### Jump Instructions

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>j label</code></td>
<td>pc = pc &amp; 0xF0000000</td>
<td>000010XXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td><code>jal label</code></td>
<td>( r_{31} = pc + 4; ) ( pc = pc &amp; 0xF0000000</td>
<td>(X&amp;2) )</td>
</tr>
<tr>
<td><code>jr ra</code></td>
<td>( pc = ra )</td>
<td>000000ssss000000000000000001000</td>
</tr>
<tr>
<td><code>jalr ra</code></td>
<td>( r_{31} = pc + 4; ) ( pc = ra )</td>
<td>000000ssss000000000000000001001</td>
</tr>
</tbody>
</table>

- Jump instruction **unconditionally** transfer execution to a new location.
- SPIM will calculate correct value for \( X \) from location of label in code.
- `jal` & `jalr` set \( r_{31}(\$ra) \) to address of the next instruction.
  - Used for function calls.
  - Return can then be implemented with `jr $ra`.

### Branch Instructions

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b label</code></td>
<td>pc += 1&amp;2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td><code>beq ra,rt,label</code></td>
<td>if ( ra == rt ) pc += 1&amp;2</td>
<td>000100ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bne ra,rt,label</code></td>
<td>if ( ra != rt ) pc += 1&amp;2</td>
<td>000101ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>ble ra,rt,label</code></td>
<td>if ( ra &lt;= rt ) pc += 1&amp;2</td>
<td>000110ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bge ra,rt,label</code></td>
<td>if ( ra &gt;= rt ) pc += 1&amp;2</td>
<td>000100ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bgt ra,rt,label</code></td>
<td>if ( ra &gt; rt ) pc += 1&amp;2</td>
<td>000100ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>blt ra,rt,label</code></td>
<td>if ( ra &lt; rt ) pc += 1&amp;2</td>
<td>000100ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>blez ra,label</code></td>
<td>if ( ra &lt;= 0 ) pc += 1&amp;2</td>
<td>000110ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bgtz ra,label</code></td>
<td>if ( ra &gt; 0 ) pc += 1&amp;2</td>
<td>000111ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bltz ra,label</code></td>
<td>if ( ra &lt; 0 ) pc += 1&amp;2</td>
<td>000001ssssssssssssssssssssssssssssss</td>
</tr>
<tr>
<td><code>bgez ra,label</code></td>
<td>if ( ra &gt;= 0 ) pc += 1&amp;2</td>
<td>000001ssssssssssssssssssssssssssssss</td>
</tr>
</tbody>
</table>

- Branch instruction **conditionally** transfer execution to a new location.
- SPIM will calculate correct value for I from location of label in code.
- SPIM allows second operand \( (ri) \) to be replaced by a constant.
- `beq`, `bne`, `bgtz`, `bltz`, `blez`, `bgez`, `bgtz`, `bltz` pseudo-instructions.


### Example Translation of Branch Pseudo-instructions

<table>
<thead>
<tr>
<th>Pseudo-Instructions</th>
<th>Real Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{bge} $t1$, $t2$, label</td>
<td>\texttt{slt} $at$, $t1$, $t2$</td>
</tr>
<tr>
<td>\texttt{blt} $t1$, $t2$, label</td>
<td>\texttt{beq} $at$, $0$, label</td>
</tr>
<tr>
<td>\texttt{b} label</td>
<td>\texttt{beq} $at$, $0$, label</td>
</tr>
<tr>
<td>\texttt{beqz} $t3$, label</td>
<td>\texttt{bne} $at$, $0$, label</td>
</tr>
<tr>
<td>\texttt{bnez} $t4$, label</td>
<td>\texttt{beq} $t3$, $0$, label</td>
</tr>
<tr>
<td></td>
<td>\texttt{bne} $t4$, $0$, label</td>
</tr>
</tbody>
</table>

---

**goto in C**

The \texttt{goto} statement allows transfer of control to any labelled point with a function. For example, this code:

```c
for (int i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

can be written as:

```c
int i = 1;
loop:
    if (i > 10) goto end;
    i++;
    printf("%d", i);
    printf("\n");
goto loop;
end:
```

---

- \texttt{goto} statements can result in very difficult to read programs.
- \texttt{goto} statements can also result in slower programs.
- In general, use of \texttt{goto} is considered \textbf{bad} programming style.
- Do not use \texttt{goto} without very good reason.
- Kernel & embedded programmers sometimes use goto.
Writing correct assembler directly is hard.

**Recommended strategy:**
- develop a solution in C
- map down to “simplified” C
- translate simplified C statements to MIPS instructions

**Simplified C**
- does not have while, compound if, complex expressions
- does have simple if, goto, one-operator expressions

Simplified C makes extensive use of
- labels … symbolic name for C statement
- goto … transfer control to labelled statement

---

**Mapping C into MIPS**

Things to do:
- allocate variables to registers/memory
- place literals in data segment
- transform C program to:
  - break expression evaluation into steps
  - replace most control structures by goto

---

**Adding Two Numbers — C to Simple C**

### C
```c
int main(void) {
    int x = 17;
    int y = 25;
    printf("%d
", x + y);
    return 0;
}
```

### Simplified C
```c
int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
    printf("%d
", z);
    printf("\n");
    return 0;
}
```
**Adding Two Numbers — Simple C to MIPS**

**Simplified C**

