Assembly Design Recipe

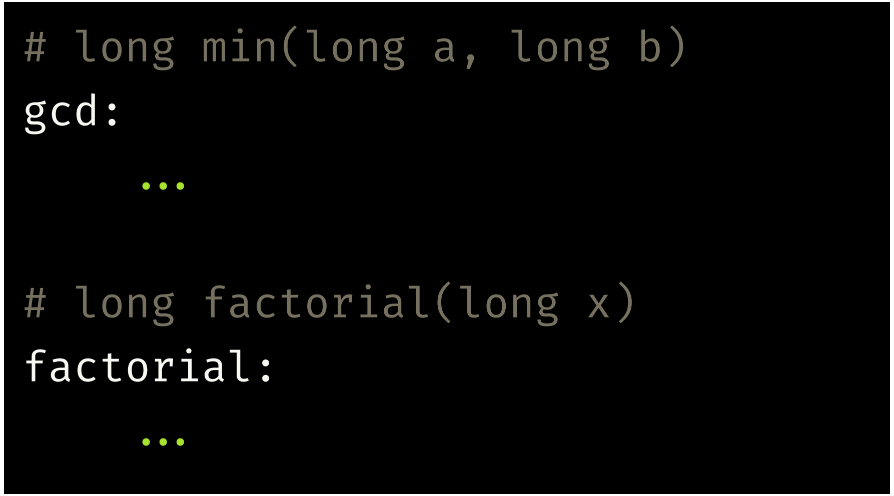
A “Design Recipe for Assembly”

1. Signature (C-ish)
2. Pseudocode (ditto)
3. Variable mappings (registers, stack offsets)
4. Skeleton
5. Fill in the blanks

(Originally by Nat Tuck)

1. Signature

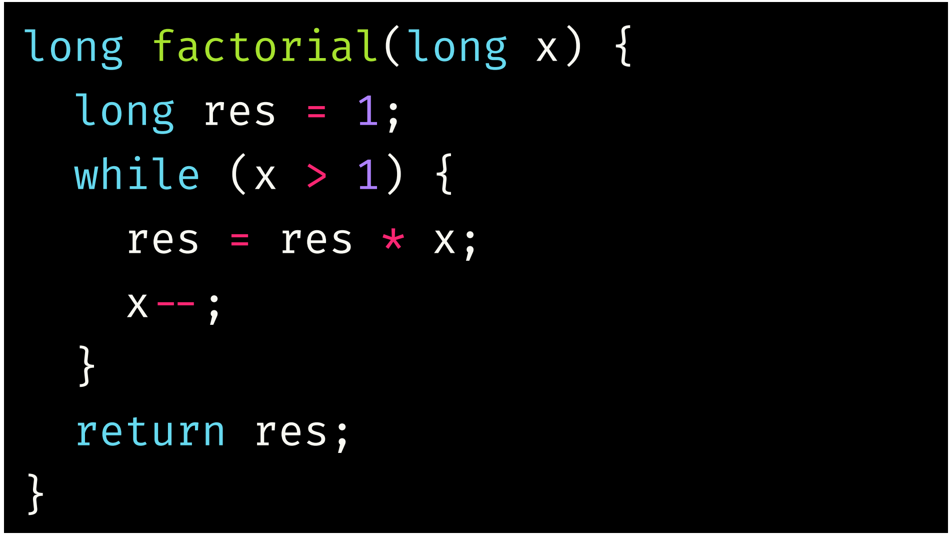
- What are our arguments?
- What will we return?



```
# long min(long a, long b)
gcd:
    ...

# long factorial(long x)
factorial:
    ...
```

2. Pseudocode

- How do we compute the function?
- Thinking in directly in assembly is *hard*
- Translating pseudocode, on the other hand, is quite straightforward
- C works pretty well



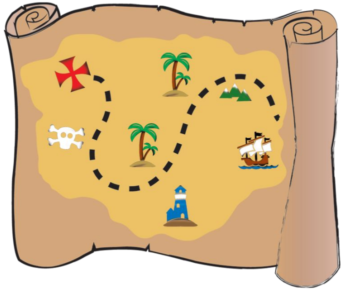
```c
long factorial(long x) {
  long res = 1;
  while (x > 1) {
    res = res * x;
    x--;
  }
  return res;
}
```

2. Pseudocode

- How do we compute the function?
- Thinking in directly in assembly is *hard*
- Translating pseudocode, on the other hand, is quite straightforward
- C works pretty well

```
# long factorial(long x)
factorial:
    # long res = 1;
    # while (x > 1) {
    #   res = res * x;
    #   x--;
    # }
    # return res;
```

3. Variable Mappings



- Need to decide where we store temporary values
- Arguments are given: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, then the stack
- Do we keep variables in registers?
    - Callee-save? `%r12`, `%r13`, `%r14`, `%r15`, `%rbx`
    - Caller-save? `%r10`, `%r11` + argument registers
- Do we use the stack?

```
# long factorial(long x)
factorial:
    # x → %r12
    # res → %rax
```

4. Function Skeleton

```
label:
    # Prologue:
    #   Set up stack frame.
    # Body:
    #   Just say "TODO"
    # Epilogue:
    #   Clean up stack frame.
```

Prologue:

- `push` callee-saves
- `enter` - allocate stack space
  - stack alignment!

