MIPS Code for Tiger

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

The Final Assignment

Adapt your interpreter to generate MIPS assembly code.

Amounts to writing

```
class Statement {
  public String mips() {
    return "# MIPS code for this statement";
  }
}
```

for each statement (Jmp, Bne, Binop, etc.)

MIPS Registers

32 general-purpose registers:

$zero	0	Hardwired constant 0
$at	1	Reserved for assembler
$v0**–**$v1	2–3	Temporaries & function return value
$a0**–**$a3	4–7	Function arguments
$t0**–**$t7	8–15	Temporaries not preserved across call
$s0**–**$s7	16–23	Temporaries you must save
$t8**–**$t9	24–25	Temporaries not preserved across call
$k0**–**$k1	26–27	Reserved for OS Kernel
$gp	28	Pointer to global area
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Function return address

The MIPS Instruction Set

Arithmetic instructions operate on registers only

```
add $t0, $t1, $t2    # t0 ← t1 + t2
mul $v0, $t3, $t5    # v0 ← t3 × t5
```

Load immediate loads a constant

```
li $t0, 2351   # t0 ← 2351
```

Comparisons set registers to 0 or 1

```
slt $t0, $t1, $t2 # t0 ← 1 if t1 < t2, 0 otherwise
```

Data Movement

```
move $t3, $t2 # copy contents of t2 to t3
```

The MIPS Instrucion Set

Unconditional branch

```
b Label # send control to Label
```

Conditional branches

```
beqz $t0, Label # send control Label if t0 is zero
bnez $t0, Label # branch to Label if t0 is non-zero
```

Jump and Link

```
jal Label # save next address in $ra, jump to Label
```

Load/stores

```
lw $t0, 10($fp) # Load t0 with memory at $fp + 10
sw $t1, -5($sp) # Store t1 at location $sp − 5
```

Assembler Directives

```
.align 2          # align on 2^2 = 4 byte boundary
.asciiz "Hello"   # zero-terminated ASCII string data
.globl main       # Make the label main visible outside
.text             # Instructions follow: place in text
.data             # Data follows: place in data
```

Tiger Activation Record Layout

8($fp)	“fp(-2)”
4($fp)	“fp(-1)”
0($fp)	Return address
-4($fp)	Static Link
-8($fp)	Saved frame pointer
-12($fp)	“fp(0)”
-16($fp)	“fp(1)”

Ent

The new intermediate instruction **ent** should be placed immediately after the label at the start of each function's code.

It does nothing in the interpreter, but generates the following MIPS code to set up the current activation record.

```
sw $ra, 0($sp)  # Save return address
sw $a0, -4($sp) # Initialize static link
sw $fp, -8($sp) # Save old fp
move $fp, $sp   # Set up new fp
subu $sp, 12    # Update sp
```

Rts

The **rts** instruction undoes the effect of the **ent** instruction:

```
move $sp, $fp     # Restore sp
lw   $fp, -8($sp) # Restore fp
lw   $ra, -0($sp) # Restore ra
jr   $ra          # Return to caller
```

Example

```
1+2*3+4  main:
           ent
           psh 1
           mov fp(0), 1
           psh 1
           mov fp(1), 2
           psh 1
           mov fp(2), 3
           mul fp(1), fp(1), fp(2)
           psh -1
           add fp(0), fp(0), fp(1)
           psh -1
           psh 1
           mov fp(1), 4
           add fp(0), fp(0), fp(1)
           psh -1
           rts
```

```
main:
  sw  $ra, 0($sp)
  sw  $a0, -4($sp)
  sw  $fp, -8($sp)
  move $fp, $sp
  subu $sp, 12
  subu $sp, 4
  li $t0, 1
  sw $t0, -12($fp)
  subu $sp, 4
  li $t0, 2
  sw $t0, -16($fp)
  subu $sp, 4
  li $t0, 3
  sw $t0, -20($fp)
  lw $t0, -16($fp)
  lw $t1, -20($fp)
  add $t0, $t0, $t1
  sw $t0, -16($fp)
  subu $sp, -4
  lw $t0, -12($fp)
  lw $t1, -16($fp)
  mul $t0, $t0, $t1
  sw $t0, -12($fp)
  subu $sp, -4
  subu $sp, 4
  li $t0, 4
  sw $t0, -16($fp)
  lw $t0, -12($fp)
  lw $t1, -16($fp)
  add $t0, $t0, $t1
  sw $t0, -12($fp)
  subu $sp, -4
  move $sp, $fp
  lw   $fp, -8($sp)
  lw   $ra, -0($sp)
  jr   $ra
```

Dissecting the Example 1

```
main:                main:              # Labels copy directly
  ent                  sw $ra, 0($sp)   # Save return address
                       sw $a0, -4($sp)  # Initialize static link
                       sw $fp, -8($sp)  # Save old fp
                       move $fp, $sp    # Set up new fp
                       subu $sp, 12     # Update sp
  psh 1                subu $sp, 4      # Psh decreases sp
  mov fp(0), 1         li $t0, 1        # Load immediate
                       sw $t0, -12($fp)# -k($fp) for fp-relative
  psh 1                subu $sp, 4
  mov fp(1), 2         li $t0, 2
                       sw $t0, -16($fp)
  psh 1                subu $sp, 4
  mov fp(2), 3         li $t0, 3
                       sw $t0, -20($fp)
  mul fp(1),fp(1),fp(2) lw $t0, -16($fp) # First operand in t0
                       lw $t1, -20($fp) # Second in t1
                       mul $t0, $t0, $t1
                       sw $t0, -16($fp) # Result in t0
```

