Branch instructions allow for the changing of the flow of the program. Without them, the program will continue in a straight line of execution.

Two types of branch instructions
- Conditional branches
- Unconditional/Direct branches

**Conditional branches**
- Branch which is dependent on a comparison. This comparison can be between 2 values in registers or with zero, and can be a signed or unsigned comparison.
  - Ex:
    - `beq $t0, $t1, label`  # if $t0 == $t1 then jump to the address of label
    - `beqz $t0, label`  # if $t0 == 0 then jump to the address of label
    - `bne $t0, $t1, label`  # if $t0 != $t1 then jump to the address of label
    - `j label`  # jump to the address of label (direct branch)
  - Many conditional branches are pseudo-instructions. They are assembled using various ‘set’ instructions.
    - Ex:
      - `bge $t4, $t2, label`  # branch to label if $t4 >= $t2
      - Assembled as:
        - `slt $1, $12, $10`  # set register $1 to 1, if register $12 < register $10
        - `beq $1, $0, label`  # branch to label if $1 = 0
      - Register $12 is $t4, register $10 is $t2.
      - So if $t4 < $t2, then set $1 to 1, or $1 is 0 if $t4 >= $t2. This is what we want.
      - Next instruction branches if register $1 is 0.
      - This requires that register $1 is available to the assembler. ($1 is same as $at).

**Unconditional branch**
- Branch which always occurs.
  - Ex:
    - `b`  # branch to label

**Jump instructions**
- Several types of jump instructions
  - `jr $ra`  # jump to address in register
    - Specifically, this instruction is how you return from a function (more on functions later).
    - Can replace $ra with any register.
  - `jal label`  # jump and link
    - This instruction is how you jump to a function.
- Saves the return address in the $ra register ($ra = PC + 4), and then jumps to the label.
  - jalr Rsrc  # jump and link register
- This instruction jumps to the address stored in any register specified by Rsrc and places the return address in $ra ($ra = PC + 4).

- Branch and jump instructions and their operand formats
  - Cross symbols represent pseudo-instructions.

<table>
<thead>
<tr>
<th>Description</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch instruction</td>
<td>b</td>
<td>label</td>
</tr>
<tr>
<td>Branch coprocessor z true</td>
<td>bczt</td>
<td>label</td>
</tr>
<tr>
<td>Branch coprocessor z false</td>
<td>bczf</td>
<td>label</td>
</tr>
<tr>
<td>Branch on equal</td>
<td>beq</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on equal zero</td>
<td>beqz</td>
<td>Rsrc, label</td>
</tr>
<tr>
<td>Branch on greater than equal</td>
<td>bge</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on GTE unsigned</td>
<td>bgeu</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on greater than zero</td>
<td>bgez</td>
<td>Rsrc, label</td>
</tr>
<tr>
<td>Branch on greater than equal</td>
<td>bgezl</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on greater than and link</td>
<td>bgtu</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on greater than zero</td>
<td>bgtz</td>
<td>Rsrc, label</td>
</tr>
<tr>
<td>Branch on less than equal</td>
<td>ble</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on LTE unsigned</td>
<td>bleu</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on less than zero</td>
<td>blez</td>
<td>Rsrc, label</td>
</tr>
<tr>
<td>Branch on greater than equal</td>
<td>bgezl</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on less than and link</td>
<td>bltz</td>
<td>Rsrc, label</td>
</tr>
<tr>
<td>Branch on less than</td>
<td>blt</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on less than unsigned</td>
<td>bltu</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on not equal</td>
<td>bne</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Branch on not equal zero</td>
<td>boez</td>
<td>Rsrc1, Src2, label</td>
</tr>
<tr>
<td>Jump</td>
<td>j</td>
<td>label</td>
</tr>
<tr>
<td>Jump and link</td>
<td>jal</td>
<td>label</td>
</tr>
<tr>
<td>Jump and link register</td>
<td>jalr</td>
<td>Rsrc</td>
</tr>
<tr>
<td>Jump register</td>
<td>jr</td>
<td>Rsrc</td>
</tr>
</tbody>
</table>

**Control Structure**

- Any reasonable programming language must have an *if* statement, and some type of *loop*.
- Otherwise programs would be straight lines of code
- Need conditional branch instructions and a direct or unconditional branch instruction.

