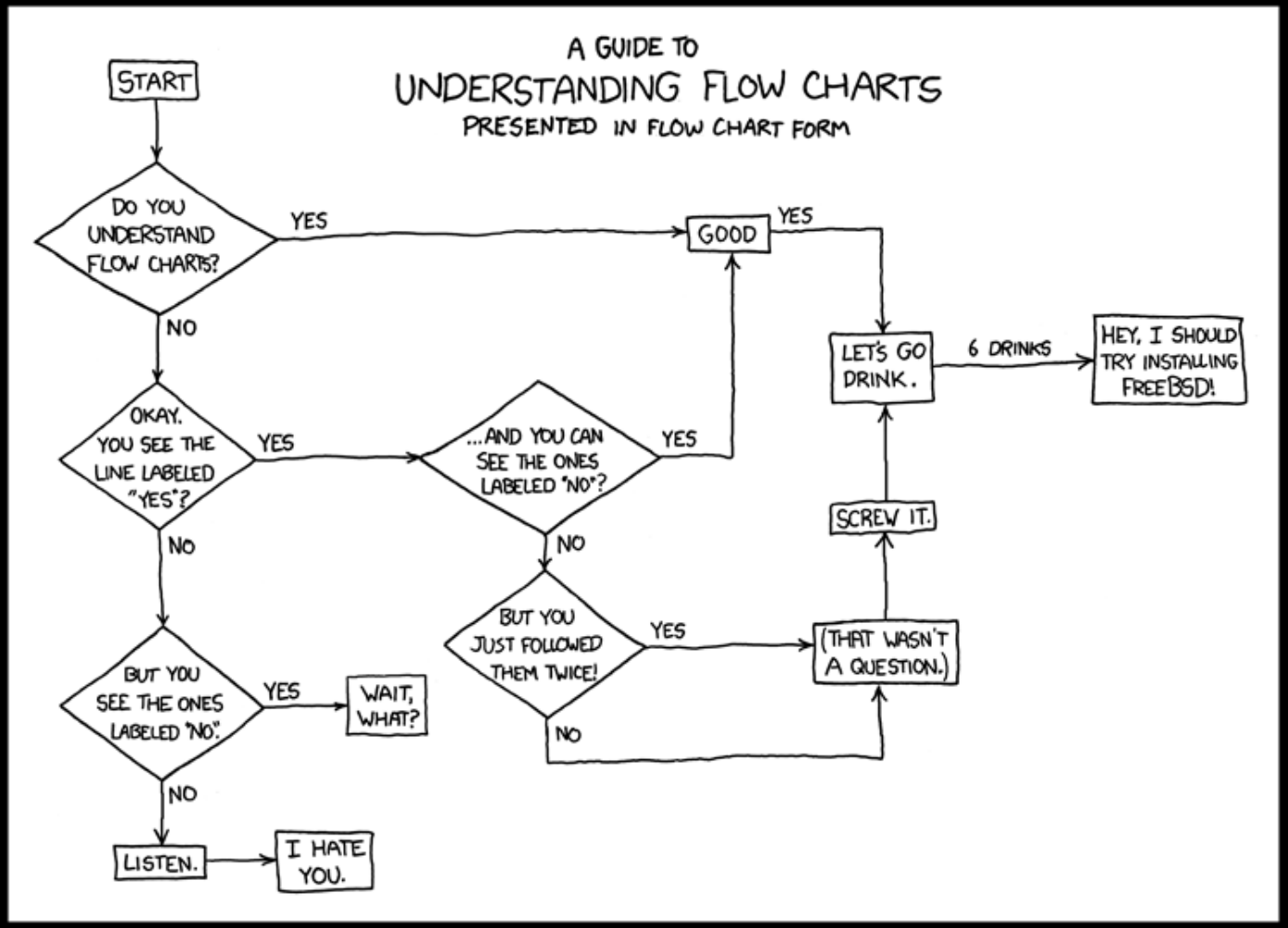
Week 8, part C: Control Flow



Control Flow

- Most programs do not follow a linear set of instructions.
  - Some operations require the code to branch to one section of code or another

```
if (a>b) max = a; else max = b;
```

  - Some require the code to jump back and repeat a section of code again.

```
while (i < j)
        i = i*2;
```

- How do we do this in assembly?

Control Flow in Assembly

- Assembly is not sophisticated.
  - We have to tell it manually where to go and when.
- Labels on the left indicate points that the program flow might need to jump to.
- Branch instructions tell the CPU to go somewhere based on some condition.
  - The assembler computes the offset between the current program counter and the target when assembling the machine code.
- We can also jump unconditionally.

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
beq	000100	$s, $t, label	if ($s == $t) pc ← label
bgtz	000111	$s, label	if ($s > 0) pc ← label
blez	000110	$s, label	if ($s ≤ 0) pc ← label
bne	000101	$s, $t, label	if ($s != $t) pc ← label

- Branch operations are key to implementing if/else statements and loops.
- The labels are memory locations, assigned to each label at compile time.

