Get To Know Your Compiler

Inspired by Matt Godbolt

Compilation Stages

1. Preprocessor

   Run preprocessor directives **–E** for clang

2. Compiler

   Compile the preprocessed source code

   - **-S** Generate Assembly
   - **-c** Generate Object Code

3. Linker

   Generate executable from object files

Does it matter?

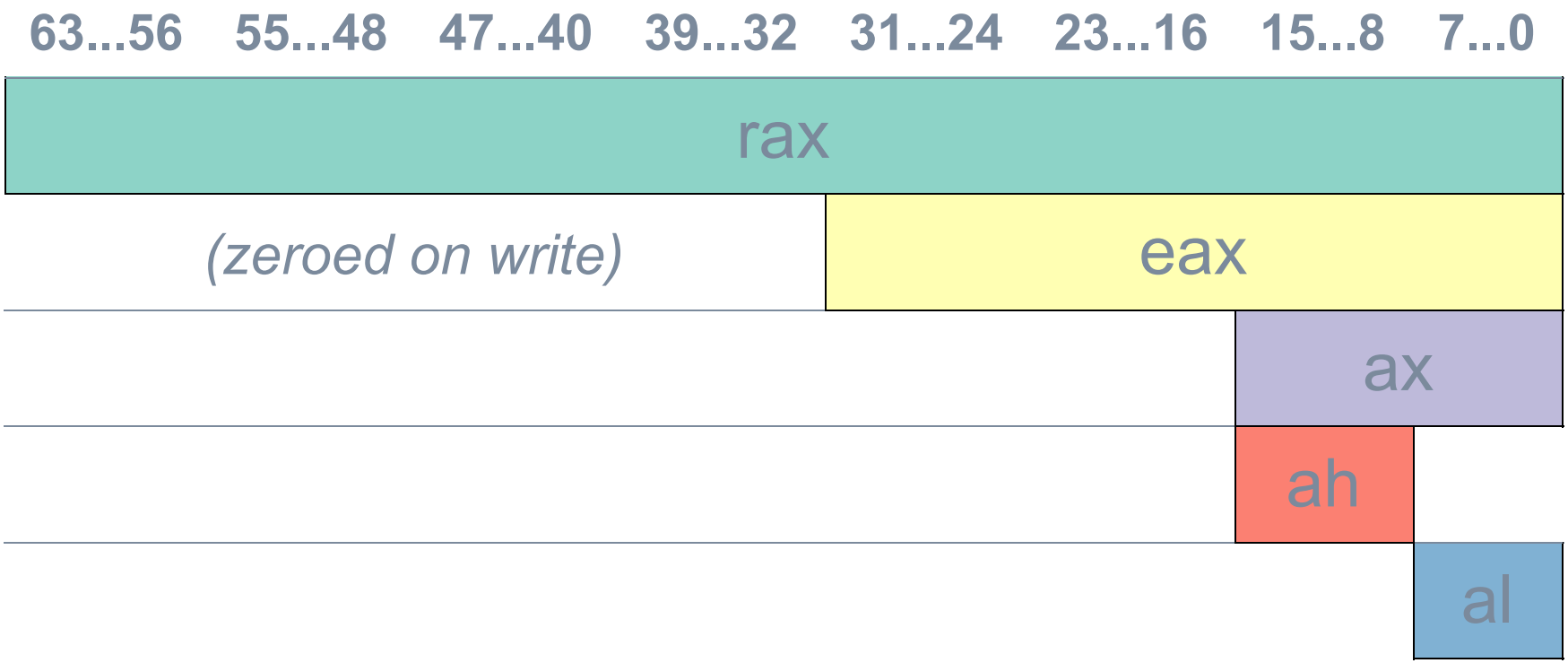
```
int sum(const vector<int> &v) {
  int result = 0;
  for (size_t i = 0; i < v.size(); ++i)
    result += v[i];
  return result;
}
```

```
int sum(const vector<int> &v) {
  int result = 0;
  for (int x : v) result += x;
  return result;
}
```

