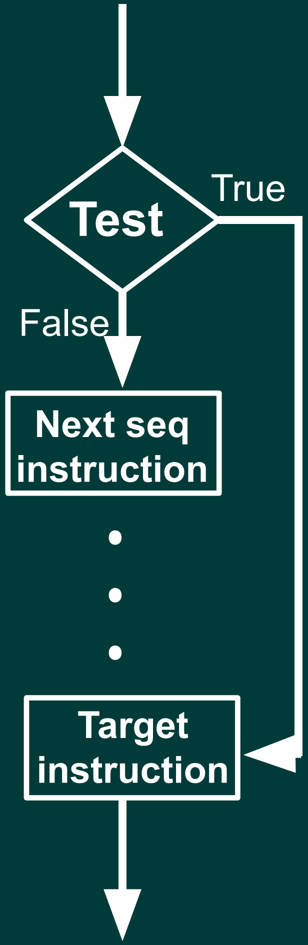
Conditional Instructions

- In order to write meaningful programs, all computer languages contain decision making statements
- In HLL, control structures are used to restrict the execution of sections of code based on conditions
  - If-else clauses
  - Loops which include
    - While loops
    - Do-until loops
    - For loops

Conditional Branches

- Control the flow of execution based on the outcome of a condition
  - The point in a program where a condition is tested is called a branch
  - The condition is a Boolean test that equates to either true or false
- The control test is a comparative one involving two values that must be in registers
  - A true outcome results in the program branching to a location in the program that is <u>**not**</u> the next sequential instruction in the program



RISC-V Conditional Branch Instructions

beq	branch if equal to
bne	branch if not equal to
blt	branch if less than
bge	branch if greater than or equal to

Conditional Branch Instruction Syntax

- All branch instructions require two registers to be specified containing the values to compare followed by a label that identifies the instruction to branch to when true
- Ex. beq t0, t1, next
  - Compare if the value in t0 is equal to the value in t1
  - If true, continue executing at the instruction with the label next
  - Otherwise, continue with the instruction after the branch

Example Code Sequence

- Assuming that registers t3 and t4 have been used in previous calculations
- The control logic that is required is to determine if the value in t3 is equal to the value in t4.
  - If true, increment t4
  - If false, increment both t3 and t4

```
        beq t3, t4, next   # test condition
        addi      t3, t3, 1       # false
next:   addi      t4, t4, 1       # true
```

  - Note the instruction immediately following the branch instruction is **not** the target of the true condition
  - There may be conditions where several instructions could be between the branch and the branch target

