Implementing Algorithms in MIPS Assembly

(Part 2)

February 6–11, 2013

Outline

- Reading strings into memory
- Jumps and conditional branches
- Branching control structures
  - If-then-else and if-then statements
- Looping control structures
  - Do-while, while, and for loops
  - Break and continue, indefinite loops
- Arrays
  - For-each loop
  - Switch statement

Reading a string from the user

Step 1: Reserve space for the string in the data segment

- use the `.space` directive
- argument is the number of **bytes** (characters) to reserve
  - remember null-terminating character!
  - should be a **multiple of 4**, to preserve word boundaries

Step 2: Read the string in your program

- use the "read string" system call (8)
- argument #1, `$a0` = address of input buffer
  - load label address with `la`
- argument #2, `$a1` = size of input buffer

(MARS demo: Parrot.asm)

Outline

Control structures in assembly

How control structures are implemented in assembly

- insert labels in text segment
- **jump** or conditionally **branch** to labels

Your only primitive control structures are goto and if-goto!

Jump instructions (unconditional branches)

Jump	`j`	`label`	`# goto label`
Jump register	`jr`	`$t1`	`# goto the address in $t1`

Conditional branching

```
# Basic instructions
beq  $t1, $t2, label     # if ($t1 == $t2) goto label
bne  $t1, $t2, label     # if ($t1 != $t2) goto label

bgez $t1, label          # if ($t1 >= 0) goto label
bgtz $t1, label          # if ($t1 >  0) goto label
blez $t1, label          # if ($t1 <= 0) goto label
bltz $t1, label          # if ($t1 <  0) goto label

# Macro instructions
beqz $t1, label          # if ($t1 == 0) goto label
bnez $t1, label          # if ($t1 != 0) goto label

beq  $t1, 123, label     # if ($t1 == 123) goto label
bne  $t1, 123, label     # if ($t1 != 123) goto label

bge  $t1, $t2, label     # if ($t1 >= $t2) goto label
bgt  $t1, $t2, label     # if ($t1 >  $t2) goto label
bge  $t1, 123, label     # if ($t1 >= 123) goto label
bgt  $t1, 123, label     # if ($t1 >  123) goto label
```

and similarly for `ble` and `blt`

Outline

If-then-else statement

Structure of an if-then-else statement

if (condition**) {**
  then-block (execute if condition is true)
} else {
  else-block (execute if condition is false)
}

Sketch of translation to assembly

```
    (translation of condition, ending in branch to thenLabel)
    (translation of else-block)
    j endLabel
thenLabel:
    (translation of then-block)
endLabel:
    (rest of program)
```

If-then-else statement

Example

```
# Pseudocode:
#   if (a < b + 3)
#     a = a + 1
#   else
#     a = a + 2
#   b = b + a
# Register mappings:
#   a: $t0, b: $t1

        addi $t2, $t1, 3      # tmp = b + 3
        blt  $t0, $t2, then   # if (a < tmp)
        addi $t0, $t0, 2      # (else case) a = a + 2
        j    end
then:   addi $t0, $t0, 1      # (then case) a = a + 1
end:    add  $t1, $t1, $t0    # b = b + a
```

If-then statement

Two strategies for if statements without else blocks:

1. use same strategy as if-then-else
2. complement condition (saves a branch on then-case)

Example of first strategy

```
# Pseudocode:
#   if (a < b + 3)
#     a = a + 1
#   b = b + a
# Register mappings:
#   a: $t0, b: $t1

         addi $t2, $t1, 3       # tmp = b + 3
         blt  $t0, $t2, then    # if (a < tmp)
         j    end
then:    addi $t0, $t0, 1       # (then case) a = a + 1
end:     add  $t1, $t1, $t0     # b = b + a
```

If-then statement

Two strategies for if statements without else blocks:

1. use same strategy as if-then-else
2. complement condition (saves a branch on then-case)

