CS143 Handout #30
Winter 2001

Final code generation

The last phase of the compiler to run is the final 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 be trivial to fairly complex, depending how high or low-level the intermediate representation is and what information it contains about the target machine and runtime environment. How aggressive the optimization is also makes a big difference.

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 usage, calling conventions, instruction scheduling, memory organization, executable format, and so on) and these details are what directs the code generation. The final code generation pass assigns the locations of variables and temporaries and generates 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 Decaf is MIPS R2000/R3000 assembly. We chose this because it allows us to use SPIM, an excellent simulator for the MIPS processor. The SPIM simulator reads MIPS assembly instructions from a file and executes them on a virtual MIPS machine. 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…) There are some links to SPIM documentation and resources on our class web site which provide more detail on using this tool.

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 32 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 32 MIPS registers identified by their symbolic name and the purpose each is used for:

`zero`	holds constant 0 (used surprising often, so dedicated register for it)
`at`	reserved for assembler
`v0-v1`	used to return results of functions
`a0-a3`	usually used to pass first 4 args to function call (we pass all args on stack)
`t0-t7`	general purpose (caller-saved)
`s0-s7`	general purpose(callee-saved)
`t8-t9`	general purpose(caller-saved)
`k0-k1`	reserved for OS
`gp`	global pointer to static data segment
`sp`	stack pointer
`fp`	frame pointer
`ra`	return address

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

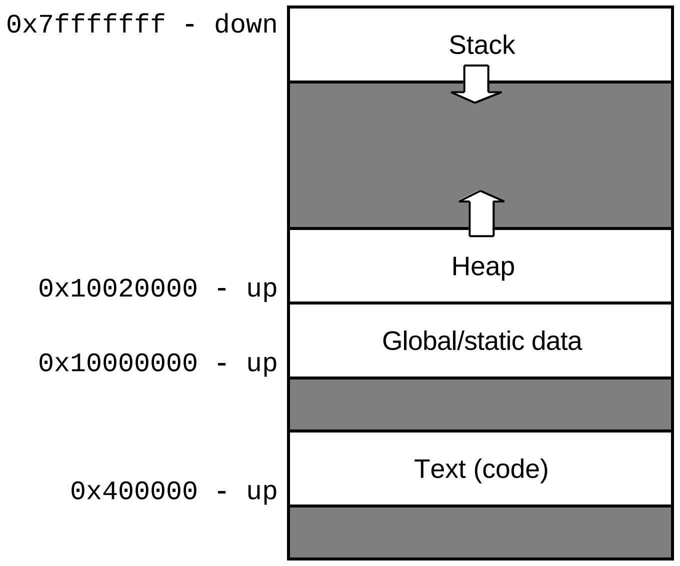
The MIPS instructions are a fairly ordinary RISC instruction set. Here are a few excerpts to show the flavor:

```
li $t0, 12             Load constant 12 into $t0
lw $t0, -4($fp)        Load word at address $fp-4 into $t0
sw $t0, 8($sp)         Store word from $t0 to address $sp+8
add $t0, $t1, $t2      Add operands in $t1 and $t2, store result in $t0
addiu $t0, $t1, 4      Add $t1 to constant 4, store result in $t0
or $t0, $t1, $t2       Or $t1 and $t2, store result in $t0
seq $t0, $t1, $t2      Set $t0 to 1 if $t1 == $t2, 0 otherwise
b label                Unconditional branch to label
beqz $t1, label        Branch to label if $t1 equals 0
jal label              Jump to label (save $ra)
jalr $t0               Jump to address in $t0 (save $ra)
jr $ra                 Return from function, resume at address $ra
```

The machine provides only one memory addressing mode: `c(rx)` which access the location at offset `c` (positive or negative) from the address held in register `rx`. 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.

