Code generation

Handout written by Maggie Johnson and revised by me.

The final phase of a compiler is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent program in the target's machine language. This step can itself be fairly complex, particularly if the intermediate code is fairly high-level and contains little or no information about the target machine or runtime environment. Unlike all our previous tasks, this one is very machine-specific, since each architecture has its own set of instructions and peculiarities that must be respected. The ABI or *application binary interface* specifies the rules for executable programs on an architecture (instructions, register uses, calling conventions, executable format, and so on) and these details are what directs the code generation. The final code generation pass must supply all the action locations of variables and temporaries, plus all the code to maintain the runtime environment, set up and return from function calls, manage the stack and so on.

MIPS R2000/R3000 assembly

The target language for SOOP is MIPS R2000/R3000 assembly language. We chose this language because it allows us to use SPIM, a software simulator that runs assembly language programs for this processor. The SPIM simulator reads assembly instructions from a file and then executes them. It was written by James Larus of the University of Wisconsin, and gracefully distributed for non-commercial use free of charge. (by the way, SPIM is "MIPS" backwards…)

First, let's start by looking at the MIPS R2000/R3000 machine. The processor chip contains a main CPU and a couple of co-processors, one for floating point operations and another for memory management. The word size is 32-bits, and the processor has thirty-two word-sized registers on-board, some of which have designed special uses, others are general-purpose. It also provides for up to 128K of high-speed cache, half each for instructions and data (although this is not simulated in SPIM). All processor instructions are encoded in a single 32-bit word format. All data operations are register to register; the only memory references are pure load/store operations.

Here are the registers identified by their symbolic name and the purpose each is used for:

`zero`	Constant 0
`at`	Reserved for assembler
`v0, v1`	Temps for expression evaluation and return results from functions
`a0, a1, a2, a3`	Usually for the first 4 arguments for function calls, additional ones are passed on stack.We will use a2 to pass "this" for method calls, and use a3 to hold this during a call. We will pass all other args on the stack for simplicity.
`t0-t9`	General purpose, caller-saved
`s0-s7`	General purpose, callee-saved
`k0, k1`	Reserved for OS
`gp`	Pointer to static/global data area
`sp`	Stack pointer
`fp`	Frame pointer
`ra`	Return address

The floating point co-processor has another set of 32 registers specifically for floating-point. Since we will only be dealing with integer-derived types, we won't need to use those.

The MIPS instructions form a fairly ordinary RISC instruction set. Here are a few excerpts to show the flavor:

```
add rd, rs, rt     Add operands in Rs and Rt, store result in Rd
addiu rd, rs, imm  Add Rs to constant imm, store result in Rt
and rd, rs, rt     Logical and Rs and Rt, store result in Rd
seq rd, rs, rt     Set rd to 1 if rs == rt, 0 otherwise
b label            Unconditional branch
beq rs, rt, label  Branch to label if rs == rt
jal target         Unconditional jump to target (save $ra)
lw rd, address     Load word at address into rd
sw rd, address     Store word at address from rd
```

The machine provides only one memory addressing mode: `c(rx)` which uses the sum of the integer `c` and the contents of register `rx`, as the address. Load and store instructions operate only on aligned data (a quantity is aligned if its memory address is a multiple of its size in bytes). You'll notice that all our generated assembly programs have `.align 2` in the preamble. This means we should align the next value (which is the global "main") on a word boundary.

As for the organization of memory in MIPS, a program's address space is composed of three parts. At the bottom is the text segment that holds the program instructions. Above the text segment is the data segment that is divided into two parts. The static data contains objects whose size and address are known to the compiler and linker. Right above the static data area is the dynamic data area or heap. This area grows up as needed. At the top of the address space is the program stack. It grows down, toward the data segment.

The calling convention used in SPIM is modeled on the Unix gcc convention, not MIPS (which is more complex but faster). A stack frame consists of the memory between the $fp pointer which points to the base of the current stack frame, and the $sp pointer, which points to the next free word on the stack. As is typical of Unix systems, the stack grows down, so $fp is above $sp. Here's what happens to effect a function call.

