Code Generation: Procedures

Procedures

- There are two things to cover here:
  - Calling procedures that are imported from an external library (and combined with your generated code via *linking*).
  - Generating code for procedures in the WLP4 program and generating code for calls to those procedures.
- The first one is easy and just requires recalling some concepts from earlier in the course.
- The second one is much harder!

Importing a Procedure

- Use the **.import** directive discussed in the linking part of the course!
- For example, we provide a **print** procedure (equivalent to Question 6) that can be used to implement the WLP4 println statement.
- To use it, output the line **.import print** at the very beginning of your generated code.
  - It technically just needs to appear somewhere before the first use of the "print" label but putting it at the very beginning is cleaner.
- As a consequence, your generated code now must be assembled with an assembler that produces **MERL files** (cs241.linkasm) and combined with the print library using a **linker** (cs241.linker).

Calling an Imported Procedure

- This is exactly like calling any other procedure in MIPS assembly.
- If the procedure expects parameters in certain registers, first place the values in the appropriate registers.
  - For example, the **print** procedure expects the number to print to be in $1.
- **Save** $31 (the current procedure's return address) on the stack before calling a procedure, and **restore** $31 after.
- To do the actual call, use **jalr**. For example:

```
lis $5
.word print
jalr $5
```

Code Generation for Procedures

- There are many ways to implement code generation for procedures. We will just cover one approach.
- The general idea is to define a **calling convention** that outlines how procedures will work at a low level. For example:
  - How parameters are passed to a procedure
  - How a procedure retrieves the parameters once it is called
  - How values are returned from a procedure
  - Which registers are preserved when a procedure is called
  - How setup and cleanup of the stack is handled
- Maybe other things in more complex languages.

Calling Convention: Hand-Written Procedures

- When writing MIPS assembly procedures by hand earlier in the course, we implicitly used the following calling convention:
  - Parameters are passed in registers (usually $1 and $2, but it must be documented by the author of the procedure) and retrieved from registers.
  - Values are returned in registers (usually $3 but it must be documented).
  - **All** registers are preserved by a call except registers used for return values, and $31 (which is modified by **jalr** and thus must be preserved by the *caller*).
  - Procedures can modify the stack at locations above $30, but they must pop all values that they push.
- The external procedures we provide (printing and memory allocation) follow this calling convention.

Calling Convention: Generated Procedures

- The calling convention we suggest for code generation is as follows:
  - Parameters are passed using **the stack**. Before a call, the caller computes the values of the arguments and pushes them to the stack in left-to-right order.
  - Each procedure *call* has **its own local frame pointer**. Parameters are retrieved using positive offsets from the local frame pointer, and non-parameter local variables are accessed using non-positive (zero or negative) offsets.
  - Since the **caller** pushes arguments, it is responsible for popping them. The **callee** (procedure being called) must pop everything it pushes.
  - The **caller** is responsible for preserving its own **return address** ($31) and **frame pointer** ($29). That is, it must save $31 and $29 to the stack, *before pushing arguments*, and restore them upon return, *after popping arguments*.

Preserving Other Registers?

- Aside from $29 and $31, we didn't specify whether any other registers are preserved.
- This part of the calling convention is up to you, and whether preserving other registers is necessary will depend on the design of your code generator.
- Let's look at an example where preserving other registers might be necessary.

Recall: Assignment Statements

- We presented the following code generation strategy for assignment statements (statement → lvalue BECOMES expr SEMI) and claimed it doesn't work:

```
addressOf(lvalue) ; $3 = address of lvalue
add $5, $3, $0    ; copy it into $5
code(expr)        ; $3 = value of expr
sw $3, 0($5)      ; store expr value into lvalue address
```

- The problem is that our code generation strategy for **expr** uses $5 to store temporary values, which may overwrite our **lvalue** address.

Recall: Assignment Statements

- Our proposed fix was to use the stack:

```
addressOf(lvalue) ; $3 = address of lvalue
push($3)          ; save lvalue address on the stack
code(expr)        ; $3 = value of expr
pop($5)           ; $5 = address of lvalue
sw $3, 0($5)      ; store expr value into lvalue address
```

- This is a robust solution to the problem, but it is not the only solution.

Recall: Assignment Statements

- Consider this solution, where we simply use a *different register* to hold the **lvalue** address, one that isn't used by our **expr** strategy.

```
addressOf(lvalue) ; $3 = address of lvalue
add $7, $3, $0    ; copy it into $7
code(expr)        ; $3 = value of expr
sw $3, 0($7)      ; store expr value into lvalue address
```

- This will work for exprs that *don't contain procedure calls.*
- If the expr contains a procedure call, that procedure could contain an assignment statement, which could modify $7!

Recall: Assignment Statements

- Consider this solution, where we simply use a *different register* to hold the **lvalue** address that isn't used by our **expr** strategy.

```
addressOf(lvalue) ; $3 = address of lvalue
add $7, $3, $0    ; copy it into $7
code(expr)        ; $3 = value of expr
sw $3, 0($7)      ; store expr value into lvalue address
```

- We can make this work for all exprs, including ones that contain procedure calls, *if* we modify our calling convention.
- If our calling convention states that **all procedures must preserve $7**, this strategy will work!

