Topic 16 - Code Generation: Procedures

Key Ideas

- procedure prologs and epilogs
- three tasks
  1. saving registers values between function calls
  2. saving the stack frame pointer
  3. passing function arguments
- namespace collisions

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 10.1-10.5
- CS241 – WLP4 Programming Language Specification

Review: Prologs and Epilogs

Recall from our Discussion of Code Generation

For the procedure **`wain`**

- Prolog
  - push the return address ($31) on the stack
  - push arguments on the stack
- Body of Procedure
  - store local variables on the stack
  - generate code for body of procedure
- Epilog
  - pop frame (local variables and arguments) off the stack
  - restore previous return address to $31
- Key Challenge: How to handle (1) registers (2) frame pointer and (3) arguments for functions calling other functions?

Multiple Procedures: Prologs and Epilogs

A Tale of Three Procs

- *Question:* What is handled in the prologs and epilogs of procedures **`f()`**, **`g()`** and **`wain()`**?

```
int f(…) {…}
int g(…) {…}
int wain(…) {…}
```

- *Handled once* in **`wain`**'s prolog
  - any .import and .export directives
  - constants ($4 ←4, $11 ←1 etc.)
- *Handled in each procedure's* prolog and epilog
  - update frame pointer, $29
  - save and restore our caller's return address, $31
  - save and restore the other registers

```
wain:
 <prolog>
    ⋮
 <epilog>
  jr $31
```

```
f:    …
 <prolog>
    ⋮
 <epilog>
  jr $31
```

```
g:   …
 <prolog>
    ⋮
 <epilog>
  jr $31
```

Multiple Procedures: 1. Saving Register Values

Three Different Approaches

Question: who saves what registers?

- say procedure **`f()`** calls procedure **`g()`**, i.e.
  `int f(…) {…g(…)…}`

1. *the caller f() saves any register values it needs*
   - f() saves all the registers which have values that it needs to be saved
   - f() may be saving registers that g() will not modify
2. *the callee g() saves any register values it modifies*
   - g() saves all registers whose values it will overwrite
   - g() may be saving register values that f() no longer needs

```
int f(…) {
  ⋮
  g();
  ⋮
}
```

```
int g(…) {
  ⋮
}
```

Multiple Procedures: 1. Saving Register Values

Three Different Approaches ...

Question: who saves what registers?

3. *hybrid*

- caller saves some registers and callee saves others
- for example:
  - caller saves $31 (because its value is overwritten when the instruction **`jalr`** is used).
  - callee saves the registers whose values it will overwrite
- this is the approach we've been following so far
- other hybrid approaches are possible

```
int f(...) {
  ⋮
  g();
  ⋮
}
```

```
int g(...) {
  ⋮
}
```

Multiple Procedures: 2. Saving the Frame Pointer

Question: who saves the frame pointer, $29?

- *if the callee g() saves $29* then it saves the registers and updates $29 to point to the start of its frame
  - if registers are saved before $29 is updated: need to calculate the start of the frame (i.e. how many registers have been saved)
  - if we update $29 before saving the registers: then we've changed its value before saving it
  - better to have caller save it
- *the caller f() saves $29*
  - f() saves its value for $29
  - g() updates the value of $29

```
int f (...) {
  ⋮
  push($29)
  push($31)
  lis $5
  .word g
  jalr $5
  $31 ← pop()
  $29 ← pop()
  ⋮
}

int g (...) {
  sub $29,$30,$4
  ⋮
}
```

Multiple Procedures: 3. Passing Parameters

Program

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

Currently for `wain`

- saving parameters (always 2 of them) and local variables on the stack
- frame pointer ($29) points to the beginning of the frame
- stack pointer ($30) points to the end of the stack
- locations in the symbol table are relative to the frame pointer ($29)