The caller sets up for call via these steps:

1) Make space on stack for callee to spill $fp, $ra and $a3 (this).
2) Pass arguments by pushing them on the stack.
3) Save any "live" registers.
4) Execute a jump to the function.

In the callee, we do the following:

1) Save values of $fp, $ra and $a3 (this) values into spill space on stack.
2) Configure the new stack frame by setting $fp to appropriate address.
3) Allocate space for local and temporary variables.
4) When ready, assign the return value to $v0.

When ready to end the function, the callee does the following:

1) Pop all locals/temps and parameters off the stack.
2) Restore the $fp, $ra and $a3 (this) values.
3) Pop the area of the stack where the $fp, $ra and $a3 were stored.
4) Jump to the address saved in $ra.

Register usage

Because the MIPS architecture requires that all values being worked on must be first loaded into a register, there will be a lot of contention for the scarce number of registers that are available. One of the jobs of the code generator is arbitrating that contention and coming up with an efficient and correct scheme for moving values in and out of registers. Doing this well turns out to be one of the most critical optimizations that the code generator can perform. When we get on to discussing optimizations, we will talk about some of the techniques and strategies than can be used. For today, we'll make simple but not-so-efficient use of our registers.

All variables are associated with a location in memory. Rather than always writing a variable's value out to a memory location each time it is assigned, we can choose to hold the value in a register. We only write it out to memory when we need to for consistency. We say that such a variable is *slaved* in the register. For variables that are used repeatedly, this will turn out to be a big win over loading and storing the value back to memory. For some variables, such as loop counters, we may never need to even create or use space in memory if the variable can be slaved in a register for its entire lifetime.

To keep track of our register usage, we keep a list of *descriptors*, one which maps each register to the identifier it is currently holding, and another which maps identifiers to whatever registers currently hold that value.

The goal is to hold as many values in registers as possible. Values in registers are only written out to memory when we run out of registers or when we cannot guarantee that the value slaved in a register is reliable. This can happen because we are going to call a function or make a jump and we can't be certain our values will be preserved or reliable afterwards.

Spilling a register means writing the currently value of a register out to memory, so that we can re-use that register to hold a different value. When all our registers are in use and we need to bring in a new value, we will have to spill one of the current registers to make space. Choosing the "best" register to spill can be a complicated enterprise.

Here's a fairly naïve algorithm for register usage. On the left we have the TAC code which refers to the variables using symbolic names, on the right, the code generator must translate that into an operations using actual machine registers:

```
a := b op c    to     OP Rs, Ri, Rj
```

1) Select a register for R_j to hold c. Using the information in the descriptors, we choose in order of preference: a register already holding c, an empty register, an unmodified register, a modified register (we have to spill it). Then, we load c into the register and update the descriptors.
2) We load R_i with b in a similar manner, but we must not reuse R_j unless b and c are the same.
3) Choose a register for a in the same way but not reusing R_i or R_j, and spilling if we need to.
4) Generate instruction: OP R_s, R_i, R_j

Sample assembly language programs

Example 1: The simple Hello, World program

```
void main(void) {
   Print("hello world");
}
```

