Warm-Up Problem

Given that NULL is 1 in WLP4, how (using WLP4 code only) can you cheat and get a pointer to 0?

CS 241 Lecture 20

Code Generation Procedures + Optimization
With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Register $29: Callee-Save

Let's discuss both approaches. Assume that we require that the callee will save $29. Thus, they will initialize $29 first:

- `g:  sub $29, $30, $4`

and then g saves registers. Is this the right order to do things in?

Saving Registers First

If we save registers first...

- \$29 is supposed to point to the beginning of the stack frame, but \$30 has already changed to store all the registers!
- That's fine for now; the only issue is that \$29 is then pointing somewhere in the middle of the stack frame.

Saving Registers First

- If we only want to find things in our own stack frame, we don't need to care where \$29 is within the stack frame, just where it is relative to our variables.
- Eventually, we'll want to access things from the caller's stack frame (think arguments!), so having \$29 in the middle is annoying.
- Either option makes accessing something annoying; we'll choose to make \$29 the bottom.

Saving $29 First (Callee-Saved)

If we initialize $29 first…

- Even if we want to update $29 before saving registers, there's one register we must save first: $29 itself!
- The convention we used before was that $29 was one word *past* $30; well, now we have a new use for that word: to store $29!
- Therefore, we do:

  ```
  push( $29 )
  add  $29 ,  $30 ,  $0
  ; push  other  registers
  ```

  then pop $29 at the very end. This callee-save approach with $29 will work.

Caller-save

- ... then again; we could just have the caller save $29:

```
push( $29 )
push( $31 )
lis  $5
.word g
jalr $5
pop( $31 )
pop( $29 )
```

- ... this seems far easier. We're going to choose to do this.

A Note on Locations

- Be very careful about where everything is relative to $29!
- Previously, the first variable was at 0($29). That was fine because wain didn't have to store registers.
- If you have callee-saved registers (*any* caller-saved registers!), 0($29) is a saved register, and so is -4($29) etc., depending on how many registers you save.
- Let's draw the stack on the board…

Arguments

- We need to store the arguments to pass to a function.
- We've already discussed that such things need to be stored on the stack (not enough registers).
- For factor → ID(expr1,..., exprn), we have...

```
code(factor) = push($29) + push($31)
             + code(expr1) + push ($3)
             + code(expr2) + push ($3)
             + ...
             + code(exprn) + push ($3)
             + lis $5
             +.word ID
             + jalr $5
             + pop n times (pop all regs)
             + pop($31) + pop($29)
```

Arguments

For procedure $\rightarrow$ int ID(params){dcls stmts RETURN expr;}, we have

- code(procedure) = ID: sub $29, $30, $4
  - + ;Save regs here?
  - + code(dcls) ;local vars
  - + ;OR save regs here?
  - + code(stmts)
  - + code(expr)
  - + pop regs ;restore saved regs
  - + add $30, $29, $4

Question: when do we save registers? Before code(dcls) or after?

Stack

Let's assume we save them before. What does the stack look like now?

$30:	.	
	local vars of g	frame of g
	saved regs of g	frame of g
$29:	args of g	frame of f
	$31	frame of f
	$29	frame of f
	.	

What is weird? Hint: The picture corresponds to our situation but something you need to keep of track of is off.

Symbol Table Revisited

Consider:

```
int g(int a, int b){ int c = 0; int d;}
```

What does the symbol table for `g` look like?

Symbol	Type	Offset (from $29)
a	int	8
b	int	4
c	int	??
d	int	??

That's not very good: the saved regs come before the local variables, so `c` and `d` have strange values!

Revisiting Arguments Translation

Let's try pushing the registers after pushing the declarations.

For `procedure` → `int ID(params){dcls stmts RETURN expr;}`, we have

```
code(procedure) = ID: sub $29, $30, $4
            + code( dcls) ;local vars
             + push regs ;save used regs
            + code(stmts)
            + code(expr)
            + pop regs ;restore regs
            + add $30, $29, $4
            + jr $31
```

(Note that `push dcls` really means find the code for the declarations and then push them to the stack.)

Revised Stack

\$30/	.	
	saved regs of g	frame of g
	local vars of g	frame of g
\$29/	args of g	frame of f
	\$31	frame of f
	\$29	frame of f
	.	

Symbol Table Revisited Revisited

Consider:

```
int g(int a, int b){ int c = 0; int d;}
```

Symbol	Type	Offset (from $29)
a	int	8
b	int	4
c	int	0
d	int	-4

Notice that we added $4 \cdot$ #params to all of the offsets in the table

Summarizing

- Parameters should have positive offsets!
- Local variables should have non-positive offsets!
- Symbol table should have added $4 \cdot$ #params to each entry in the table.

Note

- This complicates pushing registers, because we’re now generating some code *before* we preserve register values.
- Does this matter for us?
- What change to the language or compiler would make it matter for us?
- How would we need to change if it did?

More Fun: Labels

Another annoying problem. Consider this code:

```
int print(int a) {
    return  a;
}
```

- What is the problem here?
- We already have a label called `print`! But, it’s not a WLP4 procedure, and shouldn’t interfere with WLP4.
- We will have multiply defined labels if we use the function name as a label in our MIPS code!

Options

- We could just ban WLP4 code that uses function names that match some of our reserved labels like new, init, etc. This isn't very futureproof though...
- Instead, since nobody said the MIPS labels have to be identical to the WLP4 procedure names, we can change them!

Options

- We'll prepend an 'F' to the front of labels corresponding to procedures. Then, so long as we don't create any labels with a F at the beginning for any other purpose, we should be okay.
- For example, the print function above would correspond to a label Fprint.
- "Real world" example: On many systems, a C function named "foo" generates a label "_foo".
- Need to revisit the previous translations:

Revisiting Translation

factor → ID(expr1,..., exprn), we have...

```
code(factor) = push($29) + push($31)
               + code(expr1) + push ($3)
               + code(expr2) + push ($3)
               + ...
               + code(exprn) + push ($3)
               + lis $5
               +.word FID
               + jalr $5
               + pop n times (pop all regs)
               + pop($31) + pop($29)
```

Notice that we added the F above in the .word line. We then label the procedure FID in the code for procedures.

Example

Let’s do a complete example, with procedures:

```
int add(int a, int b) {
    int c = 0;
    c = a + b;
    return c;
}

int wain(int a, int b) {
    return add(a, b);
}
```

Fin (of the compiler part…)

- We now have a compiler! Job done!
- Next, we'll do optimizations. Optimizations is an optional module (you won't be tested on it), but is worth going over to deepen your understanding of compilation.
- Several times outsource jobs to the *runtime environment*. After optimizations, we will return to that runtime environment.