5 Procedures

A *procedure* is an abstraction that encapsulates a reusable unit of code. The code that calls a procedure (the *caller*) transfers control to the code of the procedure (also called the *callee*) by modifying the program counter. The caller may also pass *arguments* for the *parameters* of the procedure. When the code of the procedure finishes executing, control transfers back to the instruction after the call of the procedure. The procedure may return a value back to the caller.

To implement this functionality, the implementations of the calling code and of the procedure must agree on *conventions*:

- Where in memory or in which registers will the arguments and the return value be stored?
- Does the calling code or the procedure allocate and free memory needed for the call?
- Which registers may change their value during a procedure call? These are called *caller-save* registers, because if the caller needs their value, it needs to save it elsewhere before the call. Which registers must keep their original value after a procedure call? These are called *callee-save* registers, because if the procedure modifies them, it must change them back to their original values before returning to the caller.

Different programming languages compiled to different computers with different operating systems can have different conventions; what is important is that some convention is chosen and that the same convention is followed both by the code of the procedure and the code that calls it.

In this module, we will work through the assembly language code needed to implement procedures and procedure calls. Along the way, we will note conventions that are convenient at the time; these will become the designated conventions for CS 241E, and will be summarized at the end of the module.

5.1 Control transfer

The first issue to implement is the transfer of control to the procedure. We can use a label (in this example `proc`) to mark the address of the beginning of the code of the procedure. To call the procedure, we need to load the value of this label into the program counter; we could do this by first loading it into some register using the `LIS` instruction, then using the `JR` instruction to copy it from the register into the program counter. As a convention, we will use register 8, `Reg.targetPC`, for this purpose.

When the procedure finishes, we will need to return control back to the code that called the procedure. To remember the address to return to, we call the procedure with the `JALR` instruction instead of `JR`. Recall that `JALR` saves the current value of the program counter into register 31, `Reg.link`, and at the same time, it also copies the value of some chosen register to the program counter, just like `JR` does. To summarize, the assembly language pseudo-code to call the procedure will look as follows (ignore the lines in grey for now):

```
evaluate arguments into temporary variables
Stack.allocate(parameters)
parameter 1 := temporary 1
...
parameter n := temporary n
LIS(Reg.targetPC (8))
Use(proc)
JALR(Reg.targetPC (8))
```

In the procedure itself, we need to define the label `proc`, follow it with the code implementing the body of the procedure, and finally return control back to the calling code. Since the `JALR` instruction saved the return address in register 31, `Reg.link`, we return control using the instruction `JR(Reg.link)`. To summarize, the assembly language pseudo-code of the procedure looks as follows:

```
Define(proc)
Reg.savedParamPtr (5) := Reg.result (3)
Stack.allocate(frame)
dynamicLink := Reg.framePointer (29)
Reg.framePointer (29) := Reg.result (3)
savedPC := Reg.link (31)
paramPtr := Reg.savedParamPtr (5)

...  (body of procedure) ...

Reg.link (31) := savedPC
Reg.framePointer (29) := dynamicLink
Stack.pop // frame
Stack.pop // parameters
JR(Reg.link (31))
```

The code before the body of the procedure, which prepares the procedure for execution, is called the procedure *prologue*. For now, it contains only the definition of the label for the procedure, but as you can infer from the lines in grey, we will be adding to it throughout this module. The code after the body of the procedure, which cleans up the memory that was being used by the procedure and prepares to return to the calling code, is called the procedure *epilogue*. For now, it contains only the `JR` instruction, but we will be adding to it as well.

5.2 Allocating a procedure frame

As we discussed earlier in the module on variables, we wish to store the values of the local variables of the procedure in a stack frame, so we need to allocate one. It makes sense to allocate this stack frame inside the procedure, rather than at the call site, since we know the list of variables needed when we are compiling the procedure. Thus, we push a chunk large enough for the frame onto the stack at the beginning of the procedure, and pop it off just

before the procedure returns.

