## Warm Up Problem

Using our shorthands from class, convert the line corresponding to the mutation of c into MIPS:

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

# CS 241 Lecture 18

Code Generation Continued

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

## More on Converting Grammars

What about for `println`? Recall the rule:

statement PRINTLN LPAREN expr RPAREN SEMI

## More on Converting Grammars

What about for `println`? Recall the rule:

statement PRINTLN LPAREN expr RPAREN SEMI

You actually have already done this in A2P6 and A2P7a!

## Importing Code

A compiler needs to take lots of code from many different parts to make it work.

**Definition**

A *runtime environment* (or RTE for short) is the execution environment provided to an application or software by the operating system to assist programs in their execution. Such things include procedures, libraries, environment variables and so on.

For example, `msvcrt.dll` is a module containing standard C library functions such as `printf`, `memcpy` and so on (for Windows). For Linux machines, this is stored in `libc.so`.

Thus, it makes sense to provide `print` as part of the runtime.

## MERL files

- We will provide you with some `MERL` files to help you.
- This stands for MIPS Executable Relocatable Linkable.
- Briefly, when we compile files, almost always what is output is not pure machine code (ie. it isn't just binary associated with your code). There is often some additional header information as well. The output is usually *object code*.
- These files can help us store additional information needed by the loader and linker (see the last week of content for more on this).

## What This Means For Us Now

For us, we will provide you with a `print.merl` file that you will use to link with your assembled output:

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

Then finally, we can call

```
mips.twoints exec.mips
```

or

```
mips.array exec.mips
```

## Using `print`

- To use print, we need to add `.import print` to the beginning of our file.
- After this, we can use `print` in our MIPS code. It will print the contents of \$1. Be mindful that you may need to save \$1 depending on what you want to print.
- Note also that `print` will overwrite the data in \$31. We will need to save and restore it.

## Code for `println`

```
code(println(expr);) = push($1)
                     + code(expr)
                     + add $1, $3, $0
                     + push($31)
                     + lis $5 + .word print
                     + jalr $5 + pop($31)
                     + pop($1)
```

Note: You might be okay with overwriting $1.

## What’s Left

- Most of our statements have been completed except for `if` and `while` statements (and `delete` but that comes much later...)
- For this, we need to handle boolean tests.
- Convention: Store 1 inside \$11. (Now we have true and false stored somewhere)
- Also, we will store `print` in register \$10.

## Code Structure this far

```
;Prologue
.import print
lis $4
.word 4
lis $10
.word print
lis $11
.word 1
sub $29, $30, $4
;end Prologue and begin Body
;space for variables
;translated WLP4 code
;end Body and begin Epilogue
add $30, $29, 4
jr $31
```

## Boolean Tests

What would the code for the rule `test` $\rightarrow$ `exprA < exprB`?

## Boolean Tests

What would the code for the rule `test` $\rightarrow$ `exprA` $<$ `exprB`?

```
code(test) = code(exprA)
           + push($3)
           + code(exprB)
           + pop($5)
           + slt $3, $5, $3
```

What should we do for `test` $\rightarrow$ `exprA` $>$ `exprB`?

## Boolean Tests

What would the code for the rule `test` $\rightarrow$ `exprA  <  exprB`?

```
code(test) = code(exprA)
           + push($3)
           + code(exprB)
           + pop($5)
           + slt $3, $5, $3
```

What should we do for `test` $\rightarrow$ `exprA  >  exprB`?

Change `slt $3, $5, $3` to `slt $3, $3, $5` above!

## More Boolean Tests

Try to translate the rule `test` $\rightarrow$ `exprA` $! =$ `exprB`.

## More Boolean Tests

Try to translate the rule test $\rightarrow$ exprA $! =$ exprB.

```
code(test) = code(exprA)
           + push($3)
           + code(exprB)
           + pop($5)
           ;maybe store $6 and $7 if used
           + slt $6, $3, $5
           + slt $7, $5, $3
           ;Note 0 <= $6 + $7 <= 1
           + add $3, $6, $7
```

What should we do for test $\rightarrow$ exprA $==$ exprB?

## More Boolean Tests

Try to translate the rule `test` $\rightarrow$ `exprA` $!=$ `exprB`.

```
code(test) = code(exprA)
           + push($3)
           + code(exprB)
           + pop($5)
           ;maybe store $6 and $7 if used
           + slt $6, $3, $5
           + slt $7, $5, $3
           ;Note 0 <= $6 + $7 <= 1
           + add $3, $6, $7
```

What should we do for `test` $\rightarrow$ `exprA` $==$ `exprB`?

