Compiling Functions
Stack organization
how to lay out the program stack

• The **stack** is the primary memory area for managing the interaction between functions
  • Arguments passed from caller to callee (though this may happen in registers as an optimization)
  • Return values passed from callee to caller (though this may happen in registers as an optimization)
  • Local variables for each function
  • Saved registers for each function (caller saves registers callee might use, and callee saves registers caller might need)
  • “Spilled” registers
key mechanisms for program stack

• Reserved area of memory
  • Different area than program text, globals, heap
  • In Risc-V, stack grows **down**: pushing an element onto the stack puts it at a lower address than the current top of the stack

• Two pointers
  • **Stack pointer** *(sp)*: points to the top of the stack
    • In our approach, sp will point to the next *open* spot on the stack
    • Pushing on the stack: store to sp, decrement sp by appropriate amount
  • **Frame pointer** *(fp)*: points to the base of the activation record
    • Locations of other parts of the stack are relative to fp
    • Optimization: can eliminate fp, but makes code generation more complicated (how? why?)
Activation record

- What does an activation record look like for a function?
- Caller places arguments and return value on stack
- Caller places its return address (where it should return to) on stack
  - Why? Register holding this address will be overwritten when invoking callee
- Callee saves old frame pointer on stack, then moves frame pointer to point to the base of its record
- Callee creates space for its local variables on stack
Code generation for functions
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```
int foo() {
  ...
  z = bar(x, y);
}

int bar(int a, int b) {
  int c;
}
```

![Stack Diagram]

- old_fp
- fp
- sp
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```c
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
```

```
<table>
<thead>
<tr>
<th>old_fp</th>
<th>fp</th>
</tr>
</thead>
<tbody>
<tr>
<td>x/a</td>
<td></td>
</tr>
<tr>
<td>y/b</td>
<td></td>
</tr>
</tbody>
</table>
```

SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```
int foo() {
  ...
  z = bar(x, y);
}

int bar(int a, int b) {
  int c;
}
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
```
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```assembly
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
```

```c
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
```
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
```

SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```
int foo() {
    ...
    z = bar(x, y);
}
int bar(int a, int b) {
    int c;
}
```

```
old_fp
x/a
y/b
RA
foo_fp
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

bar:
SW FP, 0(SP)
```
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

```
int foo() {
  ...
  z = bar(x, y);
}
```

```
int bar(int a, int b) {
  int c;
}
```

```
old_fp
x/a
y/b
RA
foo_fp
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

bar:
SW FP, 0(SP)
MV FP, SP
SUBI SP, SP, 4
```
What should happen when foo calls bar?

1. Put Arguments on stack
2. Allocate space for return value
3. Save old return address (return address of foo) on stack
4. Jump to bar (using JR instructor)
5. Save old frame pointer (foo’s frame pointer) on stack
6. Set frame pointer to point to top of stack
7. Allocate space for local variables of bar

int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
What about returning from bar?

1. Put return value in appropriate location
2. Reset stack pointer to top of foo’s activation record
3. Reset frame pointer back to old location
4. Return to foo

```c
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
    ...
}
```

```
old_fp
x/a
y/b
RA
foo_fp
  c
  fp
  sp
```
What about returning from bar?

1. Put return value in appropriate location
2. Reset stack pointer to top of foo’s activation record
3. Reset frame pointer back to old location
4. Return to foo

```c
int foo() {
    ...
    z = bar(x, y);
}
int bar(int a, int b) {
    int c;
}
```

![Diagram showing stack frames and registers](diagram.png)
What about returning from bar?

1. Put return value in appropriate location
2. Reset stack pointer to top of foo’s activation record
3. Reset frame pointer back to old location
4. Return to foo

```c
int foo() {
    ...
    z = bar(x, y);
}
int bar(int a, int b) {
    int c;
    ...
    SW Tr, 8(fp)
    MV SP, FP
}
```
What about returning from bar?

