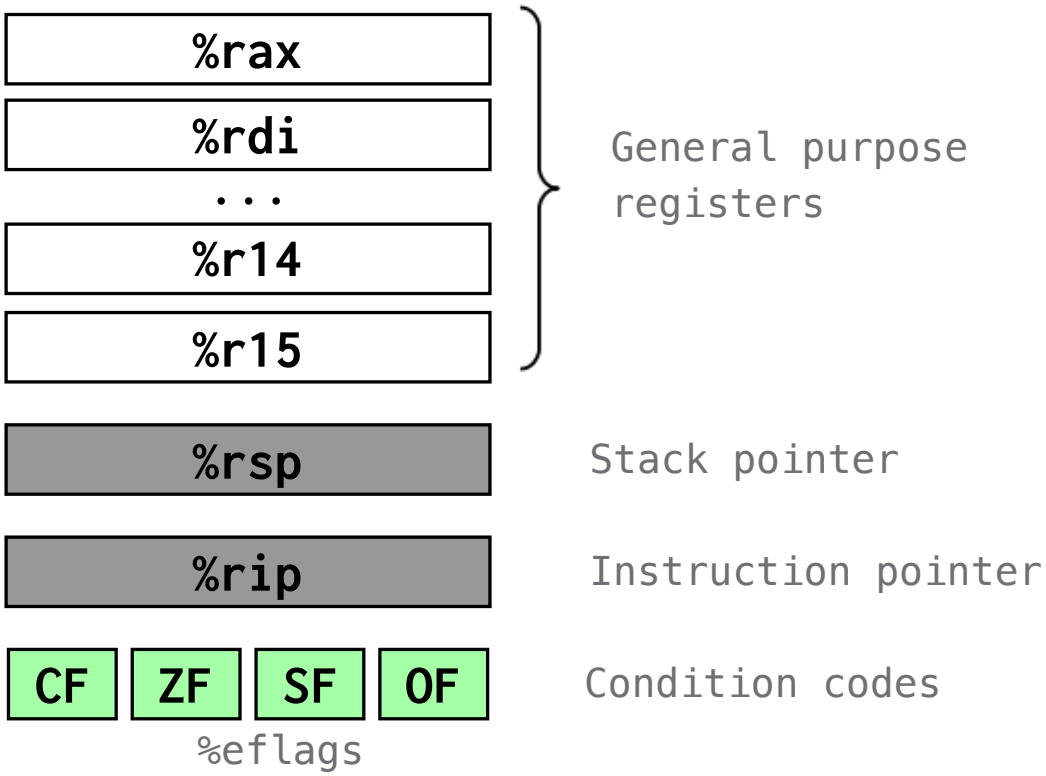
Processor state

- **Information about currently executing program**
  - **Temporary data**
    - `%rax, %rdi, …`
    - current parameters, local variables
  - **Location of runtime stack**
    - `%rsp`
  - **Location of current instruction**
    - `%rip`
  - **Status of recent operation**
    - `CF ZF SF OF`



Condition codes

Op			Comments
`cmp`	`op1,`	`op2`	Compute op2-op1, discard result, set condition codes
`test`	`op1,`	`op2`	Compute op2&op1, discard result, set condition codes
`sub`	`op1,`	`op2`	op 2 = op2-op1, set condition codes
`add`	`op1,`	`op2`	op2 = op2+op1, set condition codes

- `%eflags` register used as set of boolean values
  - ZF = zero flag SF = sign flag
  - CF = carry flag, unsigned overflow (out of MSB)
  - OF = overflow flag, signed overflow (into MSB)
- Codes explicitly set by cmp/test, implicitly set by many instructions
- Codes read by jx instructions (jump if condition x holds)

Example branch instructions

Op	Description	Condition
`jmp`	unconditional	
`je`	equal/zero	`ZF=1`
`jne`	not equal/not zero	`ZF=0`
`js`	negative (e.g. sign bit)	`SF=1`
`jl`	less (signed)	`SF!=OF`
`jle`	less or equal (signed)	`SF!=OF or ZF=1`
`jb`	below (unsigned)	`CF=1`
`...`		