Recall: Assignment Statements

- What are the advantages of the stack strategy?

```
addressOf(lvalue) ; $3 = address of lvalue
push($3)          ; save lvalue address on the stack
code(expr)        ; $3 = value of expr
pop($5)           ; $5 = address of lvalue
sw $3, 0($5)      ; store expr value into lvalue address
```

- This works even if procedure calls are not guaranteed to preserve any registers, so we don't need to update our calling convention.
- However, storing temporary values in the stack is slower than storing them in registers.

Recall: Generating Code for “wain”

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

1. **Import external procedures** and initialize constants (e.g. $4 = 4).
2. Set up the **(local)** frame pointer and put variables on the stack.
3. Generate code for all statements in the “statements” subtree.
4. Generate code for the return “expr”.
5. Generate code that cleans up the stack and returns (jr $31).

Generating Code for Procedures

procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

1. Set up the (local) frame pointer and put variables on the stack.
2. Generate code for all statements in the “statements” subtree.
3. Generate code for the return “expr”.
4. Generate code that cleans up the stack and returns (jr $31).

- It's pretty similar to wain.
- But we need to be very careful with the frame pointer and the stack setup and cleanup.

Frame Pointers and Procedures

- Each procedure **call** (not procedure) has its own set of local variables, which we allocate on the stack.
- For example, consider this recursive WLP4 procedure:

```
int f(int n) {
  int r = 1;
  if(n > 1) { r = n * f(n-1); } else {}
  return r;
}
```

- If you call f(3), then two further recursive calls are made and the stack looks like this immediately before f(1) returns:

<table>
<tr><td rowspan="2">…</td><td colspan="2">f(3) stack frame</td><td colspan="2">f(2) stack frame</td><td colspan="2">f(1) stack frame</td><td rowspan="2">Top of Stack</td></tr>
<tr><td>n = 3</td><td>r = 1</td><td>n = 2</td><td>r = 1</td><td>n = 1</td><td>r = 1</td></tr>
</table>

Frame Pointers and Procedures

- Each procedure **call** (not procedure) has its own set of local variables, which we allocate on the stack.
- For example, consider this recursive WLP4 procedure:

```
int f(int n) {
  int r = 1;
  if(n > 1) { r = n * f(n-1); } else {}
  return r;
}
```

- This is what the stack looks like after f(1) returns, and immediately before f(2) returns:

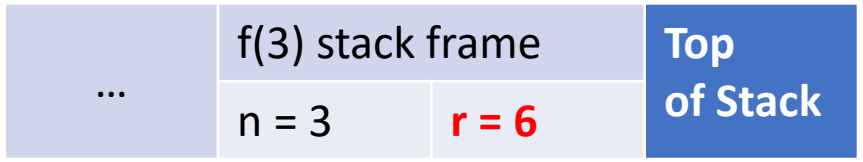
<table>
<tr><td rowspan="2">…</td><td colspan="2">f(3) stack frame</td><td colspan="2">f(2) stack frame</td><td rowspan="2">Top of Stack</td></tr>
<tr><td>n = 3</td><td>r = 1</td><td>n = 2</td><td>r = 2</td></tr>
</table>

Frame Pointers and Procedures

- Each procedure **call** (not procedure) has its own set of local variables, which we allocate on the stack.
- For example, consider this recursive WLP4 procedure:

```
int f(int n) {
  int r = 1;
  if(n > 1) { r = n * f(n-1); } else {}
  return r;
}
```

- And finally, this is what it looks like after f(2) returns, and immediately before f(3) returns:



Frame Pointers and Procedures

- Each procedure **call** (not procedure) has its own set of local variables, which we allocate on the stack.

<table>
<tr><td rowspan="2">…</td><td colspan="2">f(3) stack frame</td><td colspan="2">f(2) stack frame</td><td colspan="2">f(1) stack frame</td><td rowspan="2">Top of Stack</td></tr>
<tr><td>n = 3</td><td>r = 1</td><td>n = 2</td><td>r = 1</td><td>n = 1</td><td>r = 1</td></tr>
</table>

- If we only had one frame pointer, then each procedure call would need to use different offsets to access its local variables!
- But each procedure call is running the same block of code, just with different data in memory, so this would get messy.
- So, each procedure **call** sets up its own frame pointer.

The Frame Pointer and the Stack

- We will continue to follow the convention that parameters are at positive offsets from the frame pointer and non-parameters are non-positive (zero or negative) offsets.
- Recall our suggested calling convention:
  - The caller saves $29 (its own frame pointer) and $31 (its return address).
  - The caller pushes the arguments to the stack.
  - The caller calls the procedure with **jalr**.
  - Once the procedure returns, the caller pops the arguments.
  - The caller restores $29 and $31.

Stack Layout Before Jalr

`f(241, 0, 17)`

Immediately before jumping to f with jalr, the stack looks like this.