Stack

$29→	0xFC	1	a
	0xF8	2	b
$30→	0xF4	0xF0	c
	0xF0	↓	

Symbol Table

Name	Type	Offset
a	int	0x0
b	int	-0x4
c	int	-0x8

Multiple Procedures: 3. Passing Parameters

Dealing with a Varying Number of Parameters

- *Problem:* Could use registers for parameters but what if there are a lot of them? e.g.
  factor → ID($expr_1$, $expr_2$, …, $expr_n$)
- Solution: caller *loads parameters on the stack*, e.g.

```
code(factor) =
    push($29)
    push($31)
    code(expr1)
    push($3)
      ⋮
    code(exprn)
    push($3)
```

```
lis $5
.word FID
jalr $5
argn = pop()
  ⋮
arg1 = pop()
$31 = pop()
$29 = pop()
```

Stack

$29
$31
$expr_1$
$expr_2$
⋮
$expr_n$
↓

Multiple Procedures: 3. Passing Parameters

Generating Code for a Procedure

- procedure → INT ID(params) { dcls stmts RETURN expr ; }

Output

```
code( procedure ) =
    sub $29, $30, $4                    ; update frame pointer
    push regs ($1, $2, $5, $29, $31)    ; callee saves registers
    code(dcls)                          ; that will be overwritten
    code (stmts)
    code (expr)
    pop regs                            ; callee restores registers
    add $30, $29, $4                    ; pop rest of stack frame
    jr $31                              ; return to caller
```

- Note: The caller has already placed the params on the stack.

Multiple Procedures: 3. Passing Parameters

A Varying Number of Parameters

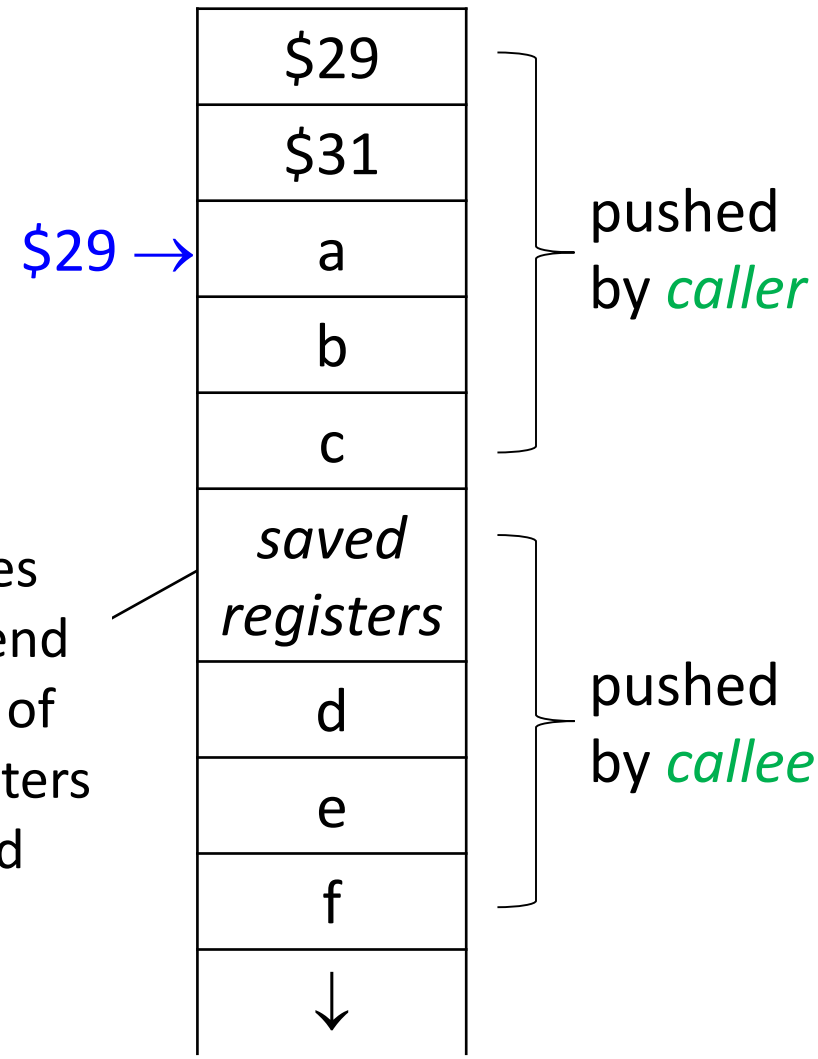
```
int g(int a, int b, int c) {
  int d = 0;
  int e = 0;
  int f = 0;
  ...
}
```

