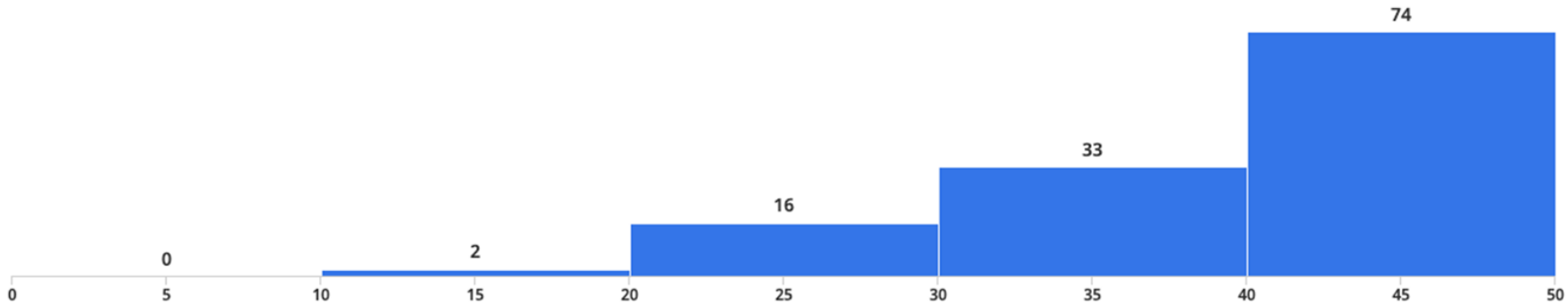
Today’s ICA

- Send the pictures to Gradescope at the end of the lecture

Help me improve the assignments

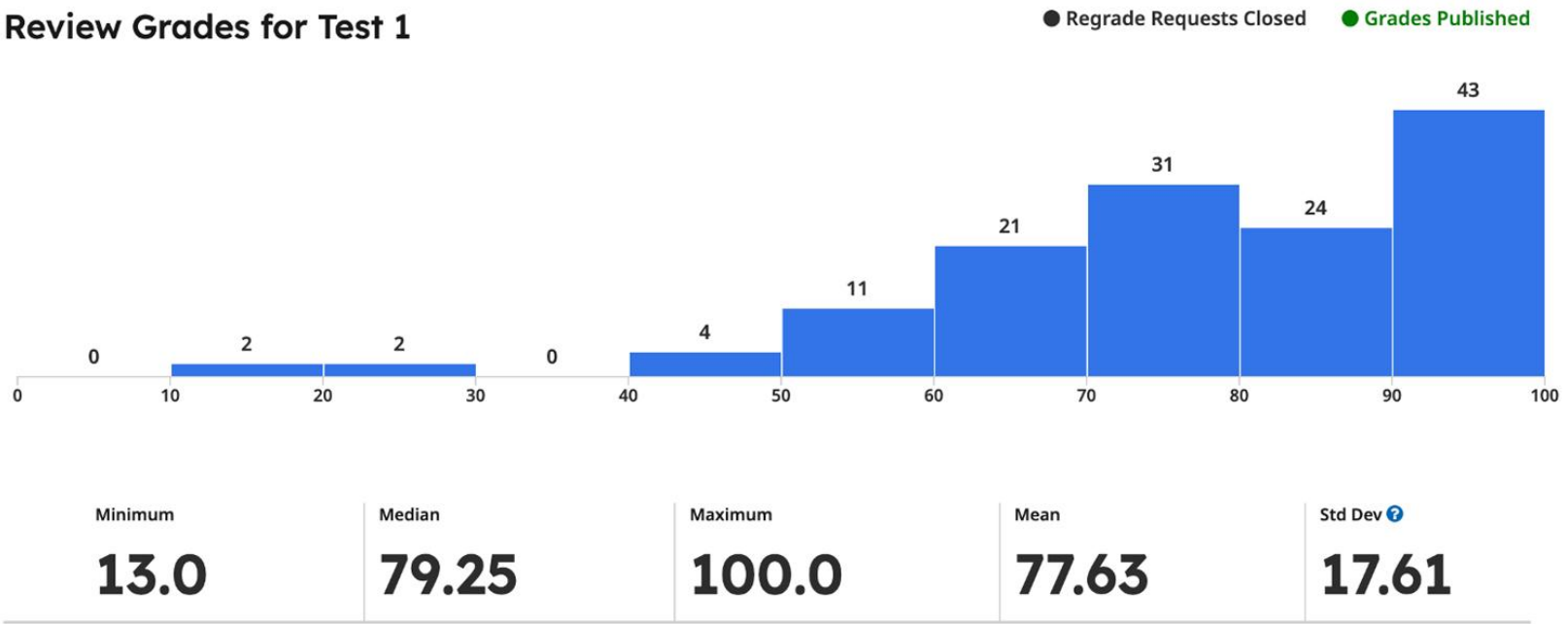
You can add comments to the class materials for extra credit

