SWEN430 - Compiler Engineering

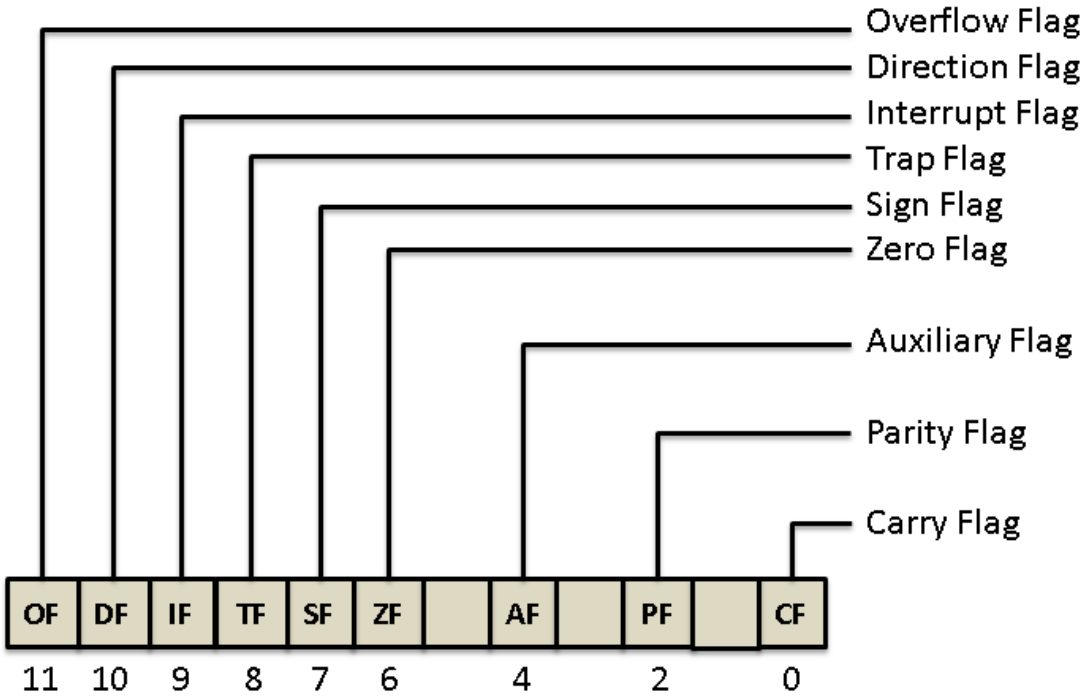
Lecture 16 - Machine Code II

Lindsay Groves and David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Flags Register

- The `EFLAGS` register holds “processor state”:



- Used (amongst other things) to implement **conditional branching**
- **Note:** there are more flags than shown here

Conditional Branching

- Conditional branch (equality) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jz target           /* branch if zero flag set */
```

- Conditional branch (less than or equal) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jle target          /* branch if sign or zero flags set */
```

- Conditional branch (not equals) implemented as follows:

```
cmpl %eax,%ebx       /* compare eax against ebx */
jnz target          /* branch if zero flag not set */
```

- Notes:
  - » **Zero Flag** set after comparison if items equal
  - » **Sign Flag** set after comparison if left operand less than right

Addressing Modes

`movl %eax, (%ebx)`	Assign `eax` register to dword at address `ebx`	`*ebx = eax`
`movl (%ebx),%eax`	Assign `eax` register from dword at address `ebx`	`eax = *ebx`
`movl 4(%esp),%eax`	Assign `eax` register from dword at address `esp+4`	`eax = *(esp+4)`
`movl (%esi,%eax),%cl`	Assign `cl` register from byte at address `esi+eax`	`cl = *(esi+eax)`
`movl %edx, (%esi,%ebx,4)`	Assign `edx` register to dword at address `esi+(4*ebx)`	`*(esi+(4*ebx)) = edx`

- 64bit x86-compatible processors can access 2^{64} **bytes** of memory
- Can read or write memory **indirectly** using address stored in register
- Corresponds to reading / writing through **pointers** in C

Understanding the Stack

`pushq %rax`	Push `rax` register onto stack
`pushq %c`	Push constant `c` onto stack
`popq %rdi`	pop qword off stack and assign to register `rdi`