Symbol Table

Name	Type	Offset
a	int	0x0
b	int	-0x4
c	int	-0x8
d	int	~~-0xC~~
e	int	~~-0x10~~
f	int	~~-0x14~~

values depend on # of registers saved

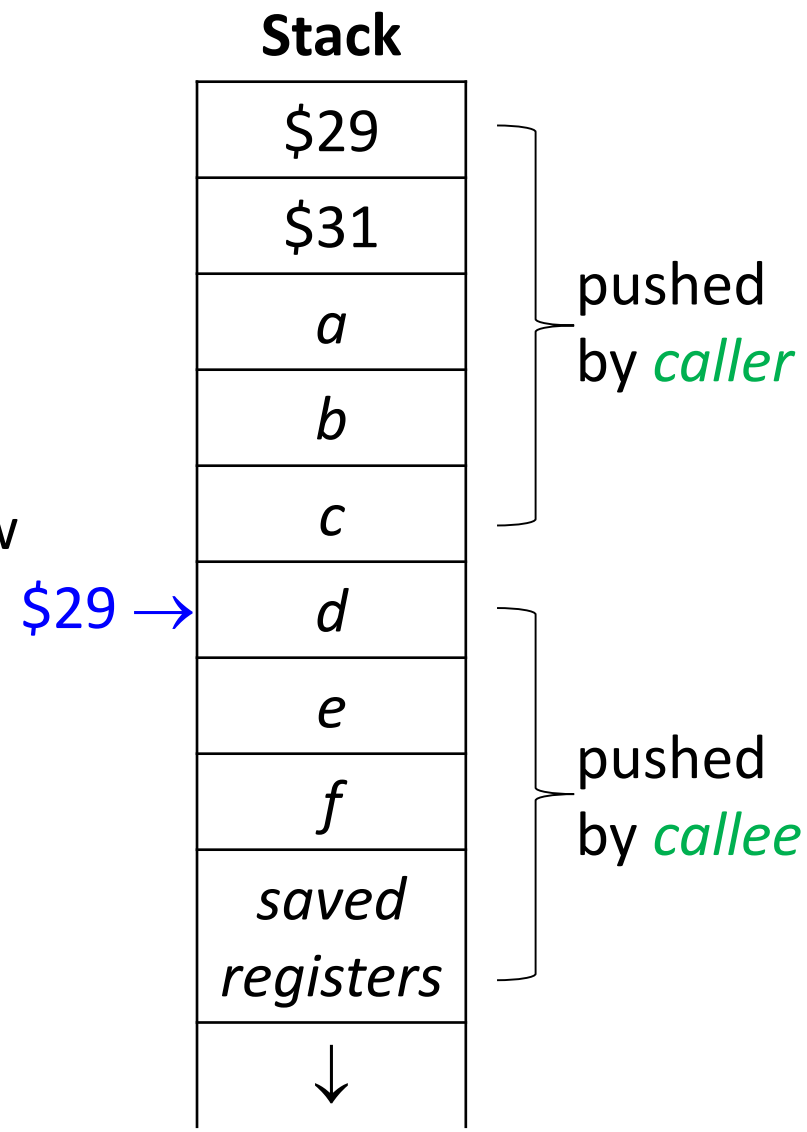
Stack



Multiple Procedures: 3. Passing Parameters

A Varying Number of Parameters

- *Problem:* the parameters for g(), i.e. *a*, *b*, *c,* and its local variables, i.e. *d*, *e*, *f*, are separated by the saved registers
- some of the values in the symbol table (on the previous slide) are now incorrect
- *Solution:* could save registers after local variables
- to convert the old symbol values to the new symbol values:
  add (the number of params × 4)