Address space layout

On a MIPS machine, a program's address space is composed of various segments. At the bottom is the text segment that holds the program instructions. Above the text segment is the static data segment, which contains objects whose size and address are known to the compiler and linker at compile-time. Both the text and static data segment have their size set at compile-time and do not grow or shrink during execution. Above the static data area is the dynamic data area or heap. This area grows up as needed based on calls to malloc or other dynamic allocation functions. At the top of the address space is the program stack. It grows down, toward the heap. The shaded areas below represent regions of the address space that are unmapped, an attempt to read or write to a location in these regions will raise an execution error in the simulator.



The text segment
The text segment contains a big clump of machine instructions. During execution, the `$pc` (program counter) references an address in the text segment. Each iteration of the fetch-execute cycle pulls the next instruction from that location, decodes it, acts on it, and advances the `$pc` to the next instruction. As the compiler writer, you don't explicit read or set the `$pc`, but use of branch, jump, and return instructions will implicitly change its contents.

All of the code in the text segment is gathered at compile (or link) time. It consists of the instructions for each function placed end-to-end. Each function is identified with a unique label. There may also be labels within functions for conditionals and loops. Usually the assembler would be responsible for laying out the instructions in sequence, assigning locations to each label based on position and then translating references to labels to the fixed addresses. It is allowable for a jump to precede the definition of the target label because the assembler usually works in two passes (or can *backpatch* to fix the address in already processed instructions). We won't use an assembler for Decaf, we pass unencoded assembly to the SPIM simulator, and it performs those tasks usually handled by the assembler. Although convenient, this does means errors such as jumping to a non-existent label (e.g. if generating incorrect code for a loop) won't be caught until you execute that instruction on the simulator, getting a failure attempting to branch to address 0.

The static data segment
All of the locations in the data segment are configured at compile-time as well. When the compiler is putting together the program, each global variable is assigned a unique fixed offset in the data segment (+0, +4, +8, etc.). String constants and shared vtables are placed in the top half of the data segment (this area could be marked read-only in a protected memory system) and each marked with a label that allows us to uniquely refer to it such as when setting the vptr for a newly allocated object. The aggregate size of all globals/statics is usually what determines the space reserved for the entire segment. In our simulator, the data segment size is set using a flag when starting the simulator. It defaults to 64K, which is plenty for all the programs we run, so we don't change it.

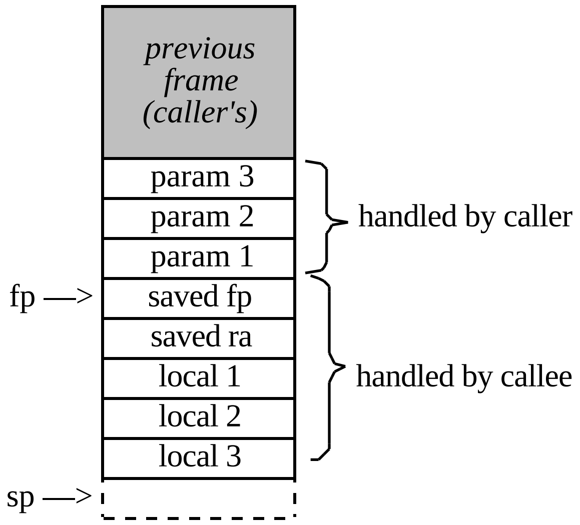
At runtime, the `$gp` register is reserved for the purpose of holding the base address of the global data segment. It is initialized to the proper value by the loader when setting up the program for execution. At runtime, a global variable can be accessed at its offset from the `$gp`. For example, in order to assign the constant value 55 to the global variable at offset 4, the MIPS sequence would look like this:

```
    li $t0, 55          Load constant 55 into $t0
    sw $t0, 4($gp)      Store value in $t0 at location +4 from $gp
```

The heap or dynamic data segment
The heap exists only at runtime and is not configured or laid out by the compiler. At runtime, the heap manager is responsible for increasing the size of the heap as needed (usually through some lower-level OS call such as `sbrk` or `vm_allocate`) and parceling out space within this area on demand. Heap managers come in all sorts of varieties: doubly-indirect handles or single-indirect pointers, fast but poor-fit allocators, slower but better-fit versions, those with automatic storage reclamation (garbage collection) and those without, and so on. Some features may be mandated by the language specification, other details may be left up to choice of the implementor. The heap manager implementation is typically written in a high-level language, although some time-critical portions may be hand-coded in assembly.

