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 \mid (X\ll 2))</td>
<td>000010XXXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>jal (label)</td>
<td>(r_{31} = pc + 4; ) (pc = pc &amp; 0xF0000000 \mid (X\ll 2))</td>
<td>000011XXXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>jr (r_s)</td>
<td>(pc = r_s)</td>
<td>000000ssssss000000000000000001000</td>
</tr>
<tr>
<td>jalr (r_s)</td>
<td>(r_{31} = pc + 4; ) (pc = r_s)</td>
<td>000000ssssss000000000000000001001</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>\texttt{b label}</td>
<td>pc += I\ll 2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>\texttt{beq} (r_s, r_t, \text{label})</td>
<td>if ((r_s == r_t)) pc += I\ll 2</td>
<td>000100ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td>\texttt{bne} (r_s, r_t, \text{label})</td>
<td>if ((r_s != r_t)) pc += I\ll 2</td>
<td>000101ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td>\texttt{ble} (r_s, r_t, \text{label})</td>
<td>if ((r_s \leq r_t)) pc += I\ll 2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>\texttt{bgt} (r_s, r_t, \text{label})</td>
<td>if ((r_s &gt; r_t)) pc += I\ll 2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>\texttt{blt} (r_s, r_t, \text{label})</td>
<td>if ((r_s &lt; r_t)) pc += I\ll 2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>\texttt{bge} (r_s, r_t, \text{label})</td>
<td>if ((r_s \geq r_t)) pc += I\ll 2</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>\texttt{blez} (r_s, \text{label})</td>
<td>if ((r_s \leq 0)) pc += I\ll 2</td>
<td>000110ssssss00000IIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td>\texttt{bgtz} (r_s, \text{label})</td>
<td>if ((r_s &gt; 0)) pc += I\ll 2</td>
<td>000111ssssss00000IIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td>\texttt{bltz} (r_s, \text{label})</td>
<td>if ((r_s &lt; 0)) pc += I\ll 2</td>
<td>000000ssssss00000IIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td>\texttt{bgez} (r_s, \text{label})</td>
<td>if ((r_s \geq 0)) pc += I\ll 2</td>
<td>000000ssssss00001IIIIIIIIIIIIIIII</td>
</tr>
</tbody>
</table>

- branch instruction **conditionally** transfer execution to a new location
- spim will calculate correct value for \(I\) from location of \texttt{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><code>bge  $t1, $t2, label</code></td>
<td><code>slt  $at, $t1, $t2</code></td>
</tr>
<tr>
<td><code>blt  $t1, $t2, label</code></td>
<td><code>beq  $at, $0, label</code></td>
</tr>
<tr>
<td></td>
<td><code>slt  $at, $t1, $t2</code></td>
</tr>
<tr>
<td></td>
<td><code>bne  $at, $0, label</code></td>
</tr>
</tbody>
</table>
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
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("%c", '\n');
    li $v0, 11
    syscall
    li $v0, 0  # return 0
    jr $ra
```

Source code for add.s: [https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521_21T2 — MIPS Control](https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521_21T2 — 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

\[
\begin{align*}
\text{i} &= 0; \\
\text{n} &= 0; \\
\text{loop:} \\
&\quad \text{if } (i \geq 5) \text{ goto end;} \\
&\quad n = n + i; \\
&\quad i++; \\
&\quad \text{goto loop;} \\
\text{end:}
\end{align*}
\]

MIPS

\[
\begin{align*}
\text{li } &\quad t0, 0 \quad \# i \text{ in } t0 \\
\text{li } &\quad t1, 0 \quad \# n \text{ in } t1 \\
\text{loop:} \\
&\quad \text{bge } t0, 5, \text{ end} \\
&\quad \text{add } t1, t1, t0 \\
&\quad \text{addi } t0, t0, 1 \\
&\quad \text{j } \quad \text{loop} \\
\text{end:}
\end{align*}
\]
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;
    n = n - i;
    goto 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**

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

**Simplified C**

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

**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**

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

**Simplified C**

```
if (i < 0) goto then1;
if (n >= 42) goto then1;
    goto else1;
then1:
    n = n - i;
else1:
    n = n + i;
end1:
```
int main(void) {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}

Simplified 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;
}
# 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 $a0, '\n'  # printf("\n");
    li $v0, 11
    syscall
    addi $t0, $t0, 1  # i++;
    j loop  # goto loop;
end:  # return 0
    li $v0, 0
    jr $ra

source code for print10.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;
    printf("Even\n");
    goto end;

odd:    printf("Odd\n");
end:     
    return 0;
}
```

source code for odd_even.c
source code for odd_even.simple.c
# 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

https://www.cse.unsw.edu.au/~cs1521/21T2/
Odd or Even: MIPS

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"
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

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
# 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 sum_100_squares.s
Sum 100 Squares: MIPS

```assembly
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