Implementing Algorithms in MIPS Assembly

(Part 1)

January 28–30, 2013

Outline

Effective documentation

Arithmetic and logical expressions

- Compositionality
- Sequentializing complex expressions
- Bitwise vs. logical operations

Documenting your code

```
# Author:      your name
# Date:        current date
# Description: high-level description of your program

.data
```

(constant and variable definitions)

```
.text

# Section 1: what this part does
# Pseudocode:
#   (algorithm in pseudocode)
# Register mappings:
#   (mapping from pseudocode variables to registers)
```

Inline comments should relate to the **pseudocode** description

(MARS demo: Circle.asm)

Just to reiterate . . . (required from here on out)

Once at top of file:

Header block

1. author name
2. date of current version
3. high-level description of your program

Once for each section:

Section block

1. section name
2. description of algorithm in pseudocode
3. mapping from pseudocode variables to registers

. . . and inline comments that relate to pseudocode

Outline

Compositionality

Main challenge

- math expressions are **compositional**
- assembly instructions are **not compositional**

Compositionality in math:

- use expressions as arguments to other expressions
- example: `a * (b + 3)`

Non-compositionality in assembly:

- can’t use instructions as arguments to other instructions
- not valid MIPS: `mult $t0 (addi $t1 3)`

Significance of compositionality (PL aside)

Compositionality is extremely **powerful** and **useful**

- leads to high expressiveness
  - can do more with less code
- leads to nice semantics
  - to understand whole: understand parts + how combined
- promotes modularity and reuse
  - extract recurring subexpressions

Pulpit: less compositional $\Rightarrow$ more compositional

assembly $\Rightarrow$ imperative (C, Java) $\Rightarrow$ functional (Haskell)

Sequentializing expressions

Goal

Find a sequence of assembly instructions that **implements** the pseudocode expression

Limited by available instructions:

- in math, add any two expressions
- in MIPS, add two registers, or a register and a constant
    - not all instructions have immediate variants
    - can use `li` to load constant into register

Limited by available registers:

- might need to swap variables in/out of memory
- (usually not a problem for us, but a real problem in practice)

Finding the right sequence of instructions

Strategy 1: Decompose expression

1. separate expression into subexpressions
   - respect grouping and operator precedence!
2. translate each subexpression and save results
3. combine results of subexpressions

(assume C-like operator precedence in pseudocode)

Example

```
# Pseudocode:
#   d = (a+b) * (c+4)
# Register mappings:
#   a: t0, b: t1, c: t2, d: t3

add  $t4, $t0, $t1     # tmp1 = a+b
addi $t5, $t2, 4       # tmp2 = c+4
mul  $t3, $t4, $t5     # d = tmp1 * tmp2
```

```
tmp1 = a+b
tmp2 = c+4
d = tmp1 * tmp2
```

Finding the right sequence of instructions

Strategy 2: Parse and translate

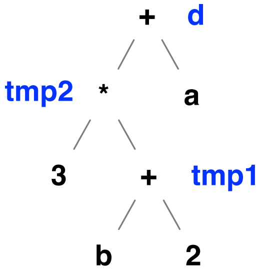
1. parse expression into **abstract syntax tree**
2. traverse tree in post-order
3. store subtree results in temp registers

This is essentially what a compiler does!

Example

```
# Pseudocode:
#   c = a + 3*(b+2)
# Register mappings:
#   a: t0, b: t1, c: t2

addi $t3, $t1, 2      # tmp1 = b+2
mul  $t4, $t3, 3      # tmp2 = 3*tmp1
add  $t2, $t0, $t4    # c = a + tmp2
```



Optimizing register usage

Can often use fewer registers by accumulating results

```
# Pseudocode:
#   c = a + 3*(b+2)
# Register mappings:
#   a: $t0, b: $t1, c: $t2
#   tmp1: $t3, tmp2: $t4
```

```
# tmp1 = b+2
# tmp2 = 3*tmp1
# c = a + tmp2

addi $t3, $t1, 2
mul  $t4, $t3, 3
add  $t2, $t0, $t4
```

$\Longrightarrow$

```
# c = b+2
# c = 3*c
# c = a + c

addi $t2, $t1, 2
mul  $t2, $t2, 3
add  $t2, $t0, $t2
```