The stack segment and MIPS calling conventions
The stack is another entity that only exists at runtime. The OS loader is responsible for allocating space for the stack segment (often configured to some large constant size, say 8MB on Solaris) and calls a bit of glue code that sets up the outermost frame which then makes a call to the `main` routine where execution begins.

The calling convention used in SPIM is modeled on the Unix gcc convention rather than the full complexity of MIPS (which is more sophisticated 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`. The parameters are pushed right to left by the caller and the locals are pushed left to right by the callee. The return value is returned in register `$v0`. Here is a diagram of the layout of the parameters and locals within the MIPS runtime stack:



The calling conventions are the part of the ABI that dictate who does what using what memory/registers in order to ensure a smooth transition from caller to callee and back.

The caller sets up for the call via these steps:

1) Make space on stack for and save any caller-saved registers
2) Pass arguments by pushing them on the stack, one by one, right to left
3) Execute a jump to the function (saves the next instruction in `$ra`)

The callee takes over and does the following:

1) Make space on stack for and save values of `$fp` and `$ra`
2) Configure frame pointer by setting `$fp` to base of frame
3) Allocate space for stack frame (total space required for all local and temporary variables)
4) Execute function body, code can access params at positive offset from `$fp`, locals/temps at negative offsets from `$fp`

When ready to exit, the callee does the following:

1) Assign the return value (if any) to `$v0`
2) Pop stack frame off the stack (locals/temps/saved regs)
3) Restore the value of `$fp` and `$ra`
4) Jump to the address saved in `$ra`

When control returns to the caller, it cleans up from the call with the steps:

1) Pop the parameters from the stack
2) Restore value of any caller-saved registers, pops spill space from stack

Access to locations in the stack frame is done using `$fp`-relative addressing rather than `$sp`. It is possible for a calling convention to use only a stack pointer (i.e. not dedicating two registers for both a frame and stack pointer) and thus support only `$sp` -relative addressing, but there are various conveniences associated with having both that warrant using up two registers. Most notably it ensures that parameters and locals are at fixed offsets through the lifetime of the call. Sometimes the stack pointer may need be adjusted mid-call to make additional space for spilling, temporaries, stack-allocated memory (via such functions as `alloca`) and so on, which would add complication if using only `$sp` -relative addressing.

Although our simplified code generator writes the parameters to the stack as shown above, the actual MIPS calling convention uses registers for higher efficiency rather than the more expensive load/store operations on memory. The first four word-sized arguments are passed in `$a0` - `$a3` and if there are more than four, subsequent ones are passed on the stack. When a called function itself needs to make a call, it must preserve the values of its own parameters (usually by spilling them to the stack) before it overwrites `$a0` and others. It must restore the values when the call returns.

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. For today, we'll make simple but not-so-efficient use of our registers, and when we go on to discuss optimization we can revisit how to improve our strategy.

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* to map each register to the identifier it is currently holding.

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, conditionally branch, enter via a label, etc. 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 that refers to the variables using symbolic names, on the right, the code generator must translate that into an operation using machine registers:

```
a = b + c    to   add Rs, Ri, Rj
```

1) Select a register for R_i to hold b. Using the information in the descriptors, we choose in order of preference: a register already holding b, 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_j with c in a similar manner, but we must not reuse R_i 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

Final code generation in Decaf

You can think of final code generation as just yet another translation task, taking input in one format, and producing equivalent output in another. As we recognize a Decaf language construct, we construct the appropriate sequence of Tac instruction objects for it, and eventually hand the list over to the Mips class to translate to its MIPS equivalent.