And its MIPS assembly:

```
    # preamble
      .text
      .align 2
      .globl main
  main:
    # BeginFunc
      add $sp, $sp, -12     # space to save ra,fp,this
      sw $fp, 12($sp)       # save prev fp
      sw $ra, 8($sp)        # save prev ra
      sw $a3, 4($sp)        # save prev this
      addiu $fp, $sp, 12    # set up new frame pointer
      move $a3, $a2         # move passed 'this' from a2 to a3
      lw $t0, _lib0
      subu $sp, $sp, $t0    # decrement sp to make space for locals/temps
      .data
  _L0:
      .asciiz "hello world"
      .text
    # // set up for built-in PrintString/Integer
    # Arg _L0
      add $sp, $sp, -12     # decrement sp to make space for spilling
      add $sp, $sp, -4      # decrement sp to make space for arg
      la $v1, _L0           # loading label
      sw $v1, 4($sp)
    # PrintString()
      jal PrintString       # jump to function
    # EndFunc
    # below handles reaching end of fn body with no explicit return
      add $sp, $fp, -12     # pop all temps and args off stack
      lw $ra, -4($fp)       # restore saved return address
      lw $a3, -8($fp)       # restore saved this
      lw $fp, 0($fp)        # restore saved frame pointer
      add $sp, $sp, 12      # pop rest of callee saves off stack
      jr $ra # return from function
    .data    # record total bytes needed for function's locals/temps
_lib0: .word 0
       .text

# library routines go here....
```

Example 2: A program with some local variables in the main:

```
void main(void) {
    int a;
    int b;
    int c;

    c = 1;
    b = 2;
    a = 3;
    c = (c + (a * b));
}
```

Its MIPS assembly:

```
  # preamble
    .text
    .align 2
    .globl main
 main:
  # BeginFunc
    add $sp, $sp, -12       # space for old ra,fp,this
    sw $fp, 12($sp)         # save prev fp
    sw $ra, 8($sp)          # save prev ra
    sw $a3, 4($sp)          # save prev this
    addiu $fp, $sp, 12      # set up new frame pointer
    move $a3, $a2           # move passed 'this' from a2 to a3
    lw $t0, _lib0
    subu $sp, $sp, $t0      # decrement sp to make space for locals/temps
  # Var a
  # Var b
  # Var c
  # Var _t0
  # _t0 = 1
    li $t0, 1               # load constant value 1 into $t0
  # c = _t0
    move $t1, $t0
  # Var _t1
  # _t1 = 2
    li $t2, 2               # load constant value 2 into $t2
  # b = _t1
    move $t3, $t2
  # Var _t2
  # _t2 = 3
    li $t4, 3               # load constant value 3 into $t4
  # a = _t2
    move $t5, $t4
  # Var _t3
  # _t3 = a * b
    mul $t6, $t5, $t3
  # Var _t4
  # _t4 = c + _t3
    add $t7, $t1, $t6
  # c = _t4
    move $t1, $t7
  # EndFunc
  # below handles reaching end of fn body with no explicit return
    sw $t0, -24($fp)        # spill _t0
```

```
      sw $t1, -20($fp)        # spill c
      sw $t2, -28($fp)        # spill _t1
      sw $t3, -16($fp)        # spill b
      sw $t4, -32($fp)        # spill _t2
      sw $t5, -12($fp)        # spill a
      sw $t6, -36($fp)        # spill _t3
      sw $t7, -40($fp)        # spill _t4
      add $sp, $fp, -12       # pop all temps and args off stack
      lw $ra, -4($fp)         # restore saved return address
      lw $a3, -8($fp)         # restore saved this
      lw $fp, 0($fp)          # restore saved frame pointer
      add $sp, $sp, 12        # pop rest of callee saves off stack
      jr $ra                  # return from function
    .data# record total bytes needed for function's locals/temps
 _lib0:   .word 32
          .text

# library routines go here....
```

Example 3: A program with a function call

```
int test(int a, int b) {
     return a + b;
}

void main(void) {
    int c;

    c = test(4, 5);
    Print(c);
}
```