Also recall from the module on variables that we want to access variables at fixed offsets from a frame pointer, which stays constant, pointing to the frame throughout the execution of the procedure, even if we use the stack for other purposes. Therefore, after pushing space for the frame onto the stack, we copy its address, which `Stack.allocate` returns in register 3, `Reg.result`, into register 29, `Reg.framePointer`. The code of the procedure now looks like this (lines in black are code that we discussed before and lines in red are newly added):

```
Define(proc)
Reg.savedParamPtr (5) := Reg.result (3)
Stack.allocate(frame)
dynamicLink := Reg.framePointer (29)
Reg.framePointer (29) := Reg.result (3)
savedPC := Reg.link (31)
paramPtr := Reg.savedParamPtr (5)

...  (body of procedure) ...

Reg.link (31) := savedPC
Reg.framePointer (29) := dynamicLink
Stack.pop // frame
Stack.pop // parameters
JR(Reg.link (31))
```

Exercise 17. *Before continuing to the next page, stop to think what can go wrong with the frame pointer with the code so far.*

The code of the procedure modifies the frame pointer. If a procedure A calls another procedure B, then B will overwrite the frame pointer to point to its own frame. Then, when B returns control to A, we will not be able to access any of the local variables of A, because their values are in the frame of A but the frame pointer now points to the frame of B.

One possible solution would be to declare the frame pointer register caller-save: if A cares about its value, it should save it somewhere safe before calling B. But where would A store the value of its frame pointer? The safest place, in that it will not get overwritten by B, would be in a local variable in A's frame. But without the value of the frame pointer, we will not be able to access any of A's variables in its frame, including the saved frame pointer. This is a chicken-and-egg problem: to get the saved value of the frame pointer, we need the saved value of the frame pointer.

A more sensible convention is to designate the frame pointer as callee-saved. The procedure B needs to modify it to make it point to its own frame, but it is required to restore it to its original value before it returns control back to A. To do so, it needs to save the old value of the frame pointer before it changes it. Where should it save it? In a variable! We will include a variable in the frame of B (and, more generally, in the frame of every procedure), called the *dynamic link*, which will hold the address of the frame of its caller. The reason for the name of this variable will be discussed in the next module, where we will also introduce a static link. The dynamic link variable saves the old value of the frame pointer, so that it can be restored before the procedure returns to its caller. The code of our procedure now looks as follows:

```
Define(proc)
Reg.savedParamPtr (5) := Reg.result (3)
Stack.allocate(frame)
dynamicLink := Reg.framePointer (29)
Reg.framePointer (29) := Reg.result (3)
savedPC := Reg.link (31)
paramPtr := Reg.savedParamPtr (5)

...  (body of procedure) ...

Reg.link (31) := savedPC
Reg.framePointer (29) := dynamicLink
Stack.pop // frame
Stack.pop // parameters
JR(Reg.link (31))
```

Recall that we generally read and write variables at a fixed offset from the frame pointer. Can we also do this at the variable write `dynamicLink := Reg.framePointer (29)` at the beginning of the procedure? No! At that point during execution, the frame pointer still points to the frame of A, the caller. The `dynamicLink` variable is in the frame of B, the callee (the procedure). At this point, which register holds the address of that frame? It is `Reg.result`, since we just allocated the frame using `Stack.allocate`. For this variable access, and for any other accesses of variables of B before the line `Reg.framePointer (29) := Reg.result (3)`, we need to access variables at offsets from `Reg.result` rather than

from `Reg.framePointer`.

5.3 Saving the link register

What else can go wrong with the current code? Consider what happens if the procedure (B) itself contains a call to some other procedure, say C. The callee B has now also become a caller (in the context of another call, the call from B to C). The body of B will contain a `JALR` instruction somewhere, which will overwrite register 31, `Reg.link`, with some address inside B, to which control should return after executing the body of C. Once `Reg.link` is overwritten, what will happen when B finishes executing? The `JR(Reg.link)` instruction at the end of B is supposed to return control back to A, but instead, it will jump to the call site inside B that called C, because that call site modified the value of `Reg.link`.