Example of second strategy

```
# Pseudocode:
#   if (a < b + 3)
#     a + 1
#   b = b + a
# Register mappings:
#   a: $t0, b: $t1

        addi $t2, $t1, 3      # tmp = b + 3
        bge  $t0, $t2, end    # if (a >= tmp) goto end
        addi $t0, $t0, 1      # a + 1
end:    add  $t1, $t1, $t0    # b = b + a
```

Outline

Do-while loop

Structure of a do-while loop

```
do {
  loop-body
} while (condition);
```

Sketch of translation to assembly

`loopLabel:`

(translation of loop-body)
(translation of condition, ending in branch to **`loopLabel`**)
(rest of program)

Do-while loop

Example

```
# Pseudocode:
#   do {
#     a = a + 3
#   } while (a < b*2);
# Register mappings:
#   a: $t0, b: $t1

loop:    addi $t0, $t0, 3      # (loop) a = a + 3
         mul  $t2, $t1, 2      # tmp = b*2
         blt  $t0, $t2, loop   # if (a < tmp) goto loop
```

Optimization: Extract loop invariants

```
         mul  $t2, $t1, 2      # tmp = b*2
loop:    addi $t0, $t0, 3      # (loop) a = a + 3
         blt  $t0, $t2, loop   # if (a >= tmp) goto loop
```

While loop

Structure of a while loop

```
while (condition) {
  loop-body
}
```

Like if-then, two strategies:

1. translate condition as usual, branch over jump to end
2. complement condition and branch to end

While loop

Strategy 1: Condition branches over jump to end

Sketch of translation to assembly

`loopLabel:`

(translation of condition, ending in branch to **`bodyLabel`**)

`j endLabel`

`bodyLabel:`

(translation of loop-body)

`j loopLabel`

`endLabel:`

(rest of program)

While loop

Strategy 2: Complement of condition branches to end

Sketch of translation to assembly

`loopLabel:`

(complement of condition, ending in branch to **`endLabel`**)

(translation of loop-body)

`j loopLabel`

`endLabel:`

(rest of program)

While loop

```
# Pseudocode: while (a <= c + 4) { a = a + 3 }
#             b = b + a
# Registers:  a: $t0, b: $t1, c: $t2
```

Strategy 1: Condition branches over jump to end

```
        addi $t3, $t2, 4      # tmp = c + 4
loop:   ble  $t0, $t3, body   # while (a <= tmp) goto body
        j    end              # goto end
body:   addi $t0, $t0, 3      # (in loop) a = a + 3
        j    loop             # end loop, repeat
end:    add  $t1, $t1, $t0    # b = b + a
```

Strategy 2: Complement of condition branches to end

```
        addi $t3, $t2, 4      # tmp = c + 4
loop:   bgt  $t0, $t3, end    # if (a > tmp) goto end
        addi $t0, $t0, 3      # (in loop) a = a + 3
        j    loop             # end loop, repeat
end:    add  $t1, $t1, $t0    # b = b + a
```

For loop

Structure of a for loop

```
for (initialize; condition; update) {
  loop-body
}
```

Two step strategy:

1. translate into equivalent pseudocode using a while loop
2. translate that into assembly

For loop

Structure of a for loop

```
for (initialize; condition; update) {
  loop-body
}
```

Equivalent program using while loop

```
initialize
while (condition) {
  loop-body
  update
}
```

Exercise

```
# Pseudocode:
#   sum = 0
#   for (i = 0; i < n; i++) {
#     sum = sum + i
#   }
# Registers: n: $t0, i: $t1, sum: $t2
```

```
# Translate to lower-level pseudocode:
#   sum = 0
#   i = 0
#   while (i < n) {
#     sum = sum + i
#     i = i + 1
#   }
        li   $t2, 0         # sum = 0
        li   $t1, 0         # i = 0
loop:   bge  $t1, $t0, end  # (start loop) if i >= n goto end
        add  $t2, $t2, $t1  # sum = sum + i
        addi $t1, $t1, 1    # i = i + 1
        j    loop           # (end loop)
end:                        # ...
```