Its MIPS assembly:

```
 # preamble
   .text
   .align 2
   .globl main
_L0:
 # BeginFunc
   sw $fp, 20($sp)         # save prev fp
   sw $ra, 16($sp)         # save prev ra
   sw $a3, 12($sp)         # save prev this
   addiu $fp, $sp, 20      # set up new frame pointer
   move $a3, $a2           # move passed 'this' from a2 to a3
   lw $t0, _size0
   subu $sp, $sp, $t0      # decrement sp to make space for locals/temps
 # Var a
 # Var b
 # Var _t0
 # _t0 = a + b
   lw $t1, -12($fp)        # load a
   lw $t2, -16($fp)        # load b
   add $t0, $t1, $t2
 # Return _t0
   move $v0, $t0           # put return value into $v0
   sw $t0, -20($fp)        # spill _t0
```

```
    add $sp, $fp, -12        # pop all temps and args off stack
    lw $ra, -4($fp)          # restore saved return address
    lw $a3, -8($fp)          # restore saved this
    lw $fp, 0($fp)           # restore saved frame pointer
    add $sp, $sp, 12         # pop rest of callee saves off stack
    jr $ra                   # return from function
  # EndFunc
  # below handles reaching end of fn body with no explicit return
    add $sp, $fp, -12        # pop all temps and args off stack
    lw $ra, -4($fp)          # restore saved return address
    lw $a3, -8($fp)          # restore saved this
    lw $fp, 0($fp)           # restore saved frame pointer
    add $sp, $sp, 12         # pop rest of callee saves off stack
    jr $ra                   # return from function
  .data  # record total bytes needed for function's locals/temps
_size0:   .word 12
          .text


main:
  # BeginFunc
    add $sp, $sp, -12        # space for old ra,fp,this
    sw $fp, 12($sp)          # save prev fp
    sw $ra, 8($sp)           # save prev ra
    sw $a3, 4($sp)           # save prev this
    addiu $fp, $sp, 12       # set up new frame pointer
    move $a3, $a2                 # move passed 'this' from a2 to a3
    lw $t0, _size1
    subu $sp, $sp, $t0       # decrement sp to make space for locals/temps
  # Var c
  # Var _t1
  # _t1 = 4
    li $t0, 4                # load constant value 4 into $t0
  # Var _t2
  # _t2 = 5
    li $t1, 5                # load constant value 5 into $t1
  # Var _t3
  # Arg _t1
    add $sp, $sp, -12        # decrement sp to make space for spilling
    add $sp, $sp, -4         # decrement sp to make space for arg
    sw $t0, 4($sp)           # push parameter value on stack
  # Arg _t2
    add $sp, $sp, -4         # decrement sp to make space for arg
    sw $t1, 4($sp)           # push parameter value on stack
  # _t3 = _L0()
    sw $t0, -16($fp)         # spill _t1
    sw $t1, -20($fp)         # spill _t2
    jal _L0                  # jump to function
    move $t0, $v0
  # // call global function test()
  # c = _t3
    move $t1, $t0
  # // set up for built-in PrintString/Integer
  # Arg c
    add $sp, $sp, -12        # decrement sp to make space for spilling
    add $sp, $sp, -4         # decrement sp to make space for arg
    sw $t1, 4($sp)           # push parameter value on stack
  # PrintInteger()
```

```
     sw $t0, -24($fp)         # spill _t3
     sw $t1, -12($fp)         # spill c
     jal PrintInteger         # jump to function
   # EndFunc
   # below handles reaching end of fn body with no explicit return
     add $sp, $fp, -12        # pop all temps and args off stack
     lw $ra, -4($fp)          # restore saved return address
     lw $a3, -8($fp)          # restore saved this
     lw $fp, 0($fp)           # restore saved frame pointer
     add $sp, $sp, 12         # pop rest of callee saves off stack
     jr $ra                   # return from function
   .data  # record total bytes needed for function's locals/temps
_size1:   .word 16
          .text

# library routines go here....
```

Example 4: A program with some global variables:

```
int a;
int b;

int tester(int d) {
   return(b + d);
}

void main(void) {
   int a;
   int c;

   a = 3;
   b = 4;
   c = a + b;
   Print(tester(c));
}
```