The general problem is that `Reg.link` is caller-save: it is modified by a procedure call because it is modified by the `JALR` instruction itself. It is not possible to make `Reg.link` callee-save because that would require changing the hardware implementation of the `JALR` instruction to save its value somewhere. Since a caller does need the old value of `Reg.link`, it should save it somewhere safe, such as to a variable, before every call, and restore it after the call. However, the procedure B might contain many call sites, and we don't need the original value of `Reg.link` immediately after each one of them; we only need it at the very end of procedure B. Therefore, as an optimization, we can save the old value of `Reg.link` once, near the beginning of B, and restore it once, near the end of B, just before it returns. The code of B now looks as follows. The `savedPC` variable is added to the frame of B to hold the old value of `Reg.link`.

```
Define(proc)
Reg.savedParamPtr (5) := Reg.result (3)
Stack.allocate(frame)
dynamicLink := Reg.framePointer (29)
Reg.framePointer (29) := Reg.result (3)
savedPC := Reg.link (31)
paramPtr := Reg.savedParamPtr (5)

...  (body of procedure) ...

Reg.link (31) := savedPC
Reg.framePointer (29) := dynamicLink
Stack.pop // frame
Stack.pop // parameters
JR(Reg.link (31))
```

Now the body of the procedure can itself include calls to other procedures.

Remark. Since we save and restore `Reg.link` at the beginning and end of the procedure, just like `Reg.framePointer`, and since `Reg.framePointer` is callee-save, one might mistakenly think that `Reg.link` is also callee-save. But this is not the case: `Reg.link` is caller-save because executing the code to call a procedure modifies its value; executing the code to call a procedure preserves the value of the callee-save frame pointer.

5.4 Passing arguments

The next feature to implement is passing arguments for the parameters of the procedure. Since a procedure could have an arbitrary number of parameters, they might not fit in registers, so we will pass them in memory. Both the calling code and the procedure must agree on a convention that specifies where in memory the arguments will be stored. Since the extent of an argument is generally the execution of the called procedure, it makes sense to pass arguments on the stack.

The values of arguments are computed in the calling code before the procedure is called. Therefore, as a convention, we will require the calling code to allocate space (a chunk) for the arguments on the stack. The calling code must somehow communicate the address of this chunk to the procedure. Since allocating memory leaves the address of the allocated chunk in register 3, `Reg.result`, it is convenient to use this register as the convention: before it transfers control to the procedure, the calling code must store the address of the chunk containing the arguments in `Reg.result`. Thus, the code to call the procedure now looks like this:

```
evaluate arguments into temporary variables
Stack.allocate(parameters)
parameter 1 := temporary 1
...
parameter n := temporary n
LIS(Reg.targetPC (8))
Use(proc)
JALR(Reg.targetPC (8))
```

Remark. As an alternative convention, it would be possible to declare that the arguments are at the top of the stack: that is, their address is in `Reg.stackPointer` rather than in `Reg.result`. However, such a convention would force the arguments to always be allocated on the stack. Later in the course, we will sometimes need to allocate the arguments in other places in memory, so we adopt `Reg.result` as our convention now to make this possible later.

After control is transferred to the procedure, recall that the first thing that the procedure does is allocate space on the stack for its frame. Doing so overwrites the value of `Reg.result`, the address of the arguments. We need to save the address of the arguments somewhere before allocating the frame. Once again, a local variable is a safe space to save this address: every procedure will have a `paramPtr` variable that holds the address of the parameters of the procedure.

You may have noticed another chicken-and-egg problem. We need to store the address of the arguments into the `paramPtr` variable so it will not be overwritten by allocation of the frame, but local variables are stored in the frame, so they do not exist before we allocate the frame. A fix is to designate another register, register 5, `Reg.savedParamPtr`, to temporarily hold the address of the arguments while we set up the frame. Once the frame is allocated, we can copy the address of the arguments from `Reg.savedParamPtr` to the `paramPtr` variable. The code of the procedure now looks like this:

```
Define(proc)
Reg.savedParamPtr (5) := Reg.result (3)
```

```
Stack.allocate(frame)
dynamicLink := Reg.framePointer (29)
Reg.framePointer (29) := Reg.result (3)
savedPC := Reg.link (31)
paramPtr := Reg.savedParamPtr (5)

...  (body of procedure) ...

Reg.link (31) := savedPC
Reg.framePointer (29) := dynamicLink
Stack.pop // frame
Stack.pop // parameters
JR(Reg.link (31))
```

Note that in general, keeping an important value in a register is more fragile than keeping it in a local variable, since registers are shared between all procedures but each procedure allocates its own set of local variables. We have to be careful that the code in the prologue between saving the address in `Reg.savedParamPtr` and copying it back out into the `paramPtr` variable does not use `Reg.savedParamPtr` for anything else. In particular, we cannot make any calls to any other procedure in this region of code, since such a procedure would overwrite `Reg.savedParamPtr`.