Dissecting the Example 2

```
psh -1               subu $sp, -4
add fp(0),fp(0),fp(1) lw $t0, -12($fp)
                     lw $t1, -16($fp)
                     add $t0, $t0, $t1
                     sw $t0, -12($fp)
psh -1               subu $sp, -4
psh 1                subu $sp, 4
mov fp(1), 4         li $t0, 4
                     sw $t0, -16($fp)
add fp(0),fp(0),fp(1) lw $t0, -12($fp)
                     lw $t1, -16($fp)
                     add $t0, $t0, $t1
                     sw $t0, -12($fp)
psh -1               subu $sp, -4
rts                  move $sp, $fp      # Restore sp
                     lw   $fp, -8($sp)  # Restore fp
                     lw   $ra, -0($sp)  # Restore ra
                     jr   $ra           # Return to caller
```

Your Job

- Write **`mips()`** methods for each instruction
- Make the **`Binop.mips()`** method work (not complete)
- Write the **`mipsGet()`** and **`mipsSet()`** methods for
- Write code for the standard library functions
- Test it

Basic Idea

I've added **`mipsGet()`** and **`mipsSet()`** methods to most of the operands.

```
class Operand {
  public String mipsGet(String reg) { ... }
  public String mipsSet(String reg) { ... }
}
```

mipsGet and mipsSet

Each method either reads or writes the contents of the operand into the given register. E.g.,

```
public class FrameRel extends Operand {
  public String mipsGet(String reg) {
    return "  lw " + reg + ", " +
      Integer.toString( -4*offset - (offset>=0?12:0) )
        + "($fp)";
  }
  public String mipsSet(String reg) {
    return "  sw " + reg + ", " +
      Integer.toString( -4*offset - (offset>=0?12:0) )
        + "($fp)";
  }
}
```

Generating Code for Operands

`0($fp)`	Return address
`-4($fp)`	Static Link
`-8($fp)`	Saved frame pointer
`-12($fp)`	"fp(0)"
`-16($fp)`	"fp(1)"

```
public String mipsGet(String reg) {
  return "  lw " + reg + ", " +
    Integer.toString( -4*offset - (offset>=0?12:0) )
      + "($fp)";
}
```

Thus, **`mipsGet($v0)`** called on **`fp(2)`** produces

```
lw $v0, -20($fp)
```

Generating Code for StackLinks

Loading **`2*fp(3)`** into **`$v0`** produces

```
mov $v0, $fp
lw $v0, -4($v0)
lw $v0, -4($v0)
lw $v0, -24($v0)
```

You need to generate code for the BlockRel operand. Compute the base operand, add the offset operand, then load or store data from that address.

Generating Code for Mov

SImple **`mips()`** method for **`Mov`**:

```
public String mips() {
  return source.mipsGet("$t0") + "\n" +
         dest.mipsSet("$t0");
}
```

`mipsGet()` called on the source operand produces code that copies the operand into **`$t0`**.

`mipsPut()` called on the destination writes **`$t0`** into the destination operand.

System Calls

SPIM supplies a number of system calls.

print_int	1	**$a0** = integer
print_float	2	**$f12** = float
print_double	3	**$f12** = double
print_string	4	**$a0** = string
read_int	5	→ **$v0** = integer
read_float	6	→ **$f0** = float
read_double	7	→ **$f0** = double
read_string	8	**$a0** = buffer, **$a1** = length
sbrk	9	**$a0** = amount → **$v0** = address
exit	10	

Invoking System Calls

```
     .data
Str: .asciiz "Hello World!"
     .text
     li $v0, 4    # code for print_string
     la $a0, Str  # address of string
     syscall
```

Sbrk

Certain instructions demand more memory (arr, rec, substr).

The sbrk system call takes one argument: the number of bytes you want, and returns the address of a new memory field.

Use this to acquire more memory.

Tiger string/record/array semantics are sloppy: once memory is allocated, it is never garbage collected.

Arr and Rec

The **arr** and **rec** instructions allocate space for an initialize arrays and records.

rec is the simpler of the two: call the sbrk system call with the right number of bytes.

arr also allocates an array, but intializes its value.

Generate a MIPS assembly code loop that fills the newly-created array.

Look at the **execute()** method for each to see how these instructions are implemented in the interpreter.

String Operations

Tiger's comparison operators ($=$, $<$, $>$, etc.) are overloaded for strings.

They perform string comparisons, not simple pointer comparison.

You should create a new instruction, Binopstr, that operates just on strings.

Change your translation to generate Binopstr when you know the arguments are strings (types should tell you this).

Library Functions

Write the standard library in MIPS assembly code.

Things like **substring()** are complicated: they need to allocate a new string and copy things with loops.

concat() is the hardest.

Ignore the **exit()** return value; just terminate.

flush() should do nothing (SPIM doesn't buffer output).

Many functions need to allocate memory for new strings using **sbrk**.