Its MIPS assembly:

```
     # preamble
       .text
       .align 2
       .globl main
     # Var a
     # Var b

   _L0:
     # BeginFunc
       sw $fp, 16($sp)        # save prev fp
       sw $ra, 12($sp)        # save prev ra
       sw $a3, 8($sp)         # save prev this
       addiu $fp, $sp, 16     # set up new frame pointer
       move $a3, $a2          # move passed 'this' from a2 to a3
       lw $t0, _lib0
       subu $sp, $sp, $t0     # decrement sp to make space for locals/temps
     # Var d
     # Var _t0
     # _t0 = b + d
       lw $t1, -4($gp)        # load b
```

```
      lw $t2, -12($fp)      # load d
      add $t0, $t1, $t2
    # Return _t0
      move $v0, $t0         # put return value into $v0
      sw $t0, -16($fp)      # spill _t0
      add $sp, $fp, -12     # pop all temps and args off stack
      lw $ra, -4($fp)       # restore saved return address
      lw $a3, -8($fp)       # restore saved this
      lw $fp, 0($fp)        # restore saved frame pointer
      add $sp, $sp, 12      # pop rest of callee saves off stack
      jr $ra                # return from function
    # EndFunc
    # below handles reaching end of fn body with no explicit return
      add $sp, $fp, -12     # pop all temps and args off stack
      lw $ra, -4($fp)       # restore saved return address
      lw $a3, -8($fp)       # restore saved this
      lw $fp, 0($fp)        # restore saved frame pointer
      add $sp, $sp, 12      # pop rest of callee saves off stack
      jr $ra                # return from function
    .data# record total bytes needed for function's locals/temps
_lib0:   .word 8
    .text


  main:
    # BeginFunc
      add $sp, $sp, -12     # space for old ra,fp,this
      sw $fp, 12($sp)       # save prev fp
      sw $ra, 8($sp)        # save prev ra
      sw $a3, 4($sp)        # save prev this
      addiu $fp, $sp, 12    # set up new frame pointer
      move $a3, $a2         # move passed 'this' from a2 to a3
      lw $t0, _lib1
      subu $sp, $sp, $t0    # decrement sp to make space for locals/temps
    # Var a
    # Var c
    # Var _t1
    # _t1 = 3
      li $t0, 3             # load constant value 3 into $t0
    # a = _t1
      move $t1, $t0
    # Var _t2
    # _t2 = 4
      li $t2, 4             # load constant value 4 into $t2
    # b = _t2
      move $t3, $t2
    # Var _t3
    # _t3 = a + b
      add $t4, $t1, $t3
    # c = _t3
      move $t5, $t4
    # Var _t4
    # Arg c
      add $sp, $sp, -12     # decrement sp to make space for spilling
      add $sp, $sp, -4      # decrement sp to make space for arg
      sw $t5, 4($sp)        # push parameter value on stack
    # _t4 = _L0()
      sw $t0, -20($fp)      # spill _t1
```

```
      sw $t1, -12($fp)      # spill a
      sw $t2, -24($fp)      # spill _t2
      sw $t3, -4($gp)       # spill b
      sw $t4, -28($fp)      # spill _t3
      sw $t5, -16($fp)      # spill c
      jal _L0               # jump to function
      move $t0, $v0
    # // call global function tester()
    # // set up for built-in PrintString/Integer
    # Arg _t4
      add $sp, $sp, -12     # decrement sp to make space for spilling
      add $sp, $sp, -4      # decrement sp to make space for arg
      sw $t0, 4($sp)        # push parameter value on stack
    # PrintInteger()
      sw $t0, -32($fp)      # spill _t4
      jal PrintInteger      # jump to function
    # EndFunc
    # below handles reaching end of fn body with no explicit return
      add $sp, $fp, -12     # pop all temps and args off stack
      lw $ra, -4($fp)       # restore saved return address
      lw $a3, -8($fp)       # restore saved this
      lw $fp, 0($fp)        # restore saved frame pointer
      add $sp, $sp, 12      # pop rest of callee saves off stack
      jr $ra                # return from function
    .data# record total bytes needed for function's locals/temps
_lib1:   .word 24
         .text

# library routines go here....
```