Our Mips class encapsulates the machine details such as instruction formats and the available registers and how they are used. It also has direct knowledge of how to manage the stack and frame pointers and return address. It tackles the job of assigning variables to registers and spilling as necessary (albeit rather inefficiently). It assumes that the offset for each global, parameter, local, and temp variable has been properly configured (this is one of your jobs in pp5) and uses that information to access the correct memory location for load and store operations when moving the value of a variable to and from a register.

Here is the method from mips.cc that assigns a given variable to a register. It embodies the algorithm given above: we first try to find the variable already assigned in our descriptors, next look for an empty one, and only if necessary, choose a register to spill and replace:

```
Mips::Register Mips::GetRegister(Declaration *var, Reason reason)
{
  Register reg;

    if (!FindRegisterWithContents(var, reg)) { // if var not in reg
                                               // look for an empty one
    if (!FindRegisterWithContents(NULL, reg)) {
        reg = SelectRegisterToSpill();         // none empty, must spill
        SpillRegister(reg);
    }
    regs[reg].decl = var;                      // update descriptor
    if (reason == ForRead) {                   // load cur value
       Emit("lw %s, %d(%s)", regs[reg].name, var->GetOffset(),
             var->IsGlobal() ? regs[gp].name : regs[fp].name);
             regs[reg].isDirty = false;
    }
  }
  if (reason == ForWrite)
    regs[reg].isDirty = true;
  return reg;
}
```

What does it take to translate each Tac instruction object? Most are quite straightforward. Given that the TAC is fairly low-level, there is not a lot of translation needed for, say, a TAC add. We get the values we need into registers and emit a MIPS `add` on those registers. For some of the more complex instructions, there is bit more going on being the scenes, but nothing too magical. As a simple example, here's the method that converts a TAC LoadConstant instruction into MIPS:

```
void Mips::EmitLoadConstant(Declaration *dst, int val)
{
  Register reg = GetRegister(dst, ForWrite);
  Emit("li %s, %d", regs[reg].name, val);
}
```

As a slightly more complicated example, converting a TAC BeginFunc instruction produces a sequence of MIPS instructions to set up the stack and frame pointer, save `$fp` and `$ra`, and make space for the new stack frame:

```
void Mips::EmitBeginFunction(int frameSz)
{
  Emit("subu $sp, $sp, 8\t# decrement sp to make space to save ra, fp");
  Emit("sw $fp, 8($sp)\t# save fp");
  Emit("sw $ra, 4($sp)\t# save ra");
  Emit("addiu $fp, $sp, 8\t# set up new fp");

  if (frameSz != 0)
    Emit("subu $sp, $sp, %d\t# decrement sp for locals/temps", frameSz);
}
```

Thus the basic strategy for final code generation is to build a sequence of Tac instructions objects as we parse the program, and convert each to assembly one by one. The `Emit` method of the code generator prefixes the assembly with the standard preamble, iterates over the sequence to translate each in turn, and pulls in the library with the assembly for the built-in routines:

```
void CodeGenerator::Emit()
{
    Mips mips;

    mips.EmitPreamble();
    for (int i = 0; i < code->NumElements(); i++)
        code->Nth(i)->Emit(&mips);
    mips.EmitLibrary();  // contains code for built-ins
}
```

Voila! We have emitted a complete assembly program that can be run on the simulator. Our final code generator is done. There is room for improvement, in particular with regard to register usage and allocation, but we have the functionality needed to support correct code generation for all Decaf language features.

