SWEN430 - Compiler Engineering (2018)

Lecture 17 - Machine Code II

Lindsay Groves and David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Translation — Stack Machine Approach

- Stack-based approach is simplest for **recursive expressions**:

While	x86

`int y = 2 * x;`	`pushq $2`
	`pushq -8(%rbp)`
	`popq %rbx`
	`popq %rax`
	`imulq %rax,%rbx`
	`pushq %rax`
	`popq %rax`
	`movq %rax, -16(%rbp)`

- Assuming `x` stored at `-8(%rbp)`, `y` at `-16(%rbp)`
- This approach is similar as for **JVM code generation**
- But, is **inefficient** and leads to **redundant instructions**

Translation — Modified Stack Machine Approach

- Improved approach exploits x86 **addressing modes**:

While	x86

`int y = 2 * x;`	`pushq $2`
	`popq %rax`
	`imulq -8(%rbp),%rax`
	`pushq %rax`
	`popq %rax`
	`movq %rax, -16(%rbp)`

- Here, we save 2 instructions by operating **directly on stack**

- Still inefficient as memory **expensive** compared to registers

Translation — Register-Based Approach

- Optimal translation uses **registers-only** for intermediate results:

While	x86

`int y = 2 * x;`	`movq $2, %rax`
	`movq -8(%rbp), %rbx`
	`imulq %rbx, %rax`
	`movq %rax, -16(%rbp)`

- Here, we have saved 4 instructions!
- **Only memory accesses** are for reading / writing variables
- Care required to avoid **clobbering values** stored in registers

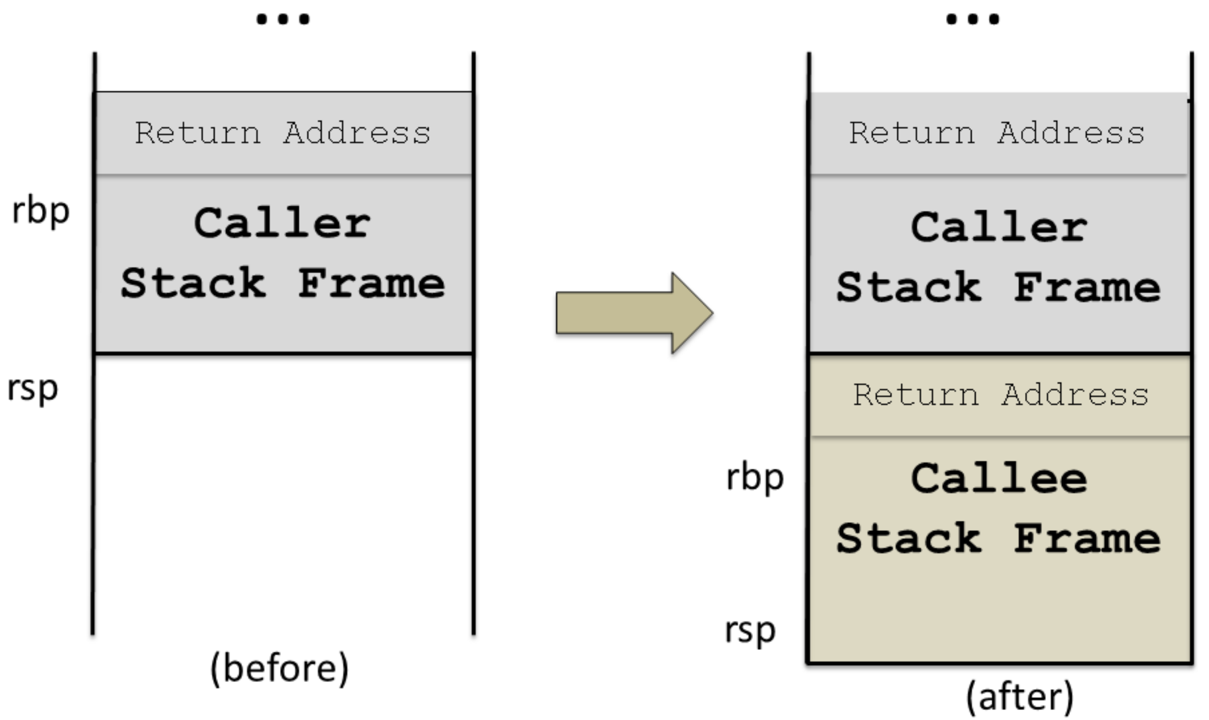
Translation — Register-Based Approach

```
movq $8, %rax          /* target: %rax */
movq -8(%rbp), %rbx    /* target: %rbx (%rax is used)*/
imulq %rbx, %rax       /* target: %rax */
movq %rax, -16(%rbp)   /* target: %rax */
```