Test 1 - Grades Distribution



Minimum	Median	Maximum	Mean	Std Dev
13.0	41.75	51.0	40.0	8.59

Grades - Last semester (Fall 2024) - (they have 25 minutes)



More MIPS

Loops, Arrays, and Bit Shifting

- More complex branches
- Loops
- Arrays
- Bit shifting
- Multiplication & Division using bit shifting
- Special: The *real* mult/div instructions

More Complex Branches

- How to implement arbitrary conditional branches?

```
if (x == y && a < b)
{
    ...
}
```

More Complex Branches

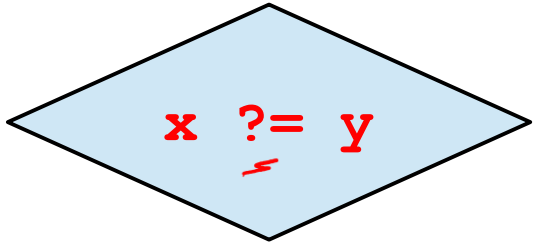
- How to implement arbitrary conditional branches?

Key Idea:

“What do I know so far?”

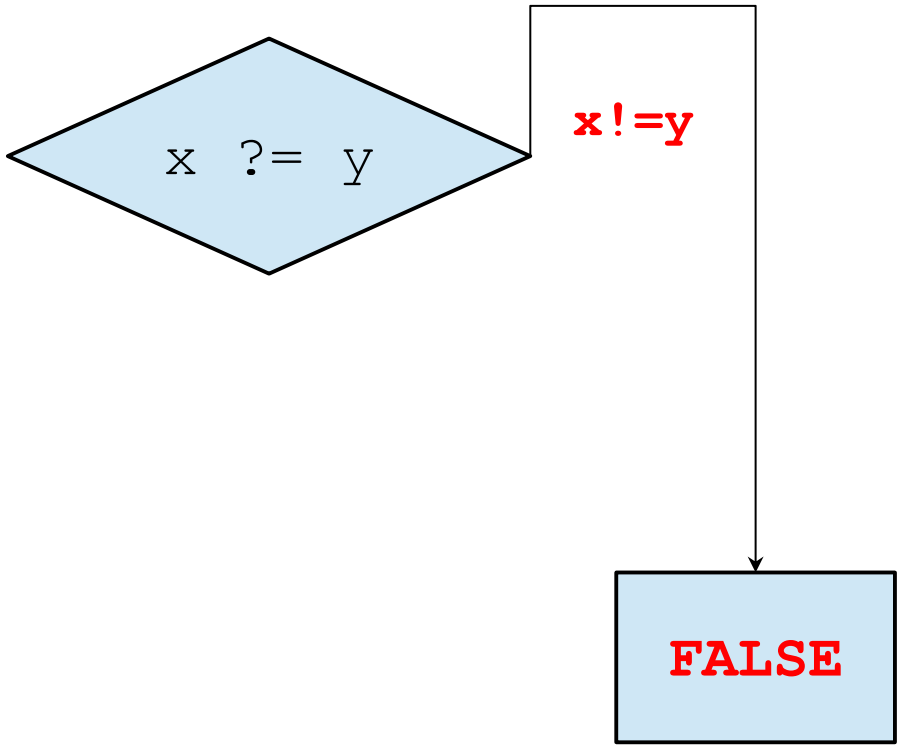
- Perform a branch when you know the answer
- Skip unnecessary tests (short-circuiting)

```
if (x == y && a < b)
{
    ...
}
```