Since the calling code pushed the arguments onto the stack, they need to be popped off to restore the stack to its original state. For CS 241E, our convention will be to require the procedure to pop the arguments off the stack before it returns to the calling code, as shown in the epilogue of the procedure just above.

A subtle remaining issue is for the calling code to compute the values of the arguments and write them into the chunk that it has allocated on the stack. There is yet another chicken-and-egg problem here. In a high-level programming language, the argument to a procedure call is some expression to be evaluated. Evaluating the expression after we have allocated the parameter chunk on the stack would overwrite `Reg.result`, which holds the address of the parameter chunk. Even worse, the argument expression could contain yet another procedure call: for example, in the expression `f(g())`, the argument for the call to `f` is an expression that calls `g`. In such a case, the call to `g` would allocate space on the stack for any arguments of `g`, overwriting the address of the space we had already allocated for the arguments to `f`.

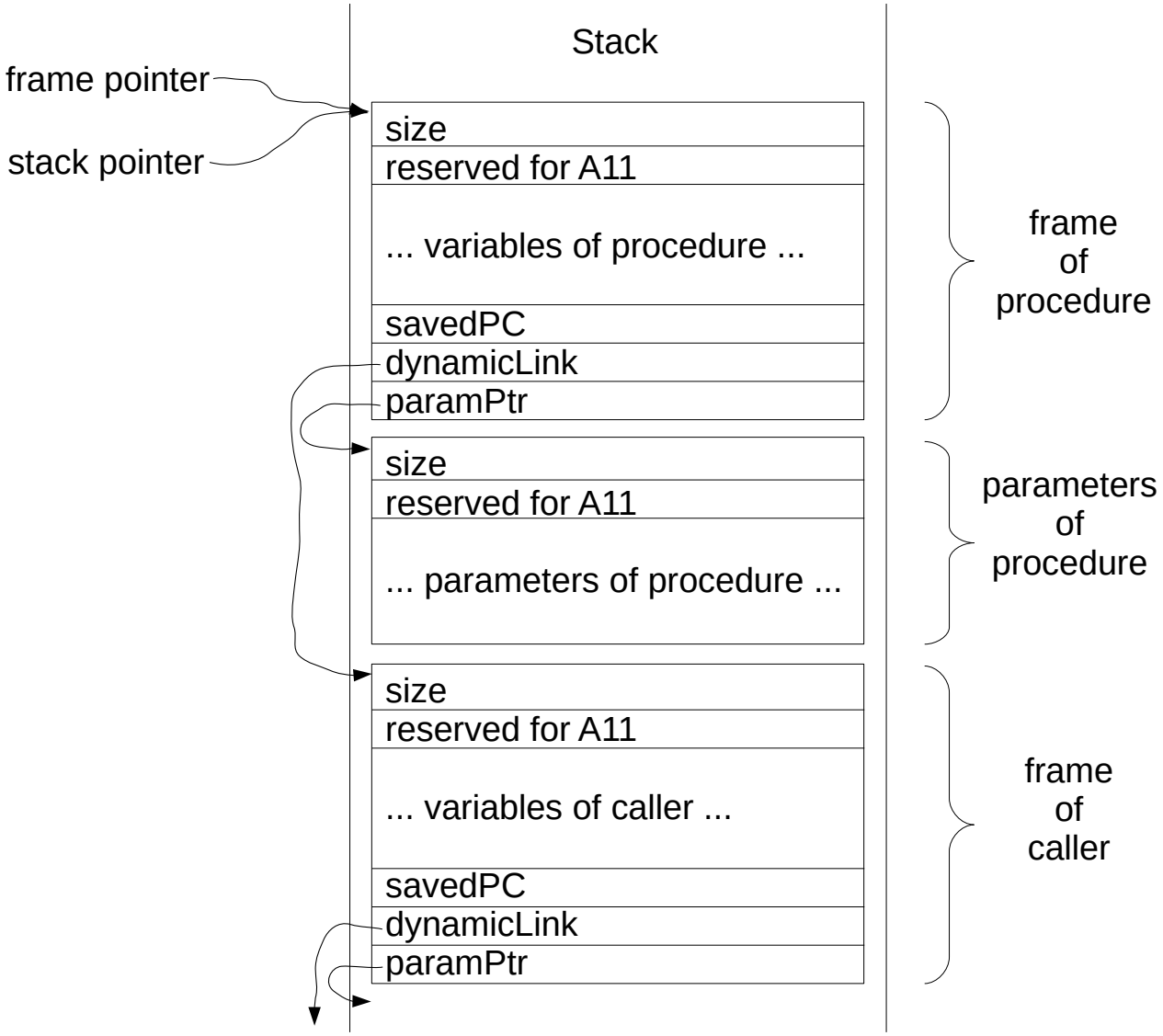
Our solution to this will be to evaluate the argument expressions *before* allocating the memory to pass the arguments to the procedure. In the case of the example `f(g())`, the generated code would complete the call to `g` before allocating memory for the arguments of `f`. For this to work, we need a place to store values of the arguments temporarily, after we have evaluated them but before we have allocated memory for passing them to the procedure. Variables are a convenient abstraction for this: we can just create fresh temporary variables in the calling code to store the values of the arguments. Then, after we allocate the memory for passing arguments to the procedure, we can copy the computed values of the arguments from these temporary variables into the allocated memory chunk. In effect, we transform the example expression `f(g())` into the equivalent code sequence `t := g(); f(t)`, where `t` is