- Stack provided for additional **temporary storage**:

```
movq $0xFF, %rax     /* store 255 in rax */
pushq %rax           /* push contents of rax on stack */
pushq $0xEE          /* push 238 directly on stack */
movq 8(%rsp),%rax    /* assign 255 to rax */
popq %rdx            /* pop 238 and assign to rdx */
```

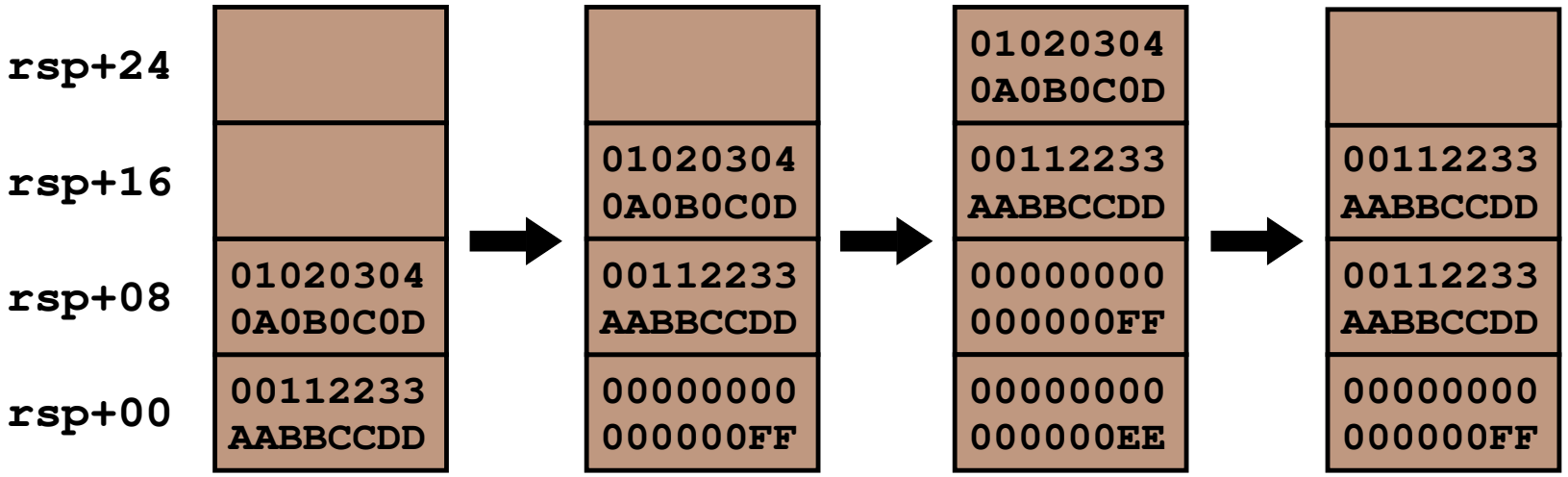
- Stack grows **downwards**!
- Stack used primarily for **local variables**, and **return address**

Visualising the Stack

- Consider **executing** these instructions:

```
movq $0xFF, %rax     /* store 255 in rax */
pushq %rax           /* push contents of rax on stack */
pushq $0xEE          /* push 238 directly on stack */
movq 8(%rsp),%rax    /* assign 255 to rax */
popq %rdx            /* pop 238 and assign to rdx */
```

- The effect on the stack can be **visualised** like so:



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 %rbx,%rax`
	`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