X86 Assembly Registers

- rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8-r15
- xmm0-xmm15
- rdi, rsi, rdx … arguments
- rax is the return value

Register Details



Instructions

```
op
op dest
op dest, src
op dest, src1, src2
```

- op is call, ret, add, sub, cmp…
- dest, src is register or memory reference
  [base + *reg1* + *reg2 (1, 2, 4, or 8)*]

Instruction Examples

```
mov eax, DWORD PTR [r14]
add rax, rdi
add eax, DWORD PTR [r14+4]
sub eax, DWORD PTR [r14+4*rbx]
lea rax, [r14+4*rbx]
xor edx, edx
```

```
int eax = *r14;   // int *r14;
rax += rdi;
eax += r14[1];
eax -= r14[rbx];
int *rax = &r14[rbx];
edx = 0;
```

Summary

- Register: rax, rbx, rcx …
- Size: rax, eax, ax ..
- Params: rdi, rsi, rdx, rcx …
- Result: rax
- Format: `op dest, src`
- `dest` and `src` are registers or memory

So Does it matter?

```
int sum(const vector<int> &v) {
  int result = 0;
  for (size_t i = 0; i < v.size(); ++i)
    result += v[i];
  return result;
}
```

```
int sum(const vector<int> &v) {
  int result = 0;
  for (int x : v) result += x;
  return result;
}
```

Compiler Explorer

- Godbolt.org

What is going on?

```
; rdi = const vector<int> *
mov rdx, QWORD PTR [rdi]   ; rdx = *rdi      ≡ begin()
mov rcx, QWORD PTR [rdi+8] ; rcx = *(rdi+8) ≡ end()
```

```
template<typename T> struct _Vector_impl {
  T *_M_start;
  T *_M_finish;
  T *_M_end_of_storage;
};
```

TRADITIONAL

```
sub rcx, rdx ; rcx = end-begin
mov rax, rcx
shr rax, 2   ; (end-begin)/4
je .L4
add rcx, rdx
xor eax, eax
```

```
size_t size() const noexcept {
  return _M_finish - _M_start;
}
```

RANGE

```
xor eax, eax
cmp rdx, rcx ; begin==end?
je .L4
```

```
auto __begin = begin(v);
auto __end = end(v);
for (auto __it = __begin;
     __it != __end;
     ++it)
```

```
; rcx ≡ end, rdx = begin, eax = 0
.L3:
  add eax, DWORD PTR [rdx]     ; eax += *rdx
  add rdx, 4                   ; rdx += sizeof(int)
  cmp rdx, rcx                 ; is rdx == end?
  jne .L3                      ;   if not, loop
  ret                          ; we're done
```

Multiplication

```
int mulByY(int x, int y) {
  return x * y;
}
```

```
mulByY(int, int):
  mov eax, edi
  imul eax, esi
  ret
```

```
int mulByConstant(int x) { return x * 2; }
```

Fancy Multiplication

```
int mulBy65599(int a) {
  return (a << 16) + (a << 6) - a;
  //          ^           ^
  //      a * 65536       |
  //                    a * 64
  // 65536a + 64a - 1a = 65599a
}
```

Division

```
int divByY(int x, int y) {
  return x / y;
}
int modByY(int x, int y) {
  return x % y;
}
```

```
divByY(int, int):
  mov eax, edi
  cdq
  idiv esi
  ret
modByY(int, int):
  mov eax, edi
  cdq
  idiv esi
  mov eax, edx
  ret
```

View

Haswell 32-bit divide - 22-29 cycles!

Summation

```
int sumTo(int x) {
     int sum = 0;
     for( int i = 0; i <= x; ++i ) {
          sum += i;
     }
     return sum;
}
```

What is the runtime?

A) O(1) B) O(x) C) O(x^2) D) What on earth is big O?

Sum(X) from CS 173

$$\sum x \equiv \frac{x(x+1)}{2} \equiv x + \frac{x(x-1)}{2}$$

The compiler is your friend