a freshly introduced temporary variable. This decouples the call to `g` so that it is no longer mixed together with the code to call `f`. To summarize, the final procedure calling code looks like this:

```
evaluate arguments into temporary variables
Stack.allocate(parameters)
parameter 1 := temporary 1
...
parameter n := temporary n
LIS(Reg.targetPC (8))
Use(proc)
JALR(Reg.targetPC (8))
```

5.5 Summary of conventions

The following diagram illustrates the contents of the stack after the prologue of the procedure has finished setting up and the procedure body begins executing:



Before the call, the stack contained only the frame of the caller and the frame pointer contained its address. The caller pushed a chunk onto the stack for the procedure parameters. After the caller transferred control to the procedure, the procedure pushed a second chunk onto the stack for its frame. The frame pointer now points to the procedure frame. The dynamic link variable in the procedure frame points to the frame of the caller. The `paramPtr` variable in the procedure frame points to the chunk containing the parameters of the procedure.

The following is a summary of the CS 241E conventions for procedure calls:

- The stack pointer (register 30) and frame pointer (register 29) are **callee-save**: they are preserved by a procedure call.
- All other registers are **caller-save**: they may be modified by a procedure call.
- The **calling code** allocates the chunk to hold the arguments passed to the procedure. It provides the address of this chunk to the procedure in **register 3**, `Reg.result`. The

procedure pops the chunk that holds the arguments before it returns to the calling code.

- The **procedure** allocates its own frame and pops it before it returns to the calling code.

5.6 Implementing variable accesses

Since procedures can have parameters, the body of a procedure needs to access (read and write) both its local variables and its parameters. We learned earlier that local variables are stored at constant offsets in the frame and can be accessed at constant offsets from the frame pointer. To access a parameter, the assembly language code needs to first read the `paramPtr` variable from the frame of the procedure to obtain the address of the chunk containing the parameters. It can then read or write the parameter at a constant offset from that address in the parameter chunk.

When a compiler compiles a variable read or write, it needs to check whether the access is to a local variable or to a parameter. For a local variable, the compiler generates the `LW` or `SW` instruction to access the variable in the frame in the same way as in Assignment 3. For a parameter, the compiler generates the read of the `paramPtr` variable followed by the access in the parameter chunk as described in the previous paragraph. You will implement this functionality in Assignment 5 in the `eliminateVarAccessesA5` function.