Lecture #22: Code Generation

[This lecture adopted in part from notes by R. Bodik]

Intermediate Languages and Machine Languages

- From trees such as output from project #2, could produce machine language directly.
- However, it is often convenient to first generate some kind of *intermediate language (IL):* a "high-level machine language" for a "virtual machine."
- Advantages:
  - Separates problem of extracting the operational meaning (the *dynamic semantics*) of a program from the problem of producing good machine code from it, because it...
  - Gives a clean target for code generation from the AST.
  - By choosing IL judiciously, we can make the conversion of IL $\rightarrow$ machine language easier than the direct conversion of AST $\rightarrow$ machine language. Helpful when we want to target several different architectures (e.g., gcc).
  - Likewise, if we can use the same IL for multiple languages, we can re-use the IL $\rightarrow$ machine language implementation (e.g., gcc, CIL from Microsoft's Common Language Infrastructure).

Stack Machines as Virtual Machines

- A simple evaluation model: instead of registers, a stack of values for intermediate results.
- Examples: The Java Virtual Machine, the Postscript interpreter.
- Each operation (1) pops its operands from the top of the stack, (2) computes the required operation on them, and (3) pushes the result on the stack.
- A program to compute 7 + 5:

```
push 7      # Push constant 7 on stack
push 5
add         # Pop two 5 and 7 from stack, add, and push result.
```

- Advantages
  - Uniform compilation scheme: Each operation takes operands from the same place and puts results in the same place.
  - Fewer explict operands in instructions means smaller encoding of instructions and more compact programs.
  - Meshes nicely with subroutine calling conventions that push arguments on stack.

Stack Machine with Accumulator

- The add instruction does 3 memory operations: Two reads and one write of the stack. The top of the stack is frequently accessed
- Idea: keep most recently computed value in a register (called the *accumulator*) since register accesses are faster.
- For an operation op($e_1, \ldots, e_n$):
  - compute each of $e_1, \ldots, e_{n-1}$ into acc and then push on the stack;
  - compute e_n into the accumulator;
  - perform op computation, with result in acc.
  - pop $e_1, \ldots, e_{n-1}$ off stack.
- The add instruction is now

```
acc := acc + top_of_stack
pop one item off the stack
```

  and uses just one memory operation (popping just means adding constant to stack-pointer register).
- After computing an expression the stack is as it was before computing the operands.

Example: Full computation of 7+5

```
acc := 7
push acc
acc := 5
acc := top_of_stack + acc
pop stack
```

Translating from AST to Stack Machine (I)

- First, it might be useful to have abstractions for our virtual machine and its operations:

```
/** A virtual machine. */
public class VM {
   /** Add INST to our instruction sequence. */
   public void emitInst(Instruction inst);
   ...
}

/** Represents machine instructions in a VM. */
public class Instruction {
   ...
}
```

Translating from AST to Stack Machine (II)

- Let's take a look at a traditional OOP approach in which code generation routines are instance methods in the AST node class.
- A simple recursive pattern usually serves for expressions.
- At the top level, our trees might have an expression-code method:

```
public abstract class Node {
    ...
    /** Generate code for me, leaving my value on the stack. */
    public abstract void cgen(VM machine);
    /** An appropriate VM instruction to use when my operands are on
     *  the stack. */
    abstract Instruction getInst();
    ...
}
```

Translating from AST to Stack Machine (III)

- Implementations of `cgen` then obey this general comment, and each assumes that its children will as well. E.g.

```
public class BinaryExpr extends Node {
    ...
    @Override
    public void cgen(VM machine) {
        left.cgen(machine);
        right.cgen(machine);
        machine.emitInst(getInst());
    }
}
```

- It is up to the implementation of VM to decide how the stack is represented: with all results in memory, or with the most recent in an accumulator.
- Code for `cgen` need not change (example of *separation of concerns*, btw).

