Lecture 22

Code Generation for Procedures in WLP4

CS 241: Foundations of Sequential Programs
Fall 2014

Troy Vasiga et al
University of Waterloo

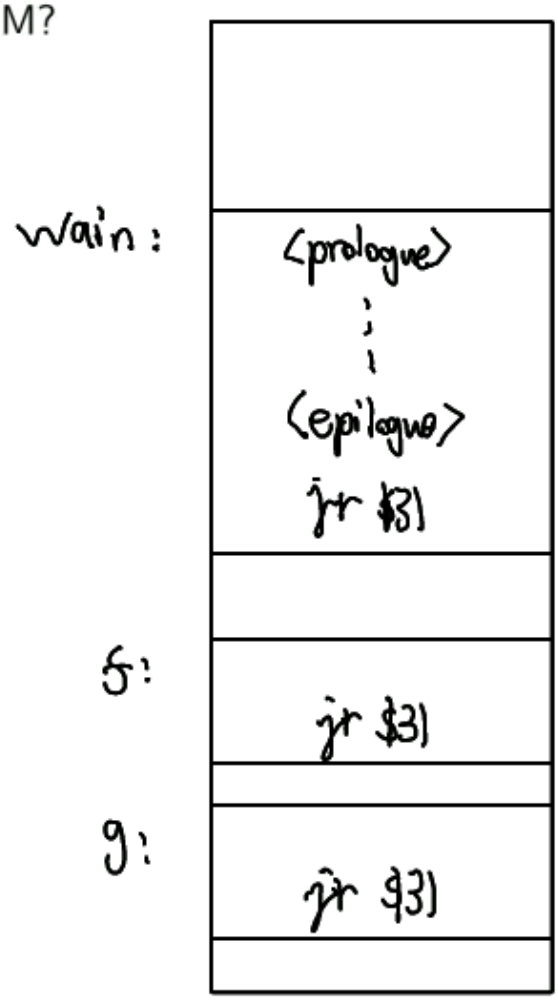
A picture

Imagine we have code like:

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

What does it look like, in terms of RAM?

int f()
int g()
int wain()



What does each procedure need to do?

Each procedure will have its own prologue and epilogue.

What should these contain?

- we don't need to worry about `.import` or `.export`
- we do need to worry about $29
- we do need to save registers
- we do need to restore registers
- we do need to return to our caller (i.e., `jr $31`)

do this once, just in main

each procedure has its own frame

Saving registers

Overall (naive) idea: save "everything" $1, $2, $5, $6, $29, $31
(maybe)

Suppose f calls g. Two approaches:

- Caller-save: f saves all registers which have useful information before calling g.
- Callee-save: g saves any register it is about to overwrite.

What have we been doing so far? Both.

hybrid — caller save: $31
callee save: everything else

What about $29?

Suppose g (the callee) saves $29.

messy { g must save registers on the stack and point $29 to the bottom of g's frame
change $30: then need to calculate the bottom stack and make $29 point there

Suppose f (the caller) saves $29.

clean { g sets $29 ⇒ i.e. sub $29, $30, $4

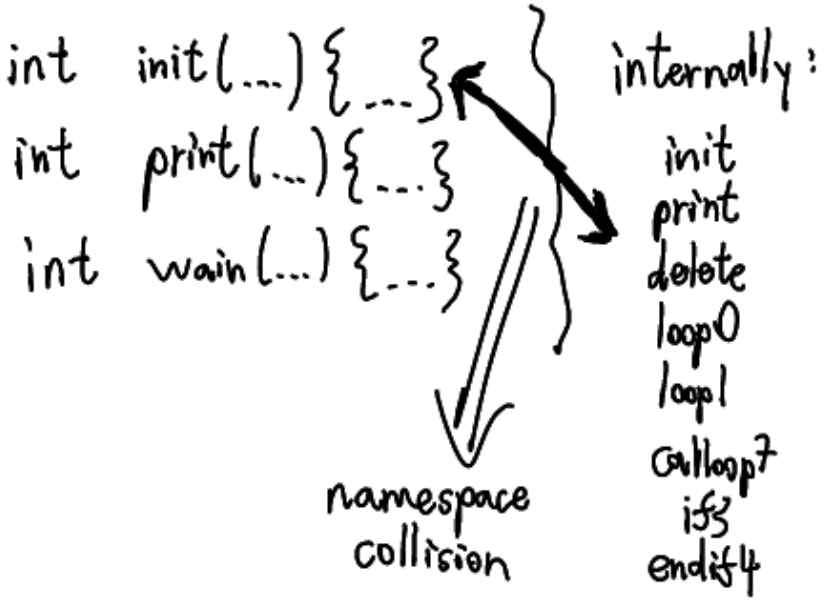
Verdict and MIPS outline:

f saves $29

```
f: ...
   push($29)      jalr $5
   push($31)      pop($31)
   lis $5         pop($29)
   .word g
```

What's my name again?

