Translating C code to MIPS

why do it
  C is relatively simple, close to the machine
  C can act as pseudocode for assembler program
  gives some insight into what compiler needs to do
  what's under the hood
    do you need to know how the carburetor works to drive your car?
    does your mechanic need to know?
Register conventions

**register conventions and mnemonics**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>hardwired 0 value</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>used by assembler (pseudo-instructions)</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-1</td>
<td>subroutine return value</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-3</td>
<td>arguments: subroutine parameter value</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-7</td>
<td>temp: can be used by subroutine without saving</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-7</td>
<td>saved: must be saved and restored by subroutine</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-9</td>
<td>temp</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0-1</td>
<td>kernel: interrupt/trap handler</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer (static or extern variables)</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address for subroutine</td>
</tr>
</tbody>
</table>

**hidden registers**

- **PC**, the program counter, which stores the current address of the instruction being executed
- **IR**, which stores the instruction being executed
**Arithmetic expression**

simple arithmetic expression, assignment

```c
int f, g, h, i, j;
f = (g + h) - (i + j);
```

assume variables are assigned to $s0, $s1, $s2, $s3, $s4 respectively

```
add $s0, $s1, $s2  # $s0 = g + h
add $s1, $s3, $s4  # $s1 = i + j
sub $s0, $s0, $s1  # f = (g + h) - (i + j)
```
**Conditional: if**

Simple *if* statement

```c
if ( i == j )
    i++ ;
    j-- ;
```

In C: if condition is true, we "fall through" to execute the statement
  if false, jump to next

In assembly, we jump if condition is true
  need to negate the condition

Assuming $s1 stores i and $s2 stores j:

```assembly
bne $s1, $s2, L1  # branch if !( i == j )
addi $s1, $s1, 1  # i++
L1: addi $s2, $s2, -1  # j--
```
Conditional: if-else

if-else

```c
if ( i == j )
    $s1 i
    i++ ;
else
    $s2 j
    j-- ;
j += i ;
```

As before, if the condition is false, we want to jump.

```c
bne $s1, $s2, ELSE   # branch if !( i == j )
addi $s1, $s1, 1      #    i++
ELSE: addi $s2, $s2, -1     # else j--
    add $s2, $s2, $s1     #    j += i
```

What's wrong with this picture?

Once we've done the if-body, we need to jump over the else-body

```c
bne $s1, $s2, ELSE   # branch if !( i == j )
addi $s1, $s1, 1      #    i++
    j NEXT                #    jump over else
ELSE: addi $s2, $s2, -1     # else j--
NEXT: add $s2, $s2, $s1    #    j += i
```
**Conditional: compound condition**

**if-else** with compound AND condition: short-circuiting

```plaintext
if ( i == j && i == k )  // if ( <cond1> && <cond2> )
    i++ ;  // if body
else
    j-- ;  // else body
j = i + k ;
```

Let `<cond1>` stand for `(i == j)` and `<cond2>` stand for `(i == k)`.

**Short-circuiting** occurs when `<cond1>` evaluates to false.

The control flow then jumps over `<cond2>` and the if-body.

If `<cond1>` evaluates to true, we also want to check `<cond2>`.

If `<cond2>` evaluates false, we again jump, this time over the if-body, and to the else-body.

If `<cond2>` is true, we fall-through to the if-body.

```plaintext
bne  $s1, $s2, ELSE   # cond1: branch if !( i == j )
bne  $s1, $s3, ELSE   # cond2: branch if !( i == k )
addi $s1, $s1, 1      # if-body: i++
j NEXT                # jump over else
ELSE: addi $s2, $s2, -1 # else-body: j--
NEXT: add $s2, $s1, $s3 # j = i + k
```
**Conditional: compound condition**

**if-else with compound OR condition: short-circuiting** 

use `<cond1>` to stand for `(i == j)` and `<cond2>` to stand for `(i == k)`.

\[
\text{if ( } <\text{cond1}> \text{ || } <\text{cond2}> \text{ )}
\]

\[
i++ ; \quad \text{// if-body}
\]

\[
\text{else}
\]

\[
j-- ; \quad \text{// else-body}
\]

\[
j = i + k ;
\]

**Short-circuiting** occurs when `<cond1>` evaluates to true

If `<cond1>` is false, we also want to check `<cond2>`

- If `<cond2>` is false, we now jump to the else-body.
- If `<cond2>` is true, we fall through to the if-body.