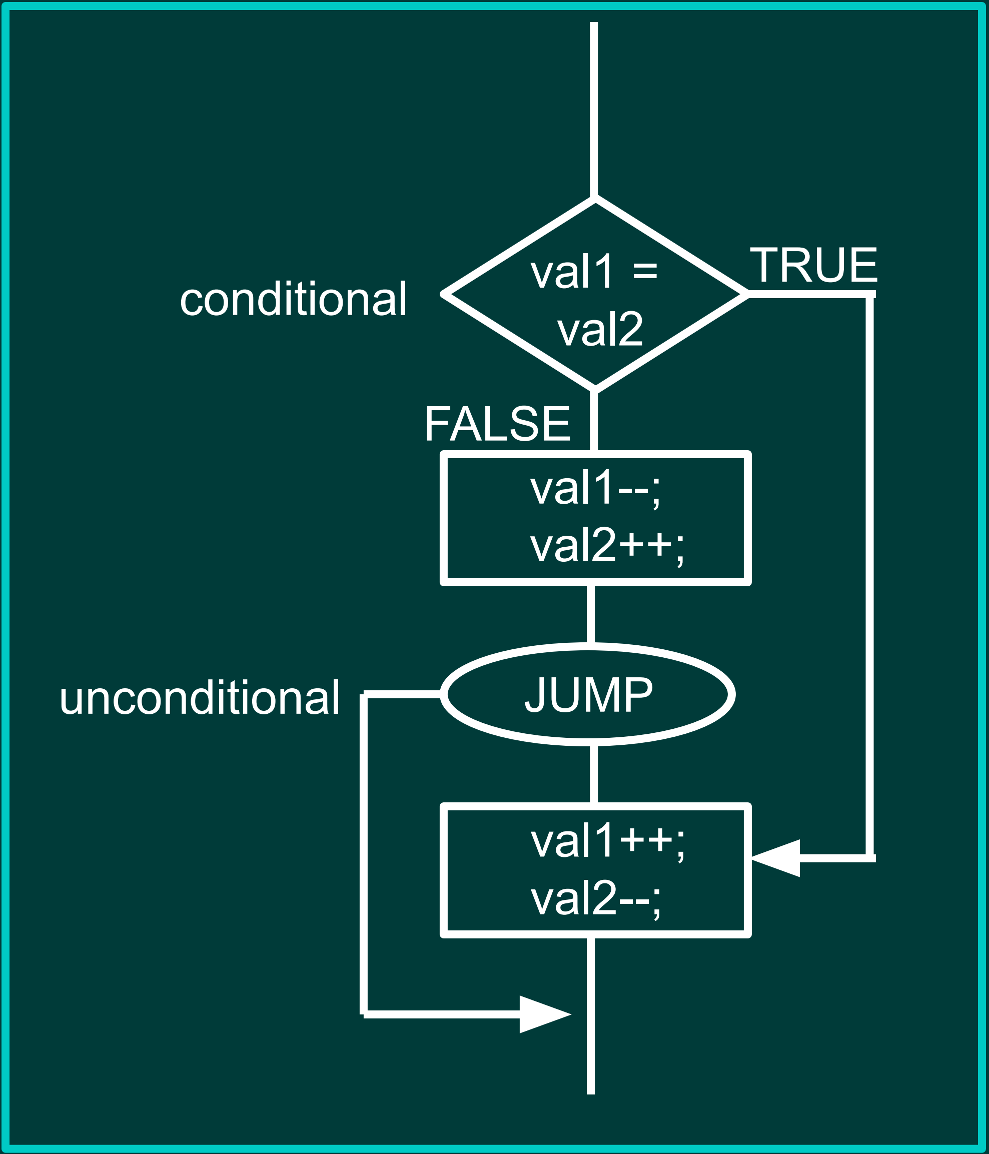
Unconditional Branch

- An unconditional branch is defined as transferring execution to a specific point in the code regardless of any condition
  - No condition check is performed
  - Synonymous with a unconditional “goto” statement in some HLLs
- An unconditional branch is necessary to be able to construct if-else logic where only one of two paths of execution should be executed
  - In other words, if one instruction sequence is executed, another instruction sequence should not
  - Unconditional branches are used in combination with conditional branches for this purpose

Conditional/Unconditional Combination

- IF/ELSE conditions

```
if (val1 = val2) {
   val1++;
   val2--;
}
else {
   val1--;
   val2++;
}
```



Example of Conditional/Unconditional Logic

(Code sequence 1 and code sequence 2 are in the main program)

Conditional branch instruction that tests the condition: **target next if true**

Code sequence 2

This code sequence executes if condition tests false.

Unconditional branch instruction that skips over code that should not execute: **target skip**

target next:

Code sequence 1

This code sequence executes if condition tests true but should not execute if code sequence 1 executed.

target skip:

Unconditional Branch

- In RISC-V, there are two ways to implement an unconditional branch
- The preferred way is to use a jump and link instruction but specify the zero register as the return address
- Ex. jal zero, target_label
  - Target_label is the user defined label for the instruction that should execute next
  - This is an example of using the zero register as a destination register for which the value is not needed to be saved

Note: the jal instruction is primarily used for subroutine calls for which there must be a return to the main program, the link part of the instruction. The return address (ra) register is typically used as the register operand for this purpose.

Unconditional Branch

- Another technique to implement an unconditional branch is to use a conditional branch that always test true
- Ex. beq zero, zero, target_label
  - zero is always equal to zero so this will always be a true condition
  - The branch will always be taken
- This method is not as efficient as using jal
- Combinations of conditional and unconditional instructions can be used independently or together to implement various logic flows among instructions.

Examples Using Pseudocode

```
if (x = y)                    beq   x, y, msub
   m = a – 4;                 jal   zero, next
                       msub:  m = a – 4
                       next:  …
```

```
                              bne  x, y, next
                       msub:  m = a – 4
                       next:  …
```

```
if (a != b)                   bne  a, b, xinc
    x++;                      jal  zero, next
                       xinc:  x = x + 1
                       next:  …
```

```
                              beq  a, b, next
                              x = x + 1
                       next:  …
```

```
if (z = b)                    beq  z, b, aadd
    a = x + y;         asub:  a = x – y
else                          jal  zero, next
    a = x – y;         aadd:  a = x + y
                       next:  …
```

```
                              bne  z, b, asub
                              a = x + y
                              jal  zero, next
                       asub:  a = x – y
                       next:  …
```

Testing The Opposite Condition

```
if (i = j)
    f = g + h;
else
    f = g – h;
```

```
        beq  s3, s4, L1
        sub  s0, s1, s2
        jal  zero, skip
L1:     add  s0, s1, s2
skip: ...
```

```
        bne  s3, s4, else
        add  s0, s1, s2
        jal  zero, skip
else:   sub s0, s1, s2
skip:   …
```

- Testing the opposite condition keeps the instructions in logical order (*then* followed by *else*)
- This is sometimes easier to follow in code
- No performance difference