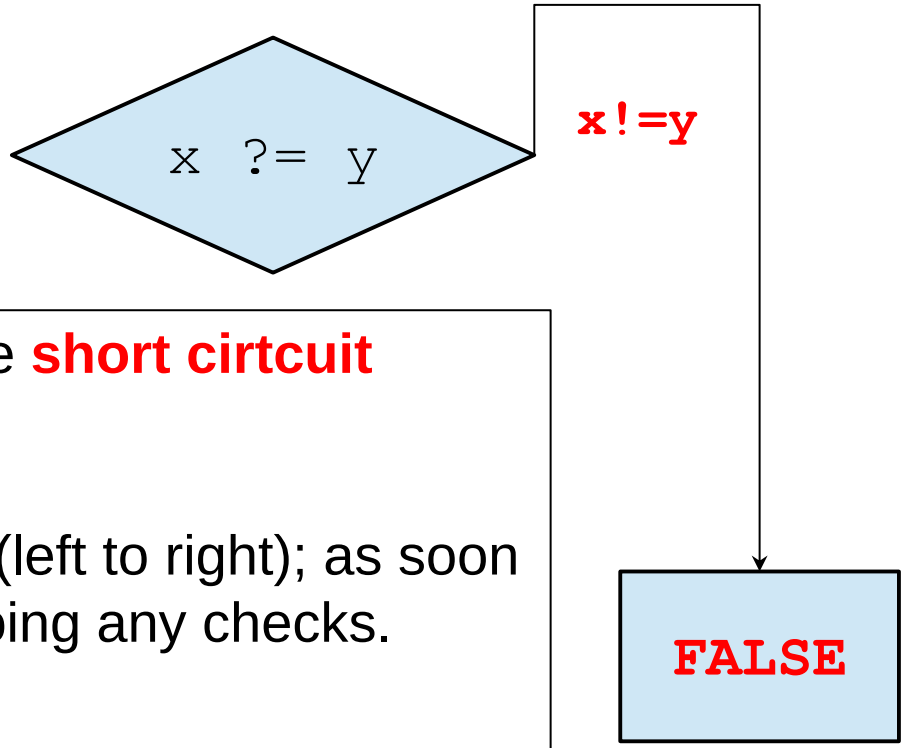
- Perform one check at a time

```
if (x == y && a < b)
{
    ...
}
```



- At each check, one of the two branches (usually) gives you a definite answer.

```
if (x == y && a < b)
{
    ...
}
```