- To **implement** register-based approach:
  - Recursively **descend** the expression tree
  - At each point, maintain list of **available registers**
  - For each subexpression, allocate a **target register**

Call Stack

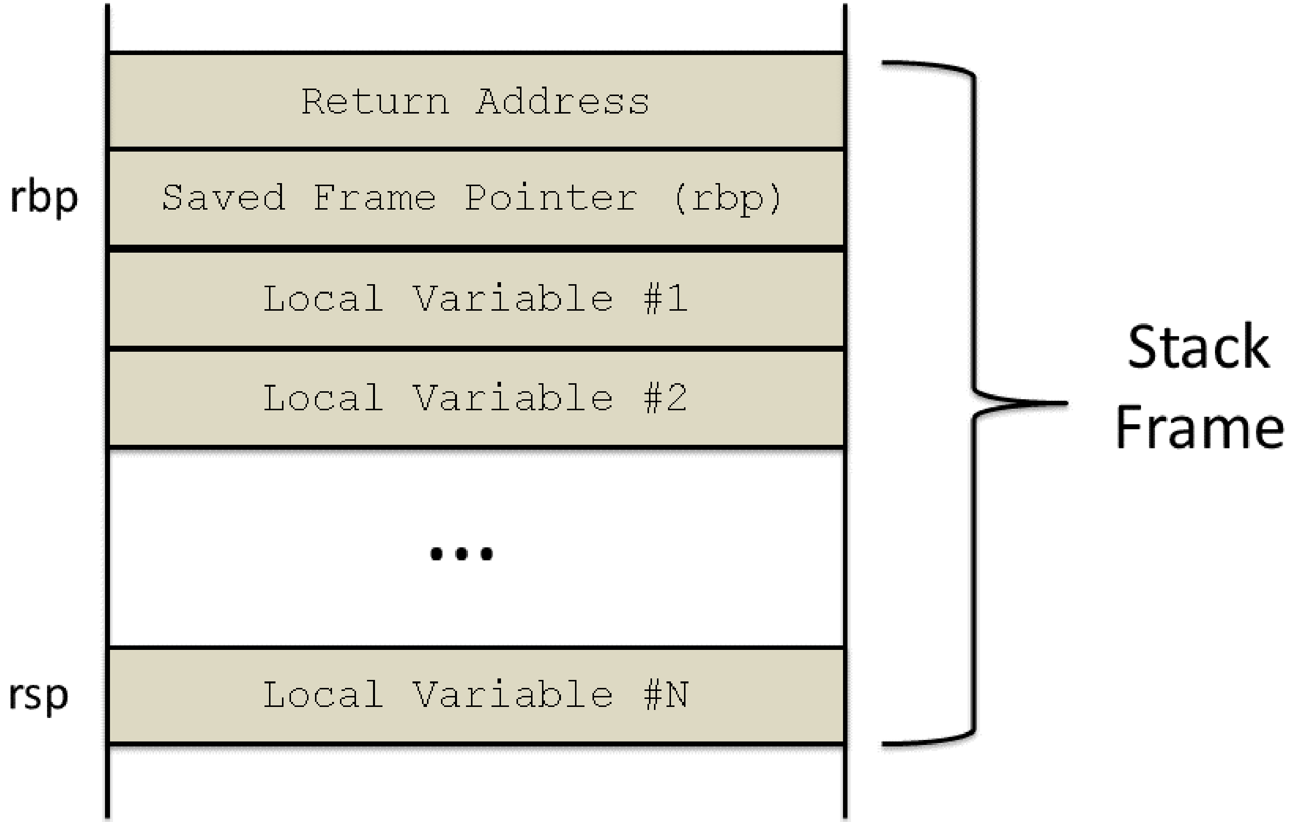
`call proc`	call procedure `proc` (`push %rip + 2` ; `jmp operand`)
`ret`	return from procedure (`pop %rip`)



- Method calls implemented via **call** instruction
- This pushes address of **next instruction** then jumps to target

Stack Frame Layout

- Each function invocation utilises a **stack frame**:



Initialising the Stack Frame

`enter`	Enter stack frame (not used, but roughly equivalent to `push %rbp` ; `mov %rsp,%rbp`)
`leave`	Leave stack frame (equivalent to `mov %rsp, %rbp` ; `pop %rbp`)