1. Put return value in appropriate location
2. Reset stack pointer to top of foo’s activation record
3. Reset frame pointer back to old location
4. Return to foo

```
int foo() {
    ...
    z = bar(x, y);
}
```

```
int bar(int a, int b) {
    int c;
}
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

bar:
SW FP, 0(SP)
MV FP, SP
SUBI SP, SP, 4
SUBI SP, SP, 4
...
SW Tr, 8(fp)
MV SP, FP
LW FP, 0(FP)
```
What about returning from bar?

1. Put return value in appropriate location
2. Reset stack pointer to top of foo’s activation record
3. Reset frame pointer back to old location
4. Return to foo

int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

bar:
SW FP, 0(SP)
MV FP, SP
SUBI SP, SP, 4
SUBI SP, SP, 4
...
SW Tr, 8(fp)
MV SP, FP
LW FP, 0(FP)
RET
```
Now what does foo do?

1. Restore old return address
2. Retrieve return value from stack
3. Remove arguments and return value from stack

```c
int foo() {
    ...
    z = bar(x, y);
}
```

```c
int bar(int a, int b) {
    int c;
}
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar
```

```
old_fp
x/a
y/b
Tr
RA
foo_fp
```

Now what does foo do?

1. Restore old return address
2. Retrieve return value from stack
3. Remove arguments and return value from stack

```
int foo() {
    ...
    z = bar(x, y);
}
int bar(int a, int b) {
    int c;
}
```
Now what does foo do?

1. Restore old return address
2. Retrieve return value from stack
3. Remove arguments and return value from stack

```c
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
```

```
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

LW RA, 4(SP)
LW Tr, 8(SP)
```
Now what does foo do?

1. Restore old return address
2. Retrieve return value from stack
3. Remove arguments and return value from stack

```c
int foo() {
    ...
    z = bar(x, y);
}

int bar(int a, int b) {
    int c;
}
```

```assembly
SW Tx, 0(SP)
SW Ty, -4(SP)
SUBI SP, SP, 8
SUBI SP, SP, 4
SW RA, 0(SP)
SUBI SP, SP, 4
JR bar

LW RA, 4(SP)
LW Tr, 8(SP)
ADDI SP, SP, 16
```

```plaintext
old_fp ← fp
sp ← fp
```
What happens if foo and bar reuse registers?
two functions

• Ignoring machinery for stack/return address/frame management

foo:
LI t0, 10
ADDI t1, t0, 7
...
JR bar
...
ADDI t2, t1, 10

bar:
LI t1, 20
...
RET

• What’s wrong with this code?
saving registers

• To avoid overwriting registers, it is important to save all registers that the caller is using and the callee will overwrite
  • Careful about “using”: a caller needs a register if the value it has before the callee is invoked is used after the callee is returned
  • More precisely, the register is live across the function call (we will define this more carefully in a later lecture)
• Save registers onto the stack when making a call
• Restore registers from the stack when returning from a call
Problem

• Do not know the caller/callee relationship!

• Caller may not know all possible functions callee invokes → cannot tell exactly which registers will be overwritten

• Callee may not know who calls it → cannot tell exactly which registers are in use
callees saves vs. caller saves

• Can be conservative and save extra registers
  • Callee can save all registers it overwrites even if the caller doesn’t use them: **callees saves**
  • Caller can save all registers it uses, even if callee does not overwrite them: **caller saves**

• Who saves the registers determines which activation record holds the registers
  • Callee saves: put saved registers on stack before allocating space for locals, restore them on return
  • Caller saves: put saved registers on stack before allocating space for arguments and return values, restore them on return

• Question: why not put saved registers on stack after arguments and return values?
Determining what register saving convention to use is part of a system’s application binary interface
- All software written for an architecture/OS should use the same convention
- What happens if not?
- Can use some combination of caller saves and callee saves
  - Risc-V dictates that some registers are the caller’s responsibility to save, and some registers are the callee’s responsibility to save

In project, we will always use callee saves: save all registers written by the callee
- One exception: RA gets overwritten by JR, so caller must save it