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

General purpose registers

Stack pointer

Instruction pointer

Condition codes

%eflags
### Condition codes

<table>
<thead>
<tr>
<th>Op</th>
<th>Op1, Op2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>op1,</td>
<td>Compute op2-op1, discard result, set condition codes</td>
</tr>
<tr>
<td>test</td>
<td>op1,</td>
<td>Compute op2&amp;op1, discard result, set condition codes</td>
</tr>
<tr>
<td></td>
<td>op2</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>op1,</td>
<td>op 2 = op2-op1, set condition codes</td>
</tr>
<tr>
<td>add</td>
<td>op1,</td>
<td>op2 = op2+op1, set condition codes</td>
</tr>
<tr>
<td></td>
<td>op2</td>
<td></td>
</tr>
</tbody>
</table>

%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**

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

**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 != 0F

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)

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

#### 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

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

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?
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

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>set dst to equal/zero condition</td>
</tr>
<tr>
<td>setne</td>
<td>set dst to not equal/zero</td>
</tr>
<tr>
<td>setle</td>
<td>set dst to less/equal (signed)</td>
</tr>
<tr>
<td>setb</td>
<td>set dst to below (unsigned)</td>
</tr>
</tbody>
</table>

**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

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

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

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmovne</td>
<td>mov src to dst if not equal condition holds</td>
</tr>
<tr>
<td>cmovs</td>
<td>mov if signed</td>
</tr>
<tr>
<td>cmovg</td>
<td>mov if greater (signed)</td>
</tr>
<tr>
<td>cmoava</td>
<td>mov if above (unsigned)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- **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:** \texttt{result} = \texttt{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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Designated purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
<tr>
<td>%rdi</td>
<td>1st argument</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument</td>
</tr>
<tr>
<td>%rcx</td>
<td>4th argument</td>
</tr>
<tr>
<td>%r8</td>
<td>5th argument</td>
</tr>
<tr>
<td>%r9</td>
<td>6th argument</td>
</tr>
<tr>
<td>%rsp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%rip</td>
<td>instruction pointer</td>
</tr>
</tbody>
</table>

**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
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;
}
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