Example Program

```
        .data
val1:   .word 25        # two values to use in example if code
val2:   .word 50

        .text
main:   lui   s0, 0x10010          # memory base address
        lw    t0, 0(s0)            # load val1 to t0
        lw    t1, 4(s0)            # load val2 to t1

# if (val1 = val2) {val4 = val1 + val2}
ex1:    beq   t0, t1, addvals      # check if equal
        jal   zero, noadd          # go here if not equal
addvals: add  t3, t0, t1           # add only if val1 = val2
noadd:  # add should not execute - check t3 to confirm - value should be 0

# if (val1 != val2) {val4 = val1 + val2}
ex2:    bne   t0, t1, doadd        # check if not equal
        jal   zero, skipadd        # go here if equal
doadd:  add   t3, t0, t1           # add only if val1 != val2
skipadd: # add should execute - check t3 to confirm - should be 75 (0x4b)

exit:   ori   a7, zero, 10
        ecall
```

Another Example Program

```
        .data
val1:   .word 25          # two values to use in example if code
val2:   .word 50

        .text
main:        lui    s0, 0x10010     # memory base address
        lw     t0, 0(s0)         # load val1 to t0
        lw     t1, 4(s0)         # load val2 to t1

#     if (val1 = val2) {                # if-else logic to implement
#       val3 = val1 + val2;
#     } else {
#       val3 = val1 – val2
#     }
            beq      t0, t1, addnums       # if val1 = val2, branch to addnums
subnums:    sub      t3, t0, t1            # else subtract
            jal zero, next         # if subtract was done, skip over add
addnums:    add      t3, t0, t1            # add numbers - subtract was skipped
next:   # code execution continues

exit:   ori    a7, zero, 10
        ecall
```

Loops

- Extension of branching
- Ex. HLL code:

```
loop:   g = g + A[j];
        i = i + j;
        if (i != h) loop;
```

- Assembly code (*do-until* loop):

```
loop:   add s1, s1, t0      # start of loop
     add s3, s3, s4
     bne s3, s2, loop  # end of block
```

- Basic blocks are fundamental to compiling
  - Sequences of instructions that end in a branch
  - Compiler subdivides program into basic blocks

While Loop

- Tests condition at beginning of loop (loop may not execute)
- HLL code:

```
while (i == k)
    i = i + j;
```

- Assembly code (*while* loop)

```
loop:   bne   t0, s5, exit
        add   s3, s3, s4
        jal   zero, loop
exit:
```

- Note opposite condition test at loop entry

Sample Program - Loop

```
# RSIC-V Program to Store Values in Array Using Loop

        .data
fives:  .word   0 0 0 0 0                   # allocate 5 element array
        .text
main:
        lui     s0, 0x10010         # base address of array in s0
        or      s1, zero, zero          # initialize counter (index)
        addi    s4, zero, 5         # initialize max # elements
        ori     t0, zero, 5         # first value to store in array

loop:   slli    s2, s1, 2               # calculate offset
        add     s3, s0, s2          # add offset to base address
        sw      t0, 0(s3)           # store the value to array
        addi    t0, t0, 5           # calculate next value to store

        addi    s1, s1, 1           # increment the counter (index)
        blt     s1, s4, loop        # branch to loop if not done

exit:   ori     a7, zero, 10
        ecall
```

Set Less Than

- The set less than instructions are another way to implement relational operators.
- However, they do not in themselves alter the flow of control.
  - They set a register to 1 or 0 to show the relation between two values.
- Set less than (slt):   slt reg1, reg2, reg3
  - Set reg1 to 1 if value in reg2 < value in reg3
  - Else set reg1 to zero
- slti – set less than immediate
  - Ex.   slti   t0, t1, 10
  - If the value in register t1 is less than 10 set the value in register t0 to 1; otherwise set t0 to zero

Example: slt Instruction

```
        .data
result:     .word     0

        .text
main:       lui    s0, 0x10010
# Load some values into registers
        ori    t0, zero, 5       # load 5 into t0
        ori    t1, zero, 10      # load 10 into t1
        sub    t2, t0, t1        # take the difference

# Compare difference to zero
        slt    t3, t2, zero      # t2 < zero? Set t3 to 1; else 0
        blt    t3, zero, output      # if t3 = 0, store difference
        jal    zero, exit        # else exit

# Store the result
output:     sw    t2, 0(s0)          # Store the result in memory

exit:   ori    a7, zero, 10          # exit program
        ecall
```

Unsigned Operations

- By default, most all instructions that work with numeric operands interpret register values as two’s complement.
- There are some cases where values represent unsigned integers and should not be interpreted as two’s complement numbers
- RISC-V includes instructions specifically defined to interpret numbers as unsigned

Unsigned Instructions

Unsigned Loads

lbu: load byte unsigned
lhu: load halfword unsigned
lwu: load word unsigned

Unsigned Branches

bltu: branch less than unsigned
bgeu: branch greater than or equal unsigned

Unsigned Set Less Than

sltu: set less than unsigned
sltiu: set less than immediate unsigned