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
```

Instructions for runtime stack

Add/remove values to stack

`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`

Call and return

`callq <fn>`

Transfer control to named function

push %rip onto stack (this becomes resume address), set %rip to fn address

`retq`

pop %rip (resume address should be topmost on stack)

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-owned and caller-owned
- **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-owned: registers for 6 arguments, return value, %r10, %r11
- **Caller-owned**
  - 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 returning to caller
    - Caller-owned: all the rest (%rbx, %rbp, %r12-%r15)

Using stack for locals/scratch

- **Why copy registers?**
  - **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, later restore the value from saved copy before returning
- **Where to copy registers?**
  - **`push` to save value to stack, `pop` to restore**
- **Local variables**
  - **Stored in registers whenever possible**
    - What if too many?
      - Compiler can re-use register when live ranges don't overlap
    - Spill to stack (push/pop) as needed
  - **What can't be stored in register?**
    - Variable too large (struct, array)
    - `&var` used requires that `var` be stored in memory

Wrap up on stack

- **Oddball cases**
  - If more than 6 arguments, extras passed on stack
  - If parameter or return value does not fit in 64-bit register (struct?), written to stack
- **Understanding stack means you know…**
  - How recursion is implemented
  - Why local variables allocated on stack are cheap
  - Why initial contents of locals is garbage, how can change with context of call
  - Consequence of function returning address into deallocated stack frame
- **Stack vulnerability**
  - Resume address is stored in stack frame
  - If access to neighbor overruns (such as access off end of array), what is consequence?
  - If resume address is trashed, what happens?

Compiler code generation

Constraints

Execution must be faithful to language semantics

Order of operations, precedence, etc.

Must obey conventions for interoperability

Function call/return, use of registers

Instructions must be legal, meet ISA contract

i.e. lea scale must be 1,2, 4, 8

Latitude

Can re-order operations that don't have dependencies

Can substitute equivalent sequence

```
mov $0, %eax
xor %eax, %eax
and $0, %eax
```

C spec liberates compiler in terms of undefined behavior

Uninitialized variable, missing return, integer overflow, dereference NULL,…

Compiler people LOVE optimization

```
-O0   // faithful/literal match to C, best for debugging
-Og   // streamlined, but debug-friendly
-O2   // apply all acceptable optimizations
```

- **Compiler knows the score when it comes to the hardware**
  - Register allocation
  - Instruction choice
  - Alignment
- **Transformations should be legal, equivalent**
  - Compiler has only knowledge of CT, not RT
  - Operates conservatively
  - "Do no harm"

Constant folding

```
unsigned int CF(unsigned int val)
{
    unsigned int ones = ~0U/UCHAR_MAX;
    unsigned int highs = ones << (CHAR_BIT - 1);
    return (val - ones) & highs;
}
```

```
0000000000400836 <CF>:
  push   %rbp
  mov    %rsp,%rbp
  mov    %rdi,-0x18(%rbp)
  movq   $0x1010101,-0x10(%rbp)
  mov    -0x10(%rbp),%rax
  shl    $0x7,%rax
  mov    %rax,-0x8(%rbp)
  mov    -0x18(%rbp),%rax
  sub    -0x10(%rbp),%rax
  and    -0x8(%rbp),%rax
  pop    %rbp
  retq
```

```
0000000000400810 <CF>:
  lea    -0x1010101(%rdi),%rax
  and    $0x80808080,%eax
  retq
```

How does knowing this influence how you write the code in the first place?

Common subexpression elimination

```
int CSE(int num, int val)
{
    int a = (val + 50);
    int b = num*a - (50 + val);
    return (val + (100/2)) + b;
}
```

```
0000000000400860 <CSE>:
  push   %rbp
  mov    %rsp,%rbp
  mov    %edi,-0x14(%rbp)
  mov    %esi,-0x18(%rbp)
  mov    -0x18(%rbp),%eax
  add    $0x32,%eax
  mov    %eax,-0x8(%rbp)
  mov    -0x14(%rbp),%eax
  imul   -0x8(%rbp),%eax
  mov    -0x18(%rbp),%edx
  add    $0x32,%edx
  sub    %edx,%eax
  mov    %eax,-0x4(%rbp)
  mov    -0x18(%rbp),%eax
  lea    0x32(%rax),%edx
  mov    -0x4(%rbp),%eax
```

```
0000000000400820 <CSE>:
  400820:  lea    0x32(%rsi),%eax
  400823:  imul   %edi,%eax
  400826:  retq
```

Also can apply to repeated address calculations!

Strength reduction

```
int SR(int val)
{
    unsigned int b = 5*val;
    int c = b / (1 << val);
    return (b + c) % 2;
}
```

```
0000000000400892 <SR>:
  push   %rbp
  mov    %rsp,%rbp
  mov    %edi,-0x14(%rbp)
  mov    -0x14(%rbp),%edx
  mov    %edx,%eax
  shl    $0x2,%eax
  add    %edx,%eax
  mov    %eax,-0x8(%rbp)
  mov    -0x14(%rbp),%eax
  mov    $0x1,%edx
  mov    %eax,%ecx
  mov    %edx,%esi
  shl    %cl,%esi
  mov    -0x8(%rbp),%eax
  cltd
  idiv   %esi
  mov    %eax,-0x4(%rbp)
  mov    -0x8(%rbp),%edx
  mov    -0x4(%rbp),%eax
```

```
0000000000400830 <SR>:
  lea    (%rdi,%rdi,4),%eax
  mov    %edi,%ecx
  mov    %eax,%edx
  shr    %cl,%edx
  add    %edx,%eax
  and    $0x1,%eax
  retq
```

Cost-per-instruction varies, not all created equal

Code motion

