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>(X«2)</td>
</tr>
<tr>
<td>jal label</td>
<td>ra = pc + 4; pc = pc &amp; 0xF0000000</td>
<td>(X«2)</td>
</tr>
<tr>
<td>jr $r_s</td>
<td>pc = $r_s</td>
<td></td>
</tr>
<tr>
<td>jalr $r_s</td>
<td>ra = pc + 4; pc = $r_s</td>
<td></td>
</tr>
</tbody>
</table>

- Jump instructions **unconditionally** transfer execution to a new location
  - In other word, jump instructions change the pc (program counter)
- For **j label** and **jal label** mipsy calculates correct value for \( X \) from location of \( label \) in code
- **jal** & **jalr** set $ra ($31) to address of the next instruction
  - Call to function \( f \) implemented by **jal f**
  - Return can then be implemented with **jr $ra**
- **jr** & **jalr** can be used with any register
  - Used to implement function pointer derefencing in C, and methods in object-oriented languages

https://www.cse.unsw.edu.au/~cs1521/22T3/COMP1521%2022T3%20—%20MIPS%20Control
Branch Instruction

- **b label**  \( \text{pc += I} \ll 2 \)  pseudo-instruction
- **beq**  \( r_s, r_t, label \)  \( \text{if } (r_s == r_t) \text{ pc += I} \ll 2 \)
- **bne**  \( r_s, r_t, label \)  \( \text{if } (r_s != r_t) \text{ pc += I} \ll 2 \)
- **ble**  \( r_s, r_t, label \)  \( \text{if } (r_s \leq r_t) \text{ pc += I} \ll 2 \)
- **bgt**  \( r_s, r_t, label \)  \( \text{if } (r_s > r_t) \text{ pc += I} \ll 2 \)
- **blt**  \( r_s, r_t, label \)  \( \text{if } (r_s < r_t) \text{ pc += I} \ll 2 \)
- **bge**  \( r_s, r_t, label \)  \( \text{if } (r_s \geq r_t) \text{ pc += I} \ll 2 \)
- **blez**  \( r_s, label \)  \( \text{if } (r_s \leq 0) \text{ pc += I} \ll 2 \)
- **bgtz**  \( r_s, label \)  \( \text{if } (r_s > 0) \text{ pc += I} \ll 2 \)
- **bltz**  \( r_s, label \)  \( \text{if } (r_s < 0) \text{ pc += I} \ll 2 \)
- **bgez**  \( r_s, label \)  \( \text{if } (r_s \geq 0) \text{ pc += I} \ll 2 \)
- **bnez**  \( r_s, label \)  \( \text{if } (r_s > 0) \text{ pc += I} \ll 2 \)
- **beqz**  \( r_s, label \)  \( \text{if } (r_s > 0) \text{ pc += I} \ll 2 \)

**branch instruction conditionally** transfer execution to a new location (except b is unconditional)

- **mipsy** will calculate correct value for I from location of label in code
- **mipsy** allows second operand \((r_t)\) to be replaced by a constant (fine to use in COMP1521)
### 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, 42, label</code></td>
<td><code>beq $at, $0, label</code></td>
</tr>
<tr>
<td><code>beqz $t3, label</code></td>
<td><code>addi $at, $zero, 42</code></td>
</tr>
<tr>
<td><code>bne $t4, label</code></td>
<td><code>slt $at, $t1, $at</code></td>
</tr>
<tr>
<td><code>b label</code></td>
<td><code>bne $t4, $0, label</code></td>
</tr>
</tbody>
</table>

---

https://www.cse.unsw.edu.au/~cs1521/22T3/ COMP1521 22T3 — MIPS Control
Branch versus Jump

- Jump instructions are unconditional
- Branch instructions are conditional and can implement if and while
  - Except `b label` which has same effect as `j label`
  - You can use either
- `jal` and `jr` instructions provide a simple function call & return implementations
  - No equivalent branch instructions
- Branch instruction encode a 16-bit relative offset
  - Target (label) must be within -32768..32767 instructions
  - Not a problem in COMP1521 - we write small programs
- Jump instruction encode a 28-bit value
  - Allows jumps to be used for targets (labels) further away
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:
```

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;
}
```
```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 $a0, '\n'  # printf("%c", '\n');
    li $v0, 11
    syscall
    addi $t0, $t0, 1  # i++;
    b loop  # goto loop;

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

source code for print10.s

https://www.cse.unsw.edu.au/~cs1521/22T3/
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;
  b loop
end:

source code for sum_100_squares.s
end:

    move $a0, $t0       # printf("%d", sum);
    li    $v0, 1

syscall
    li    $a0, '\n'    # printf("%c", '\n');
    li    $v0, 11
syscall
    li    $v0, 0       # return 0

jr    $ra
**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`
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*
### 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:
```
Odd or Even: C to simplified C

C

```c
int main(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    if (x % 2 == 0) {
        printf("Even\n");
    } else {
        printf("Odd\n");
    }
    return 0;
}
```

source code for odd_even.c

Simplified C

```c
int main(void) {
    int x, v0;
    printf("Enter a number: ");
    scanf("%d", &x);
    v0 = x % 2;
    if (v0 == 1) goto odd;
    printf("Even\n");
    goto end;

odd:
    printf("Odd\n");

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

source code for odd_even.simple.c

https://www.cse.unsw.edu.au/~cs1521/22T3/  COMP1521 22T3 — MIPS Control
# 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
    rem $t0, $v0, 2  # if (x % 2 == 0) {
    beq $t0, 1, odd
    la $a0, string1  # printf("Even\n");
    li $v0, 4
    syscall
    b end

source code for odd_even.s
```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"
```

source code for odd_even.s

https://www.cse.unsw.edu.au/~cs1521/22T3/
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:
```
int main(void) {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}

Source code for print10.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;
}

Source code for print10.simple.c
# 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("%c", '\n');
    li $v0, 11
    syscall
    addi $t0, $t0, 1
    # i++;
    b loop
    # goto loop;
end:
    li $v0, 0
    # return 0
    jr $ra

source code for print10.s
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 $1^2 + 2^2 + \ldots + 99^2 + 100^2$
# 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 = $i \times i$;
    add $t0$, $t0$, $t2$  # sum = sum + square;
    addi $t1$, $t1$, 1  # i = i + 1;
    b loop

derived from sum_100_squares.s

https://www.cse.unsw.edu.au/~cs1521/22T3/
```
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
### 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

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

**MIPS**

```mips
# assume i in $t0
# assume n in $t1

blt $t0, 0, else1
bge $t1, 42, else1
sub $t1, $t1, $t0
j   end1
else1:
    add $t1, $t1, $t0
end1:
```
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
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

can be written as:

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