Break and continue

In C-like languages, within loops:

- **`break`** – exit the loop
- **`continue`** – skip to the next iteration

Translation of break to assembly

```
j endLabel
```

Translation of continue to assembly

In while loop:

- `j loopLabel`

In for loop:

- Must execute update first ← gotcha! (next slide)

Translation of continue in for-loop

Sketch of for-loop, translated to assembly

```
    (translation of initialize)
loopLabel:
    (complement of condition, ending in branch to endLabel)
    (translation of loop-body)
updateLabel:          # new label added for continue
    (translation of update)
    j loopLabel
endLabel:
    (rest of program)
```

Translation of continue to assembly

```
j updateLabel
```

Translation of conditional break/continue

Common pattern: break/continue guarded by if-statement

- E.g. **`if (`**condition**`) break`**

```
# Pseudocode:
#   while (true) {
#     ...
#     if (a < b) break
#     ...
#   }
# Register mappings: a = $t0, b = $t1
```

Naive: translate if-then and break separately

```
loop:   ...                     # (begin loop)
        bge  $t0, $t1, else     # if (a < b)
        j    end                # (then branch) break
else:   ...                     # (rest of loop body)
        j    loop               # (end loop)
end:
```

Translation of conditional break/continue

Naive: translate if-then and break separately

```
loop:   ...                     # (begin loop)
        bge  $t0, $t1, else     # if (a < b)
        j    end                # (then branch) break
else:   ...                     # (rest of loop body)
        j    loop               # (end loop)
end:
```

Better: implement if-break as one conditional branch

```
loop:   ...                     # (begin loop)
        blt  $t0, $t1, end      # if (a < b) break
        ...                     # (rest of loop body)
        j    loop               # (end loop)
end:
```

Indefinite loops

Structure of an indefinite loop

`while (true) {` loop-body `}`

Trivial to implement in assembly

```
loopLabel:
    (translation of loop-body)
    j loopLabel
endLabel:  # needed for break
    (rest of program)
```

Break and continue

- **`break`** – jump or branch to **`endLabel`**
- **`continue`** – jump or branch to **`loopLabel`**

(MARS demo: Circle.asm)

Exercise

```
# Pseudocode:
#   total = 0
#   for (i = 0; i < n; i++) {
#     if (i % 5 > 2) continue
#     total += i
#   }
# Registers: total = $t0, i = $t1, n = $t2
# Note: rem $t3, $t1, 5  ==>  $t3 = $t1 % 5
```

```
        li   $t1, 0          # (init) i = 0
loop:   bge  $t1, $t2, end   # while (i < n)
        rem  $t3, $t1, 5     # tmp = i % 5
        bgt  $t3, 2, update  # if (tmp > 2) continue
        add  $t0, $t0, $t1   # total += i
update: addi $t1, $t1, 1     # (update) i++
        j    loop            # (end while)
end:                         # ...
```

Declaring arrays in the data segment (review)

Declare and initialize an array of integers

```
fibs:    .word   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

Reserve space but don't initialize

```
# save space for a 10 integer array
# or a 39 character null-terminated string
array:  .space  40
```

Argument to `.space` is number of **bytes** to reserve

Element addresses

Declaration in data segment

```
# 10 integer array or 39 character null-terminated string
array:  .space  40
```

If we interpret as integers ...

- **array**, **array+4**, **array+8**, **array+12**, **...**, **array+36**
- **lw** to move an integer from array (in memory) to a register

If we interpret as ASCII characters ...

- **array**, **array+1**, **array+2**, **array+3**, **...**, **array+36**
- **lb** to move a character from array to a register
- **lw** to move a four character chunk into a register

lw — addresses must always respect word boundaries!

Basic addressing mode

```
lw $t1, 4($t2)   # $t1 = Memory[$t2+4]
```

- `$t1` is the **destination** register
- `$t2` contains the **base address** (pointer to memory)
- `4` is the **offset** from the base address