Exercise

```
# Pseudocode:
#   d = a - 3 * (b + c + 8)
# Register mappings:
#   a: t0, b: t1, c: t2, d: t3

addi $t3, $t2, 8    # d = b + c + 8
add  $t3, $t1, $t3
li   $t4, 3         # d = 3 * d
mul  $t3, $t4, $t3
sub  $t3, $t0, $t3  # d = a - d
```

Logical expressions

In high-level languages, used in conditions of control structures

- branches (if-statements)
- loops (while, for)

Logical expressions

- values: True, False
- boolean operators: not (`!`), and (`&&`), or (`||`)
- relational operators: `==`, `!=`, `>`, `>=`, `<`, `<=`

In MIPS:

- conceptually, False = `0`, True = `1`
- non-relational logical operations are **bitwise**, not boolean

Bitwise logic operations

```
and   $t1, $t2, $t3     # $t1 = $t2 & $t3 (bitwise and)
or    $t1, $t2, $t3     # $t1 = $t2 | $t3 (bitwise or)
xor   $t1, $t2, $t3     # $t1 = $t2 ^ $t3 (bitwise xor)
```

Example: 0110 ‘**op**’ 0011

```
       1010             1010             1010
  and  0011         or  0011        xor  0011
  ---------        ---------        ---------
       0010             1011             1001
```

1 iff both are 1

1 iff either is 1

1 iff exactly one 1

Immediate variants

```
andi $t1, $t2, 0x0F     # $t1 = $t2 & 0x0F (bitwise and)
ori  $t1, $t2, 0xF0     # $t1 = $t2 | 0xF0 (bitwise or)
xori $t1, $t2, 0xFF     # $t1 = $t2 ^ 0xFF (bitwise xor)
```

Bitwise logic vs. boolean logic

For **`and`**, **`or`**, **`xor`**:

- equivalent when False = `0` and True = `1`
- not equivalent when False = `0` and True $\neq$ `0`! (as in C)

Careful: MARS provides a macro instruction for bitwise **`not`**

- this is **not equivalent** to logical not
- inverts every bit, so “not True” $\Rightarrow$ **`0xFFFFFFFE`**

How can we implement logical not?

```
xori $t1, $t2, 1    # $t1 = not $t2 (logical not)
```

Relational operations

Logical expressions

- values: True, False
- boolean operators: not (`!`), and (`&&`), or (`||`)
- relational operators: `==`, `!=`, `>`, `>=`, `<`, `<=`

```
seq  $t1, $t2, $t3     # $t1 = $t2 == $t3 ? 1 : 0
sne  $t1, $t2, $t3     # $t1 = $t2 != $t3 ? 1 : 0
sge  $t1, $t2, $t3     # $t1 = $t2 >= $t3 ? 1 : 0
sgt  $t1, $t2, $t3     # $t1 = $t2 >  $t3 ? 1 : 0
sle  $t1, $t2, $t3     # $t1 = $t2 <= $t3 ? 1 : 0
slt  $t1, $t2, $t3     # $t1 = $t2 <  $t3 ? 1 : 0

slti $t1, $t2, 42      # $t1 = $t2 <   42 ? 1 : 0
```

(MARS provides macro versions of many of these instructions that take immediate arguments)

Exercise

```
# Pseudocode:
#   c = (a < b) || ((a+b) == 10)
# Register mappings:
#   a: t0, b: t1, c: t2

add  $t3, $t0, $t1  # tmp = a+b
li   $t4, 10        # tmp = tmp == 10
seq  $t3, $t3, $t4
slt  $t2, $t0, $t1  # c = a < b
or   $t2, $t2, $t3  # c = c | tmp
```

Exercise

```
# Pseudocode:
#   c = (a < b) && ((a+b) % 3) == 2
# Register mappings:
#   a: t0, b: t1, c: t2

#   tmp1: t3, tmp2: t4

add  $t3, $t0, $t1  # tmp1 = a+b
li   $t4, 3         # tmp1 = tmp1 % 3
div  $t3, $t4
mfhi $t3
seq  $t3, $t3, 2    # tmp1 = tmp1 == 2
slt  $t4, $t0, $t1  # tmp2 = a < b
and  $t2, $t3, $t4  # c = tmp2 & tmp1
```