The ChocoPy Project Approach

- As you have seen, our projects use a different program structure.
- Functions such as `cgen` are grouped into analyzers.
- Not really a traditional OOP approach, but it is nice to see the options.
- Here we might write routines such as:

```
public class CodeGenerator extends NodeAnalyzer<Void> {
   public CodeGenerator (VM machine0) {
       machine = machine0;
   }
   ...
   @Override
   public analyze(BinaryExpr node) {
       node.left.dispatch(this);
       node.right.dispatch(this);
       machine.emitInst(node.dispatch(getInstAnalyzer));
       /* I leave getInstAnalyzer to your imagination. */
   }
}
```

From Stack IL to Machine Code (I)

- Eventually, we want to produce machine language.
- To do so, we essentially write another translator from stack language to, say, RISC V.
- This can be simple (and reusable across languages).
- Sample Translation:

`acc := 7`	`li a0, 7`
`push acc`	`addi sp, sp, -4`
	`sw a0, 0(sp)`
`acc := 5`	`li a0, 5`
`acc := top_of_stack + acc`	`lw t0, 0(sp)`
	`add a0, t0, a0`
`pop stack`	`addi sp, sp, 4`

- As you can see, each statement on the left has a simple translation on the right.
- Unfortunately, there's quite a bit of stack-pointer twiddling going on.

From Stack IL to Machine Code (II)

- An alternative is to allocate all the space needed for the stack (i.e., its maximum in the current function) and keep track of the stack pointer "mentally." (In the project, you can do either, if you choose to use the stack abstraction.)
- Example.

```
   Stack                        Previous            Alternative

                                                   # At start of function
                                                   addi sp, sp, -<size>
  ...                                              ...
  acc := 7                     li a0, 7            li a0, 7
  push acc                     addi sp, sp, -4     sw a0, 12(sp) # E.g.
                               sw a0, 0(sp)
  acc := 5                     li a0, 5            li a0, 5
  acc := top_of_stack + acc    lw t0, 0(sp)        lw t0, 12(sp)
                               add a0, t0, a0      add a0, t0, t0
  pop stack                    addi sp, sp, 4
```

From Stack IL to Machine Code (III)

- So if we had to use several stack slots, we'd simply adjust the immediate offset we use from `sp` in our code.
- For example, suppose we want to translate `x * (a + b)`:

```
acc := x                      lw a0, x
push acc                      sw a0, 8(sp)  # For example
acc := a                      lw a0, a
push acc                      sw a0, 4(sp)
acc := b                      lw a0, b
acc := top_of_stack + acc     lw t0, 4(sp)
                              add a0, t0, a0
pop stack
acc := top_of_stack * acc     lw t0, 8(sp)
                              mul a0, t0, a0
pop stack
```

- (Alternatively, can use negative offsets from `fp` as stack offsets, which is what the reference compiler does.)

Virtual Register Machines and Three-Address Code

- Another common kind of virtual machine has an infinite supply of *registers*, each capable of holding a scalar value or address, in addition to ordinary memory.
- A common IL in this case is some form of *three-address code*, so called because the typical "working" instruction has the form

  target := operand$_1$ $\oplus$ operand$_2$

  where there are two source "addresses," one destination "address" and an operation ($\oplus$).
- Often, we require that the operands in the full three-address form denote (virtual) registers or immediate (literal) values, similar to the usual RISC architecture.

Three-Address Code, continued

- A few other forms deal with memory and other kinds of operation:

```
memory_operand := register_or_immediate_operand
register_operand := register_or_immediate_operand
register_operand := memory_operand
goto label
if operand1 ≺ operand2 then goto label
param operand                  ; Push parameter for call.
call operand, # of parameters  ; Call, put return in
                               ; specific dedicated register
```

- Here, $\prec$ stands for some kind of comparison. Memory operands might be labels of static locations, or indexed operands such as (in C-like notation): `*(r1+4)` or `*(r1+r2)`.

Translating from AST into Three-Address Code

- Change the `cgen` routine to return where it has put its result:

```
public abstract class Node {
    ...
    /** Generate code to compute my value, returning the location
     *  of the result. */
    public Operand cgen(VM machine);
}
```

- Where an Operand denotes some abstract place holding a value.
- Once again, we rely on our children to obey this *general* comment:

```
public class BinaryExpr extends Callable {
   public Operand cgen(VM machine) {
      Operand leftOp = left.cgen(machine);
      Operand rightOp = right.cgen(machine);
      Operand result = machine.allocateRegister();
      machine.emitInst(result, getInst(), leftOp, rightOp);
      return result;
   }
}
```

- `emitInst` now produces three-address instructions.

A Larger Example

- Consider a small language with integers and integer operations:

```
P:    D ";" P | D
D:    "def" id(ARGS) "=" E;
ARGS: id "," ARGS | id
E:    int | id | "if" E1 "=" E2 "then" E3 "else" E4 "fi"
          | E1 "+" E2 | E1 "-" E2 | id "(" E1,...,En ")"
```

- The first function definition f is the "main" routine
- Running the program on input i means computing f(i)
- Let's continue implementing `cgen` ('+' and '-' already done).