Branch instructions

- How does a branch instruction work?

```
.text

main:   beq $t0, $t1, end  # check if $t0 == $t1
        ...                # if $t0 != $t1, then
        ...                # execute these lines

 end:   ...                # if $t0 == $t1, then
        ...                # execute these lines
```

Branch instructions

- Alternate implementation using `bne`:

```
.text

main:   bne $t0, $t1, end  # check if $t0 == $t1
        ...                # if $t0 == $t1, then
        ...                # execute these lines

 end:   ...                # if $t0 != $t1, then
        ...                # execute these lines
```

- Used to produce `if` statement behaviour.

Conditional Branch Terms

- When the branch condition is met, we say the branch is taken.
- When the branch condition is not met, we say the branch is not taken.
  - What is the next PC in this case?
    - It's the usual `PC+4`
- How far can a processor branch? Are there any constraints?

If/Else statements in MIPS

```
if ( i == j )
   i++;
else
   j--;
j = j+i;
```

- Strategy for if/else statements:
  - Test condition, and jump to `if` logic block whenever condition is true.
  - Otherwise, perform `else` logic block, and jump to first line after `if` logic block.

Translated if/else statements

```
#  $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF     # branch if ( i == j )
         addi $t2, $t2, -1     # j--
         j END                 # jump over IF
IF:      addi $t1, $t1, 1      # i++
END:     add $t2, $t2, $t1     # j += i
```

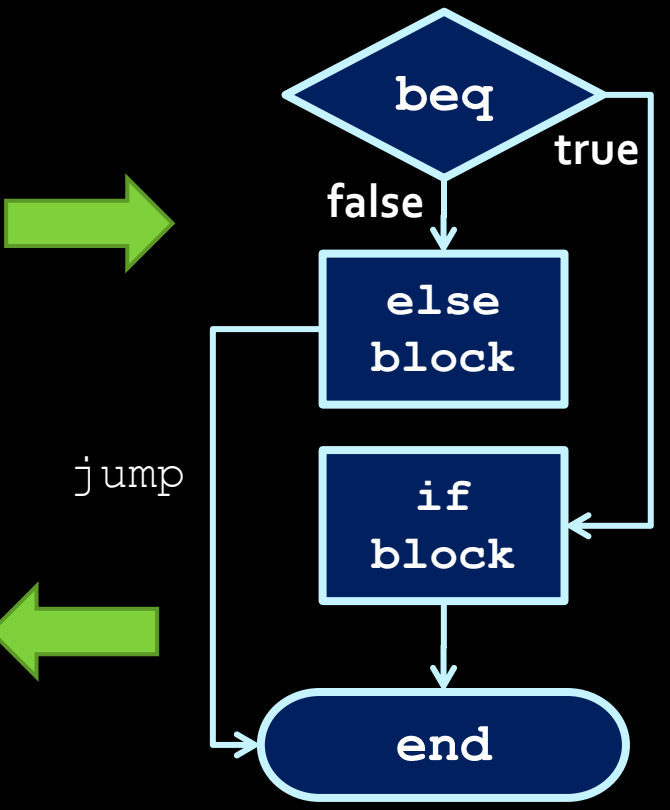
- Alternately, you can branch on the else condition first:

```
#  $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE   # branch if ( i != j )
         addi $t1, $t1, 1      # i++
         j END                 # jump over ELSE
ELSE:    addi $t2, $t2, -1     # j--
END:     add $t2, $t2, $t1     # j += i
```

A trick with if statements

- Use flow charts to help you sort out the control flow of the code:

```
if ( i == j )
  i++;
else
  j--;
j += i;
```



jump

```
#   $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF
         addi $t2, $t2, -1
         j END
IF:      addi $t1, $t1, 1
END:     add $t2, $t2, $t1
```

Multiple Conditions Inside If

```
if ( i == j && i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

Multiple Conditions Inside If

```
if ( i == j && i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

- Branch for each if

```
#  $t1 = i, $t2 = j, $t3 = k
main:  bne $t1, $t2, ELSE    # cond1: branch if ( i != j )
       bne $t1, $t3, ELSE    # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1      # if (i==j|i==k) --> i++
       j END                 # jump over else
ELSE:  addi $t2, $t2, -1     # else-body: j--
END:   add $t2, $t1, $t3     # j = i + k
```