If names of procedures map to labels, then...



Prepend "F" in from
⇒ F<user proc name>

eg. Fprint
Finit

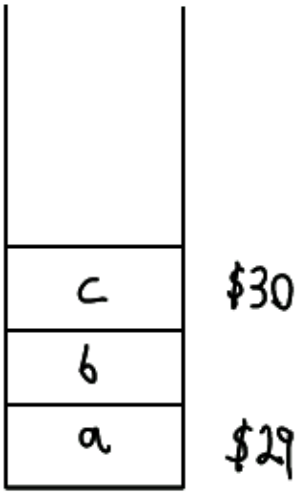
Parameters

Recall what we did with the symbol table for each procedure.

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

Symbol Table and Stack:

a	int	0
b	int	-1
c	int	-2

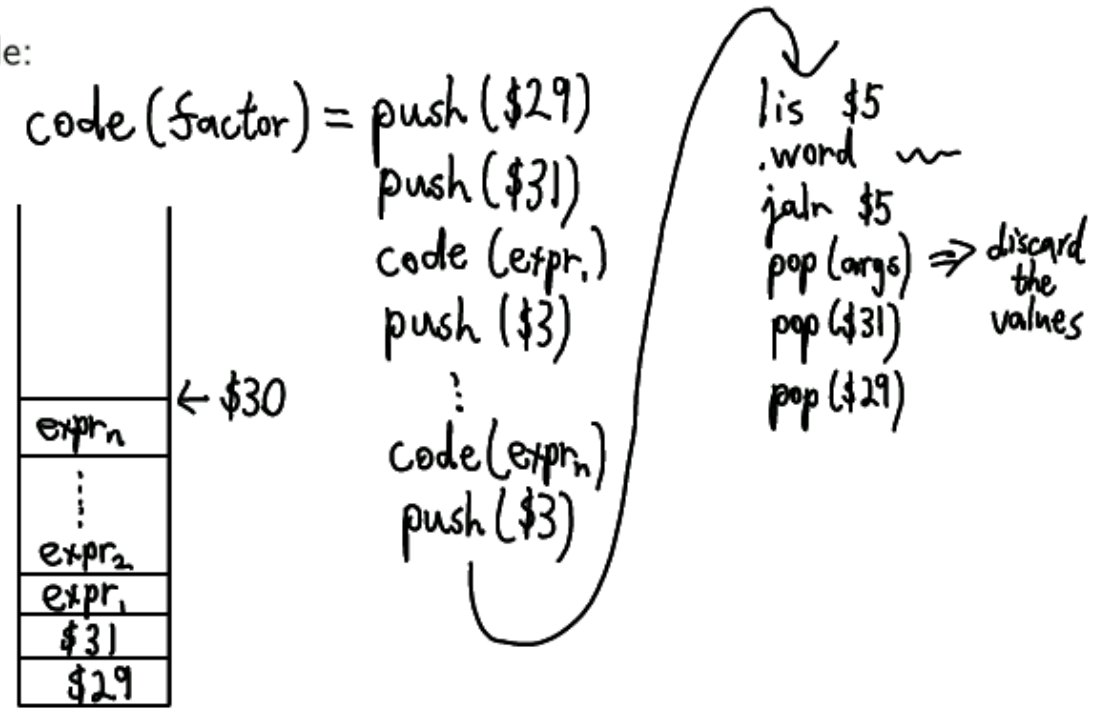


Parameter Ideas

We could use registers for parameters.

Or, we could be smart, since we could have a lot of parameters, as in: `factor → ID(expr1, ..., exprn)`

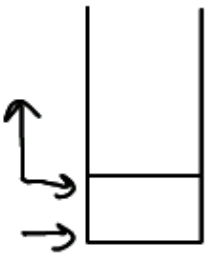
Code:



Code for procedure

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

code(procedure) = sub $29, $30, $4
+ push regs ($1, $2, $5, $6, $29, $31)



+ code(dcls) ← + code(stmts)
+ code(expr)
+ pop regs
add $30, $29, $4
jr $31

Problems?

why no code(params)

