

Warm-Up Problem

- Find three distinct one-character changes that make this code fail to type-check (while still passing parsing):

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

CS 241 Lecture 16

Code Generation

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot,
and Carmen Bruni

Food For Thought

There are infinitely many equivalent MIPS programs for a single WLP4 program. Which should we output?

- Correctness is most important!
- For us, we seek a simplistic solution (we don't want to overcomplicate this if we can avoid it)
- Efficiency to compile (something that is exponential in the number of lines of code is likely not useful)
- Efficiency of the code itself (that is, how fast does it run?)

Efficiency

- Real compilers have an *intermediate representation* (IR) that's close to assembly code but (at least) has infinite registers
- This IR is good for optimization
- We won't do optimization, so won't use an IR. Our step of generating assembly will be (more-or-less) the "generate IR" step of a larger compiler. MIPS is our IR!

First Example

Let's try the following:

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

Recall our mips.twoints convention that registers 1 and 2 store the input values, and our usual convention to return in register 3. We need to put the value in register 1 into register \$3. What MIPS command does this?

Solution

```
add $3, $1, $0
```

Almost done! What's missing?

```
add $3, $1, $0
```

```
jr $31
```

But, things will get more complicated...

Things we Glossed Over

- How did we know where a was stored?
- What if we had >2 arguments? (in a different procedure)
- What if we had done something complex with intermediary results?

Things we Glossed Over

- How did we know where a was stored?
 - The parse tree will be the same if we'd done "return b " instead of "return a "!


The Main Issue

- The parse tree isn't going to be enough to determine the difference between the two pieces of code.
- How can we resolve this? How can we distinguish between these two codes?

Return of the Revenge of the Symbol Table

- That's what our symbol table is for! We'll augment it to include where each symbol is stored.

Symbol	Type	Location
a	int	\$1
b	int	\$2



Is this kind of location "good enough"?

Issues

- Storing variables and parameters in registers will force us to run out of registers quickly. What if we had 20 local variables?
- This means we need to store them somewhere else. Where?
- Store them on the stack!

Code Generation

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

Becomes

```
lis $4
```

```
.word 4
```

```
sw $1, -4($30)
```

```
sub $30, $30, $4
```

```
sw $2, -4($30)
```

```
sub $30, $30, $4
```

```
lw $3, 4($30)
```

```
add $30, $30, $4
```

```
add $30, $30, $4
```

```
jr $31
```

We make the convention that \$4 always contains the number 4.

Updates to Symbol Table

Instead of the symbol table storing registers, it should actually store the offset from the stack pointer!

Symbol	Type	Offset (from \$30)
a	int	4
b	int	0

“Offset from the stack pointer” will cause us problems, though...

Local Variables

Variables also have to go on the stack but we don't know what the offsets should be until we process all of the variables and parameters! For example,

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

Let's generate the code for this program
(see next slide)

Code Generated

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30)      ; For int c = 0
sub $30, $30, $4
lw $3, 8($30)       ; Offset changed due to
                    ; presence of c!
...
jr $31              ; Restore stack
```

Local Variables

Variables also have to go on the stack, but we don't know what the offsets should be until we process all of the variables and parameters! For example,

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

Symbol	Type	Offset (from \$30)
a	int	8
b	int	4
c	int	0

As we process the code, we need to be able to compute the offset as we see the values. Also, we need to handle intermediate values of complicated arithmetic expressions by storing on the stack! Doing this from \$30 would be difficult.

In Search of a Fix

How then do we arrange it so that when we see the variable, we know what the offset is? Remember that the key issue here is that \$30 (the top of the stack) changes.

Reference the offset from the bottom of the stack frame! We'll store this value in \$29! This is called the "frame pointer".

If we calculate offsets from \$29, then no matter how far we move the top of the stack, the offsets from \$29 will be unchanged!

Code Generated

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30)
sub $30, $30, $4
lw $3, 0($29)
```

```
...
jr $31
```

This is good enough for mainline code, but remember that procedures need to preserve registers, and it would be awkward to do so with \$29 here!

```
; For int c = 0
```

```
; Offset is always 0
; from $29, no matter
; where our stack goes
; Restore stack
```

Local Variables with \$29

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

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

This is easier with \$29.

Something Harder

What about a more complicated program:

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

How do we handle this?