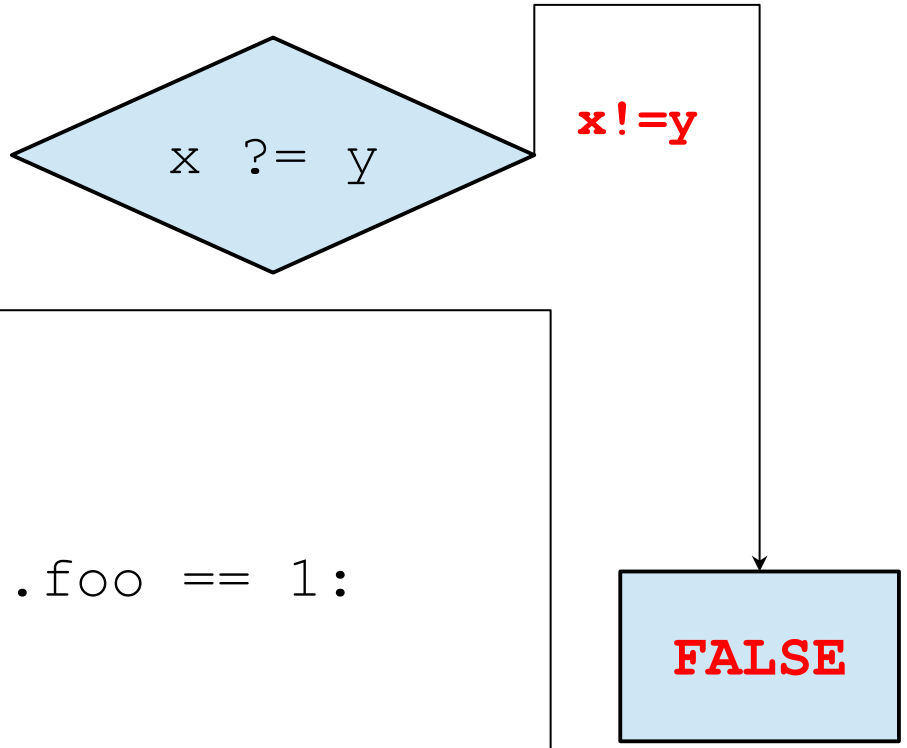


Programming languages typically use **short cirtcuit evaluation.**

You evaluate the conditions in order (left to right); as soon as you know the answer, you stop doing any checks.

Group Question:
Can you think of a reason why this is necessary for correctness?

```
if (x == y && a < b)
{
    ...
}
```