Example 5: A program with an array and a class

```
class Cow {
  int height;
   int weight;
  void Moo(void) {
         Print ( height, " ", weight, "\n" );
  }
}

void main(void) {
  int x[8];

  class Cow betsy;
  betsy = New(Cow);
  betsy.weight = 100;
  betsy.height = 122;
  betsy.Moo();

   x = NewArray(x);
  x[2] = 12;
  x[4] = 24;
}
```

Its MIPS assembly:

```
  # preamble
    .text
    .align 2
    .globl main
_L0:
  # BeginFunc
    add $sp, $sp, -12        # space for old ra,fp,this
    sw $fp, 12($sp)          # save prev fp
    sw $ra, 8($sp)           # save prev ra
    sw $a3, 4($sp)           # save prev this
    addiu $fp, $sp, 12       # set up new frame pointer
    move $a3, $a2            # move passed 'this' from a2 to a3
    lw $t0, _lib0
    subu $sp, $sp, $t0       # decrement sp to make space for locals/temps
    .data
_L1:
    .asciiz " "
    .text
    .data
_L2:
    .asciiz "\n"
    .text
  # // set up for built-in PrintString/Integer
  # Arg height
    add $sp, $sp, -12        # decrement sp to make space for spilling
    add $sp, $sp, -4         # decrement sp to make space for arg
    la $v1, height           # loading label
    sw $v1, 4($sp)
  # PrintInteger()
    jal PrintInteger         # jump to function
  # Arg _L1
    add $sp, $sp, -12        # decrement sp to make space for spilling
```

```
    add $sp, $sp, -4         # decrement sp to make space for arg
    la $v1, _L1              # loading label
    sw $v1, 4($sp)
  # PrintString()
    jal PrintString          # jump to function
  # Arg weight
    add $sp, $sp, -12        # decrement sp to make space for spilling
    add $sp, $sp, -4         # decrement sp to make space for arg
    la $v1, weight  # loading label
    sw $v1, 4($sp)
  # PrintInteger()
    jal PrintInteger         # jump to function
  # Arg _L2
    add $sp, $sp, -12        # decrement sp to make space for spilling
    add $sp, $sp, -4         # decrement sp to make space for arg
    la $v1, _L2  # loading label
    sw $v1, 4($sp)
  # PrintString()
    jal PrintString          # jump to function
  # EndFunc
  # below handles reaching end of fn body with no explicit return
    add $sp, $fp, -12        # pop all temps and args off stack
    lw $ra, -4($fp)          # restore saved return address
    lw $a3, -8($fp)          # restore saved this
    lw $fp, 0($fp)           # restore saved frame pointer
    add $sp, $sp, 12         # pop rest of callee saves off stack
    jr $ra                   # return from function
  .data# record total bytes needed for function's locals/temps
_lib0:   .word 0
  .text
    .data
    .align 2
  Cow:
    .word _L0
  # // method #0 is Moo
    .text


  main:
  # BeginFunc
    add $sp, $sp, -12        # space for old ra,fp,this
    sw $fp, 12($sp)          # save prev fp
    sw $ra, 8($sp)           # save prev ra
    sw $a3, 4($sp)           # save prev this
    addiu $fp, $sp, 12       # set up new frame pointer
    move $a3, $a2            # move passed 'this' from a2 to a3
    lw $t0, _lib1
    subu $sp, $sp, $t0       # decrement sp to make space for locals/temps
  # // function being defined has 0 bytes of parameters
  # Var x
  # Var betsy
  # // set up call to built-in New()
  # Var _t0
  # Var _t1
  # _t1 = 12
    li $t0, 12               # load constant value 12 into $t0
  # Arg _t1
    add $sp, $sp, -12        # decrement sp to make space for spilling
```

```
  add $sp, $sp, -4         # decrement sp to make space for arg
  sw $t0, 4($sp)           # push parameter value on stack
# // sizeof Cow (12 bytes) is first argument
# Arg Cow
  add $sp, $sp, -4         # decrement sp to make space for arg
  la $v1, Cow  # loading label
  sw $v1, 4($sp)
# // label of vtable is second argument
# _t0 = New()
  sw $t0, -24($fp)         # spill _t1
  jal New                  # jump to function
  move $t0, $v0
# // built-in New() allocates memory and sets vtable ptr
# betsy = _t0
  move $t1, $t0
# Var _t2
# Var _t3
# _t3 = 8
  li $t2, 8                # load constant value 8 into $t2
# _t2 = betsy + _t3
  add $t3, $t1, $t2