When a and b were in registers, we could just subtract them. Now we need to load them, then subtract them.

Load them into registers? We'll run out of registers again with more complicated behavior...

New Convention!

- We'll continue to use \$3 for the result of any expression
- Also use \$5 for intermediate scratch values

More Code

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
lw $3, 0($29)      ; a
add $5, $3, $0      ; Move a to $5
lw $3, -4($29)      ; b
sub $3, $5, $3       ; a - b
...                 ; Restore stack
jr $31
```

Still Have Problems

Where does this approach break down?

Consider something like $a + (b - c)$. Would need to load a , load b , load c compute $b - c$, then compute the answer. This would require a third register. (Remember, we want to process code left to right.)

Where should we store these values instead? On the stack again! This way we only will ever need two registers for scratch work!

Abstraction

We'll use some shorthand for our code.

code(x) represents the generated code for x.

code(a): (where a is a variable)

lw \$3, N(\$29) where N is the offset in the
symbol table

push(\$x):

sw \$x, -4(\$30) +
sub \$30, \$30, \$4

pop(\$x):

add \$30, \$30, \$4 +
lw \$x, -4(\$30)

Abstraction

```
code(a - b):  
code(a) +  
push($3) +  
code(b) +  
pop($5) +  
sub $3, $5, $3
```

Example

Let's compute the MIPS code for

```
int wain(int a, int b) {  
    int c = 3;  
    return a + (b - c);  
}
```

using these abstractions.

Solution

...	; Prologue doesn't
	; change
lw \$3, 0(\$29)	; a
sw \$3, -4(\$30)	; push(\$3)
sub \$30, \$30, \$4	
lw \$3, -4(\$29)	; b
sw \$3, -4(\$30)	; push(\$3)
sub \$30, \$30, \$4	
lw \$3, -8(\$29)	; c
add \$30, \$30, \$4	; pop(\$5)
lw \$5, -4(\$30)	
sub \$3, \$5, \$3	; b - c
add \$30, \$30, \$4	; pop(\$5)
lw \$5, -4(\$30)	
add \$3, \$5, \$3	; a + (b - c)
...	; Epilogue doesn't
	; change

Notes

- We can generalize this technique so we only need two registers for *any* computation!
- We can generalize this to WLP4 by making rules for each grammar rule, e.g.:

```
code(expr1 → expr2 PLUS term):  
  code(expr2)  
  + push($3)  
  + code(term)  
  + pop($5)  
  + add $3, $5, $3
```

- The recursive code has assured that the operands will be in \$5 and \$3 by the time add is run.

More on Converting Grammars

- Singleton grammar productions are relatively straightforward to translate:
 - `code(S → BOF procedure EOF):`
`code(procedure)`
 - `code(expr → term): code(term)`
- lvalues are a bit odd. Recursion might do the wrong thing:

```
code(statement → lvalue BECOMES expr SEMI):  
  code(lvalue)  
  + push($3) ???  
  + code(expr)  
  + pop($5) ???  
  + ???
```

Ivalues

- The basic idea of our “code” function is that it produces the code to put the *value* of the expression in \$3
- Ivalues shouldn't actually generate values!
- If we have { int x = 0; x = 3; ...}, having the code for “x” in “x = 3” generate 0 would be useless

lvalue thoughts

- We could have the code function do something different for lvalues...
 - `code(lvalue → anything)` produces an address in `$3` instead of its value?
 - This would make the code function unpredictable and harder to debug, though.

lvalue thoughts

- Or, we could have the code function for expressions that *include* lvalues do something different based on the kind of lvalue.
 - $\text{code}(\text{stmt} \rightarrow (\text{lvalue} \rightarrow \text{ID}) \text{ BECOMES } \text{expr SEMI})$ has to be distinct from $\text{code}(\text{stmt} \rightarrow (\text{lvalue} \rightarrow \text{STAR expr}) \text{ BECOMES } \text{expr SEMI})$.
 - This makes the code function less modular and less maintainable...

lvalue thoughts

- Or, we could have a totally separate code generation function that instead of generating values, generates the code to store values

code(stmt \rightarrow lvalue BECOMES expr SEMI):

code(expr)

+ store(lvalue)

- What if the lvalue had a side-effect? Now you're running your code in the wrong order! Need to refine on this to not reorder code.