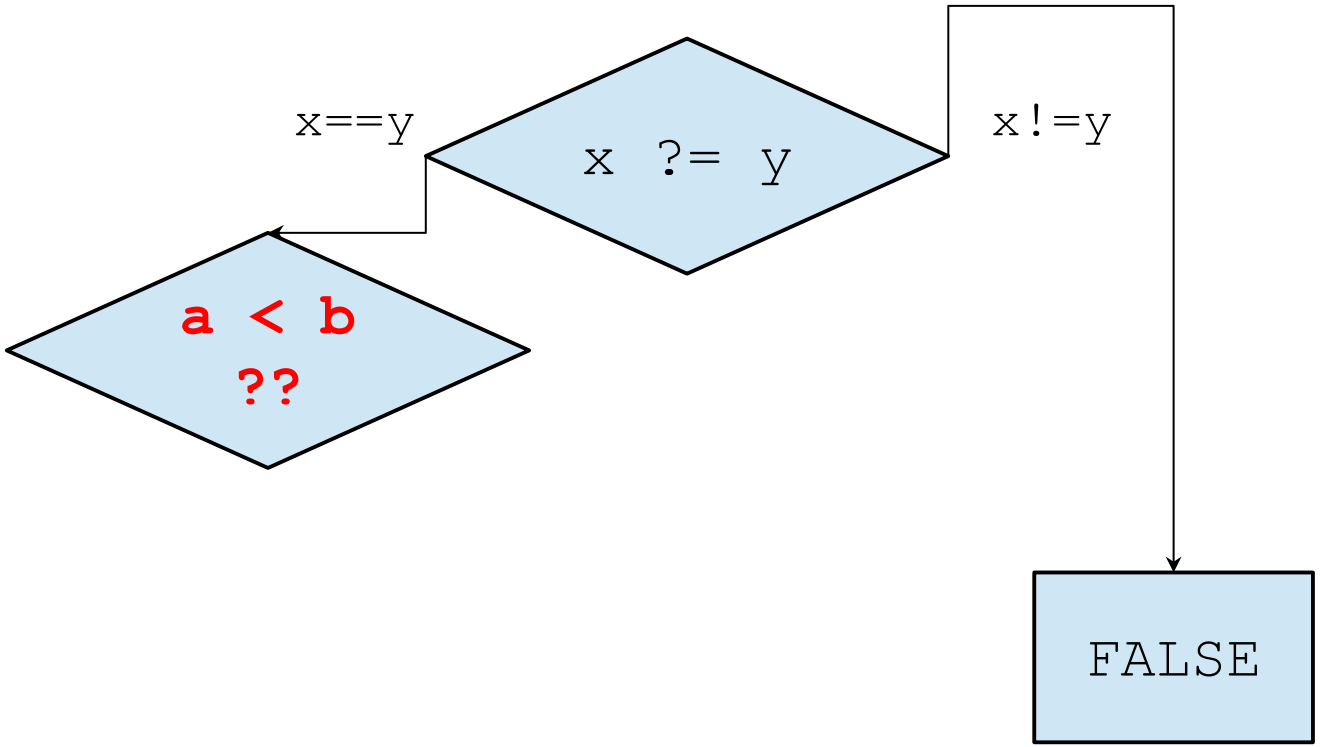


Consider this Python code...

```
obj = ... ;
if obj is not None and obj.foo == 1:
    obj.bar += 1
```

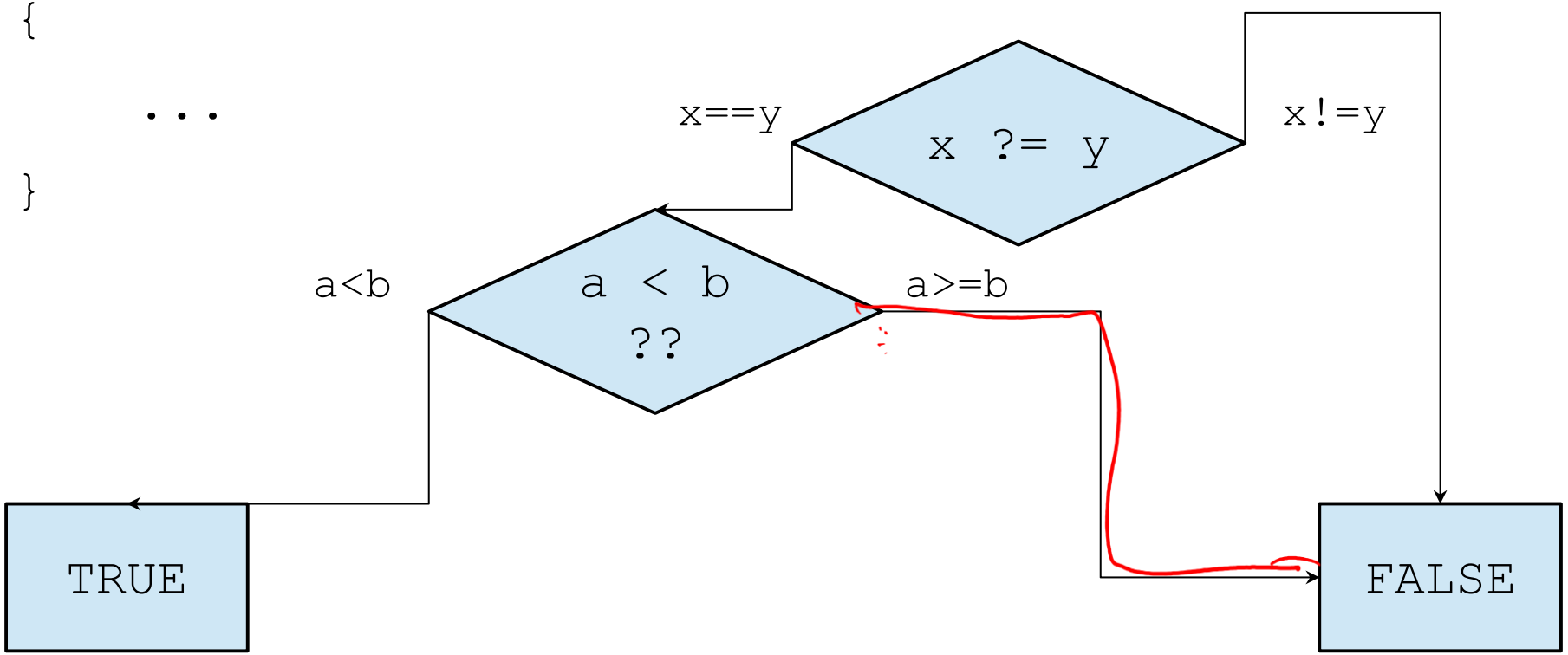
What would happen if Python didn't short circuit the `and` operator?

```
if (x == y && a < b)
{
    ...
}
```