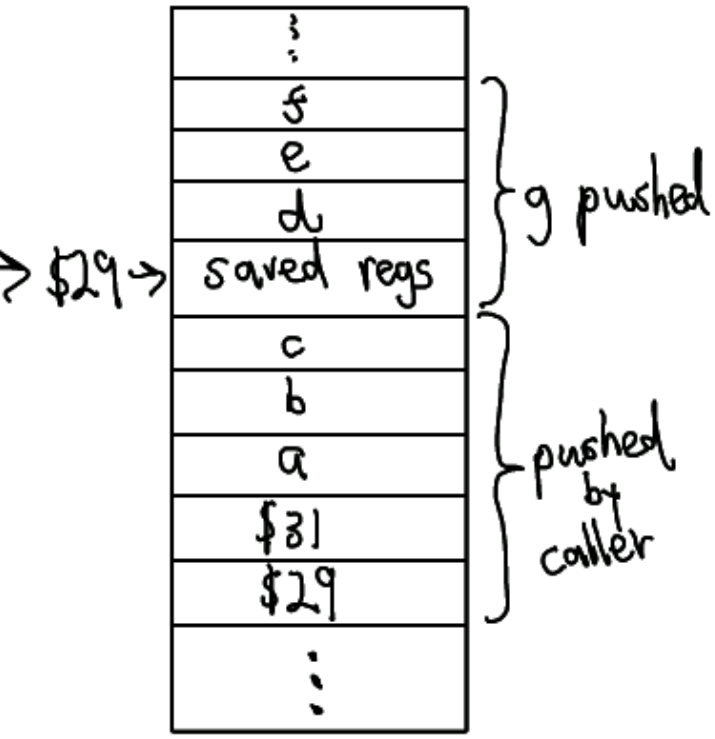
Try an example

Suppose g (the callee) is:

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

Symbol Table and Picture:

a	int	0
b	int	-1
c	int	-2
d	int	-3
e	int	-4
f	int	-5



Problems and Mathematical Fix

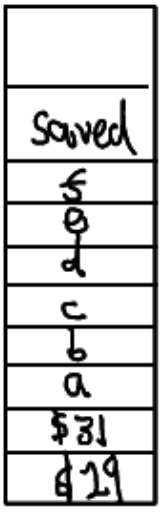
Parameters and local variables:

separated by my "saved registers"

Fix: ⇒ save registers after pushing local variables

Old symbol table values and new symbol table values:

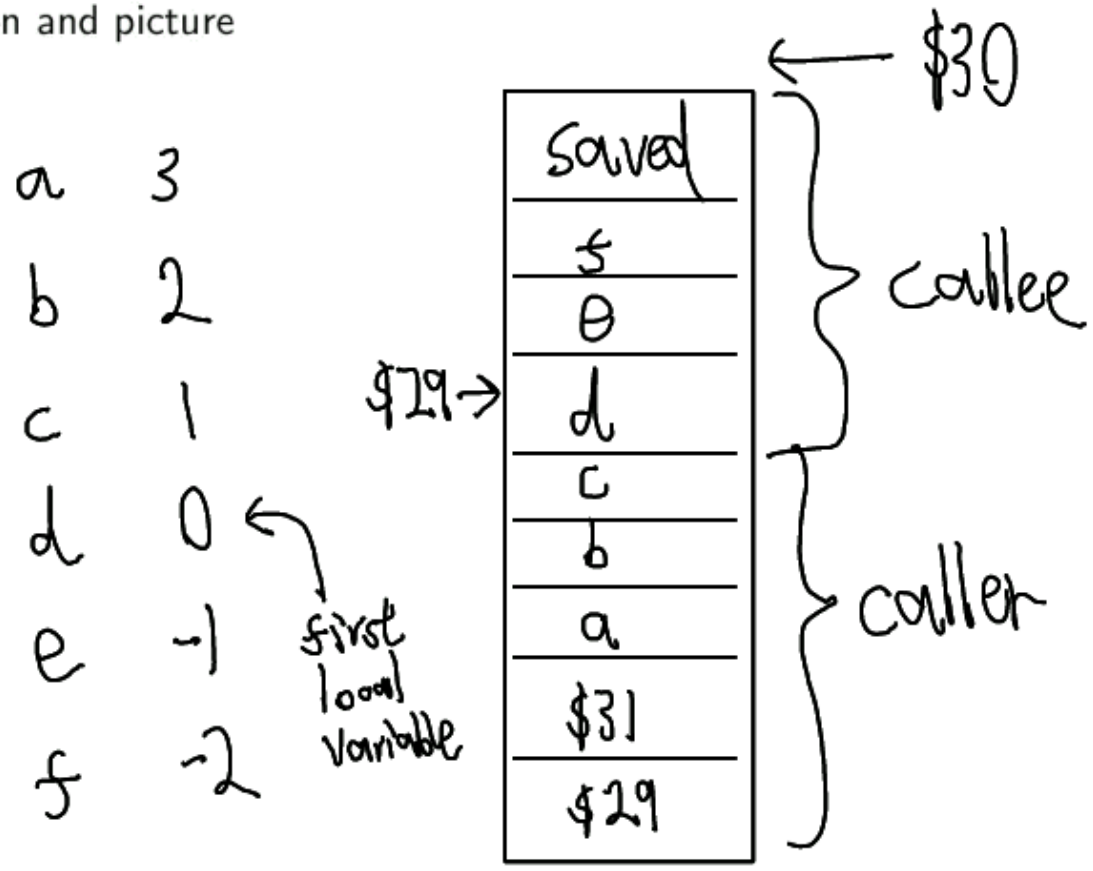
Add (# of args) to offset



One last problem:

The big fix

Solution and picture



Alternate fix?

Keep $29 pointing at a

$\Rightarrow$ messy

Could also save all the registers in the caller, but...

```
f() { g();
      g();
      g(); }
```

$\Rightarrow$ gets inefficient

Optimization

CS444

Large and interesting topics: we will just skim.
Recall that we can have an infinite number of MIPS programs that are equivalent to a particular WLP4 program. What are the "good" MIPS programs that we would like to generate?

- increasing speed of instructions in actual code: hard

  ⇒ some operations are faster:
  C++; is slower than ++C;

- Reducing number of statements (stack usages, register allocation)

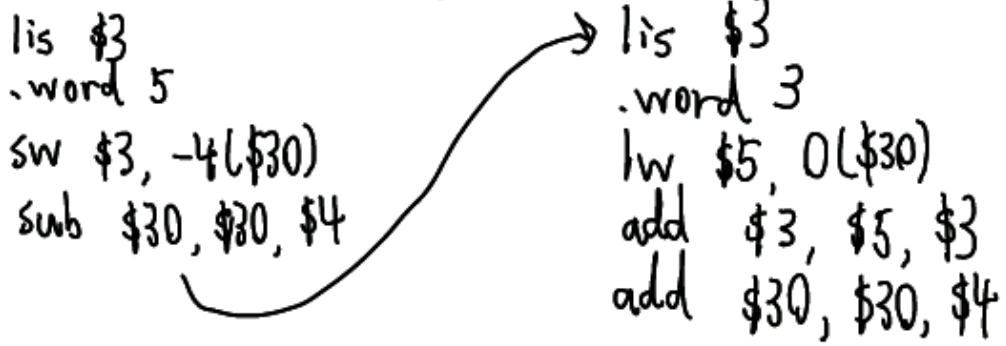
good approximation

Notice: minimal routine/size is uncomputable.
instead use heristics.

Optimization: Constant Folding

Consider an expression 5+3.

What our compiler currently does:



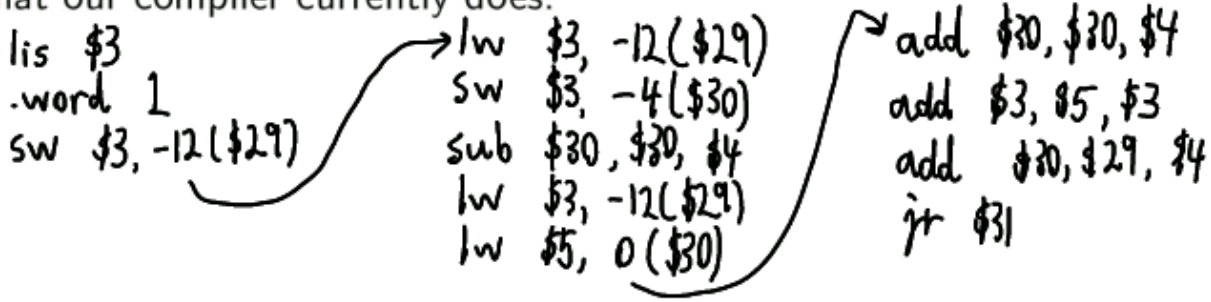
What it could do:

```
lis $3
.word 8
```

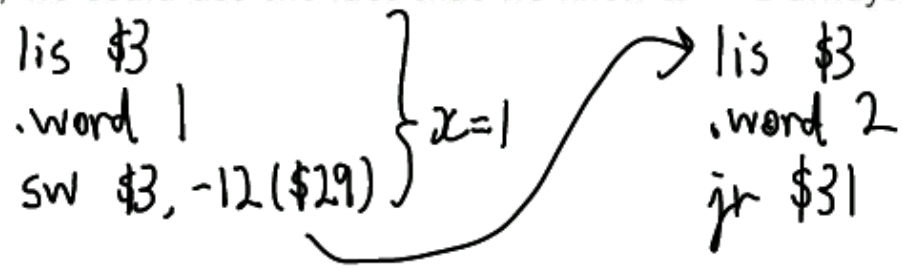
Optimization: Constant Propagation

```
int x = 1;
return x+x;
```