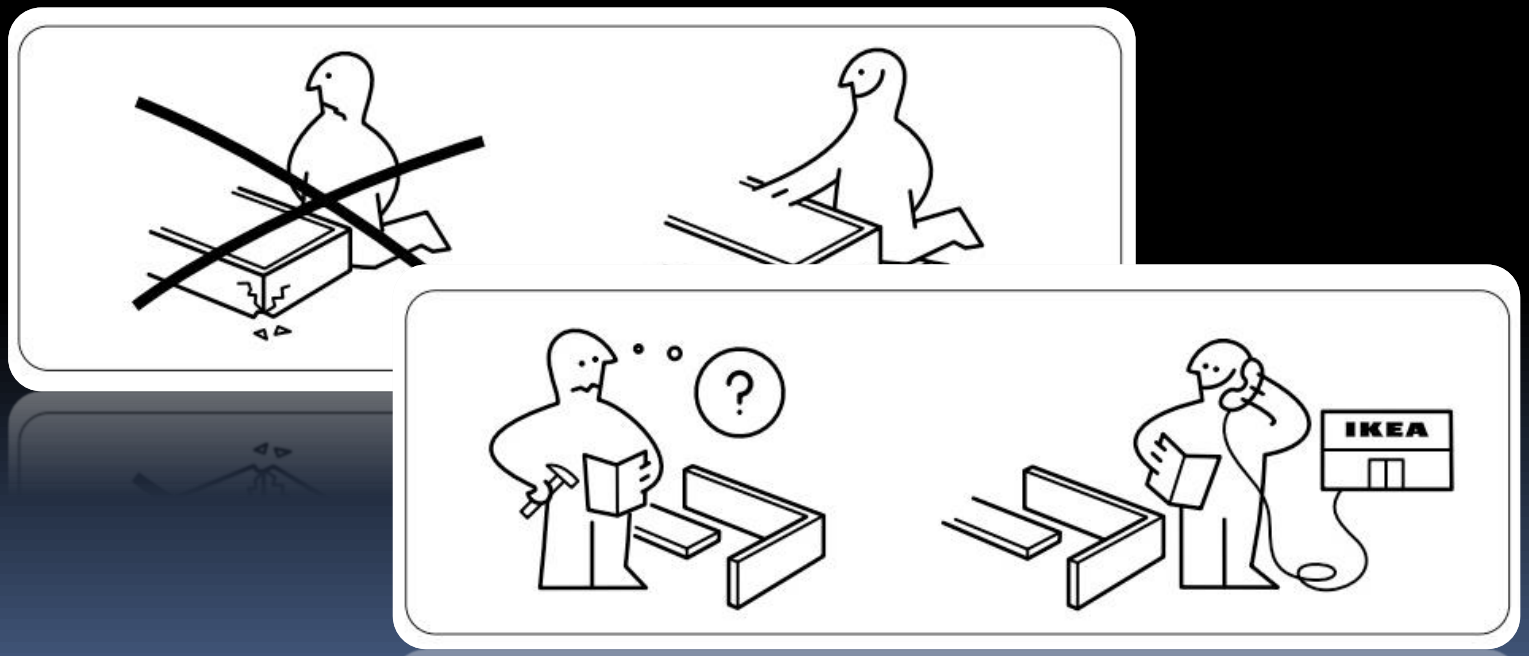
Multiple Conditions Inside If

```
if ( i == j || i == k )
   i++ ;  // if-body
else
   j-- ;  // else-body
j = i + k ;
```

- For OR condition, tweak the logic:

```
#  $t1 = i, $t2 = j, $t3 = k
main:  beq $t1, $t2, IF      # cond1: branch if ( i == j )
       bne $t1, $t3, ELSE    # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1      # if (i==j|i==k) → i++
       j END                 # jump over else
ELSE:  addi $t2, $t2, -1     # else-body: j--
END:   add $t2, $t1, $t3     # j = i + k
```

Time for more instructions!



Jump instructions

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	label	pc ← label
`jal`	000011	label	$ra = pc; pc ← label
`jalr`	001001	$s	$ra = pc; pc = $s
`jr`	001000	$s	pc = $s

- `jal` = "jump and link".
  - Register $31 (aka $ra) stores the address that's used when returning from a subroutine.
- Note: `jr` and `jalr` are not j-type instructions.
  - We can tell because they have $s

Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
`slt`	101010	$d, $s, $t	$d = ($s < $t)
`sltu`	101001	$d, $s, $t	$d = ($s < $t)
`slti`	001010	$t, $s, i	$t = ($s < SE(i))
`sltiu`	001001	$t, $s, i	$t = ($s < SE(i))

- "slt" = "Set Less Than"
- Comparison operation stores one (1) in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.
- Signed: `0x8000000` is less than all numbers
- Unsigned: `0 - 0x7FFFFFFF` are less than `0x8000000`
  - Note immediate is sign-extended even in `sltiu`

Branch Pseudoinstructions

- Implemented using `slt` variants and branches.
- You are allowed to use them unless we say otherwise.

Instruction	Opcode/Function	Syntax	Operation
blt	N/A	$s, $t, label	if ($s < $t) pc ← label
bltu	N/A	$s, $t, label	if ($s < $t) pc ← label
bgt	N/A	$s, $t, label	if ($s > $t) pc ← label
bgtu	N/A	$s, $t, label	if ($s > $t) pc ← label
ble	N/A	$s, $t, label	if ($s ≤ $t) pc ← label
bleu	N/A	$s, $t, label	if ($s ≤ $t) pc ← label
bge	N/A	$s, $t, label	if ($s ≥ $t) pc ← label
bgeu	N/A	$s, $t, label	if ($s ≥ $t) pc ← label

What is This Code?

```
main:    add $t0, $zero, $zero
         addi $t1, $zero, 100
START:   beq $t0, $t1, END
         addi $t0, $t0, 1
         j START
END:
```

- Take a moment to think about it:
  - You can analyze this code…
- Then move to next part