# Var _t4
# _t4 = 100
  li $t4, 100              # load constant value 100 into $t4
# * _t2 = _t4
  sw $t0, -20($fp)         # spill _t0
  sw $t1, -16($fp)         # spill betsy
  sw $t2, -32($fp)         # spill _t3
  sw $t3, -28($fp)         # spill _t2
  sw $t4, -36($fp)         # spill _t4
  lw $t0, -36($fp)         # load _t4
  lw $t1, -28($fp)         # load _t2
  sw $t0, ($t1)            # dereference to save value
# Var _t5
# Var _t6
# _t6 = 4
  li $t2, 4                # load constant value 4 into $t2
# _t5 = betsy + _t6
  lw $t4, -16($fp)         # load betsy
  add $t3, $t4, $t2
# Var _t7
# _t7 = 122
  li $t5, 122              # load constant value 122 into $t5
# * _t5 = _t7
  sw $t1, -28($fp)         # spill _t2
  sw $t2, -44($fp)         # spill _t6
  sw $t3, -40($fp)         # spill _t5
  sw $t5, -48($fp)         # spill _t7
  lw $t0, -48($fp)         # load _t7
  lw $t1, -40($fp)         # load _t5
  sw $t0, ($t1)            # dereference to save value
# betsy.0()
  add $sp, $sp, -12        # space for old ra,fp,this
  lw $t2, -16($fp)         # load betsy
  move $a2 $t2             # put this into a2
  lw $v0, ($a2)            # load vtable pointer
  lw $v0, 0($v0)           # load address of method
  sw $t1, -40($fp)         # spill _t5
```

```
  jalr $v0                  # jump to method
# // invoke method #0 on object of class Cow
# // set up call to built-in NewArray
# Var _t8
# Var _t9
# _t9 = 32
  li $t0, 32                # load constant value 32 into $t0
# Arg _t9
  add $sp, $sp, -12         # decrement sp to make space for spilling
  add $sp, $sp, -4          # decrement sp to make space for arg
  sw $t0, 4($sp)            # push parameter value on stack
# // sizeof array is first arg. 8 elems * 4 bytes each
# _t8 = NewArray()
  sw $t0, -56($fp)          # spill _t9
  jal NewArray              # jump to function
  move $t0, $v0
# x = _t8
  move $t1, $t0
# Var _t10
# _t10 = 2
  li $t2, 2                 # load constant value 2 into $t2
# Var _t11
# Var _t12
# _t12 = 4
  li $t3, 4                 # load constant value 4 into $t3
# Var _t13
# _t13 = _t12 * _t10
  mul $t4, $t3, $t2
# _t11 = x + _t13
  add $t5, $t1, $t4
# Var _t14
# _t14 = 12
  li $t6, 12                # load constant value 12 into $t6
# * _t11 = _t14
  sw $t0, -52($fp)          # spill _t8
  sw $t1, -12($fp)          # spill x
  sw $t2, -60($fp)          # spill _t10
  sw $t3, -68($fp)          # spill _t12
  sw $t4, -72($fp)          # spill _t13
  sw $t5, -64($fp)          # spill _t11
  sw $t6, -76($fp)          # spill _t14
  lw $t0, -76($fp)          # load _t14
  lw $t1, -64($fp)          # load _t11
  sw $t0, ($t1)             # dereference to save value
# Var _t15
# _t15 = 4
  li $t2, 4                 # load constant value 4 into $t2
# Var _t16
# Var _t17
# _t17 = 4
  li $t3, 4                 # load constant value 4 into $t3
# Var _t18
# _t18 = _t17 * _t15
  mul $t4, $t3, $t2
# _t16 = x + _t18
  lw $t6, -12($fp)          # load x
  add $t5, $t6, $t4
# Var _t19
```

```
   # _t19 = 24
     li $t7, 24               # load constant value 24 into $t7
   # * _t16 = _t19
     sw $t1, -64($fp)         # spill _t11
     sw $t2, -80($fp)         # spill _t15
     sw $t3, -88($fp)         # spill _t17
     sw $t4, -92($fp)         # spill _t18
     sw $t5, -84($fp)         # spill _t16
     sw $t7, -96($fp)         # spill _t19
     lw $t0, -96($fp)         # load _t19
     lw $t1, -84($fp)         # load _t16
     sw $t0, ($t1)            # dereference to save value
   # EndFunc
   # below handles reaching end of fn body with no explicit return
     sw $t1, -84($fp)         # spill _t16
     add $sp, $fp, -12        # pop all temps and args off stack
     lw $ra, -4($fp)          # restore saved return address
     lw $a3, -8($fp)          # restore saved this
     lw $fp, 0($fp)           # restore saved frame pointer
     add $sp, $sp, 12         # pop rest of callee saves off stack
     jr $ra                   # return from function
   .data# record total bytes needed for function's locals/temps
_lib1:   .word 88
         .text
```