```
int CM(int val)
{
    int sum = 0;
    do {
       sum += 6 + 14*val;
    } while (sum < (9/val));
    return sum;
}
```

```
00000000004008c2 <CM>:
    push   %rbp
    mov    %rsp,%rbp
    mov    %edi,-0x14(%rbp)
    movl   $0x0,-0x4(%rbp)
    mov    -0x14(%rbp),%eax
    add    %eax,%eax
    lea    0x0(,%rax,8),%edx
    sub    %eax,%edx
    mov    %edx,%eax
    add    $0x6,%eax
    add    %eax,-0x4(%rbp)
    mov    $0x9,%eax
    cltd
    idivl  -0x14(%rbp)
    cmp    -0x4(%rbp),%eax
    jg     4008d0 <CM+0xe>
    mov    -0x4(%rbp),%eax
    pop    %rbp
```

```
0000000000400840 <CM>:
    mov    $0x9,%eax
    xor    %ecx,%ecx
    cltd
    idiv   %edi
    imul   $0xe,%edi,%esi
    add    $0x6,%esi
    add    %esi,%ecx
    cmp    %eax,%ecx
    jl     400850 <CM+0x10>
    mov    %ecx,%eax
    retq
```

Why is beneficial to move work outside loop body?

Dead code elimination

```
int DC(int a, int b)
{
    if (a < b && a > b) // can never be true!
        printf("The end of the world is near!");

    int result;
    for (int i = 0; i < 9999; i++)
        result *= i;

    if (a == b)
        a++; // if/else obviously same
    else
        a++;

    if (a == 0)
        return 0; // if/else same, not so obvious
    else
        return a;
}
```

```
00000000004008f9 <DC>:
    push   %rbp
    mov    %rsp,%rbp
    sub    $0x20,%rsp
    mov    %edi,-0x14(%rbp)
    mov    %esi,-0x18(%rbp)
    mov    -0x14(%rbp),%eax
    cmp    -0x18(%rbp),%eax
    jge    400921 <DC+0x28>
    mov    -0x14(%rbp),%eax
    cmp    -0x18(%rbp),%eax
    jle    400921 <DC+0x28>
    mov    $0x400e0c,%edi
    callq  400680 <puts@plt>
    movl   $0x0,-0x4(%rbp)
    jmp    400953 <DC+0x5a>
    mov    -0x8(%rbp),%eax
    imul   -0x4(%rbp),%eax
    mov    %eax,-0x8(%rbp)
    addl   $0x1,-0x4(%rbp)
    cmpl   $0x270e,-0x4(%rbp)
    jle    40092a <DC+0x31>
    mov    -0x14(%rbp),%eax
    cmp    -0x18(%rbp),%eax
    jne    40094f <DC+0x56>
    addl   $0x1,-0x14(%rbp)
    jmp    400953 <DC+0x5a>
    addl   $0x1,-0x14(%rbp)
```

```
0000000000400860 <DC>:
  lea    0x1(%rdi),%eax
  retq
```

Function inlining

```
0000000000400692 <main>:
    push   %rbp
    mov    %rsp,%rbp
    sub    $0x20,%rsp
    mov    %edi,-0x14(%rbp)
    mov    %rsi,-0x20(%rbp)
    mov    $0x0,%eax
    callq  400450 <rand@plt>
    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax
    mov    %eax,%edi
    callq  400566 <CF>
    mov    %eax,%edx
    mov    -0x4(%rbp),%eax
    add    %edx,%eax
    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax
    mov    $0x6b,%esi
    mov    %eax,%edi
    callq  400588 <CSE>
    add    %eax,-0x4(%rbp)
    mov    $0x6b,%edi
    callq  4005ba <SR>
    add    %eax,-0x4(%rbp)
    mov    $0x6b,%edi
```

```
int main(int argc, char *argv[])
{
    int x = rand();
    x += CF(x);
    x += CSE(x, 107);
    x += SR(107);
    x += CM(107);
    x += DC(x, 107);
    return x;
}
```

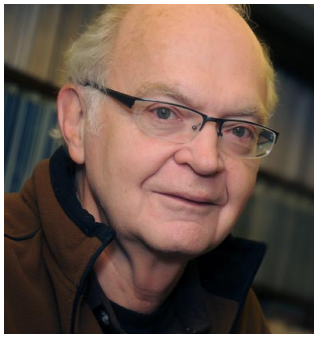
```
0000000000400430 <main>:
    sub    $0x8,%rsp
    xor    %eax,%eax
    callq  400410 <rand@plt>
    lea    -0x1010101(%rax),%edx
    add    $0x8,%rsp
    and    $0x80808080,%edx
    add    %edx,%eax
    imul   $0x9e,%eax,%eax
    lea    0xbc3(%rax,%rax,1),%eax
    retq
```

Decomposition is good! Have your cake and eat it, too!

Rules of thumb

- **Is there even a problem?**
  - Measure! If ok at expected scale, you're done!
- **KISS (keep it simple stupid)**
  - If low-traffic/small input: simplest code, easy to understand and debug
  - (optimize use of programmer's time!)
- **Choose correct algorithm/design**
  - Optimization reduces constants, doesn't change Big-O or fix bad design
- **Let gcc do its magic!**
  - No pre-optimize, don't get in compiler's way
  - Read generated assembly to know what you are getting
- **Only then take action of your own**
  - Measure again, attend only to actual bottleneck

Optimization reality check



“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ”

— Donald Knuth

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”

— W.A. Wulf

“Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.”

— Rob Pike