Warm-Up Problem

What MIPS assembly instructions/directives require extra data in MERL files?
What MIPS assembly instructions/directives do not?

CS 241 Lecture 19

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

Address Of

- Recall that an lvalue is something that can appear as the LHS of an assignment rule. Note that we have a rule factor → AMP lvalue. Why that instead of another factor?
- Suppose we have an ID value *a*. How do we find out where *a* is in memory?
- In our symbol table! We can load the offset first and then use this to find out the location.

Address Of is an lvalue Problem

- Remember our discussion of the many ways of implementing lvalues? The address of operator is all lvalue!
- If you used the “code(lvalue) generates an address” version, you’re already done.

Comparisons

- Comparisons of pointers work the same as with integers with one exception…
- Pointers cannot be negative*, and so slt is not what we want to use! We should use sltu instead.

* Actually, this is just a matter of interpretation. We generally treat all of memory as an array indexed from 0 to the size of memory (and thus unsigned), but the only thing that breaks if we treat it as signed is our intuition. Regardless, C asks us to treat it as unsigned, so we shall.

Comparisons

- Given test → expr COMP expr, how can we tell which of slt or sltu to use? Simply check the type of the exprs (if you implemented type checking properly, checking one is enough, since they're the same)
- Pro Tip: You might want to augment your tree node class to include the type of the node itself (if it has one).

Pointer Arithmetic

- Recall for addition and subtraction we have several contracts. *The code for addition will vary based on the types of its subexpressions!*
- For int + int or int - int, we proceed as before. This leaves 4 contracts we need to consider that use pointers

Addition

$\texttt{expr}_1 \rightarrow \texttt{expr}_2 + \texttt{term}$ where
$\texttt{type}(\texttt{expr}_2) == \texttt{int *}$ and type(term) == int

```
code(expr_1) = code(expr_2)
             + push($3)
             + code(term)
             + mult $3, $4
             + mflo $3
             + pop($5)                 ; $5 <- expr
             + add $3, $5, $3
```

Recall that we're computing a different memory address corresponding to $\texttt{expr}_2 + 4 \cdot \texttt{term}$.

Addition

$expr_1 \rightarrow expr_2$ + term where type($expr_2$) == int and type(term) == int*

```
code(expr1) =  code(expr2)
             + mult $3, $4
             + mflo $3
             + push($3)
             + code(term)
             + pop($5)            ;$5 <- expr
             + add $3, $5, $3
```

Recall that we're computing a different memory address corresponding to $4 \cdot expr2 + term$.

Subtraction

$expr_1 \rightarrow expr_2$ - term where type($expr_2$) == int * and type(term) == int

```
code(expr1) = code(expr2)
            + push($3)
            + code(term)
            + mult $3, $4
            + mflo $3
            + pop($5)           ; $5 <- expr
            + sub $3, $5, $3
```

Recall that we're computing a different memory address corresponding to $expr_2 - 4 \cdot term$.

Subtraction

expr1 → expr2 - term where type(expr2) == int* and type(term) == int*

```
code(expr1) = code(expr2)
              + push($3)
              + code(term)
              + pop($5)                  ; $5 <- expr
              + sub $3, $5, $3
              + div $3, $4
              + mflo $3
```

Recall that we're computing a distance between two memory addresses corresponding to ($expr_2$ − term)/4.

Memory Allocation

- Lastly, we need to handle calls such as new and delete.
- Good news! We're going to outsource this work to the runtime environment. Include alloc.merl:

```
./wlp4gen < source.wlp4i > source.asm
cs241.linkasm < source.asm > source.merl
cs241.merl source.merl print.merl alloc.merl > exec.mips
```

- NOTE: alloc.merl must be linked last! (Needs to know where end of code is). Let's recall our memory layout on the board...

Prologue Additions

- Now we include
```
.import init
.import new
.import delete
```
- The command init initializes the heap. Must be called at the beginning. Takes a parameter in $2 and initializes the data structure.
- If called with mips.array then $2 is the length of the array.
- Otherwise, we want $2 = 0.

new

- Finds the number of new words needed as specified in $1.
- Returns a pointer to memory of beginning of this many words in $3 if successful.
- Otherwise, it places 0 in $3.

```
code(new int [expr]) = code(expr)
                + add $1, $3, $0
                + call(new)
                + bne $3, $0, 1
                + add $3, $11, $0
```

Note, the last line sets $3 to be NULL and executes if and only if the call to new failed.

delete

- Requires that $1 is a memory address to be deallocated.

```
code(delete [] expr) = code(expr)
                + beq $3, $11, skipDelete
                + add $1, $3, $0
                + call(delete)
                + skipDelete:
```

Again, as with `if` and `while` statements, you will need to number your deletion labels. We skip delete when attempting to delete a null pointer.

Example

Let’s do a complete example with pointer manipulation:

```
int wain(int *a, int b) {
    int *c = NULL;
    int ret = 0;
    c = new int[b];
    if (c < a) {
        ret = a - c;
    } else {
        ret = c - a;
    }
    return ret;
}
```

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?

- Import print, 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 other procedures?

- 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, e.g., 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.
- \$30 is preserved through symmetry.
- Who should save \$29?