Assembly Programming

Conditional Statements and Loops

In MIPS, we create conditional constructs with conditional branches

High-level code

```
if (i == j)
   f = g + h;

f = f – i;
```

MIPS assembly

```
# $s0 = f, $s1 = g
# $s2 = h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Branch not equal

Compares two values ($s3=i, $s4=j) and jumps if they are different

If-Else Statement

- We use the unconditional ~~branch~~ jump (i.e., j) to skip the "else" subtask if the "if" subtask is the correct one

Conditional branch and unconditional jump

High-level code

```
if (i == j)
  f = g + h;
else
  f = f – i;
```

MIPS assembly

```
# $s0 = f, $s1 = g,
# $s2 = h
# $s3 = i, $s4 = j

      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

1. Compare two values ($s3=i, $s4=j) and, if they are different, jump to L1, to execute the "else" subtask

2. Jump to done, after executing the "if" subtask

While Loop

- As in LC-3, the conditional branch (i.e., beq) checks the condition and the unconditional ~~branch~~ jump (i.e., j) jumps to the beginning of the loop

High-level code

```
// determines the power
// of 2 equal to 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly

```
# $s0 = pow, $s1 = x

      addi $s0, $0, 1
      add  $s1, $0, $0
      addi $t0, $0, 128
while: beq  $s0, $t0, done
      sll  $s0, $s0, 1
      addi $s1, $s1, 1
      j    while
done:
```

init

左移操作 每向左移动一位 数值就乘以2

jump

1. Conditional branch to check if the condition still holds

2. Unconditional ~~branch~~ jump to the beginning of the loop

For Loop

- The implementation of the "for" loop is similar to the "while" loop

High-level code

```
// add the numbers from 0 to 9

int sum = 0;
int i;
for (i = 0; i != 10; i = i+1)
{
  sum = sum + i;
}
```

MIPS assembly

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

结果效果相同
但使用的指令不同

1. Conditional branch to check if the condition still holds

2. Unconditional branch to the beginning of the loop

For Loop Using SLT

- We use slt (i.e., set less than) for the "less than" comparison

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i*2)
{
  sum = sum + i;
}
```

MIPS assembly

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

Initialize sum and i

Set less than
`$t1 = $s0 < $t0 ? 1:0`

Shift left logical

Array

Access an array requires loading the base address into a register

In MIPS, this is something we can not do with only one single immediate operation.

Load upper immediate + OR immediate

eg lui $s0, 0x1234
ori $s0, $s0, 0x8000

result: load the address 0x12348000

- We first load the base address of the array into a register (e.g., $s0) using lui and ori

High-level code

```
int array[5];

array[0] = array[0] * 2;

array[1] = array[1] * 2;
```

MIPS assembly

```
# array base address = $s0
# Initialize $s0 to 0x12348000
lui  $s0, 0x1234
ori  $s0, $s0, 0x8000

lw   $t1, 0($s0)
sll  $t1, $t1, 1
sw   $t1, 0($s0)
lw   $t1, 4($s0)
sll  $t1, $t1, 1
sw   $t1, 4($s0)
```

'lw': load word: load one word (32 bits / 8 bytes) from memory to register

syntax: lw reg offset(base)

从内存地址 base + offset 处加载一个 word 到 reg

sw : store word: store one word from reg to memory

syntax: sw reg offset(base)

将reg内容存储到内存地址base offset处

Function Calls

Conventions:

Caller:
- passes arguments
- jumps to calee

Callee:
- performs the procedure
- returns the result to caller
- returns to the point of call
- must not overwrite registers or memory needed by the caller.

Conventions In MIPS:

Call procedure: Jump and Link (jal)
Return from procedure: Jump register (jr)
Argument values: $a0 ~ $a3
Return value: $v0

Code Example:

High-level code

```
int main()
{
  int y;
  ...
  // 4 arguments
  y = diffofsums(2, 3, 4, 5);
  ...
}

int diffofsums(int f, int g,
  int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  // return value
  return result;
}
```

MIPS assembly

```
# $s0 = y
main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # call procedure
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result=(f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Argument values

Return value

Caller

Return address

Callee

For this snippet MIPS assembly, the `$t0`, `$t1`, `$s0` in `callee` overwrite the registers in `caller`

So we should use `stack` to avoid this situation

MIPS assembly

```
diffofsums:
  addi $sp, $sp, -12  # allocate space on stack to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result=(f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)    # restore $t1 from stack
  lw   $t0, 4($sp)    # restore $t0 from stack
  lw   $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr  $ra             # return to caller
```

- Saving and restoring all registers requires a lot of effort
- In MIPS, there is a convention about temporary registers (i.e., $t0-$t9): There is no need to save them

MIPS assembly

```
diffofsums:
  addi $sp, $sp, -4   # allocate space on stack to store 1 register
  sw   $s0, 0($sp)    # save $s0 on stack


  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result=(f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0


  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr  $ra             # return to caller
```

- Temporary registers $t0-$t9 are nonpreserved registers. They are not saved, thus, they can be overwritten by the function
- Registers $s0-$s7 are preserved (saved; callee-saved) registers