```
sw $t1, 4($t2)   # Memory[$t2+4] = $t1
```

- `$t1` is the **source** register
- `$t2` contains the **base address** (pointer to memory)
- `4` is the **offset** from the base address

(Similarly for `lb` and `sb`)

All other data memory addressing modes are translated to this form!

Pseudo-addressing modes

Macro instructions to read/write a specific address

```
lw $t1, $t2             # $t1 = Memory[$t2]
sw $t1, $t2             # Memory[$t2] = $t1
```

Macro instructions for reading/writing with labels

```
lw $t1, label           # $t1 = Memory[label]
lw $t1, label+4         # $t1 = Memory[label+4]
lw $t1, label($t2)      # $t1 = Memory[label+$t2]
sw $t1, label           # Memory[label] = $t1
sw $t1, label+4         # Memory[label+4] = $t1
sw $t1, label($t2)      # Memory[label+$t2] = $t1
```

This leads to many different ways to iterate through arrays

For-each loop (arrays only)

Structure of a for-each loop

```
foreach (elem in array) {
  loop-body
}
```

elem and array are pseudocode-level names

- elem might map to a register
- array might map to a label

To implement, we must either:

- know the length of the array in advance
- use a marker in memory to indicate the end
  - e.g. null-terminated string

For-each loop – enumerating the elements

Strategy #1, for-loop with counter

```
# Pseudocode:
#   foreach (fib in fibs) {
#     ...
#   }
# Registers: fib = $t0, i = $t1

.data

fibs:   .word   0, 1, 1, 2, 3, 5, 8, 13, 21, 35, 55, 89, 144

.text

        li   $t1, 0             # i = 0
loop:   ...                     # (loop condition, TODO)
        lw   $t0, fibs($t1)     # fib = fibs[i]
        ...                     # (loop body)
        addi $t1, $t1, 4        # i++    <= +4
        j    loop               # (end loop)
```

For-each loop – enumerating the elements

Strategy #2, increment address

```
# Pseudocode:
#   foreach (fib in fibs) {
#     ...
#   }
# Registers: fib = $t0, addr = $t1

.data

fibs:   .word   0, 1, 1, 2, 3, 5, 8, 13, 21, 35, 55, 89, 144

.text

        li   $t1, fibs          # addr = fibs
loop:   ...                     # (loop condition, TODO)
        lw   $t0, $t1           # fib = *addr
        ...                     # (loop body)
        addi $t1, $t1, 4        # addr += 4
        j    loop               # (end loop)
```

Switch statements

Structure of a switch statement

```
switch (n) {
  (case k:   k-block)*
  default:   default-block
}
```

- n is an integer variable
- each k is an integer constant
- each k-block is a sequence of statements
  - often ends in **break**

Execution rules

- if value of k=n, execute corresponding k-block
  - keep executing subsequent blocks until **break**
- if no such k, execute default-block

Switch statements

Can implement using if-statements ...
but there's a clever strategy when all k's are in a small range

Translation strategy

1. in text segment, implement and label each k-block and the default-block, in order of switch statement
2. in data segment, declare **array of addresses** (jump table)
   - in array at position *i*, label of case-block for *i*=k
   - for "gaps" in cases, give label for default case
3. translate switch statement into an array lookup
   - check bounds of n and jump to default case if out
   - if in range, translate n to corresponding index (e.g. n*4)
4. use `jr` to jump to the address from array lookup

Switch statements

Example: Print properties of one digit number

```
# Pseudocode:
#   switch (n) {
#     case 0:
#       print("n is zero\n")
#       break
#     case 4:
#       print("n is even\n")
#     case 1:
#     case 9:
#       print("n is a square\n")
#       break
#
#
#   ... (continue in next col)
```

```
case 2:
  print("n is even\n")
case 3:
case 5:
case 7:
  print("n is prime\n")
  break
case 6:
case 8:
  print("n is even\n")
  break
default:
  print("out of range\n")
}
```

Example from: `http://en.wikipedia.org/wiki/Switch_statement`

(MARS demo: Switch.asm)