	Stack (before jalr to f)
$30+0	17
$30+4	0
$30+8	241
	Caller's $31
	Caller's $29

Retrieving Arguments from the Stack

- It is important to understand the state of the stack when the called procedure starts running.
  - Because we pushed $29 and $31 first, *then* pushed the arguments, the stack pointer $30 is pointing to the *last argument* we pushed.
  - If there are *n* arguments total, then $30 points to the n^{th} argument.
- If we start the procedure by setting the FP $29 in the usual way…

  ```
  sub $29, $30, $4
  ```

  - The first argument is at offset *4n*, the second is at offset 4n – 4, the third is at offset *4n – 8*, and so on… with the last argument at offset 4.
  - If argument 0 is the first one, the offset for argument *i* is *4(n – i)*.

Stack Layout After Jalr & Setting FP

`f(241, 0, 17)`

	$29+0	**Stack (after setting FP)**
$30+0	$29+4	17
$30+4	$29+8	0
$30+8	$29+12	241
		Caller's $31
		Caller's $29

Accessing Local Variables

procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The “params” child is either a leaf node (no parameters) or has a “paramlist” child, which is a linked list of “dcl” nodes.
- If there are parameters, add each parameter to the offset table.
  - **Do not** generate any code here to push the parameters to the stack. The parameters are pushed by the caller.
- For the non-parameters, look at the “dcls” child of the procedure and add each non-parameter declaration to the offset table, like for wain.
  - **Do** push each non-parameter to the stack here, as you did in wain.

Accessing Local Variables

procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- For wain, we gave the formula (counter – **2**) * (-4) for offsets, where counter is the number of variables seen so far, because wain always has **2** parameters.
- For a procedure with *n* parameters, it would be (counter – *n*) * (-4).
- Alternatively, you can set a variable to *4n* (offset of first parameter) and just subtract 4 for each additional variable you see.

Where to Preserve Registers

- If your calling convention requires you to preserve additional registers, it makes sense in WLP4 to do it *after* pushing local variables.
- Otherwise your saved registers will affect your offset calculations!
- This strategy works in WLP4 because local variables can only be initialized to constants.
- In languages where local variables can be initialized to complex expressions, the initialization process might modify the registers you seek to preserve, so things would be trickier.

Stack Cleanup

- At the end of the procedure, before returning, we must pop everything the procedure pushed as part of our calling convention.
- If you preserved additional registers, pop these.
- Otherwise, you just need to pop the **non-parameter local variables**.
- Don't pop the **parameters** in the procedure itself!
- In our calling convention, the caller pushes the parameters, and it is the caller's responsibility to pop them.
- Unlike wain (where stack cleanup was optional), here it is **essential** for the generated procedures to work correctly.

One Last Note: Procedure Names

- At the start of each procedure, we need to output a **label definition** that we can use to call the procedure.
- It may seem like the natural choice for the label name is to use the name the user gave the procedure, which is stored in the ID token.

procedure → INT **ID** LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Is there any problem with using this name?

One Last Note: Procedure Names

- These are all valid WLP4 procedure names:

```
int print() { return 0; } // conflicts with imported "print"
int else1() { return 0; }
int loop1() { return 0; }
// what if you used "else1" or "loop1" as label names
// in an if statement or while loop???
```

- User-defined procedure names might happen to match a label used internally by your code generator, causing a duplicate label error!
- So *directly* using a user-defined name is a bad idea.

One Last Note: Procedure Names

- There's a simple fix: take the user-defined name, and add a unique prefix like "F" for function or "P" for procedure.

```
int print() { return 0; }
// generated label would be "Pprint:"
int else1() { return 0; }
// generated label would be "Pelse1:"
```

- As long as none of the internal labels you generate for things like if statements and while loops start with "P", there will not be conflicts.
- Since none of the imported procedures start with "P", there is also no possibility of conflicts there.

Generating Code for Procedures

procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

1. Set up the (local) frame pointer and put variables on the stack.
2. Generate code for all statements in the “statements” subtree.
3. Generate code for the return “expr”.
4. Generate code that cleans up the stack and returns (jr $31).

Generating Code for Other Procedures

procedure → INT ID LPAREN params RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- More concretely (assuming no additional saved registers):

```
Output initial label "P" + ID.lexeme + ":"
sub $29, $30, $4          ; set the frame pointer
Generate code for dcls ; push non-parameter local variables
Generate code for statements
Generate code for expr ; return value goes in $3
Generate code that pops non-parameter local variables
jr $31
```

End of Code Generation…?

- The only thing we haven't discussed is how to implement **new** and **delete**. On the project, we provide a library you can use for this.
- Next, we will discuss the implementation details of this library.
- Other than that, we have a code generator! But we made a lot of compromises to make it easier to implement.
- We made heavy use of stack memory, but registers are more efficient, so real-world compilers try to use registers as much as possible.
- Beyond making effective use of registers, there are other **compiler optimizations** we can do to generate more efficient code. We'll discuss some of those after we cover memory management.