- x86 provides instructions for **creating** / **destroying** stack frame:

```
pushq    %rbp        /* save old frame pointer */
movq     %rsp, %rbp /* setup new frame pointer */
subq     $16, %rsp  /* allocate space on stack */
...
leave               /* restore stack & frame pointer */
ret                 /* return from function */
```

- For some reason, `enter` instruction **rarely used**!
- Instead, stack frame more commonly setup manually

Calling Conventions

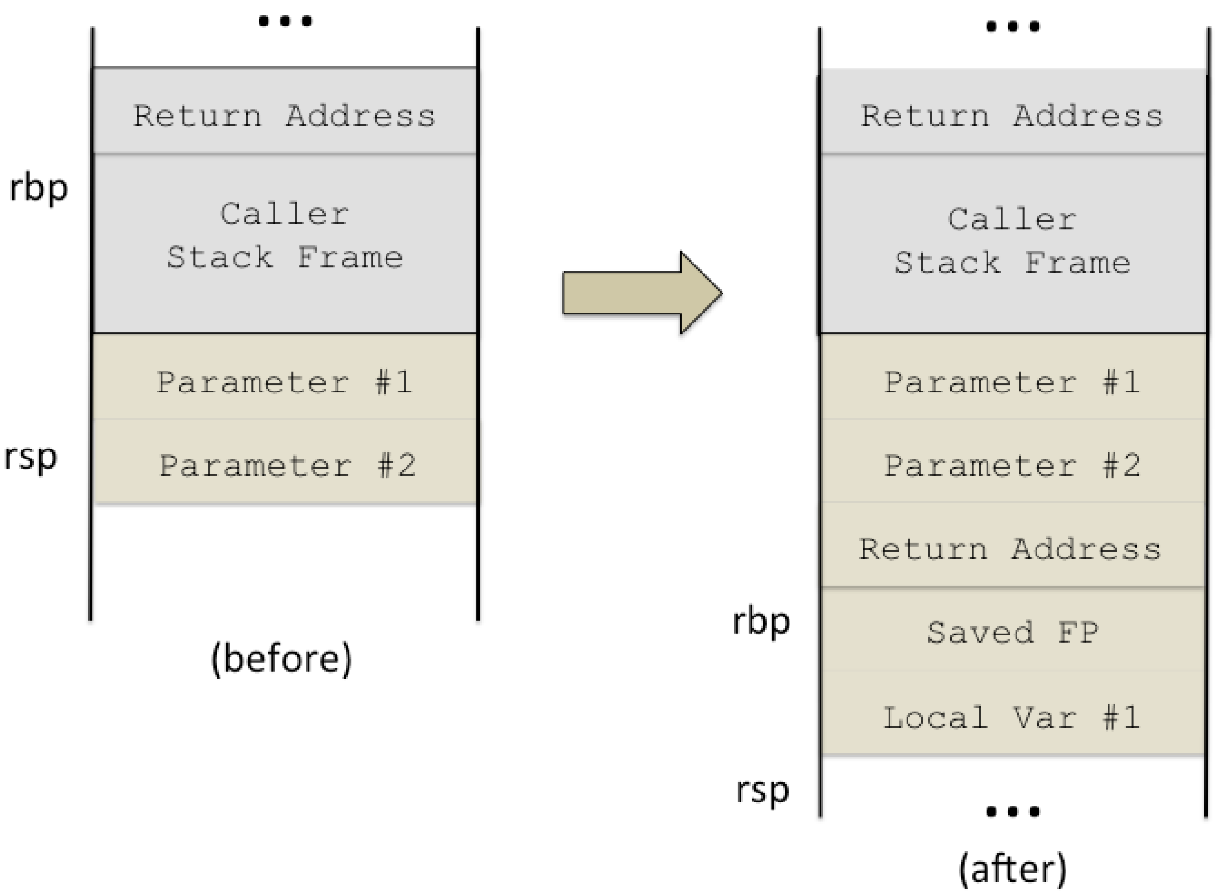
- Consider making a function call:

```
...
call somefn
...
```

- If callee **reads** / **writes** registers it may **overwrite** caller registers
- To prevent arbirtrary loss of data, a **calling convention** is used
- There are two main conventions: **caller-save** and **callee-save**
- For caller-save convention, **caller responsible** for saving registers in use
- For callee-save convention, **callee responsible** for saving registers it will use

