Computer Organization and Architecture

MIPS Architecture
Lecture: 05

Conversion of C Programs to MIPS Assembly Language

Fall, 2020

Register conventions

Number	Name	Use
0	`$zero`	hardwired 0 value
1	`$at`	used by assembler (pseudo-instructions)
2-3	`$v0-1`	subroutine return value
4-7	`$a0-3`	arguments: subroutine parameter value
8-15	`$t0-7`	temp: can be used by subroutine without saving
16-23	`$s0-7`	saved: must be saved and restored by subroutine
24-25	`$t8-9`	temp
26-27	`$k0-1`	kernel: interrupt/trap handler
28	`$gp`	global pointer (static or extern variables)
29	`$sp`	stack pointer
30	`$fp`	frame pointer
31	`$ra`	return address for subroutine
	`Hi, Lo`	used in multiplication (provide 64 bits for result)

Hidden registers
   PC, the program counter, which stores the current **address** of the instruction being
       executed

Arithmetic expression

simple arithmetic expression, assignment

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

$s0	(g + h) - (i + j)
$s1	i + j
$s2	h
$s3	i
$s4	j

```
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

```
        if ( i == j )                                    $s1    i
           i++ ;                                          $s2    j
        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:

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

Conditional: if-else

if-else

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

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

```
        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
```

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

```
        bne   $s1, $s2, ELSE  # branch if !( i == j )
        addi $s1, $s1,        #     i++
        1    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

```
if ( i == j && i == k )   // if ( <cond1> && <cond2> )
  i++ ;                    // if body
else                                               $s1  | i
  j-- ;                    // else body            $s2  | j
j = i + k ;                                        $s3  | 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.

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

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)`.

    `if ( <cond1> || <cond2> )`

```
        i++ ;              //    if-body              $s1 i
     else                                             $s2 j
       j-- ;              //    else-body             $s3 k
     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.

```
            beq  $s1, $s2, IF      # cond1: branch if ( i == j )
                                   #        Notice branch on TRUE
            bne  $s1, $s3, ELSE    # cond2: branch if ! ( i == k )
      IF:   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: switch

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


          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:

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

$s1	i
$s2	j
$s3	k

```
L1: if ( <cond> ) {
        <while-body>
        goto L1 ;
    }
```

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

```
    while ( i < j ) {          L1: if ( i < j ) {
        k++ ;                          k++ ;
        i = i * 2 ;                    i = i * 2 ;
    }                                  goto L1 ;
                                   }
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:
```

Array: C

all the elements in the array
all the positive elements in the array all
the negative elements in the array

```c
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]
         ble  $s7, NEG         # if ! (arr[i] > 0)
         z    $s3, $s3, $s7    #      pos += arr[i];
         add
         u
```

```
        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
```

Conversion of C Programs to MIPS Assembly Language

//Program to print numbers from 1 to 5

```cpp
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    // while loop from 1 to 5
    while (i <= 5) {
        cout << i << " ";
        ++i;
    }
    return 0;
}
```

```cpp
// C++ program to find the sum of first n natural numbers
// positive integers such as 1,2,3,...n are known as natural numbers

#include <iostream>
using namespace std;

int main() {
    int num, sum;
    sum = 0;
    cout << "Enter a positive integer: ";
    cin >> num;
    for (int count = 1; count <= num; ++count) {
        sum += count;
    }
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Display Multiplication table up to 10

```cpp
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;

    for (int i = 1; i <= 10; ++i) {
        cout << n << " * " << i << " = " << n * i << endl;
    }

    return 0;
}
```