Warm Up Problem

Is it better to save registers from the **caller** or the **callee**? Which registers do we not have a choice about **caller** versus **callee** saving?

CS 241 Lecture 20

Code Generation Procedures

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Procedures

We need to now deal with multiple function calls. There are a bunch of factors to consider:

- Who should save which registers? The caller? The callee (the function being called)?
- What do functions need to update/initialize?
- How do we have to update our symbol table?
- How do other functions differ from `wain`?
- How do we handle parameter passing?

Recall: `wain`

What do we need to do for `wain`?

- Load `import`, `init`, `new`, `delete`
- Initialize $4, $10, $11. (Don't need this for other procedures!)
- Call `init`
- Save $1, $2 (these are our parameters).
- Reset stack (at end)
- Call `jr $31` (at end)

General Procedures

How does the previous slide change for a generic procedure?

General Procedures

How does the previous slide change for a generic procedure?

- Don't need any imports
- Need to update $29
- Save registers
- Restore registers and stack and `jr $31`

Saving and Restoring Registers

Who is responsible for saving and restoring registers? There are two options:

Definition

The **caller** is a function `f` that calls another function `g`.

The **callee** is a function `g` that is being called by another function `f`.

Note that `f` could be `g`. One of the above two must be saving registers. Which one should we pick?

Current State

Our current convention:

- Caller needs to save \$31. Otherwise, we lose the return address (to say the loader) once we complete our call to `jalr`.
- Callee has been saving registers it will modify and restoring at the end. The function `f` shouldn't be worried about which registers `g` might be using. This makes sense as well from a programming point of view.
- Note that we have only used registers from \$1 to \$7 (and registers \$4, \$10, \$11 are constant) as well as registers \$29, \$30, and \$31.
- We don't need to worry about \$30.
- Who should save \$29?

Register $29 - Callee-Save

We 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 save g's registers.

Why must it be in this order?

Saving Registers First

If we save registers first...

Saving Registers First

If we save registers first...

- Then \$29 needs to point at the beginning of the stack frame - however \$30 has changed to store all the registers!
- We would then need to keep track of how many registers we stored to correctly update \$29.
- So \$29 = \$30 + 4· regs saved.
- Seems tedious... let's initialize \$29 first.

Saving $29 First

If we initialize $29 first...

- Note we need to push $29 onto the stack first before updating $29 (why!?)
- We can do this but this updates the stack pointer.
- That's okay because our convention is to store $29 as one word higher than the very bottom of our stack frame!
- 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 do a caller-save:

```
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.

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` $\rightarrow$ `ID(expr1,..., exprn)`, we have...

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` $\rightarrow$ `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
                + jr $31
```

Arguments

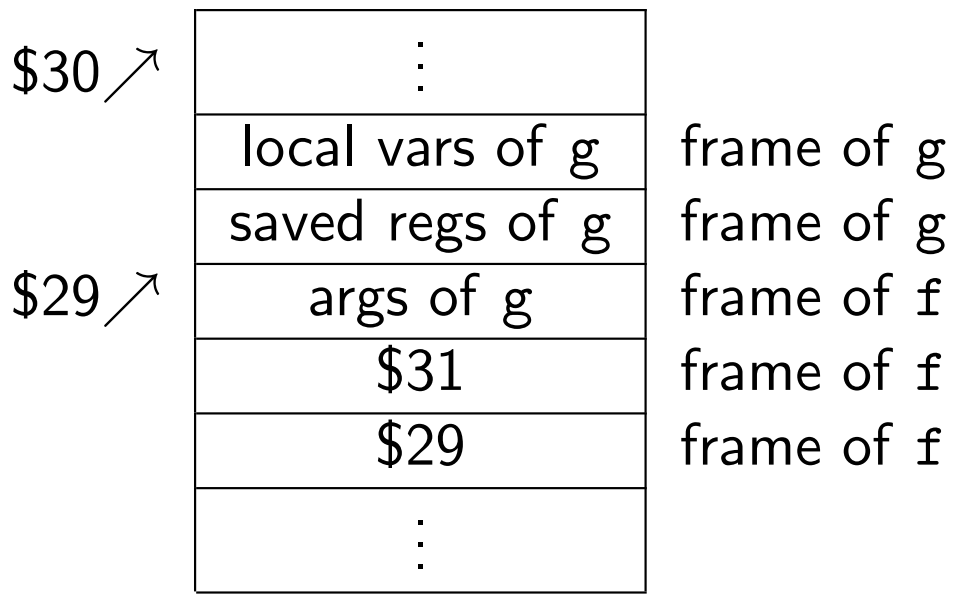
For `procedure` $\rightarrow$ `int ID(params){dcls stmts RETURN expr;}`, we have

```
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
                + jr $31
```

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?



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! Also, parameters a and b are *below* $29 (is this a problem?!)