Multiple Procedures: 3. Passing Parameters

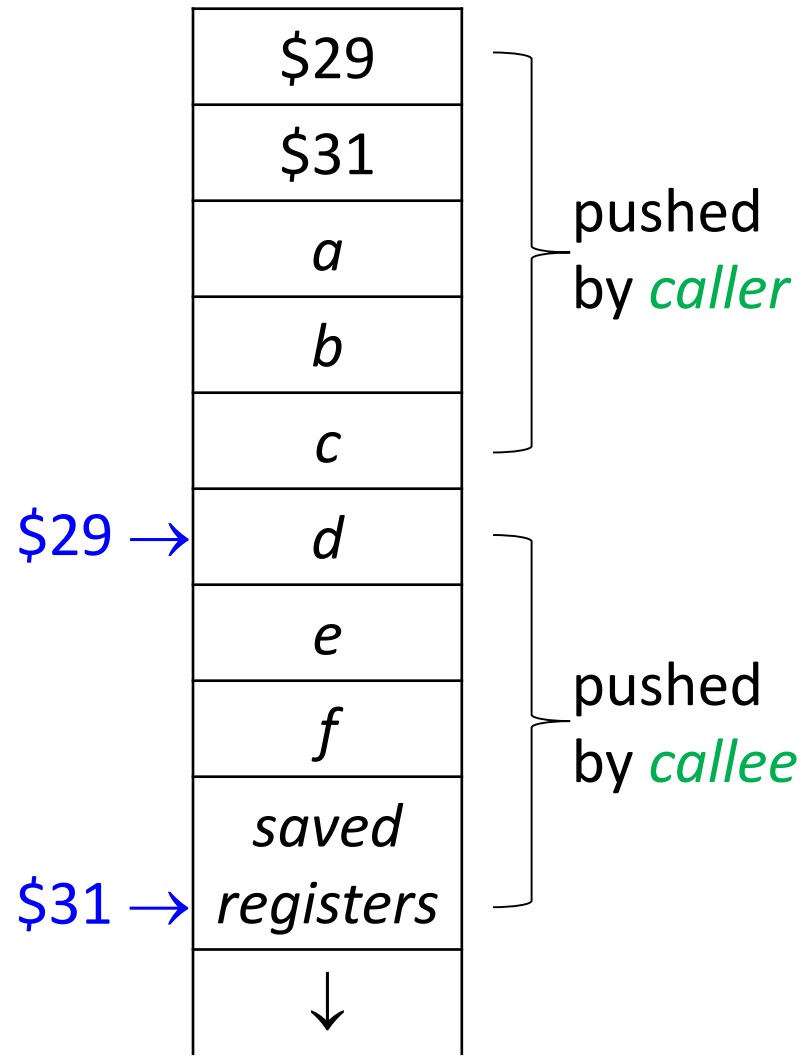
A Varying Number of Parameters

- *positive offsets* are the params
- *zero and negative offsets* are the local variables