Sample assembly language programs

Example 1: The simple Hello, World program

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

Its TAC instructions:

```
main:
        BeginFuncWithParams  ;
        Var _tmp0 ;
        _tmp0 = "hello world" ;
        LCall _PrintString(_tmp0) ;
        EndFunc ;
```

And its MIPS assembly:

```
        # preamble
          .text
          .align 2
          .globl main
main:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
          subu $sp, $sp, 4      # decrement sp to make space for locals/temps
        # _tmp0 = "hello world"
          .data
          _string1: .asciiz "hello world"
          .text
        # _tmp0 = _string1
          la $t0, _string1      # load label
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t0, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -8($fp)       # spill _tmp0 from $t0
          jal _PrintString      # jump to function
          add $sp, $sp, 4       # pop params off stack
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
        # From here down is the standard library which is linked into all
        # programs. It contains the assembly for the built-in library
        # functions (Print, Read, Alloc)

       (code for library functions omitted for sake of brevity
        check out /usr/class/cs143/lib/libDecaf.s if you're curious)
```

Example 2: A program with a global variable and a function call:

```
int num;

int Binky(int a, int b) {
     int c;
     c = a + b;
     return c;
}

void main() {
    num = Binky(4, 10);
    Print(num);
}
```

TAC instructions:

```
Var num ;
_Binky:
        BeginFuncWithParams a, b ;
        Var c ;
        Var _tmp0 ;
        _tmp0 = a + b ;
        c = _tmp0 ;
        Return c ;
        EndFunc ;
main:
        BeginFuncWithParams  ;
        Var _tmp1 ;
        _tmp1 = 4 ;
        Var _tmp2 ;
        _tmp2 = 10 ;
        Var _tmp3 ;
        _tmp3 = LCall _Binky(_tmp1, _tmp2) ;
        num = _tmp3 ;
        LCall _PrintInt(num) ;
        EndFunc ;
```

MIPS assembly:

```
_Binky:
      # BeginFunc
        subu $sp, $sp, 8        # decrement sp to make space to save ra, fp
        sw $fp, 8($sp)          # save fp
        sw $ra, 4($sp)          # save ra
        addiu $fp, $sp, 8       # set up new fp
        subu $sp, $sp, 8        # decrement sp to make space for locals/temps
      # _tmp0 = a + b
        lw $t1, 4($fp)          # load a into $t1
        lw $t2, 8($fp)          # load b into $t2
        add $t0, $t1, $t2
      # c = _tmp0
        move $t3, $t0
      # Return c
        move $v0, $t3           # put return value into $v0
        move $sp, $fp           # pop callee frame off stack
```

```
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
        # Save modified registers before flow of control change
  main:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
          subu $sp, $sp, 12     # decrement sp to make space for locals/temps
        # _tmp1 = 4
          li $t0, 4             # load constant value 4 into $t0
        # _tmp2 = 10
          li $t1, 10            # load constant value 10 into $t1
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t1, 4($sp)        # push param value on stack
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t0, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -8($fp)       # spill _tmp1 from $t0
          sw $t1, -12($fp)      # spill _tmp2 from $t1
          jal _Binky            # jump to function
          move $t0, $v0
          add $sp, $sp, 8       # pop params off stack
        # num = _tmp3
          move $t1, $t0
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t1, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -16($fp)      # spill _tmp3 from $t0
          sw $t1, 0($gp)        # spill num from $t1
          jal _PrintInt         # jump to function
          add $sp, $sp, 4       # pop params off stack
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
```

Example 3: Use of branch to route control through if/else

```
   void main() {
      int a;

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

TAC instructions:

```
   main:
           BeginFuncWithParams  ;
           Var a ;
           Var _tmp0 ;
           _tmp0 = 12 ;
           Var _tmp1 ;
           _tmp1 = a == _tmp0 ;
           IfZ _tmp1 Goto _L0 ;
           Var _tmp2 ;
           _tmp2 = 1 ;
           a = _tmp2 ;
           Goto _L1 ;
   _L0:
           Var _tmp3 ;
           _tmp3 = 23 ;
           a = _tmp3 ;
   _L1:
           EndFunc ;
```

MIPS assembly:

```
main:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
          subu $sp, $sp, 20     # decrement sp to make space for locals/temps
        # _tmp0 = 12
          li $t0, 12            # load constant value 12 into $t0
        # _tmp1 = a == _tmp0
          lw $t2, -8($fp)       # load a into $t2
          seq $t1, $t2, $t0
        # IfZ _tmp1 Goto _L0
        # Save modified registers before flow of control change
          sw $t0, -12($fp)      # spill _tmp0 from $t0
          sw $t1, -16($fp)      # spill _tmp1 from $t1
          beqz $t1, _L0
        # _tmp2 = 1
          li $t0, 1             # load constant value 1 into $t0
        # a = _tmp2
          move $t1, $t0
        # Goto _L1
        # Save modified registers before flow of control change
          sw $t0, -20($fp)      # spill _tmp2 from $t0
          sw $t1, -8($fp)       # spill a from $t1
          b _L1
        # Save modified registers before flow of control change
  _L0:
        # _tmp3 = 23
          li $t0, 23            # load constant value 23 into $t0
        # a = _tmp3
          move $t1, $t0
        # Save modified registers before flow of control change
          sw $t0, -24($fp)      # spill _tmp3 from $t0
          sw $t1, -8($fp)       # spill a from $t1
  _L1:
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
```

Example 4: A program with a class and showing dynamic dispatch:

```
class Cow {
  int height;
  int weight;
  void InitCow(int h, int w) {
      height = h;
      weight = w;
  }
  void Moo() {
     Print("Moo!\n");
   }
}

void main() {

  class Cow betsy;
  betsy = New(Cow);
  betsy.InitCow(-5, 22);
  betsy.Moo();
}
```

TAC instructions:

```
__Cow_InitCow:
        BeginFuncWithParams this, h, w ;
        *(this + 4) = h ;
        *(this + 8) = w ;
        EndFunc ;
__Cow_Moo:
        BeginFuncWithParams this ;
        Var _tmp0 ;
        _tmp0 = "Moo!\n" ;
        LCall _PrintString(_tmp0) ;
        EndFunc ;
VTable Cow =
        __Cow_InitCow,
        __Cow_Moo,
;
main:
        BeginFuncWithParams  ;
        Var betsy ;
        Var _tmp1 ;
        _tmp1 = 12 ;
        Var _tmp2 ;
        _tmp2 = LCall _Alloc(_tmp1) ;
        Var _tmp3 ;
        _tmp3 = Cow ;
        *(_tmp2) = _tmp3 ;
        betsy = _tmp2 ;
        Var _tmp4 ;
        _tmp4 = 5 ;
        Var _tmp5 ;
        _tmp5 = - _tmp4 ;
        Var _tmp6 ;
        _tmp6 = 22 ;
        Var _tmp7 ;
        _tmp7 = *(betsy) ;
```

```
Var _tmp8 ;
_tmp8 = *(_tmp7) ;
ACall _tmp8(betsy, _tmp5, _tmp6) ;
Var _tmp9 ;
_tmp9 = *(betsy) ;
Var _tmp10 ;
_tmp10 = *(_tmp9 + 4) ;
ACall _tmp10(betsy) ;
EndFunc ;
```

Its MIPS assembly:

