5 | Transfer Control & Memory Arrays

In high level languages, we have three control structures:

- **Sequence:** Instructions are executed in the order in which they occur in source program
- **Selection**
    - **One-way selection:** A block of instructions is executed conditionally (if something is true, execute this, like an *if statement*)
    - **Two-way selection:** Exactly one of two blocks of instructions is executed conditionally (this is exemplified by *if-else statements*, in which there's no case where both blocks will be executed)
- **Iteration:** A block of instructions is executed repeatedly (loops, like *while*, called a ***pretest loop***, where the condition is tested before execution)

In assembly languages, we have two types of transfer of control, which are used to implement the control structures.

- **[Conditional] branch:** Control passes to an instruction out of sequence, but only if a condition is satisfied
    - Format: **[label:] bc $rs, $rt, label**
    - Meaning: Transfer to the instruction with the given label iff bc condition is true
    - Branch conditions: All mnemonics in the "Assembly" column used @ bc

Assembly	Meaning	Java/C++/Python
beq	branch to label if equal	==
bne	branch to label if not equal	!=
blt	branch to label if less than	<
ble	branch to label if less than or equal	<=
bgt	branch to label if greater than	>
bge	branch to label if greater than or equal	>=

- Example: beq $t0, $t1, done # branch to done if $t0 == $t1
- Example: bgt $a0, $a3, big # branch to big if $a0 > $a3
- Example: bne $t3, $0, loop # branch to loop if $t3 != 0

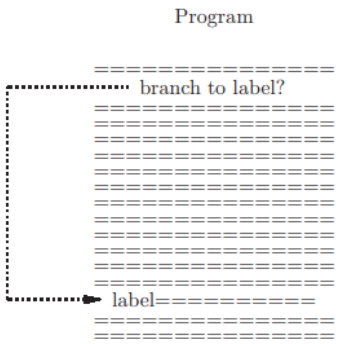
- In these examples, "big", "done", and "loop" are labels to some other instructions in the program (target of the branch)

- **[Unconditional] jump:** Control passes to an instruction out of sequence, unconditionally (always takes place, no testing is done)

  - Format: **[label:] j label**
  - Meaning: Transfer to the instruction with the given label unconditionally

  - Example: j exit # jump to exit
  - Example: j loop # jump to loop

Sequence happens automatically; for **one-way selection**, we use a branch instruction. To the right is a diagram of what happens. Each line represents an instruction in memory. When we come to the branch, it may or may not branch down to the instruction with the label.



Example: Clear register $v0 if *x* is negative.

```
        lw      $t0, x          # load x into $t0
        bge     $t0, $0, notNeg # branch if not negative, aka $t0 >= 0
        move    $v0, $0         # $v0 = 0
notNeg: [other instructions]
```

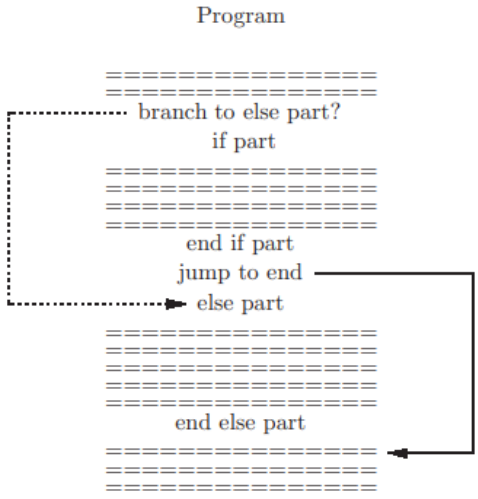
Example: Store the largest of the 4 numbers beginning at start into largest.

```
                .data
largest:        .word   0
start:          .word   3, -99, 7, 0

                .text               # where $v0 is largest seen so far
        lw      $v0, start          # $v0 = 3
        lw      $t1, start+4        # $t1 = -99
        bge     $v0, $t1, first     # condition is true (is $v0 >= $t1?)
        move    $v0, $t1            # if false, move $t1's value into $v0
```

```
first:
        lw      $t1, start+8            # $t1 = 7
        bge     $v0, $t1, second        # condition is false (is $v0 >= $t1?)
        move    $v0, $t1                # $v0 = 7
second:
        lw      $t1, start+12           # $t1 = 0
        bge     $v0, $51, done          # condition is true (is $v0 >= $t1?)
        move    $v0, $t1
done:
        sw      $v0, largest            # largest = 7
```

Notice how if the condition was false, it would do the *move* instruction and continue on to the *second* label regardless. If the condition was false, it would ***skip*** the *move* instruction.



