Jump Instructions

<table>
<thead>
<tr>
<th>assembler</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>j label</td>
<td>pc = pc &amp; 0xF0000000</td>
<td>00000100000000000000000000000000</td>
</tr>
<tr>
<td>jal label</td>
<td>r_{31} = pc + 4; pc = pc &amp; 0xF0000000</td>
<td>00000110000000000000000000000000</td>
</tr>
<tr>
<td>jr r_s</td>
<td>pc = r_s</td>
<td>00000010000000000000000000000000</td>
</tr>
<tr>
<td>jalr r_s</td>
<td>r_{31} = pc + 4; pc = r_s</td>
<td>00000010000000000000000000000000</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>b label</td>
<td>pc += I\times2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>beq r_s,r_t,label</td>
<td>if ( r_s = r_t ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>bne r_s,r_t,label</td>
<td>if ( r_s \neq r_t ) pc += I\times2</td>
<td>00010110000000000000000000000000</td>
</tr>
<tr>
<td>ble r_s,r_t,label</td>
<td>if ( r_s \leq r_t ) pc += I\times2</td>
<td>00010110000000000000000000000000</td>
</tr>
<tr>
<td>bgt r_s,r_t,label</td>
<td>if ( r_s &gt; r_t ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>blt r_s,r_t,label</td>
<td>if ( r_s &lt; r_t ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>bge r_s,r_t,label</td>
<td>if ( r_s \geq r_t ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>blez r_s,label</td>
<td>if ( r_s = 0 ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>bgtz r_s,label</td>
<td>if ( r_s &gt; 0 ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>bltz r_s,label</td>
<td>if ( r_s &lt; 0 ) pc += I\times2</td>
<td>00010010000000000000000000000000</td>
</tr>
<tr>
<td>bgez r_s,label</td>
<td>if ( r_s \geq 0 ) pc += I\times2</td>
<td>00010010000000000000000000000000</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 \( (r_t) \) to be replaced by a constant
Example Translation of Branch Pseudo-instructions

<table>
<thead>
<tr>
<th>Pseudo-Instructions</th>
<th>Real Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>bge $t1, $t2, label</td>
<td>slt $at, $t1, $t2, label</td>
</tr>
<tr>
<td>blt $t1, $t2, label</td>
<td>beq $at, $0, label</td>
</tr>
<tr>
<td></td>
<td>slt $at, $t1, $t2</td>
</tr>
<tr>
<td></td>
<td>bne $at, $0, label</td>
</tr>
</tbody>
</table>

**goto in C**

The **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:
```

**goto in C**

- **goto** statements can result in very difficult to read programs.
- **goto** statements can also result in slower programs.
- In general, use of **goto** is considered **bad** programming style.
- Do not use **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\n", 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     # x = 17;
    li $t1, 25     # y = 25;
    add $t2, $t1, $t0 # z = x + y
    move $a0, $t2  # printf("%d", z);
    li $v0, 1
    syscall
    li $a0, '\n'
    # printf("\n");
    li $v0, 11
    syscall
    li $v0, 0  # return 0
    jr $ra
```

source code for add.s

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

**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
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;
if (n < 42) goto else1;
else1:
    n = n - i;
end1:
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**

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

#### MIPS

```assembly
# 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;
go to else1;
then1:
    n = n - i;
go to 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
# print integers 1..10 one per line
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    $v0, '
'    # printf("\n");
    syscall
    addi $t0, $t0, 1 # i++;
    j loop         # goto loop;
end:       # return 0
    li    $v0, 0
    syscall
jr   $ra
```

source code for print10.s

(Odd or Even: MIPS)

```mips
# read a number and print whether its odd or even
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
    j end
```

source code for odd_even.s

(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;  // print even case
    printf("Even\n");
    goto end;
odd:
    printf("Odd\n");
end:
    return 0;
}
```

(Odd or Even: MIPS)

```mips
# read a number and print whether its odd or even
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
    j end
```

source code for odd_even.s
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"
```

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;
}
```

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;
}
```

Sum 100 Squares: MIPS

```
# 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;
    j loop
end:
```

source code for odd_even.s
source code for sum_100_squares.c
source code for sum_100_squares.simple.c
source code for sum_100_squares.s
end:
    move $a0, $t0  # printf("%d", sum);
    li $v0, 1
    syscall
    li $a0, '
'  # printf("\n");
    li $v0, 11
    syscall
    li $v0, 0  # return 0
    jr $ra

source code for sum_100_squares.s

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