Lecture 34: Registers, Functions, Parameters

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 $m \leq n$, 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               Using Registers

addi sp, sp, -8
li t0, 42                 lw t0, 0(a0)  # Load code for f
sw t0, 0(sp)              lw a1, 4(a0)  # Static link from f
lw t0, 0(fp)              li a0, 42     # Param to f
lw t1, 4(t0)
sw t1, 4(sp)
lw t0, 0(t0)
jalr ra, t0, 0            jalr ra, t0, 0
addi sp, sp, 8
```

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                  f:   addi sp, sp, -12
     sw ra, 4(sp)                          sw ra, 8(sp)
     sw fp, 0(sp)                          sw fp, 4(sp)
     addi fp, sp, 8                        addi fp, sp, 12
     lw t0, 4(fp)     # x                  lw t0, 4(fp)     # x
     addi sp, sp, -4
     sw t0, 0(sp)                          sw t0, 0(sp)
     jal ra, g                             jal ra, g
     addi sp, sp, 12  # restore sp
     lw t0, 0(fp)     # y                  lw t0, 0(fp)     # y
     addi sp, sp, -4
     sw t0, 0(sp)                          sw t0, 0(sp)
     etc.                                  etc.
```

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

Parameter Passing Semantics: Value vs. Reference

- So far, our examples have dealt only with *value parameters*, which are the only kind found in C, Java, and Python

  Ignorant comments from numerous textbook authors, bloggers, and slovenly hackers notwithstanding [End Rant].

- Pushing a parameter's value on the stack creates a copy that essentially acts as a local variable of the called function.
- C++ (and Pascal) have *reference parameters*, where assignments to the formal are assignments to the actual.

```
void incr(int& x) {                y = 4;
    x += 1;                        incr(y); // Now y == 5.
}
```


Implementation of Reference Parameters

- Implementation of reference parameters is simple:
  - Push the address of the argument, not its value, and
  - To fetch from or store to the parameter, do an extra indirection.

```
void incr(int& x) {                y = 4;
    x += 1;                        incr(y);
}

incr:                              # Assume y at -12(fp)
   # Prologue goes here            li t0, 4
   lw t0, 0(fp)                    sw t0, -12(fp)
   lw t1, 0(t0)                    addi t0, fp, -12   # &y
   addi t1, t1, 1                  addi sp, sp -4
   sw t1, 0(t0)                    sw t0, 0(sp)
   # Epilogue goes here            jal incr
                                   addi sp, sp, 4
```


Copy-in, Copy-out Parameters

- Some languages, such as Fortran and Ada, have a variation on this: *copy-in, copy-out*. Like call by value, but the final value of the parameter is copied back to the original location of the actual parameter after function returns.
  - "Original location" because of cases like `f(A[k])`, where `k` might change during execution of `f`. In that case, we want the final value of the parameter copied back to `A[k0]`, where `k0` is the original value of `k` before the call.
  - Question: can you give an example where call by reference and copy-in, copy-out give different results?

Implementation of Copy-in/Copy-out Parameters

- We can implement copy-in/copy-out as a variation of the by-reference implementation.

```
void incr(int& x) {                    y = 4;
    x += 1; etc.                       incr(y);
}

incr:
   # Prologue goes here.               # Assume y at -12(fp)
   # Allocate local at -12(fp) for x   li t0, 4
   lw t0, 0(fp)                        sw t0, -12(fp)
   lw t0, 0(t0)                        addi t0, fp, -12   # &y
   sw t0, -12(fp)  # Copy in           addi sp, sp, -4
   lw t0, -12(fp)                      sw t0, 0(sp)
   addi t0, t0, 1                      jal incr
   sw t0, -12(fp)                      addi sp, sp, 4
   # etc.  (modify -12(fp) only)
   lw t0, 0(fp)
   lw t1, -12(fp)
   sw t1, 0(t0)    # Copy out
   # Epilogue goes here
```


Parameter Passing Semantics: Call by Name

- Algol 60's definition says that the effect of a call $P(E)$ is as if the body of P were substituted for the call (dynamically, so that recursion works) and E were substituted for the corresponding formal parameter in the body (changing names to avoid clashes).
- It's a simple description that, for simple cases, is just like call by reference:

```
procedure F(x)                  F(aVar);
   integer x;                becomes
begin                           aVar := 42;
   x := 42;
end F;
```

- But the (unintended?) consequences were "interesting".

Call By Name: Jensen's Device

- Consider:

```
procedure DoIt (i, L, U, x, x0, E)
    integer i, L, U; real x, x0, E;
begin
    x := x0;
    for i := L step 1 until U do
        x := E;
end DoIt;
```

- To set y to the sum of the values in array A[1:N],

```
integer k;
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

- To set z to the Nth harmonic number:

```
DoIt(k, 1, N, z, 0.0, z+1.0/k);
```

- Now how are we going to make this work?

Call By Name: Implementation

- Basic idea: Convert call-by-name parameters into parameterless functions (traditionally called *thunks*.)
- To allow assignment, these functions can return the addresses of their results.
- So the call

```
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

  becomes something like (please pardon highly illegal notation):

```
integer t1;  real t2, t3, t4;
t2 := 1.0; t3 := 0.0;
DoIt(lambda: &k, lambda: &t2, lambda: &N, lambda: &y,
     lambda: &t3, lambda: (t4 := y+A[k], &t4));
```

- Later languages have abandoned this particular parameter-passing mode.