What our compiler currently does:



Or, we could use the fact that we know `x = 1` always and:



Further, if `x` is never used anywhere else:

```
lis $3
.word 2
```

Optimization: Common subexpression elimination

(1) Compute a+b into $3
(2) mult $3, $3
    mflo $3

Imagine you had:
`(a+b) * (a+b)`

Optimization: Dead-code elimination

What makes code dead?

→ never executed

→ eg. test is always false

⇒ logical implication

Optimization: Register allocation

- Notice that many registers (i.e., \$14 through \$28) are not used by our compilation system.
- Notice that using registers would save instructions. Why?

no push/pop

- Suppose we could store 15 variables. Which ones would we store?

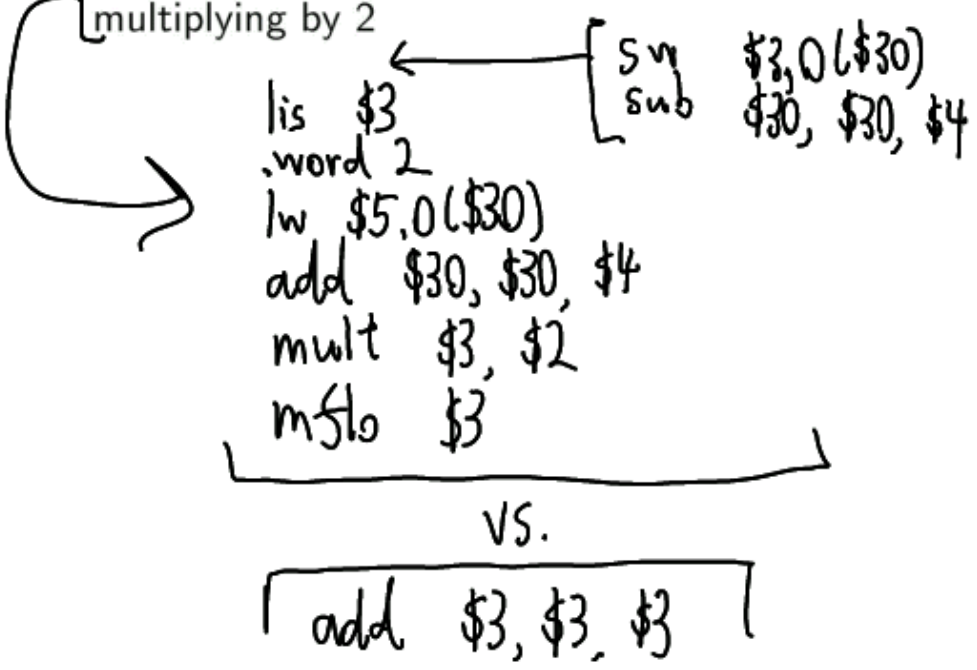
most used ones
⇒ maintain priority

- & do you see a problem?

↳ address-of ⇒ push out into RAM

Optimization: Strength Reduction

- add is usually faster than `mult` (in the real world)
- however, in CS241, if we just count MIPS instructions, consider multiplying by 2



Optimization: Inlining Procedures

```
int f(int x) { return x+x; }
int wain(int a, int b) { return f(a); }
```

```
int wain(int a, int b) {
    return a+a;
}
```

Pros/Cons:

- ↳ if all calls to f are inlined, no need to generate the code for f at all
- if body of f is big and/or called many times, code expands.

Optimization: Tail Recursion in Procedures

In WLP4, we must have exactly one `return` as the last statement in every function.

In other programming languages, you can say something like:

```
int fact(int n, int a) {
  if (n==0) return a;
  else return fact(n-1, n*a);
}
```

fact(3,1)
⇒ fact(2,3)
⇒ fact(1,6)
⇒ fact(0,6) ⇒ 6

Notice that the last thing the function does is returning a value: there is no additional work to be done. Thus, the frame can be **reused**!

A few words about overloading

```
int f(int a) { ... }
int f(int a, int* b) { ... }
```

→ Fi_f

→ Fip_f

Solved by *name mangling*:

F + <types of params> + __ + <name>

Problems: C++

⇒ no standard on name mangling

⇒ had to link two pieces of code from different compilers

Solutions: `extern` ⇒ do not mangle my name!