- **Examples:**

  If previous instruction was cmp op1, op2, computed "result" is op2-op1

  je: Jump if ZF is 1

  result op2-op1 is zero means op1 is **equal** to op2

  jl: Jump if SF != OF

  result op2-op1 is negative means op2 is **less** than op1

  other case: if result ended up positive due to overflow, op2 is also less than op1

If-then(-else)

```
int if_then(int arg)
{
    if (arg != 6)
        arg++;
    arg *= 35;
    return arg + 7;
}
```

```
4004d6:  cmp    $0x6,%edi
4004d9:  je     4004de <if_then+0x8>
4004db:  add    $0x1,%edi
4004de:  imul   $0x23,%edi,%edi
4004e1:  lea    0x7(%rdi),%eax
4004e4:  retq
```

- **How if-then translated**
  - C code control flow reads as test for whether to enter
  - but assembly translation actually tests for whether to skip over
- **Consider:**
  - How does assembly change if test on line 1 is: `arg == 9`? `arg <= 6`?
  - What if put line 3 inside `else` clause?

Loops

```
int for_loop(int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += i;
    return sum;
}
```

```
400504:  mov    $0x0,%edx
400509:  mov    $0x0,%eax
40050e:  jmp    400515 <for_loop+0x11>
400510:  add    %edx,%eax
400512:  add    $0x1,%edx
400515:  cmp    %edi,%edx
400517:  jl     400510 <for_loop+0xc>
400519:  repz retq
```

- **How loop is translated**
  - **First iteration jumps over body to get to test— why rearrange in this way?**
    - One copy of test instructions
    - One branch per loop iteration instead of two
- **For/while/do-while largely same assembly translation**
  - **How to implement break/continue?**

Logical operations

```
int logic(int x)
{
   if ((x%2 == 0) && (x > 9))
      x++;
   return x;
}
```

```
40051d:  mov    %edi,%eax
40051f:  test   $0x1,%al
400521:  jne    40052b <logic+0xe>
400523:  cmp    $0x9,%edi
400526:  jle    40052b <logic+0xe>
400528:  add    $0x1,%eax
40052b:  repz retq
```

- **No instruction for logical AND/OR**
  - **Translated into test/cmp/mov, etc**
  - **Two branches in the assembly — why?**
    - Logical connectives required to "short-circuit"

Read condition codes

Op	Description
`sete`	set dst to equal/zero condition
`setne`	set dst to not equal/zero
`setle`	set dst to less/equal (signed)
`setb`	set dst to below (unsigned)
`...`	

