:0]
out[31:0]
imm32
32
30
Shift Left 2
in[29:0]
out[31:0]
32
branch offset
inst[31:26]
6
inst[5:0]
6
zero
opcode[5:0]
invalid_instruction_address
funct[5:0]
MIPS decoder
alu_op[2:0]
write_enable
rd_src
alu_src2
except
control_type
lui
slt
3
alu_op[2:0]
wr_enable
rd_src
alu_src2
OR
except
control_type
lui
slt