- In the still-active chain, check the next condition...

```
if (x == y && a < b)
{
    ...
}
```



- Continue to build conclusions at each step
- Eventually, you'll know the answer in all paths

More Complex Branches

- Flowcharts are nice, but can we convert them into branches?

```
if (x == y && a < b)
{
    ...
}
```

```
if (x == y && a < b)
```

Our code plan:

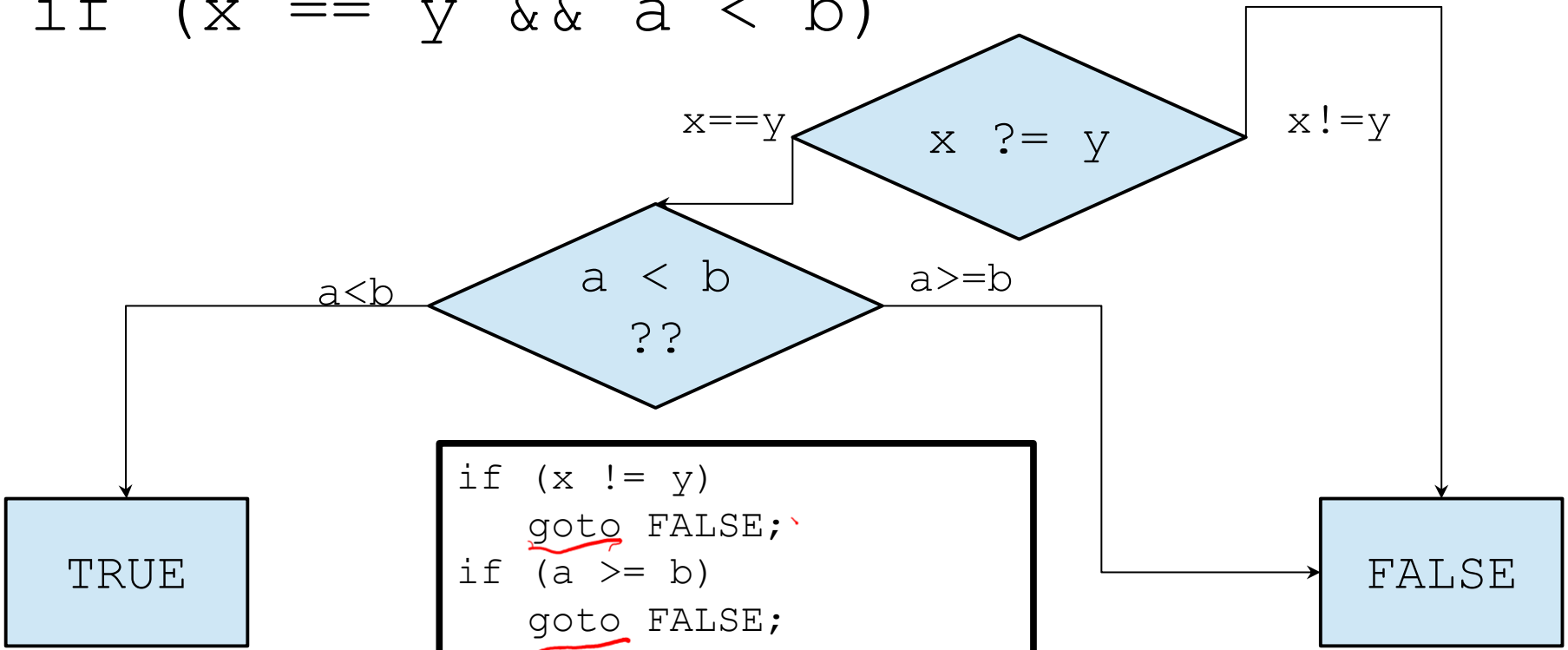
```
compare x,y
    if x != y, then FALSE
compare a,b
    if not(a<b) then FALSE
else
    TRUE
```

More Complex Branches

- Think about the labels you will need
  - Do you ever jump to the `TRUE` case?
  - Do you ever jump to the `FALSE` case?
  - What is your `else` ?