Simple Cases: Literals and Sequences

Conversion of D ";" P:

```
public class StmtList extends Node {
   ...
   public Operand cgen(VM machine) {
      for (int i = 0; i < arity(); i += 1)
         stmts.get(i).cgen(machine);
   }
   return Operand.NoneOperand;
}

public class IntegerLiteral extends Node {
   ...
   @Override
   Operand cgen(VM machine) {
       return machine.immediateOperand(value);
   }
}
```

- `NoneOperand` is an Operand that contains None.

Identifiers

```
public class Identifier : public Node {
   ...
   Operand cgen(VM machine) {
      Operand result = machine.allocateRegister();
      VarInfo info = getInfoFor(name);  // However you do this.
      machine.emitInst(MOVE, result, info.getLocation(machine));
      return result;
   }
}
```

- That is, we assume that the `VarInfo` object that holds information about this occurrence of the identifier contains enough information to get an operand that accesses it from the VM.

Calls

```
public class CallExpr extends Node {
   ...
   @Override
   public Operand cgen(VM machine) {
      for (Node arg : args)
          machine.emitInst(PARAM, arg.cgen(machine));
      Operand callable = function.cgen(machine);
      machine.emitInst(CALL, callable, args.arity());
      return Operand.ReturnOperand;
   }
}
```

- `ReturnOperand` is an abstract location where functions return their value.

Control Expressions: if (Strategy)

- Control expressions generally involve jump and conditional jump instructions.
- To translate

```
if E1 = E2 then E3 else E4 fi
```

we might aim to produce something that realizes the following pseudocode:

```
    code to compute E1 into r1
    code to compute E2 into r2
    if r1 != r2 goto L1
    code to compute E3 into r3
    goto L2
L1:
    code to compute E4 into r3
L2:
```

where the $\mathrm{r}i$ denote virtual-machine registers.

Control Expressions: if (Code Generation)

```
public class IfExpr extends Node {
    ...
    public Operand cgen(VM machine) {
        Operand leftOp = left.cgen(machine);
        Operand rightOp = right.cgen(machine);
        Label elseLabel = machine.newLabel();
        Label doneLabel = machine.newLabel();
        machine.emitInst(IFNE, left, right, elseLabel);
        Operand result = machine.allocateRegister();
        machine.emitInst(MOVE, result, thenExpr.cgen(machine));
        machine.emitInst(GOTO, doneLabel);
        machine.placeLabel(elseLabel);
        machine.emitInst(MOVE, result, elseExpr.cgen(machine));
        machine.placeLabel(doneLabel);
        return result;
    }
}
```

- `newLabel` creates a new, undefined instruction label.
- `placeLabel` inserts a definition of the label in the code.

Code generation for 'def'

```
public class FuncDef extends Node {
   ...
   @Override
   Operand cgen(VM machine) {
      machine.placeLabel(name);
      machine.emitFunctionPrologue();
      Operand result = statements.cgen(machine);
      machine.emitInst(MOVE, Operand.ReturnOperand, result);
      machine.emitFunctionEpilogue();
      return Operand.NoneOperand;
   }
}
```

- Where function prologues and epilogues are standard code sequences for entering and leaving functions, setting frame pointers, etc.

A Sample Translation

Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                if x = 2 then 1 else
                   fib(x - 1) + fib(x - 2)
```

Possible code generated:

```
f:  function prologue
    r1 := x
    if r1 != 1 then goto L1
    r2 := 0
    goto L2
L1: r3 := x
    if r3 != 2 then goto L3
    r4 := 1
    goto L4
L3: r5 := x
    r6 := r5 - 1
    param r6
    call fib, 1
    r7 := rret
    r8 := x
    r9 := r8 - 2
    param r9
    call fib, 1
    r10 := r7 + rret
    r4 := r10
L4: r2 := r4
L2: rret := r2
    function epilogue
```

Some Comments About the RISC V ABI

- An *Application Binary Interface (ABI)* is a set of low-level conventions describing how modules in a program communicate at the level of machine code such as register use, calling conventions, data alignment, and system calls.
- For the purposes of project 3, we will depart in a few ways from the standard conventions used for RV32IM installations:
  - In the standard convention, the first 8 arguments to a function are passed in registers `a0`–`a7` (`x10`–`x17`), either directly (if they fit in 32 bits) or by reference. Later arguments are placed on the stack.
  - In our conventions, all parameters are on the stack, with the last argument on top. We don't have to deal with quantities larger than 32 bits.
  - In the standard convention, the stack pointer is always aligned on a 16-byte boundary. This helps when data types require proper alignment in memory for correctness or performance.
  - We don't use that convention, although the reference compiler happens to abide by it.

Converting Three-Address Code to RV32 Code

- The problem is that in reality, the RV architecture has fewer physical registers than our three-address code generator from last time typically allocates as virtual registers.
- *Register allocation* is the general term for assigning virtual registers to real registers or memory locations.
- When we run out of real registers, we *spill* values into memory locations reserved for them.
- We keep a register or two around as *compiler temporaries* for cases where the instruction set doesn't let us just combine operands directly.

A Simple Strategy: Local Register Allocation

- It's convenient to handle register allocation within *basic blocks*—sequences of code with one entry point at the top and any branches at the very end.
- At the end of each such block, spill any registers whose values are needed in other basic blocks.
- To do this efficiently, need to know when a register is *dead*—that is, when its value is no longer needed. We say that a register *dies* in an instruction that uses its value if no other instruction will use that value before another value is assigned.
- We'll talk about how to compute that in a later lecture. Let's assume we know it for now.
- Let's also assume that each virtual register representing a local variable or intermediate result has a memory location reserved for it on the stack suitable for spilling.

Simple Algorithm for Local Register Allocation (I)

First, we need some supporting data structures and functions:

- A set `availReg` of available physical (i.e. real) registers. Initially, this contains all physical registers available for assignment. (There may also be some "very temporary" registers around to help with certain instructions).
- A function `dies(pc)` that returns the set of virtual registers that die in the instruction at `pc`.
- A mapping `realReg` from virtual registers to the current physical registers that hold them (if any).
- A boolean function `isReg(x)` that returns true iff `x` is a virtual register (as opposed to an immediate or missing operand).
- A function `spillReg(pc)` that chooses an allocatable physical register not in `availReg` (that is, currently assigned to some virtual register), generates code to write its contents to the place reserved for that virtual register on the stack, marks the spilled virtual register as dying at `pc`, returns the physical register.

Simple Algorithm for Local Register Allocation (II)

- We execute the following for each three-address instruction in a basic block (in turn).

```
# Allocate registers to an instruction x := y op z or x := op y
# [Adopted from Aho, Sethi, Ullman]
def regAlloc(pc, x, y, z):
    if realReg[x] != None or dies(x, pc):
        "No new allocation needed"
    elif isReg(y) and y in dies(pc):
        realReg[x] = realReg[y];
    elif isReg(z) and z in dies(pc):
        realReg[x] = realReg[z];
    elif len(availReg) != 0:
        realReg[x] = availReg.pop()
    else:
        realReg[x] = spillReg(pc)
```

- After generating code for the instruction at `pc`,

```
for r in dies(pc):
    if realReg[r] != realReg[x]:
        availReg.add(realReg[r])
    realReg[r] = None
```

Function Prologue and Epilogue for the RV32

- Consider a function F that needs K bytes of local variables, saved registers, and other compiler temporary storage for expression evaluation.
- We'll consider the case where we keep a frame pointer.
- Overall, the code for a function, F, looks like this:

F:

```
# Prologue
addi sp, sp, -K        # Reserve space for locals, saved regs, etc.
sw ra, K-4(sp)         # Save return pointer
sw fp, K-8(sp)         # Save dynamic link (caller's frame pointer)
addi fp, sp, K         # Set new frame pointer.
code for body of function, leaving value in a0
# Epilog
lw ra, -4(fp)          # Restore ra
lw fp, -8(fp)          # Restore frame pointer
addi sp, sp, K         # Pop stack
jr ra                  # Return (short for 'jalr x0, ra, 0')
```

Code Generation for Local Variables (Review)

- We store local variables are stored on the stack (thus not at fixed addresses).
- One possibility: access relative to the stack pointer, but
  - Sometimes convenient for stack pointer to change during execution of of function, sometimes by unknown amounts.
  - Debuggers, unwinders, and stack tracers would like a simple way to compute stack-frame boundaries.
- Solution: use a frame pointer, which is constant over execution of function.
- In our convention, the frame pointer always points to the last (lowest-addressed) word on the stack of the *caller*, which holds the last function argument (if any).
- Thus, since our words are 4 bytes long, parameter i of a K-arguement function is at location frame pointer $+\ 4(K - i - 1)$.
- The caller registers `ra` and `fp` are saved at `-4(fp)` and `-8(fp)`, respectively, with other saved registers, local variables, and temporaries starting at `-12(fp)`.

Accessing Non-Local Variables (Review)

- In program on left, how does f3 access x1?
- Our convention is that that functions pass static links just before the first parameter of their callees (so that for the callee, it ends up at frame pointer $+ 4K$ for a K-parameter function.)
- The static link passed to f3 will be f2's frame pointer.