As for **two-way selection**, it uses a branch instruction and a jump instruction. For the diagram to the right, each line represents an instruction, there is a conditional branch, if we don't want to do the if part, branch to the else part.

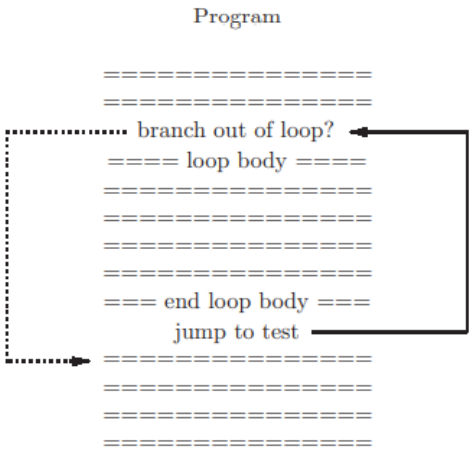
Example: What we want to do from Java to assembly language:

```
if(x > y)
        y =3;
else
        y = 0;

        .data
x:      .word       -4
y:      .word       17

        .text
lw      $t0, x                  # load x into $t0
lw      $t1, y                  # load y into $t1
ble     $t0, $t1, elsePart      # branch if x<= y (this is the complement of above)
                                # if condition is false, branch to else part
li      $t0, 3                  # $t0 = 3
```

```
        j       done            # skip over the elsePart, so unconditional to done
elsePart:
        li      $t0, 0          # $t0 = 0
done:
        sw      $t0, y          # y = $t0
```



Now, for **iteration structures**, it's a ***pretest loop*** (the test for termination occurs *before* the body of the loop is executed, even once), requiring a branch and a jump instruction. See the right for the diagram, where each line is an instruction. What we're talking about is a *while loop*. A ***posttest loop*** requires fewer statements oppositely.

The jump occurs over and over again until the branch condition is false, and then we exit the loop.

Example: Find the sum of numbers from 1 through the value in register $a0, leaving the sum in $v0.

How we would code it in high level language (Java), see right (decrement by 1 and add 10 + 9 = 19, 19 + 8, etc.):

```
while($a0 > 0) {
        $v0 = $v0 + $a0;
        $a0 = $a0 - 1;
}
```

```
        .text
        li      $v0, $0         # accumulator, where $v0 = 0
lp:
        ble     $a0, $0, done   # finished? this is the loop condition, is $a0 <= 0?
        add     $v0, $v0, $a0   # $v0 = $v0 + $a0
        addi    $a0, $a0, -1    # $a0 -- (or $a0 = $a0 - 1)
        j       lp              # repeat (jump to lp)
done:                           # where it should go when loop fails
```

Arrays

An **array** is a sequence of contiguous memory locations treated as a homogeneous list of values. Contiguous means memory locations that are adjacent to each other (no gaps, no interruptions). Like Java, this is a fixed size.

In assembly language:

```
            .data
numbers:    .space      100         # 100 full words = 80 bytes
```

To access it:

```
        la      $t0, numbers        # loads the address of 'numbers' to $t0
```

To access elements inside the array (these are for the first value):

```
        lw      $t1, numbers        # use the symbolic address of the array
        lw      $t1, 0($t0)         # use the explicit address of the array,
                                    # 0 displacement
```

If we wanted the second element (index 1) of numbers, and so on. Remember that each word is 4 bytes, thus the increase of 4.

```
        lw      $t1, numbers+4      # symbolic
        lw      $t1, 4($t0)         # explicit
```

Example: Find the sum of the numbers in the array named grades.

```
            .data
sum:        .word       0                   # where the sum should be stored
size:       .word       6                   # number of values to be summed
grades:     .word       5, -3, 44, 100, 0, 20   # array

            .text
        lw      $t0, size                   # loop counter
        li      $v0, 0                      # accumulator = 0, accumulate sum
        la      $t1, grades                 # address of array (pointer)
lp:
        ble     $t0, 0, done                # finished? $t0 <= 0 (no more elements)
        lw      $t2, 0($t1)                 # grades[$t1], using explicit address
        add     $v0, $v0, $t2               # accumulate sum, $v0 = $v0 + $t2
        addi    $t1, $t1, 4                 # address of next word ($t1 = $t1 + 4 bytes)
        addi    $t0, $t0, -1                # decrement loop counter ($t0--)
        j       lp                          # repeat loop unconditionally until ble true
done:
```

```
sw      $v0, sum                    # now, sum = $v0
```

In conclusion, a **memory array** is just a sequence of contiguous memory locations. Sometimes, people call it a ***buffer***, a portion of memory set aside to store lots of data.