Revisiting Arguments Translation

Let's try pushing the registers after pushing the declarations. For `procedure` $\rightarrow$ `int ID(params){dcls stmts RETURN expr;}`, we have

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

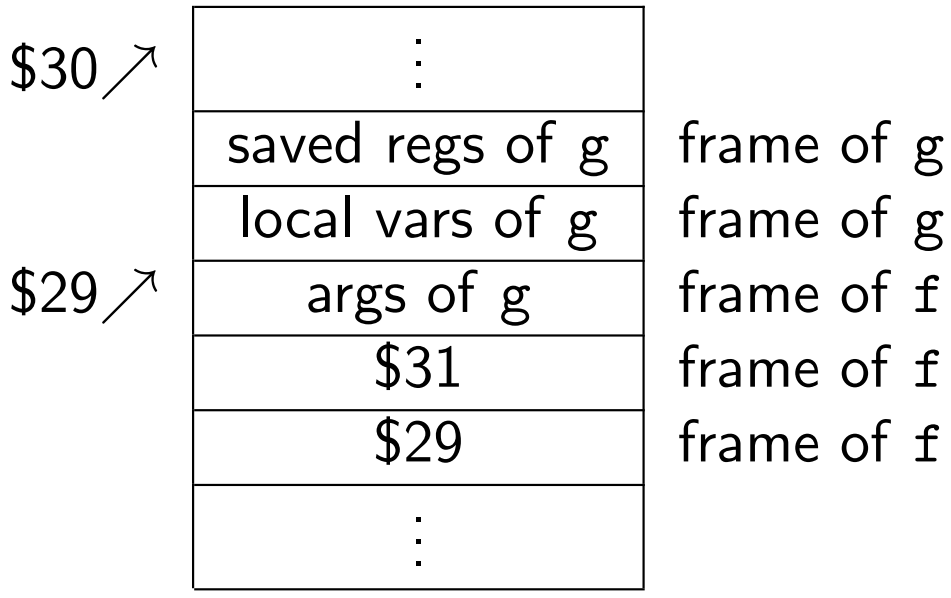
Revisiting Arguments Translation

Let's try pushing the registers after pushing the declarations. For `procedure` $\rightarrow$ `int ID(params){dcls stmts RETURN expr;}`, we have

```
code(procedure) = ID: sub $29, $30, $4
                + push 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



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 $4 \cdot \#\text{params}$ 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 $4 \cdot \#\text{params}$ to each entry in the table.

Thought Experiment

- The above was done with `callee-saved` registers.
- What if we did this with `caller-saved` registers?
- Do we get a savings? What if a function calls another function multiple times in succession? Would only then need to save registers once.
- We leave this as food for thought and switch topics next.

More Fun: Labels

Another annoying problem. Consider this code:

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

What is the problem here?

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`! Thus, 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 seems unnecessary though.
- Since we are generating the labels for a program, we can just rename them by adding something extra to labels for functions.
- We choose the latter option and will append an `F` to the front of labels corresponding to functions. 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`.
- Need to revisit the previous translations:

Revisiting Translation

`factor` $\rightarrow$ `ID(expr1,..., exprn)`, we have...

Revisiting Translation

`factor` $\rightarrow$ `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.

Optimization

How do we optimize our code?

- In general this problem is really difficult.
- In practice, we want to minimize the runtime of our compiler and the subsequent code.
- For us, we chose to minimize line numbers (since it is an easy to measure metric)
- Could easily spend an entire course on this.
- We discuss a few ideas

Constant Folding

Consider the following code:

```
int wain(int a, int b){
  return 2 + 3;
}
```

Naively, we might do something like this

Constant Folding

```
int wain(int a, int b){return 2 + 3;}

;Insert preamble here
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 3
lw $5, 0($30)
add $30, $30, $4
add $3, $3, $5
```

Issues

- In the previous code, we use the stack but we really didn't have to - we could have just loaded 3 into a different register.
- Further to this, our programming languages could have just done the operation to begin with instead of writing MIPS code to do it - really all we want to do is:

  ```
  lis $3
  .word 5
  ```

  and our code would be equivalent and correct
- This is called constant folding.

Constant Propagation

- What if a value created never changes for throughout the running of the function?
- You should be able to replace the variable with the constant first and then use the constant folding trick.
- For example:

```
int wain(int a, int b){
  int x = 5;
  return x + x;
}
```

Naive Translation (ignore preamble)

```
int wain(int a, int b){int x = 5; return x + x;}

;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lw $3, -8($29) ;offset for x
sw $3, -4($30) ;push $3
sub $30, $30, $4
lw $3, -8($29)
lw $5, 0($30)  ;pop($5)
add $30, $30, $4
add $3, $3, $5
```

Improvements

```
int wain(int a, int b){int x = 5; return x + x;}
```

- One solution is to note that $x = 5$ and then do:

```
;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 10
```

- Perhaps even easier - if you only use x here, then you don't need a stack entry!