**Implementation of beq**

```
addi $s0, $0, 4     # $s0 = 0 + 4 = 4
addi $s1, $0, 1     # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2     # $s1 = 1 << 2 = 4
beq  $s0, $s1, target # branch is taken
addi $s1, $s1, 1     # not executed
```
Implementation of $bne$

```plaintext
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
dddi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll $s1, $s1, 2      # $s1 = 1 << 2 = 4
bne $s0, $s1, target # branch not taken
addi $s1, $s1, 1     # $s1 = 4 + 1 = 5
sub $s1, $s1, $s0    # $s1 = 5 - 4 = 1
```

**Implementation of an ‘if’**

In Java:
```
if (i == j)
    f = g + h;
```
```
f = f - i;
```

If we assume, the values are loaded from memory into registers

In MIPS:
```
#$s0 = f, $s1 = g,  $s2 = h, $s3 = i,  $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
```

**Implementation of an ‘if-else’**

In Java:
```
if (i == j)
    f = g + h;
else
    f = f - i;
```

In MIPS:
```
# $s0 = f, $s1 = g,  $s2 = h, $s3 = i,  $s4 = j
beq $s3, $s4, TRUE
add $s0, $s1, $s2 #Else case
```
```
j DONE
```
```
TRUE: sub $s0, $s0, $s3 #If case
```
```
DONE:
```

**Implementation of a ‘for’**

In Java:
```
// sums the powers of
// 2 from 1 to 256
int sum = 0;
int i;
for (i=1; i < 257; i=i*2) {
```
```


```plaintext
sum = sum + i;
}

In MIPS:
# $s0 = i, $s1 = sum
  add $s1, $0, $0
  addi $s0, $0, 1
  addi $t0, $0, 257
loop: slt $t1, $s0, $t0
  beq $t1, $0, done
  add $s1, $s1, $s0
  sll $s0, $s0, 1
  j loop
done:

Implementation of a ‘while’

In Java:
// determines the power
// of n such that 2^n = 128
int pow = 1;
in n = 0;
while (pow != 128) {
  pow = pow * 2;
  n = n + 1;
}

In MIPS:
# $s0 = pow, $s1 = n
  addi $s0, $0, 1
  add $s1, $0, $0
  addi $t0, $0, 128
while:
  beq $s0, $t0, done
  sll $s0, $s0, 1
  addi $s1, $s1, 1
  j while
done:

Implementation of a Case/switch statement

In Java:
switch (amount) {
  case 20:
    fee = 2;
    break;
  case 50:
    fee = 3;
    break;
  case 100:
    fee = 5;
}
```

Stony Brook University: CSE220 System Fundamentals I

Prof. Jennifer Wong-Ma
break;
default:
fee = 7;
}

In MIPS:
case20:
  li $t0, 20
  bne $s0, $t0, case50
  li $s1, 2
  j done

case50:
  li $t0, 50
  bne $s0, $t0, case100
  li $s1, 3
  j done

case100:
  li $t0, 100
  bne $s0, $t0, default
  li $s1, 5
  j done
default:
  li $s1, 7
done:

• Notice that last case must wait for $n$-1 tests before executing, making it slow.
• Alternative tries to go to all cases equally fast: jump address table for scalability.
  o Idea: encode alternatives as a table of addresses of the cases.
    ▪ Table is an array of words with addresses corresponding to case labels.
  o Program indexes into table and jumps.
• MIPS instruction “jump register” (jr) unconditionally branches to an address in a register; use load to get address.