```
def f1(x1):
   def f2(x2):
       def f3(x3):
          ... x1 ...
       ...
       f3(12)
   ...
   f2(9)
```

```
# To access x1 in f3:
  lw t0, 4(fp)    # Fetch FP for f2
  lw t0, 4(t0)    # Fetch FP for f1
  lw t0, 0(t0)    # Fetch x1.

# When f2 calls f3:
  addi sp, sp, -8 # Allocate space for parameters
  li t0, 12
  sw t0, 0(sp)    # Pass parameter
  sw fp, 4(sp)    # Pass f2's frame to f3
  jal ra, f3
  addi sp, sp, 8  # Restore stack pointer
```

Accessing Non-Local Variables (II)

- We'll say a function is at nesting level 0 if it is at the outer level, and at level $k+1$ if it is most immediately enclosed inside a level-k function. Likewise, the variables, parameters, and code in a level-k function are themselves at level $k+1$ (enclosed in a level-k function).
- In general, for code at nesting level n to access a variable at nesting level $m \le n$, perform $n-m$ loads of static links.

Calling Function-Valued Variables and Parameters

- As we've seen, a function value can be represented by a code address and a static link (let's assume code address comes first).
- So if (as an extension to our Project 3) we need to call a function parameter:

```
def caller(f):
    f(42)
```

`caller` could receive a pointer to a *closure object* containing the code pointer and static link for `f`. Then the call `f(42)` might get translated to:

```
addi sp, sp, -8       # Allocate argument list.
li t0, 42
sw t0, 0(sp)
lw t0, 0(fp)          # Get address of function value f
lw t1, 4(t0)          # Get static link for f
sw t1, 4(sp)          # Pass to f
lw t0, 0(t0)          # Get address of f's code
jalr ra, t0, 0        # Call
addi sp, sp, 8        # Restore sp
```

Using Registers for Parameters

- For simplicity, we're using the stack for everything.
- But it's useful to see why the RISC-V architects chose an ABI in which parameters go to registers.

Using Stack

```
addi sp, sp, -8
li t0, 42        # Param to f
sw t0, 0(sp)     # Push param
lw t0, 0(fp)     # Load f
lw t1, 4(t0)     # Load static link
sw t1, 4(sp)     # Push as param #2
lw t0, 0(t0)     # Load code address
jalr ra, t0, 0
addi sp, sp, 8
```

Using Registers

```
lw t0, 0(a0)  # Load code for f
lw a1, 4(a0)  # Static link from f
li a0, 42     # Param to f



jalr ra, t0, 0
```

Avoiding Pushes and Pops

- Don't really need to push and pop the stack as I've been doing. Here's an alternative when translating

```
def f(x, y):
   g(x); g(y); ...
```

```
f:   addi sp, sp, -8
     sw ra, 4(sp)
     sw fp, 0(sp)
     addi fp, sp, 8
     lw t0, 4(fp)     # x
     addi sp, sp, -4  # Push to stack
     sw t0, 0(sp)
     jal ra, g
     addi sp, sp, 4   # Pop from stack
     lw t0, 0(fp)     # y
     addi sp, sp, -4  # Push to stack
     sw t0, 0(sp)
     etc.
```

```
f:   addi sp, sp, -12
     sw ra, 8(sp)
     sw fp, 4(sp)
     addi fp, sp, 12
     lw t0, 4(fp)     # x

     sw t0, 0(sp)
     jal ra, g

     lw t0, 0(fp)     # y

     sw t0, 0(sp)
     etc.
```

...and you can continue to use the depressed stack pointer for arguments on the right.