Compilers Course
Lecture 16: Translating RTL to Machine Code

RTL is still higher-level than machine code:

- Unbounded number of virtual registers (temps)
- Temps survive recursive function calls
- Conditional jumps that combine comparisons with jumps
- Non machine-specific ops (arithmetic, loads/stores)

Code generation in an optimizing compiler

1. Optimize the RTL code.
2. Convert RTL instructions to machine instructions.
3. Try to allocate temps to hardware registers.
4. Introduce stack frames:
   - Map unallocated temps to slots in the stack frame.
   - Save and restore caller-save registers around recursive calls.
   - Save and restore callee-save registers around function bodies.
5. Optimize the machine code.
6. Generate binary code.

A lot of work.

Simple code generation

a) Skip code optimization.

b) Skip register allocation. All temps are located in the stack.
RTL to machine code conversion can be done on-the-fly as each RTL instruction is processed.

c) Divide the stack frame into a fixed size part for the temps (addressed by FP - offset), and a variable size part for parameters in recursive calls (addressed by SP + offset).

d) Use a symbol table to record each temp's offset from FP:
- process the formal parameters and record their FP offsets
- first lookup of a local temp:
  * the temp is not bound
  * increase fixed frame size by one word
  * record negation of new size as temp's offset
- subsequent lookups:
  * the temp is bound, return its offset

At the end of the translation we know how much space is needed for temps in the frame. Call it FRAME_SIZE.

e) Use a small number (3) of scratch registers. Call them R0, R1, R2.
Then rewrite RTL computational instructions

```
dst := src1 op src2
```

as

```
R1 := load FP - offset(src1)  // look up src1
R2 := load FP - offset(src2)  // look up src2
R0 := R1 rtl_to_machine(op) R2
store R0, FP - offset(dst)    // look up dst
```

f) For a call instruction

```
dst := f(arg0, ..., arg(N-1))
```

Assume
- The calling convention specifies that the first *I* arguments are passed in registers *R(0)..R(I-1)*, and that the remaining *J = N-I* arguments are passed on the stack in *SP[0]..SP[J-1]*.
- The return address is placed in *RA* by the call instruction.
- The return value is placed in *R0*.

Rewrite the call as follows

```
SP := SP - J*4 // 4 == sizeof(a machine word)

load R0, FP - offset(arg(I+0))
store R0, SP + 0*4
load R0, FP - offset(arg(I+1))
store R0, SP + 1*4
...
load R0, FP - offset(arg(I+J-1))
store R0, SP + (J-1)*4

load R0, FP - offset(arg0)
load R1, FP - offset(arg1)
...
load R(I-1), FP - offset(arg(I-1))

call f
store R0, FP - offset(dst)
SP := SP + J*4
```

g) At the start of the code for a procedure *P* with *N* arguments, output prologue:

```
     .text
     .globl P
P:
     SP := SP - P_FRAME_SIZE            // allocate P's frame
     store FP, SP + (P_FRAME_SIZE - 4)  // save caller's FP
     store RA, SP + (P_FRAME_SIZE - 8)  // save RA
     FP := SP + P_FRAME_SIZE            // set up P's FP
```

If the first *I* arguments are in registers, store them in the frame:

```
     store R0, FP - 12
     store R1, FP - 16
     ...
     store R(I-1), FP - 12 - 4*(I-1)
```

h) At the end of the code for a procedure *P*, output epilogue:

```
P_FRAME_SIZE=<P's FRAME_SIZE>          // symbolic constant
Lreturn:
     load RA, FP - 8                   // restore RA
     load FP, FP - 4                   // restore caller's FP
     SP := SP + P_FRAME_SIZE           // deallocate P's frame
     ret                               // return to caller
```

i) For a global variable *X* with size *S* and alignment *A*, output:

```
     .data
     .align A
X:   .space S
```

If the variable has an initial value *V*, instead output:

```
X:   .word V
```