Setx instructions

Set single byte dst to 0 or 1 based on whether condition holds

- Reads current state of flags
- Destination is single-byte sub-register (e.g. `%al` for low byte of `%rax`)
- Does not perturb the other bytes of register
- Typically followed by `movzbl` to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp    $0xf,%edi
setle  %al
movzbl %al,%eax
retq
```

Conditional move

Op	Description
cmovne	mov src to dst if not equal condition holds
cmovs	mov if signed
cmovg	mov if greater (signed)
cmova	mov if above (unsigned)
...	

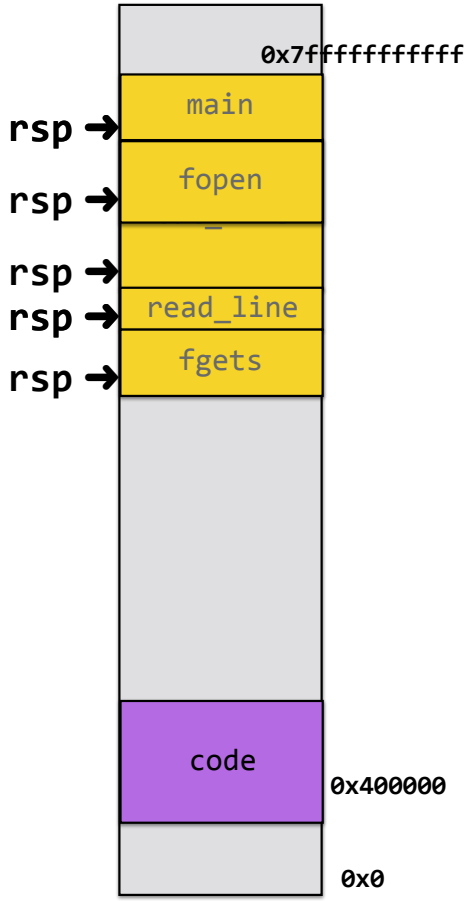
```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp     %edi,%esi
mov     %edi,%eax
cmovge  %esi,%eax
retq
```

- **cmovx src,dst (src, dst have to be register)**
  - mov src to dst if condition x holds, no change otherwise
  - "predicated" instruction, may be more efficient than branch
- **Seen in translation of C ternary: `result = test? then : else;`**
  - Both then/else are computed, set result to else
  - Overwrite result with then if test is true (or vice versa)
  - Not used when: then/else has side effects or too expensive to compute

Runtime stack

- **Languages that support recursion**
  - **Functions must be re-entrant**
    - Can have multiple simultaneous instantiations
  - **Each instantiation needs own distinct storage**
    - Arguments, return value
    - Local variables
    - Intermediate results, scratch
- **Stack frame per function instantiation**
  - **Currently executing function is topmost stack frame**
  - **Making new call pushes another frame**
  - **Return from call pops frame**
  - **LIFO — callee returns before caller does**
    - main calls fopen, then calls cat_file which calls read_line which calls fgets



Register use, conventions

Register	Designated purpose
`%rax`	return value
`%rdi`	1st argument
`%rsi`	2nd argument
`%rdx`	3rd argument
`%rcx`	4th argument
`%r8`	5th argument
`%r9`	6th argument
`%rsp`	stack pointer
`%rip`	instruction pointer

- **Conventions required for interoperability**
  - **"ABI" application binary interface**
  - **Mechanisms for call/return**
    - `call` transfers to callee, `ret` returns control to caller
    - resume address pushed on stack by call instruction
    - address popped and resumed by ret instruction
  - **Pass arguments, receive return value**
    - Designated registers
    - If more than 6 args, extras are stored on stack
    - If return value too big to fit in register, stored on stack
  - **Register use/ownership**
    - Registers divided into caller-owned or callee-owned
  - **Stack management**
    - Grows down, 16-byte alignment

Function examples

```
int dinky(int x)
{
    return x + 2;
}

int binky(int x, int y)
{
    int result = x * y;
    return result;
}

int oscar(void)
{
    int a = binky(5, 7);
    a = dinky(a);
    return a + 9;
}
```

```
000000000040056b <dinky>:
  40056b:  lea    0x2(%rdi),%eax
  40056e:  retq

000000000040056f <binky>:
  40056f:  mov    %edi,%eax
  400571:  imul   %esi,%eax
  400574:  retq

0000000000400575 <oscar>:
  400575:  mov    $0x7,%esi
  40057a:  mov    $0x5,%edi
  40057f:  callq  40056f <binky>
  400584:  mov    %eax,%edi
  400586:  callq  40056b <dinky>
  40058b:  add    $0x9,%eax
  40058e:  retq
```

Register ownership

- **ONE set of registers**
  - One %rax that is shared by all
  - Need a set of conventions to ensure functions don't trash other's data
  - Registers divided into callee-owner and caller-owner
- **Callee-owned**
  - Caller cedes these registers at time of call, cannot assume value will be preserved across call to callee
  - Callee has free reign over these, can overwrite with impunity
    - Callee-owner: registers for 6 arguments, return value, %r10, %r11
- **Caller-owner**
  - Caller retains ownership, expects value to be same after call as it was before call
  - Callee can "borrow" these from caller but must write down saved value and restore it before return
    - Caller-owner: all the rest (%rbx, %rbp, %r12-%r15)

Using stack for locals/scratch

Why copy?

Caller about to make a call, must cede callee-owned registers

If value in a callee-owned register that will be needed after the call, must make a copy before making the call

Callee needs to "borrow" caller-owned register

Must first copy value, then restore the value from saved copy before returning

Where to copy?

On stack, use push/pop instructions

`push src`

Decrement %rsp to make space, store src value at new top of stack

`pop dst`

Copy topmost value from stack into dst register; increment %rsp

More examples

`/afs/ir/class/cs107/samples/lect13/stack.c`