```
__Cow_InitCow:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
        # *(this + 4) = h
          lw $t0, 8($fp)        # load h into $t0
          lw $t1, 4($fp)        # load this into $t1
          sw $t0, 4($t1)        # store with offset
        # *(this + 8) = w
          lw $t2, 12($fp)       # load w into $t2
          sw $t2, 8($t1)        # store with offset
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
        # Save modified registers before flow of control change
  __Cow_Moo:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
          subu $sp, $sp, 4      # decrement sp to make space for locals/temps
        # _tmp0 = "Moo!\n"
          .data
          _string1: .asciiz "Moo!\n"
          .text
        # _tmp0 = _string1
          la $t0, _string1      # load label
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t0, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -8($fp)       # spill _tmp0 from $t0
          jal _PrintString      # jump to function
          add $sp, $sp, 4       # pop params off stack
        # EndFunc
        # below handles reaching end of fn body with no explicit return
          move $sp, $fp         # pop callee frame off stack
          lw $ra, -4($fp)       # restore saved ra
          lw $fp, 0($fp)        # restore saved fp
          jr $ra                # return from function
          .data
          .align 2
  Cow:          # this is the vtable for class Cow
          .word __Cow_InitCow
          .word __Cow_Moo
          .text
        # Save modified registers before flow of control change
  main:
        # BeginFunc
          subu $sp, $sp, 8      # decrement sp to make space to save ra, fp
          sw $fp, 8($sp)        # save fp
          sw $ra, 4($sp)        # save ra
          addiu $fp, $sp, 8     # set up new fp
          subu $sp, $sp, 44     # decrement sp to make space for locals/temps
        # _tmp1 = 12
          li $t0, 12            # load constant value 12 into $t0
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t0, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -12($fp)      # spill _tmp1 from $t0
          jal _Alloc            # jump to function
          move $t0, $v0
          add $sp, $sp, 4       # pop params off stack
        # _tmp3 = Cow
          la $t1, Cow           # load label
        # *(_tmp2) = _tmp3
          sw $t1, 0($t0)        # store with offset
        # betsy = _tmp2
          move $t2, $t0
        # _tmp4 = 5
          li $t3, 5             # load constant value 5 into $t3
        # _tmp5 = - _tmp4
          neg $t4, $t3
        # _tmp6 = 22
          li $t5, 22            # load constant value 22 into $t5
        # _tmp7 = *(betsy)
          lw $t6, 0($t2)        # load with offset
        # _tmp8 = *(_tmp7)
          lw $t7, 0($t6)        # load with offset
        # Set up for function call
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t5, 4($sp)        # push param value on stack
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t4, 4($sp)        # push param value on stack
          subu $sp, $sp, 4      # decrement sp to make space for param
          sw $t2, 4($sp)        # push param value on stack
        # Save modified registers before flow of control change
          sw $t0, -16($fp)      # spill _tmp2 from $t0
          sw $t1, -20($fp)      # spill _tmp3 from $t1
          sw $t2, -8($fp)       # spill betsy from $t2
          sw $t3, -24($fp)      # spill _tmp4 from $t3
          sw $t4, -28($fp)      # spill _tmp5 from $t4
          sw $t5, -32($fp)      # spill _tmp6 from $t5
          sw $t6, -36($fp)      # spill _tmp7 from $t6
          sw $t7, -40($fp)      # spill _tmp8 from $t7
          jalr $t7              # jump to function
          add $sp, $sp, 12      # pop params off stack
```

```
# _tmp9 = *(betsy)
  lw $t1, -8($fp)       # load betsy into $t1
  lw $t0, 0($t1)        # load with offset
# _tmp10 = *(_tmp9 + 4)
  lw $t2, 4($t0)        # load with offset
# Set up for function call
  subu $sp, $sp, 4      # decrement sp to make space for param
  sw $t1, 4($sp)        # push param value on stack
# Save modified registers before flow of control change
  sw $t0, -44($fp)      # spill _tmp9 from $t0
  sw $t2, -48($fp)      # spill _tmp10 from $t2
  jalr $t2              # jump to function
  add $sp, $sp, 4       # pop params off stack
# EndFunc
# below handles reaching end of fn body with no explicit return
  move $sp, $fp         # pop callee frame off stack
  lw $ra, -4($fp)       # restore saved ra
  lw $fp, 0($fp)        # restore saved fp
  jr $ra                # return from function
```

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, Assemblers, Linkers, and the SPIM Simulator. User Manual, 1998.

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

S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA: Morgan Kaufmann, 1997.

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