Parameter Passing

- Must specify how to pass **parameters** from caller to callee



- Standard approach: **push parameters onto stack** before call

Return Value

- Must specify how to pass **return value** from callee to caller
- More **complicated** to handle than for parameters
- Caller **cannot** push return value on stack — last item must be return address
- Two main approaches:
  - Caller **reserves space** for return value before making call
  - Return value **passed-by-register** (e.g. `rax`, which would mean `rax` was caller-save)

CDecl Calling Convention

- The `cdecl` calling-convention commonly used for **C programs**
- Used for C because supports **variable length** arguments
- In this convention:
  - **Caller responsible** for cleaning parameters off stack
  - Registers `rax`, `rcx` and `rdx` are **caller saved**, all others **callee saved**
  - **Arguments** passed on the stack in right-to-left order
  - **Return value** passed back in `rax`

Omitting the Frame Pointer

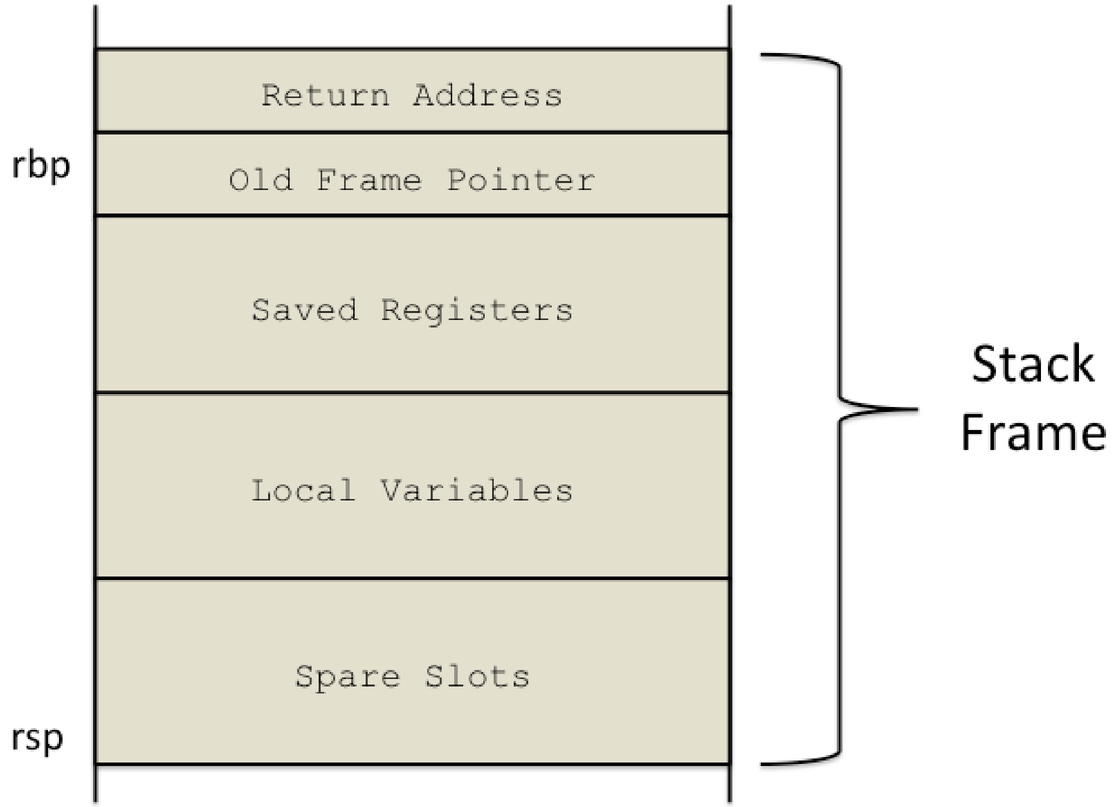
- One of gcc's optimisations is omitting the frame pointer when possible
- This can cause problems with debuggers
- But in the case of 32 bit machines, there are only 8 registers you can use and having a frame pointer loses one (and esp is already taken by the stack).
- Less of a problem with 64 bit machines as there are more registers.

Register Spilling

- Given **limited registers** on x86, we'll eventually run out of them!
- When this happens, can fall back to using **stack-based approach**
- More efficient to calculate number of **additional "registers" required**
- Then, **pre-allocate** stack space for additional "overflow" registers

Register Spilling (Continued)

- This is roughly what the **stack frame** will look like then:



- Note: not always possible to **predetermine slots**