- Add labels for anything you will jump to
- In the `else` case, is the condition true or false?

```
if (x == y && a < b)
```




```
if (x != y)
    goto FALSE;
if (a >= b)
    goto FALSE;

...TRUE code...

FALSE:
```

```
if (x != y)
    goto FALSE;
if (a >= b)
    goto FALSE;
...
FALSE:
```

Group Exercise:

Convert this C code to assembly. The variables are in the following registers:

`x`	`$s0`
`y`	`$s1`
`a`	`$s2`
`b`	`$s3`

Use `tX` registers for all temporaries. Do not modify **ANY** `sX` register.

```
bne $s0,$s1,FALSE    # if (x!=y) goto

slt $t0,$s2,$s3      # t0 = (a < b)
beq $t0,$zero,FALSE  # if (a >= b) goto

...

FALSE:
```

The original code:

```
if (x == y && a < b)
{
    ...
}
```

More Complex Branches

- We've seen how to implement AND (two reasons we might skip the next block)

- How to implement OR (two reasons to go **INTO** the block)?
  - Two ways to jump **into** the block
  - Jump over the block if both fail
  - Need a third (unconditional) branch

```
if (foo >= bar || bar >= baz)
{
    ...
}
```

Question:

How to write this code using `goto`'s ?

```
if (foo >= bar || bar >= baz)
{
    ...
}
```

```
if (foo >= bar || bar >= baz)
    goto GO_IN;
goto AFTER;
GO_IN:
...

AFTER:
```

```
if (foo >= bar || bar >= baz)
    goto GO_IN;
goto AFTER;
GO_IN:
...

AFTER:
```

Group Exercise:

Convert this C code to assembly. The variables are in the following registers:

`foo`	`$s5`
`bar`	`$s6`
`baz`	`$s7`

Use `tX` registers for all temporaries.

```
slt $t0,$s5,$s6      # s0 = (foo < bar)
beq $t0,$zero,GO_IN  # if (foo >= bar) goto

slt $t0,$s6,$s7      # t0 = (bar < baz)
beq $t0,$zero,GO_IN  # if (baz >= baz) goto

j AFTER

GO_IN:
...

AFTER:
```

The original code:

```
if (foo >= bar || bar >= baz)
{
        …
}
```

```
slt $t0,$s5,$s6      # s0 = (foo < bar)
beq $t0,$zero,GO_IN # if (foo >= bar) goto


slt $t0,$s6,$s7      # t0 = (bar < baz)
beq $t0,$zero,GO_IN # if (baz >= baz) goto


j AFTER


GO_IN:
...


AFTER:
```

Why did we use BNE instead of BEQ here?

Loops

Loops in assembly:

- Test the condition (often at top)
  - Jump **out** of loop if fail
- Loop back to top when loop done
- Need two labels (top / done)

Normal C/Java Code:

```
for (int i=0; i<100; i++)
    ...
```

Group Exercise:

Convert this C loop into C code that uses `if` statements with `goto` and labels.

```
if (x == y)
    goto THERE;
```

Normal C/Java Code:

```
for (int i=0; i<100; i++)
```

More Like Assembly:

```
int i=0;

LOOP:
if (i >= 100)
    goto AFTER;
...

i++;
goto LOOP;

AFTER:
```

Normal C/Java Code:

```
for (int i=0; i<100; i++)
    ...
```

More Like Assembly:

```
int i=0;
```

Init

```
LOOP:
if (i >= 100)
    goto AFTER;
```

Test condition;
Leave if false.

```
...
```

Body

```
i++;
goto LOOP;
```

Increment;
Back to top

```
AFTER:
```

End of Loop

```
for (int i=0; i<100; i++)
    ...
```

Group Exercise:

Convert this C code to assembly.