Symbol Table

Name	Type	Offset
a	int	0xC
b	int	0x8
c	int	0x4
d	int	0x0
e	int	-0x4
f	int	-0x8

Stack



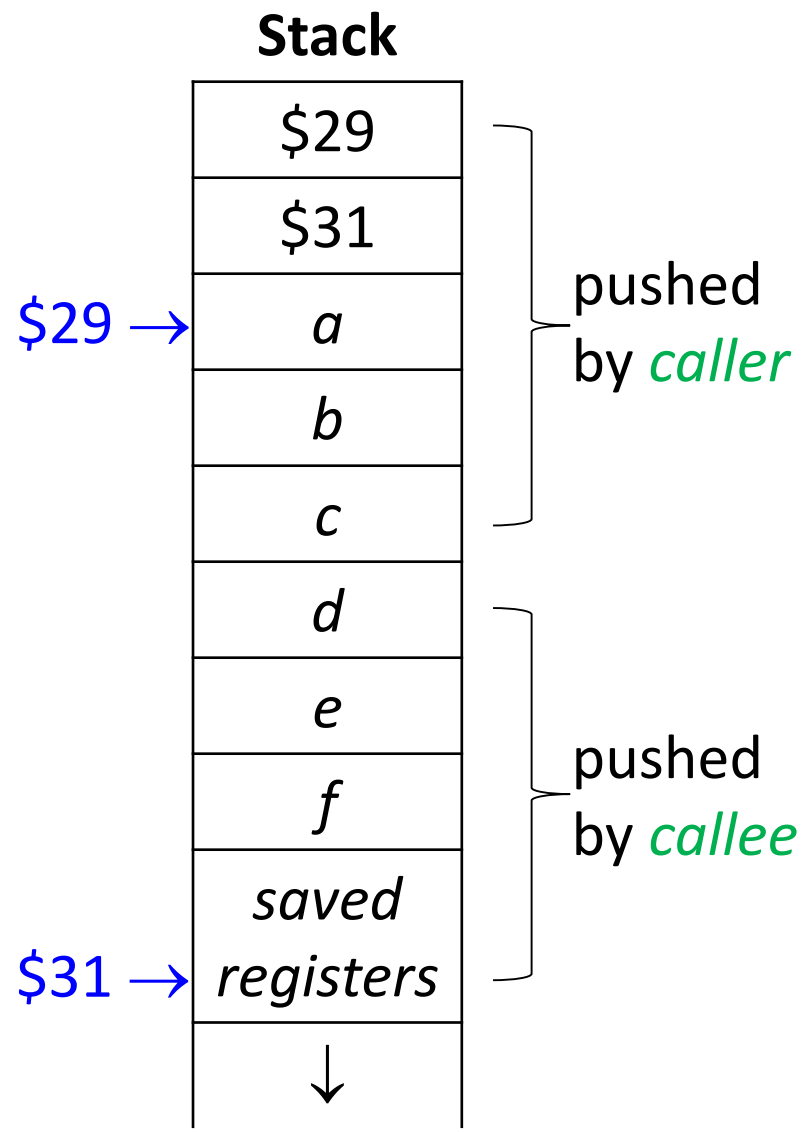
Multiple Procedures: 3. Passing Parameters

A Varying Number of Parameters

- *alternative:* could keep $29 pointing at the first parameter, i.e. at "*a*".
- having the caller save the registers is **not** a good idea especially if one procedure, say f(), calls another procedure multiple times, e.g.

```
int f()  {
   g(1);
   g(2);
   g(3);
   }
```

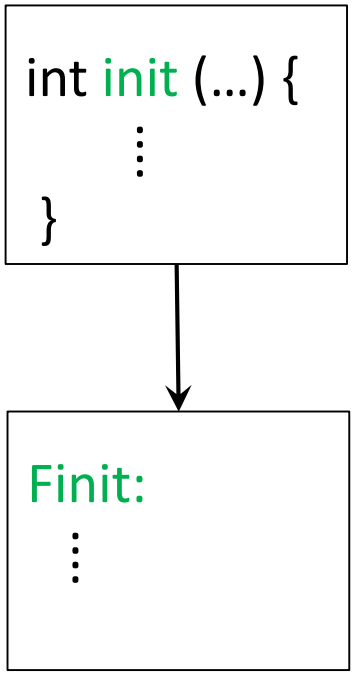
- too inefficient, repeating the same code 3 times.



Multiple Procedures: Namespace Collision

Namespace Collisions

- *Question:* If names of procedures map onto labels, what if a procedure uses the same name as a label in the runtime environment?
- called a *namespace collision*
- e.g. you have a function called init() and the underlying system already uses init as a label
- Solution: reserve the letter F for functions
- when processing WLP4 procedure names append the letter F in front of the corresponding MIPS assembly language label
- e.g. the procedure "int init(…) { … }" in WLP4 becomes "Finit: …" in MIPS assembly language.



Summary

Caller

- Pushes
  - frame pointer $29
  - stack pointer $31
  - function arguments

  onto the stack.

Callee

- Pushes
  - local variables
  - register values it will overwrite

  onto the stack.
- Frame pointer can point to the first argument (*a*) or the first local variable (*d*).