```c
int x, y, z;
x = 17;
y = 25;
z = x + y;
printf("%d", z);
printf("\n");
```

**MIPS**

```mips
# add 17 and 25 and print result
main:
    # x, y, z in $t0, $t1, $t2
    li $t0, 17
    li $t1, 25
    add $t2, $t1, $t0
    move $a0, $t2
    # printf("%d", z);
    li $v0, 1
    syscall
    li $a0, \n
    # printf("%c", '\n');
    li $v0, 11
    syscall
    li $v0, 0
    # return 0
    jr $ra
```

**Loops — while from C to Simplified C**

**Standard C**

```c
i = 0;
n = 0;
while (i < 5) {
    n = n + i;
i++;
}
```

**Simplified C**

```c
i = 0;
n = 0;
loop:
    if (i >= 5) goto end;
    n = n + i;
i++;
    goto loop;
end:
```

**Loops — while from Simplified C to MIPS**

**Simplified C**

```c
i = 0;
n = 0;
loop:
    if (i >= 5) goto end;
    n = n + i;
i++;
    goto loop;
end:
```

**MIPS**

```mips
li $t0, 0 # i in $t0
li $t1, 0 # n in $t1
loop:
    bge $t0, 5, end
    add $t1, $t1, $t0
    addi $t0, $t0, 1
    j loop
end:
```

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

10 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

11 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

12 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

13 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

14 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

15 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

16 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

17 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

18 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

19 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

20 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

21 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

22 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

23 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

24 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

25 / 26

---

**Source code for add.s**

https://www.cse.unsw.edu.au/~cs1521/21T3/COMP1521 21T3 — MIPS Control

26 / 26
### Conditionals — if from C to Simplified C

**Standard C**

```c
if (i < 0) {
    n = n - i;
} else {
    n = n + i;
}
```

**Simplified C**

```c
if (i >= 0) goto else1;
```

```c
n = n - i;
```

```c
goto end1;
```

```c
else1:
    n = n + i;
end1:
```

*note:* else is not a valid label name in C

---

### Conditionals — if from Simplified C to MIPS

**Simplified C**

```c
if (i >= 0) goto else1;
n = n - i;
goto end1;
else1:
n = n + i;
end1:
```

**MIPS**

```mips
# assuming i in $t0,
# assuming n in $t1...
bge $t0, 0, else1
sub $t1, $t1, $t0
goto end1
else1:
    add $t1, $t1, $t0
end1:
```

---

### Conditionals — if and &&: from C to Simplified C

**Standard C**

```c
if (i < 0 && n >= 42) {
    n = n - i;
} else {
    n = n + i;
}
```

**Simplified C**

```c
if (i >= 0) goto else1;
```

```c
if (n < 42) goto else1;
```

```c
n = n - i;
```

```c
goto end1;
```

```c
else1:
    n = n + i;
end1:
```
Conditionals — if and &&: from Simplified C to MIPS

**Simplified C**

```c
if (i >= 0) goto else1;
if (n < 42) goto else1;
n = n - i;
goto end1;
else1:
n = n + i;
end1:
```

**MIPS**

```mips
# assume i in $t0
# assume n in $t1
bge $t0, 0, else1
blt $t1, 42, else1
sub $t1, $t1, $t0
j end1
else1:
add $t1, $t1, $t0
end1:
```

odd-even: from C to simplified C

**Standard C**

```c
if (i < 0 || n >= 42) {
    n = n - i;
} else {
    n = n + i;
}
```

**Simplified C**

```c
if (i < 0) goto then1;
if (n >= 42) goto then1;
goto else1;
then1:
n = n - i;
goto end1;
else1:
n = n + i;
end1:
```

Printing First 10 Integers: C to simplified C

**C**

```c
int main(void) {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

**Simplified C**

```c
int main(void) {
    int i;
i = 1;
loop:
    if (i > 10) goto end;
i++;
    printf("%d", i);
    printf("\n");
goto loop;
end:
    return 0;
}
```
Printing First 10 Integers: MIPS

```mips
main:
    # int main(void) {
    # int i; // in register $t0
    li $t0, 1
    # i = 1;

loop:
    # loop:
    bgt $t0, 10, end # if (i > 10) goto end;
    move $a0, $t0
    # printf("%d" i);
    li $v0, 1
    syscall
    li $a0, '\n'
    # printf("\n");
    li $v0, 11
    syscall
    addi $t0, $t0, 1
    # i++;
    b loop
    # goto loop;

end:
    li $v0, 0
    # return 0
    jr $ra
```

