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&lt;2) 000010XXXXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>jal label</td>
<td>ra = pc + 4; pc = pc &amp; 0xF0000000</td>
<td>(X&lt;2) 000011XXXXXXXXXXXXXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>jr r&lt;sub&gt;...&lt;/sub&gt;</td>
<td>pc = r&lt;sub&gt;...&lt;/sub&gt; 000000ssss0000000000000000001000</td>
<td></td>
</tr>
<tr>
<td>jalr r&lt;sub&gt;...&lt;/sub&gt;</td>
<td>ra = pc + 4; pc = r&lt;sub&gt;...&lt;/sub&gt; 000000ssss0000000000000000001001</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

Branch Instructions

<table>
<thead>
<tr>
<th>b label</th>
<th>pc += I&lt;sub&gt;»&lt;/sub&gt;2 pseudo-instruction</th>
</tr>
</thead>
</table>
| beq r<sub>...</sub>, r<sub>...</sub>, label | 000100ssssstttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt...
**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</td>
</tr>
<tr>
<td>blt $t1, 42, label</td>
<td>beq $at, $0, label</td>
</tr>
<tr>
<td>beqz $t3, label</td>
<td>addi $at, $zero, 42</td>
</tr>
<tr>
<td>bnez $t4, label</td>
<td>slt $at, $t1, $at</td>
</tr>
<tr>
<td>b label</td>
<td>beqz $t3, $0, label</td>
</tr>
<tr>
<td></td>
<td>bnez $t4, $0, label</td>
</tr>
<tr>
<td></td>
<td>beq $0, $0, label</td>
</tr>
</tbody>
</table>

**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 -32,768..32,767 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.

**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:
```
**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;
        printf("%d", i);
        i++;
        goto loop;
    end:
        return 0;
}
```

**Printing First 10 Integers: MIPS**

```assembly
main:
        li $t0, 1
    loop:
        bgt $t0, 10, end
        move $a0, $t0
        li $v0, 1
        syscall
        li $a0, '\n'
        li $v0, 11
        syscall
        addi $t0, $t0, 1
        b loop
    end:
        li $v0, 0
        jr $ra
```

**Sum 100 Squares: C to simplified C**

C

```c
int main(void) {
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i * i;
    }
    printf("%d\n", sum);
    return 0;
}
```

Simplified C

```c
int main(void) {
    int i;
    loop_i_to_100__init:
        int i;
        loop_i_to_100__cond:
            if (i > 100) goto end;
            printf("%d", i);
            i++;
            goto loop_i_to_100__cond;
    loop_i_to_100__end:
        printf("%d", sum);
        return 0;
    }
```

**Sum 100 Squares: MIPS**

```assembly
main:
        li $t0, 1
        li $t1, 0
    loop_i_to_100__init:
        addi $t0, $t0, 1
        b loop_i_to_100__cond
    loop_i_to_100__cond:
        li $t0, 1
        li $t1, 100
        bgt $t0, $t1, end
        li $a0, $t0
        li $v0, 1
        syscall
        li $a0, $t0
        li $v0, 11
        syscall
        addi $t0, $t0, 1
        b loop_i_to_100__cond
    loop_i_to_100__end:
        li $v0, 0
        jr $ra
```

source code for print10.c

source code for print10.simple.c

source code for print10.s

source code for sum_100_squares.c

source code for sum_100_squares.simple.c

source code for sum_100_squares.s
# Calculate 1*1 + 2*2 + ... + 99*99 + 100*100
# Written by: Abiram Nadarajah <abiramn@cse.unsw.edu.au>
# Written as a COMP1521 lecture example

UPPER_BOUND = 100

main:
    # Locals:
    # - $t0: int sum
    # - $t1: int i
    # - $t2: temporary value
    li $t0, 0  # int sum = 0;

loop_i_to_100__init:
    li $t1, 1  # int i = 0;

loop_i_to_100__cond:
    bgt $t1, UPPER_BOUND, loop_i_to_100__end  # while (i < UPPER_BOUND) {

loop_i_to_100__body:
    mul $t2, $t1, $t1  # sum = (i * i) +
    add $t0, $t0, $t2  # ... 0;

    loop_i_to_100__step:
        addi $t0, $t0, 1  # i++;
        b loop_i_to_100__cond  # }

loop_i_to_100__end:
    li $v0, 1  # syscall 1: print_int
    move $a0, $t0  #
    syscall  # printf("%d", sum);
    li $v0, 11  # syscall 11: print_char
    li $a0, '\n'  #
    syscall  # putchar('
');
    li $v0, 0  # return 0;
    jr $ra  # return 0;

---

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
**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

```c
# 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 n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (n % 2 == 0) {
        printf("even\n");
    } else {
        printf("odd\n");
    }
    return 0;
}
```

### Simplified C

```c
int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (n % 2 != 0) goto n_mod_2_ne_0;
    printf("even\n");
goto epilogue;
n_mod_2_ne_0:
    printf("odd\n");
    epilogue:
    return 0;
}
```