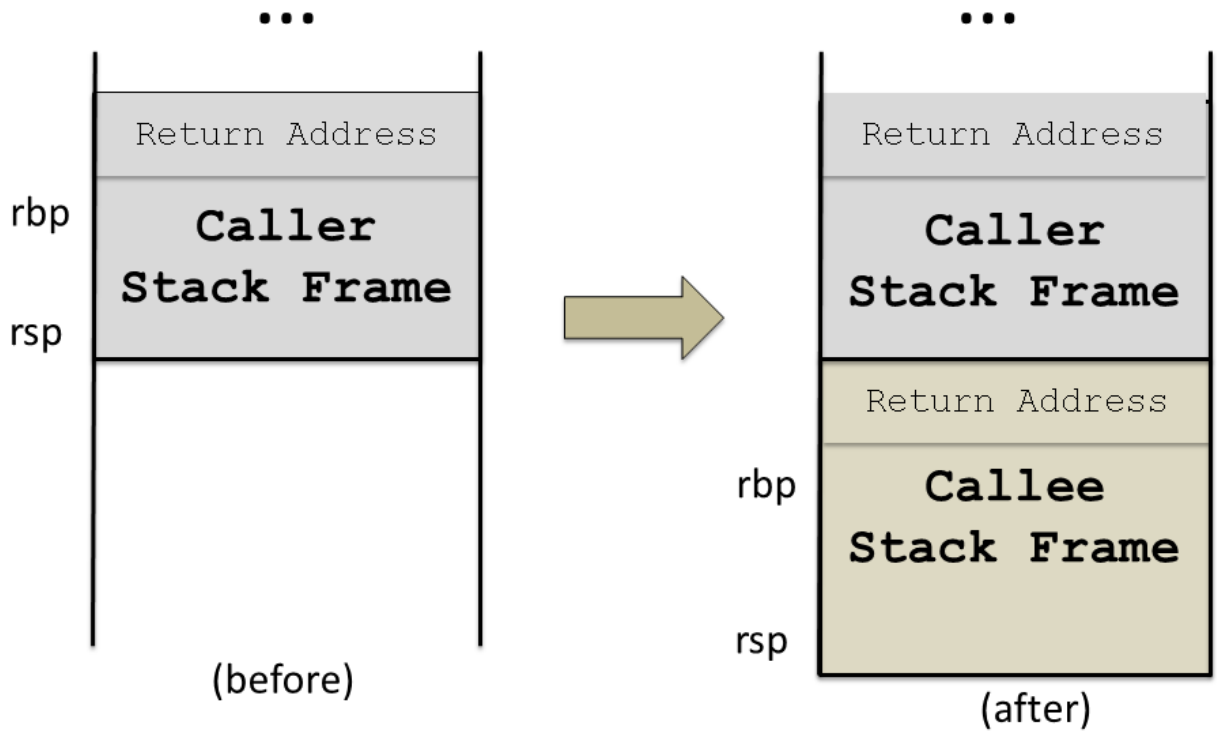
Translation — Register-Based Approach

```
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

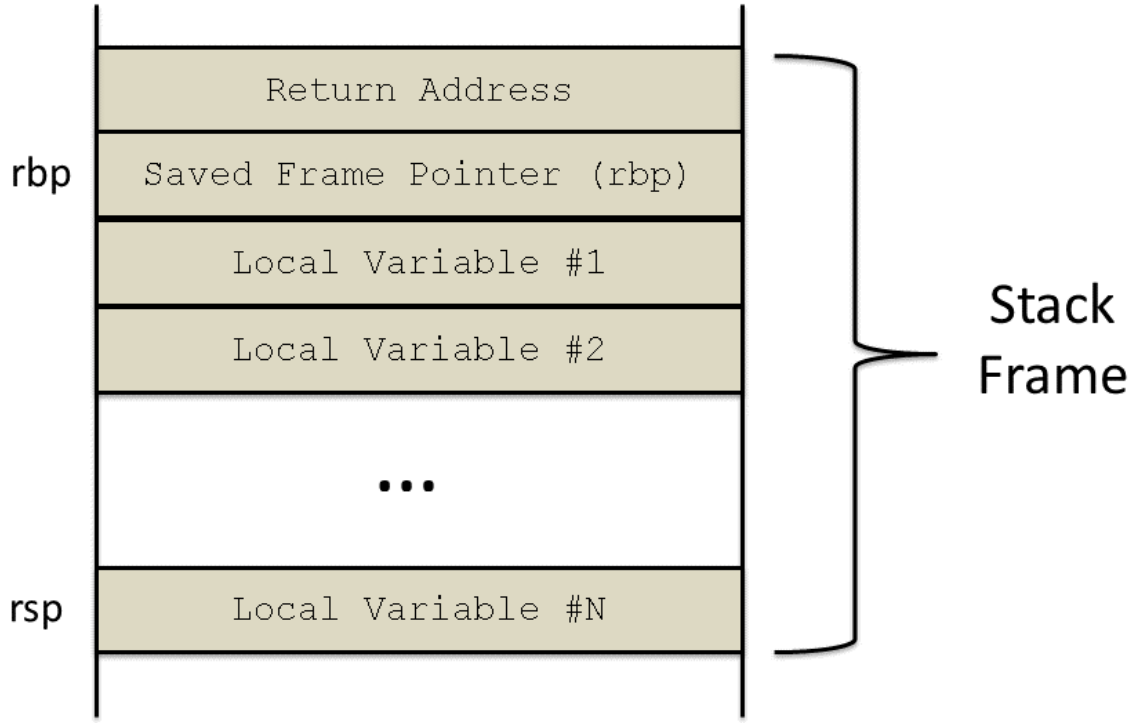
`call proc`	call procedure `proc` (`push %rip + 2` ; `jmp operand`)
`ret`	return from procedure (`pop %rip`)



- 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**:



- This contains, amongst other things, space for **local variables**

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 %rbp, %rsp` ; `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

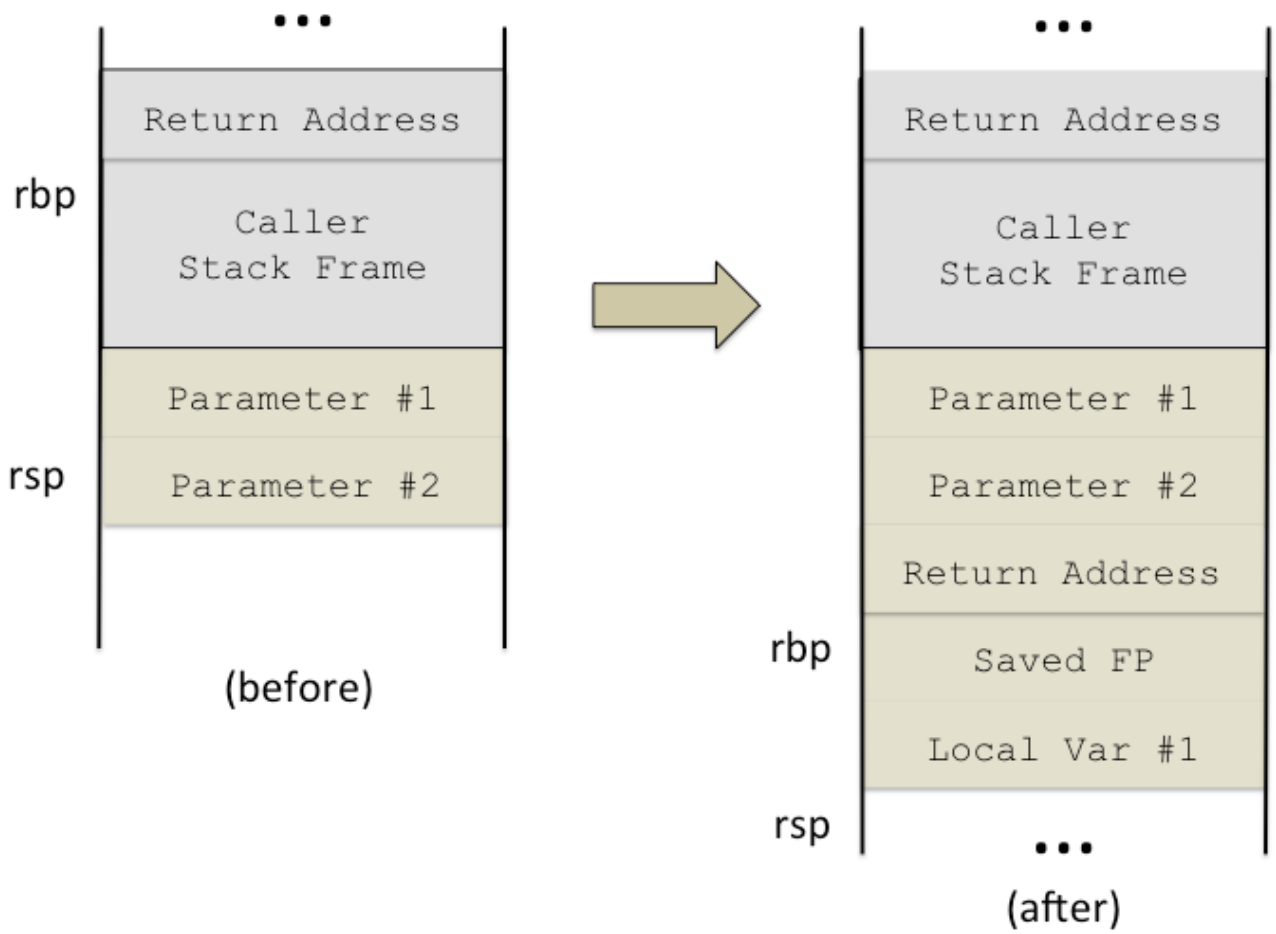
- Consider making a function call:

```
...
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)