Odd or Even: C to simplified C

**C**

```c
int main(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    if ((x & 1) == 0) {
        printf("Even\n");
    } else {
        printf("Odd\n");
    }
    return 0;
}
```

**Simplified C**

```c
int main(void) {
    int x, v0;
    printf("Enter a number: ");
    scanf("%d", &x);
    v0 = x & 1;
    if (v0 == 1) goto odd;
    printf("Even\n");
    goto end;
odd:
    printf("Odd\n");
end:
    return 0;
}
```

Odd or Even: MIPS

```mips
main:
    la $a0, string0 # printf("Enter a number: ");
    li $v0, 4
    syscall
    li $v0, 5 # scanf("%d", &x);
    syscall
    and $t0, $v0, 1 # if (x & 1 == 0) {
    beq $t0, 1, odd
    la $a0, string1 # printf("Even\n");
    li $v0, 4
    syscall
    b end
```

Odd or Even: MIPS

```mips
main:
    la $a0, string0 # printf("Enter a number: ");
    li $v0, 4
    syscall
    li $v0, 5 # scanf("%d", &x);
    syscall
    and $t0, $v0, 1 # if (x & 1 == 0) {
    beq $t0, 1, odd
    la $a0, string1 # printf("Even\n");
    li $v0, 4
    syscall
    b end
```

Odd or Even: MIPS

```mips
main:
    la $a0, string0 # printf("Enter a number: ");
    li $v0, 4
    syscall
    li $v0, 5 # scanf("%d", &x);
    syscall
    and $t0, $v0, 1 # if (x & 1 == 0) {
    beq $t0, 1, odd
    la $a0, string1 # printf("Even\n");
    li $v0, 4
    syscall
    b end
```

Odd or Even: MIPS

```mips
main:
    la $a0, string0 # printf("Enter a number: ");
    li $v0, 4
    syscall
    li $v0, 5 # scanf("%d", &x);
    syscall
    and $t0, $v0, 1 # if (x & 1 == 0) {
    beq $t0, 1, odd
    la $a0, string1 # printf("Even\n");
    li $v0, 4
    syscall
    b end
```
Odd or Even: MIPS

```assembly
odd:       # else
    la $a0, string2 # printf("Odd\n");
    li $v0, 4
    syscall
end:
    li $v0, 0 # return 0
    jr $ra
.data
string0:   .asciiz "Enter a number: "
string1:   .asciiz "Even\n"
string2:   .asciiz "Odd\n"
```

Source code for odd_even.s

Sum 100 Squares: C to simplified C

**C**

```c
int main(void) {
    int sum = 0;
    for (int i = 0; i <= 100; i++) {
        sum += i * i;
    }
    printf("%d\n", sum);
    return 0;
}
```

Source code for sum_100_squares.c

**Simplified C**

```c
int main(void) {
    int i, sum, square;
    sum = 0;
    i = 0;
    loop:
    if (i > 100) goto end;
        square = i * i;
        sum = sum + square;
        i = i + 1;
    goto loop;
end:
    printf("%d", sum);
    printf("\n");
    return 0;
}
```

Source code for sum_100_squares/simple.c

Sum 100 Squares: MIPS

```assembly
# calculate 1*1 + 2*2 + ... + 99 * 99 + 100 * 100
# sum in $t0, i in $t1, square in $t2
main:
    li $t0, 0 # sum = 0;
    li $t1, 0 # i = 0
loop:
    bgt $t1, 100, end # if (i > 100) goto end;
    mul $t2, $t1, $t1 # square = i * i;
    add $t0, $t0, $t2 # sum = sum + square;
    addi $t1, $t1, 1 # i = i + 1;
    b loop
end:
```

Source code for sum_100_squares.s
Sum 100 Squares: MIPS

end:
move $a0, $t0  
# printf("%d", sum);
li $v0, 1  
syscall
li $a0, 'n'  
# printf("\n");
li $v0, 11  
syscall
li $v0, 0  
# return 0
jr $ra

source code for sum_100_squares.s

Side Topic: C do/while

- C has a different while loop - do/while.
- Loop condition checked at bottom of loop executed - always executed once
- Many programmers do not use it

```c
int i = 1;
loop:
    printf("%d", i);
    printf("\n");
i++;
if (i < 10) goto loop;
end:
```

Can be written as:

```c
int i = 1;
loop:
    printf("%d", i);
    printf("\n");
i++;
if (i < 10) goto loop;
end:
```