\[
\text{beq } s1, s2, \text{ IF} \quad \# \text{ cond1: branch if ( i == j )}
\]

\[
\text{bne } s1, s3, \text{ ELSE} \quad \# \text{ cond2: branch if ! ( i == k )}
\]

\[
\text{IF: addi } s1, s1, 1 \quad \# \text{ if-body: i++}
\]

\[
\text{j NEXT} \quad \# \text{ jump over else}
\]

\[
\text{ELSE: addi } s2, s2, -1 \quad \# \text{ else-body: j--}
\]

\[
\text{NEXT: add } s2, s1, s3 \quad \# \text{ j = i + k}
\]
Conditional: switch

```c
switch( i ) {
    case 1: i++ ; // falls through
    case 2: i += 2 ; $s1 i
                break; $s4 temp
    case 3: i += 3 ;
}
```

```assembly
addi $s4, $zero, 1     # case 1: set temp to 1
bne $s1, $s4, C2_COND # false: branch to case 2 cond
    j C1_BODY          # true: branch to case 1 body
C2_COND: addi $s4, $zero, 2 # case 2: set temp to 2
    bne $s1, $s4, C3_COND # false: branch to case 3 cond
    j C2_BODY          # true: branch to case 2 body
C3_COND: addi $s4, $zero, 3 # case 3: set temp to 3
    bne $s1, $s4, EXIT  # false: branch to exit
    j C3_BODY          # true: branch to case 3 body
C1_BODY: addi $s1, $s1, 1 # case 1 body: i++
C2_BODY: addi $s1, $s1, 2 # case 2 body: i += 2
    j EXIT            # break
C3_BODY: addi $s1, $s1, 3 # case 3 body: i += 3
EXIT:
```
Loops: while

If statement uses branch instruction.

What about loops?

Example:

```c
while ( <cond> ) {
   <while-body>
}
```

If condition is true, execute body and go back, otherwise do next statement.

```c
while ( i < j ) {
   k++ ;
   i = i * 2 ;
}
```

```asm
L1:  bge  $s1, $s2, DONE    # branch if ! ( i < j )
     addi $s3, $s3, 1     #     k++
     add  $s1, $s1, $s1    #     i = i * 2
     j    L1             # jump back to top of loop
DONE:
```

$s1  i
$s2  j
$s3  k
**Loops: for**

for ( <init> ; <cond> ; <update> ) {
    <for-body>
}

Equivalent while loop:

<init>;             <init>;
while ( <cond> ) {   L1: if ( <cond> ) {
    <for-body>       <for-body>
    <for-body>       <update>
    <update>         goto L1 ;
}
DONE:
Array: C

Problem: Given an array of int, calculate the sum of:
   all the elements in the array
   all the positive elements in the array
   all the negative elements in the array

main () {
    int i, size = 10, sum, pos, neg;
    int arr[10] = {12, -1, 8, 0, 6, 85, -74, 23, 99, -30};

    sum = 0; pos = 0; neg = 0;
    for (i = 0; i < size; i++) {
        sum += arr[i];
        if (arr[i] > 0)  
            pos += arr[i];
        if (arr[i] < 0)  
            neg += arr[i];
    }
    return 0;
}
Array: assembler

.text
.globl main

main:

la $s0, size  # initialize registers
lw $s1, 0($s0)  # $s1 = size
ori $s2, $0, 0  # $s2 = sum
ori $s3, $0, 0  # $s3 = pos
ori $s4, $0, 0  # $s4 = neg

# <init>
ori $s5, $0, 0  # $s5 = i
la $s6, arr  # $s6 = &arr

# if (<cond>)
L1:  bge $s5, $s1, DONE

#    <for-body>
lw $s7, 0($s6)  # $s7 = arr[i]
addu $s2, $s2, $s7  # sum += arr[i]
blez $s7, NEG  # if ! (arr[i] > 0)
addu $s3, $s3, $s7  #   pos += arr[i];
j UPDATE                   # goto UPDATE
NEG: bgez $s7, UPDATE      # if ! (arr[i] < 0)
    addu $s4, $s4, $s7    #    neg += arr[i];

UPDATE:                    #    <update>
    addi $s5, $s5, 1      # i++
    addi $s6, $s6, 4      # move array pointer
    j L1                # goto L1

DONE:
    # initialize data
    .data
    size: .word 10
    arr: .word 12, -1, 8, 0, 6, 85, -74, 23, 99, -30