Key idea is $a == b$ is the same as $!(a! = b)$. Add the line `sub $3, $11, $3` above to flip 0 to 1 and vice versa!

## Last Two Tests

How do we do `test` $\rightarrow$ `exprA <= exprB` or
`test` $\rightarrow$ `exprA >= exprB`?

## Last Two Tests

How do we do `test` $\rightarrow$ `exprA <= exprB` or `test` $\rightarrow$ `exprA >= exprB`?

Use the fact that $a <= b$ is the same as $!(a > b)$ and similarly for $>=$.

This leaves us with our final `if` and `while` statements (see next slides)

## `if` Statements

Rule: `statement` $\rightarrow$ `IF (test)` $\{$`stmts1`$\}$ `ELSE` $\{$`stmts2`$\}$ .

Translation (Hint: Use Labels!):

## if Statements

Rule: `statement` $\rightarrow$ `IF (test)` $\{$`stmts1`$\}$ `ELSE` $\{$`stmts2`$\}$ .

Translation (Hint: Use Labels!):

```
code(statement) = code(test)
                + beq $3, $0, else
                + code(stmts1)
                + beq $0, $0, endif
                + else: code(stmts2)
                + endif:
```

Caution: Be wary of multiple labels! How do we fix this?

## Simple `if` Counter Idea

- Keep track of how many if statements you have.
- Have a counter of these; say `ifcounter`.
- Each time you have an if statement, increment the counter.
- Use label names like `else#` and `endif#` where# corresponds to the `ifcounter`.

## `while` Statements

Rule: `statement` $\rightarrow$ `WHILE (test)` $\{$`statements`$\}$.

Translation (Hint: Use Labels!):

## while Statements

Rule: statement $\rightarrow$ WHILE (test) {statements}.

Translation (Hint: Use Labels!):

```
code(statement) = loop: code(test)
                + beq $3, $0, endWhile
                + code(stmts)
                + beq $0, $0, loop
                + endWhile:
```

Again, be sure to have a while loop counter variable like with if statements!

## An Extremely Important Final Tip

- Since you are generating MIPS code; note that you can also generate comments with your MIPS code!
- We recommend that you also output some comments which will make it easier to decipher what you were doing when you see your final MIPS code.

## Recap

Before we continue, a moment to recap our conventions:

- \$0 is always 0.
- \$1 and \$2 are for arguments 1 and 2 in `wain`.
- \$3 is always for output (and possibly intermediate computations).
- \$4 is always 4.
- \$5 is also for intermediate computations.
- \$10 will store `print`
- \$11 will be reserved later for 1.
- \$29 is our frame pointer (fp).
- \$30 is our stack pointer (sp).
- \$31 is our return address (ra).

## Prologue

At the beginning of our code, we must:

- Load 4 into $4 and 1 into $11.
- Import print. Store in $10
- Store the return address on the stack. (Optional - useful only if overwriting).
- Initialize the frame pointer hence creating a stack frame
- Store registers 1 and 2

## Body

- Need to store local variables in the stack frame
- Contain MIPS code corresponding to the WLP4 program

## Epilogue

- Need to pop the stack frame
- Also, need to restore the previous variables.

## Pointers

At last we have reached pointers. We need to support all of the following:

- NULL
- Allocating and deallocating heap memory
- Dereferencing
- Address of
- Pointer arithmetic
- Pointer comparisons
- Pointer assignments and pointer access

Here we go!

## NULL

The first and obvious choice for us for the value of `NULL` is `0x0`. Why is this a problem for us?

## NULL

The first and obvious choice for us for the value of `NULL` is `0x0`. Why is this a problem for us?

In our language, `0x0` is a valid memory address! (Note, in say C programming, the operating system prevents access to `0x0`).

So we would like our `NULL` to crash if attempting to dereference. We can force this by picking a value for `NULL` that is not **word-aligned**.

## NULL

The first and obvious choice for us for the value of `NULL` is `0x0`. Why is this a problem for us?

In our language, `0x0` is a valid memory address! (Note, in say C programming, the operating system prevents access to `0x0`).

So we would like our `NULL` to crash if attempting to dereference. We can force this by picking a value for `NULL` that is not **word-aligned**.

Word-aligned for us means that the address is a multiple of 4. The smallest value for `NULL` therefore that we can (and will) use is `0x1`.

## Code

`factor` $\rightarrow$ `NULL`

```
code(factor) = add $3, $0, $11
```

Note that attempts to use NULL with either `lw` or `sw` will result in a crash since MIPS is expecting a word-aligned address.