Example 6: Use of branch to route control through if/else

```
   void main(void) {
      int a;

      if (a == 12)
         a = 1;
      else
         a = 23;
   }
```

And its MIPS assembly:

```
   # preamble
     .text
     .align 2
     .globl main
  main:
   # BeginFunc
     add $sp, $sp, -12        # space for old ra,fp,this
     sw $fp, 12($sp)          # save prev fp
     sw $ra, 8($sp)           # save prev ra
     sw $a3, 4($sp)           # save prev this
     addiu $fp, $sp, 12       # set up new frame pointer
     move $a3, $a2            # move passed 'this' from a2 to a3
     lw $t0, _lib0
     subu $sp, $sp, $t0       # decrement sp to make space for locals/temps
   # Var a
   # Var _t0
   # _t0 = 12
     li $t0, 12               # load constant value 12 into $t0
```

```
   # Var _t1
   # _t1 = a == _t0
     lw $t2, -12($fp)         # load a
     seq $t1, $t2, $t0
   # IfZ _t1 Goto _L0
     sw $t0, -16($fp)         # spill _t0
     sw $t1, -20($fp)         # spill _t1
     beqz $t1, _L0
   # Var _t2
   # _t2 = 1
     li $t0, 1                # load constant value 1 into $t0
   # a = _t2
     move $t1, $t0
   # Goto _L1
     sw $t0, -24($fp)         # spill _t2
     sw $t1, -12($fp)         # spill a
     b _L1

  _L0:
   # Var _t3
   # _t3 = 23
     li $t0, 23               # load constant value 23 into $t0
   # a = _t3
     move $t1, $t0
     sw $t0, -28($fp)         # spill _t3
     sw $t1, -12($fp)         # spill a
  _L1:
   # EndFunc
   # below handles reaching end of fn body with no explicit return
     add $sp, $fp, -12        # pop all temps and args off stack
     lw $ra, -4($fp)          # restore saved return address
     lw $a3, -8($fp)          # restore saved this
     lw $fp, 0($fp)           # restore saved frame pointer
     add $sp, $sp, 12         # pop rest of callee saves off stack
     jr $ra                   # return from function
   .data # record total bytes needed for function's locals/temps
_lib0:   .word 20
         .text
```

Bibliography

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

J. Larus, SPIM S20: A MIPS R2000 Simulator. User Manual, 1993.

D. Patterson, J. Hennessy, Computer Organization & Design: The Hardware/Software Interface. Morgan-Kaufmann, 1994.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.