Assign a `tX` register of your choice to hold `i`.

Use `tX` registers for all temporaries.

```
addi $t0, $zero,0       # i = 0

LOOP:
addi $t1, $zero, 100    # move this above???
slt  $t2, $t0, $t1      # t2 = (i < 100)
beq  $t2, $zero, AFTER  # if (i >= 100) break
...

addi $t0, $t0, 1
j LOOP

AFTER:
```

Or:

```
slti $t2, $t0, 100
```

The original code:

```
for (int i=0; i<100; i++)
    ...
```

```
while (x < y)
{
    ...

    if (foo == bar)
        break;

    ...
}
```

Group Exercise:

Convert this C code to assembly.

The variables are in the following registers:

```
x
$s2
y
$s3
foo
$s4
bar
```

```
LOOP:
slt  $t0, $s2, $s3      # t2 = (x < y)
beq  $t0, $zero, AFTER # if (x >= y) break
...
beq  $s4,$s5, AFTER
...
j LOOP


AFTER:
```

The original code:

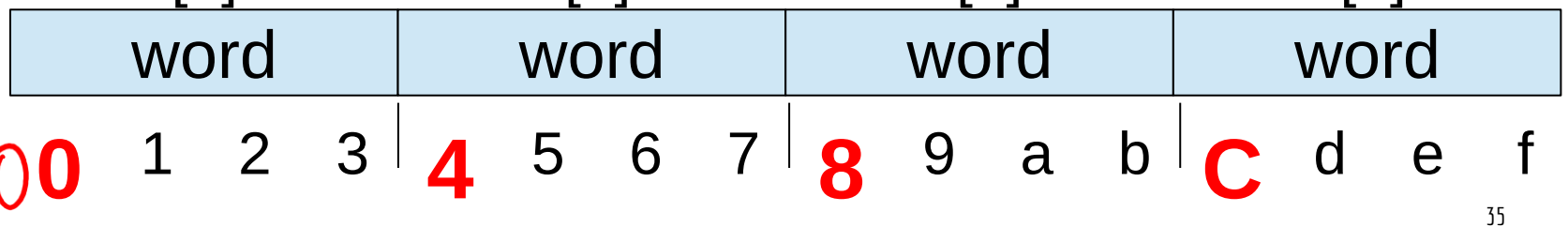
```
while (x < y)
{
    ...

    if (foo == bar)
        break;

    ...
}
```

Loops: Tips and Tricks

- Use same code order as C code
- **Comment heavily**
- Block comments typically **required** for each loop
  - What is the condition?
  - What variables are used?
  - What registers are used, and for what?
- Use blank lines to group related code together

Arrays

- An **array** is several variables of the same type, arranged sequentially in memory



- The address of the array is the address of its first element
- Element `[i]` is at address `base+i*size`

Declaring Arrays in MIPS

- Declare an array with **one label** followed by **many variables**

int x[10];

```
myArray:   .word    0     # [0]
           .word    0     # [1]
           .word    0     # [2]
           .word    0     # [3]
           .word    0     # [4]
           .word    0     # [5]
           .word    0     # [6]
```

Array Sizes

- Arrays work like C, not Java
  - Have we heard this before?

- No built-in length operator
- No out-of-bounds checking

- Often we'll either add a `len` variable or use null terminators

Strings as Arrays

- `.asciiz` also declares an array – of bytes

–Adds automatic null terminator

- `.ascii` (no Z) declares a string, doesn't add the null terminator

```
int myArray[4];

...
int i = ... ;

int x = myArray[0];
int y = myArray[1];
int z = myArray[i];
```

Group Exercise:

Convert this C code to assembly.

Assume that `i` is stored in `$s7`.

Assign `sX` registers for `x,y,z.`

Use `tX` registers for all temporaries.