**Odd or Even: MIPS**

```c
# Print out whether a value is odd or even.
# Written by: Abiram Nadarajah <abiram@cse.unsw.edu.au>
# Written as a COMP1521 lecture example

.text
main:
    # Locals:
    # - $t0: int n
    # - $t1: n % 2
    li $v0, 4       # syscall 4: print_string
    la $a0, prompt_msg       # printf("Enter a number: ");
    syscall
    li $v0, 5       # syscall 5: read_int
    syscall
    move $t0, $v0    # scanf("%d", &n);
    rem $t1, $t0, 2   # if ((n % 2)
    bnez $t1, n_mod_2_ne_0  # != 0) goto n_mod_2_ne_0;
```

source code for odd_even.s
Odd or Even: MIPS

```assembly
bnez $t1, n_mod_2_ne_0   # != 0) goto n_mod_2_ne_0;
li $v0, 4       # syscall 4: print_string
la $a0, even_msg   #
syscall           # printf("even\\n");
b   epilogue        # goto epilogue;
epilogue:
   li $v0, 4       # syscall 4: print_string
   la $a0, odd_msg  #
syscall           # printf("odd\\n");

.n_mod_2_ne_0:
li $v0, 4       # syscall 4: print_string
la $a0, even_msg   #
syscall           # printf("even\\n");
```

```
prompt_msg:  
.asciiz "Enter a number: 

even_msg:  
.asciiz "even\\n"
odd_msg:  
.asciiz "odd\\n"
```

Loops — while from C to Simplified C

**Standard C**
```
i = 0;
n = 0;
while (i < 5) {
    n = n + i;
i++;
}
```

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

**MIPS**
```
li $t0, 0       # i in $t0
li $t1, 0       # n in $t1
loop:
bge $t0, 5, end
addi $t0, $t1, $t0
addi $t0, $t0, 1
j loop
end:
```
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 = 1;
    loop:
        if (i > 10) goto end;
        printf("%d", i);
        printf("\n");
        i++;
        goto loop;
    end:
        return 0;
}
```

source code for print10.c
source code for print10.simple.c

Printing First 10 Integers: MIPS

```mips
# print integers 1..10 one per line
main:
    li $t0, 1  // in register $t0
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
```

source code for print10.s

Sum 100 Squares: C to simplified C

C
```c
int main(void) {
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i * i;
    }
    printf("%d\n", sum);
    return 0;
}
```

Simplified C
```c
int main(void) {
    int i = 0;
    loop_i_to_100__init:
        int i = 0;
        loop_i_to_100__cond:
            if (i > UPPER_BOUND) goto end;
            printf("%d", i);
            printf("\n");
            i++;
            goto loop_i_to_100__step:
            end:
                return 0;
```

source code for sum_100_squares.c
source code for sum_100_squares.simple.c
# Calculate 1*1 + 2*2 + ... + 99*99 + 100*100
# Written by: Abiram Nadarajah <abiram@cse.unsw.edu.au>
# Written as a COMP1521 lecture example

UPPER_BOUND = 100

main:
    # Locals:
    # - $t0: int sum
    # - $t1: int i
    # - $t2: temporary value
    li $t0, 0  # int sum = 0;
loop_i_to_100__init:
    li $t1, 1  # int i = 0;
loop_i_to_100__cond:
    bgt $t1, UPPER_BOUND, loop_i_to_100__end  # while (i < UPPER_BOUND) {
loop_i_to_100__body:
    mul $t2, $t1, $t1  # sum = (i * i) +
    add $t0, $t0, $t2 # ... 0;
loop_i_to_100__step:
    addi $t0, $t0, 1 # i++;
    b loop_i_to_100__cond # }
loop_i_to_100__end:
    li $v0, 1  # syscall 1: print_int
    move $a0, $t0 #
    syscall # printf("%d", sum);
    li $v0, 11 # syscall 11: print_char
    li $a0, 'n' #
    syscall # putchar(
'\n');
    li $v0, 0  # return 0;
    jr $ra

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;
```

else1:
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:
```
The break statement

Sometimes it is useful to exit from the middle of a loop

- **break** allows you to check a condition mid-loop and quit

```c
// read up to 100 characters
// stop if the next character is '!
while (i <= 100) {
    int ch = getchar();
    if (ch == '!') break;
    putchar(ch);
}
```

The continue statement

Sometimes it is useful to go to next iteration and skip rest of loop

- **continue** allows you to go to next iteration from mid-loop

```c
// iterate over integers 1..100
// skip every multiple of three
for (i = 1; i <= 100; i++) {
    if (i % 3 == 0) continue;
    printf(%d\n", i);
}
```

Continue can simplify loops

```c
while (Condition) {
    some_code_1
    if (Condition1) {
        some_code_2
        if (Condition2) {
            some_code_3
        }
    }
}
```
Side Topic: C do/while

C has a different while loop - do/while (post-test).

- loop condition checked at bottom of loop - always executed once
- many programmers do not use it

```
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

can be written as:

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