```
lis $3
.word 10
```

Improvements

```
int wain(int a, int b){int x = 5; return x + x;}
```

- Another trick is to note that you are calling the same expression so you could do:

```
;Insert preamble for $1 $2 and $29
lis $3
.word 5
sw $3, -4($30)
sub $30, $30, $4
lw $3, -8($29) ;offset for x
add $3, $3, $3
```

  This last trick is called **common subexpression elimination** (see next slide)
- As another example, if you see say $(a - b * c) * (a - b * c)$, then you do not need to compute this value twice and instead just compute the first term and then perform the multiplication with itself.

Dead Code Elimination (Yes Cs 245 Is Useful!)

Recall from CS 245:

- Dead code is code that cannot be reached no matter what input is given.
- This is often a result of code following tests that always simplify to false, for example:

```
int wain(int a, int b){
  if ( a < b){
    if (b < a){
      //Dead Code
    }
  }
}
```

- Another dead code situation is to compute values that are never used.
- Detecting this code and removing it would improve runtime.

Register Allocation

- Accessing variables on RAM is expensive.
- It would be nice to use a register for a variable for the duration of the time it is needed and then store it back on RAM.
- For our scheme, the registers from \$14 to \$28 are unused. Using these should give optimizations.
- Caveat: When you take the address of operator, doing so of a register won't make sense - you must give a RAM address (and you need to make sure the value is updated for pointers if applicable).
- Which variables should we store here? Most used? Recently used? Try to do it so that variables are in registers when in a **live range** and then remove them outside of this range.

Example of Live Ranges

```
1  int wain(int a, int b){
2    int x = 0; int y = 0; int z = 0;
3    x = 3;
4    y = 10;
5    println(x);
6    z = 7;
7    y = y - x;
8    y = y - z;
9    println(z);
10   return z
11 }
```

Live ranges:

`x` lines 3-7 `y` lines 4-8 `z` lines 6-10

Note: It would be easy here to put all three variables into registers but in general, reducing to the live range could optimize code.

Strength Reduction

In the real world, addition is much faster than multiplication. If you can avoid multiplying this is best. Consider `code(n*2;)`:

```
;load n into $3
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, -4($30)
add $30, $30, $4
mult $3, $5
mflo $3
```

This is more work than:

```
add $3, $3, $3
```

Inlining Procedures

Consider the following:

```
int f(int a, int b){
  return a + b;
}

int wain(int a, int b){
   return f(a,b)
}
```

It would be far faster to use the equivalent code:

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

as we wouldn't need to have all of the overhead associated with the function call!

Inlining Procedures

Does the previous slide always give us shorter code?

Inlining Procedures

Does the previous slide always give us shorter code?

- If we call `f` a lot, then we might get many copies of `f`.
- This can be a win if the body of `f` is shorter than the code to call `f`.
- Sometimes this is harder to tell. Can also cause problems with recursive calls.
- If all calls to `f` are inline, then we never need the code for the procedure `f`.

Tail Recursion

Consider the following:

```
int fact(int n, int acc){
  if (n == 0){return acc;}
  else {return fact(n-1, acc*n);}
}
```

- Notice that the last operation made in each branch is just returning a value (that is, after the recursive call, there is no more work to do in this stack frame).
- We could reuse the current stack frame to save some operations.
- This is called **tail recursion**.
- Can we do this in our WLP4 grammar?

Tail Recursion

Consider the following:

```
int fact(int n, int acc){
  if (n == 0){return acc;}
  else {return fact(n-1, acc*n);}
}
```

- Notice that the last operation made in each branch is just returning a value (that is, after the recursive call, there is no more work to do in this stack frame).
- We could reuse the current stack frame to save some operations.
- This is called **tail recursion**.
- Can we do this in our WLP4 grammar? No! return statements are not allowed in if statements and we cannot refactor the code to get this behaviour!

Hypothetically

Let's suppose that we did allow for multiple return statements like this however. What would tail recursive code look like? Let's look at

```
return fact(n-1, acc*n);
```

which is of the form `factor` $\rightarrow$ `ID(expr1,...,exprn)`. Then this would be:

```
code(factor) = code(expr1) + push ($3)
             + ...
             + code(exprn) + push($3)
```

Code for Factor

factor → ID(expr1,...,exprn)

```
code(factor) = code(exprn) + push($3)
             + ...
             + code(expr1) + push($3)
             + pop($5)
             + sw $5, 4n($29)
             + pop($5)
             + sw $5, 4(n-1)($29)
             + ...
             + pop($5)
             + sw$5, 4($29)
             + lis $5
             +.word FID
             + add $30, $29, $4 ;reset stack ptr
             + jr $5 ;NOT jalr don't save
                     ;$31, $29 or pop args
```

Code for Procedure

```
code(procedure) = FID: sub $29, $30, $4
                + push(registers)
                + code(statements)
                + code(expr)
                + pop(registers)
                + add $29, $30, $4
                + jr $31
```

Making This Work

- To make the above work, what is often done is to write equivalent but simpler code *inside the current language (WLP4)*.
- This is sometimes called *intermediate code*.
- Once you have this code, run this through your compiler and then use the MIPS output.