Epilogue:

- `leave` - deallocate stack space
- Restore (`pop`) any pushed registers
- `ret` - return to call site

4. Function Skeleton

```
min:
    # Prologue:
    push %r12     # Save callee-save regs.
    push %r13
    enter $16, $0  # Allocate / align stack
    # Body:
                  # Just say "TODO"
    # Epilogue:
    leave         # Clean up stack frame.
    pop %r13      # Restore saved regs.
    pop %r12
    ret           # Return to call site
```

5. Complete the Body

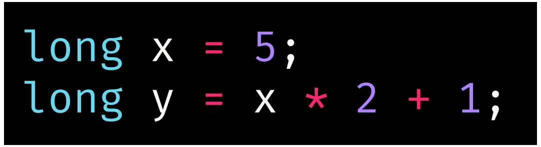
- Translate your pseudocode into assembly - line by line
- Apply variable mappings

Translating Pseudocode

- Relatively straightforward
- Each line of C corresponds to one or a few instructions
- When you get stuck, use https://godbolt.org/ for inspiration

Variables, Temporaries, Assignment

- Each C variable maps to a register or a stack location (by using `enter`)
- Temporary results go into registers
- Registers can be shared / reused - keep track carefully

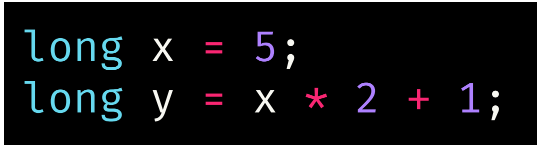


```
long x = 5;
long y = x * 2 + 1;
```

With:
  x in %r10
  y in %r11
  Temporary for x * 2 is %rdx

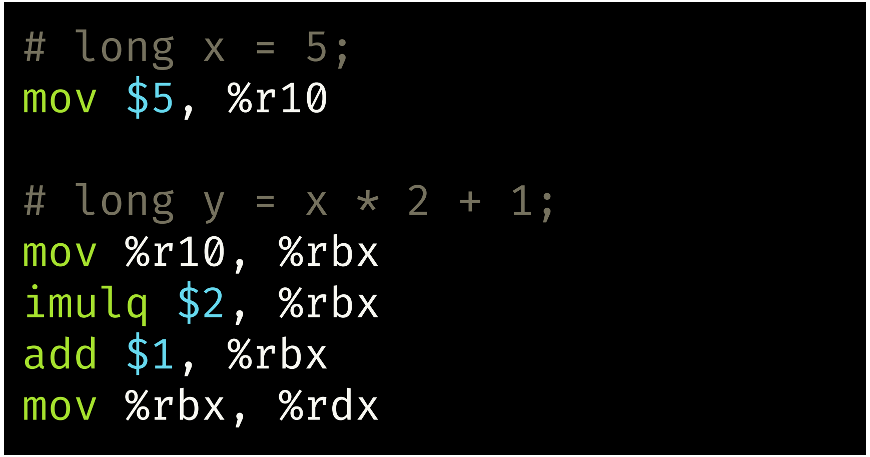
Variables, Temporaries, Assignment

- Each C variable maps to a register or a stack location (by using `enter`)
- Temporary results go into registers
- Registers can be shared / reused - keep track carefully



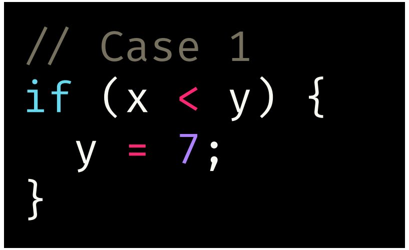
```c
long x = 5;
long y = x * 2 + 1;
```

With:
x in %r10
y in %rbx
Temporary for x * 2 is %rdx



```
# long x = 5;
mov $5, %r10

# long y = x * 2 + 1;
mov %r10, %rbx
imulq $2, %rbx
add $1, %rbx
mov %rbx, %rdx
```

If statements 1



```
// Case 1
if (x < y) {
  y = 7;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

If statements 1

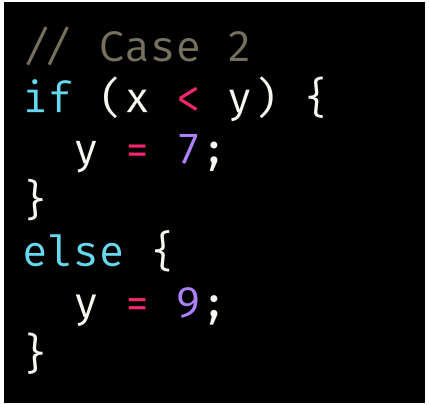
```c
// Case 1
if (x < y) {
  y = 7;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

```
# if (x < y)
  # cmp can only take one indirect arg
  mov -16(%rbp), %r10
  cmp %r10, -8(%rbp)   # cmp order backwards from C
  # condition reversed, skip block _unless_ cond
  # jge → if (-8(%rbp) ≥ %r10) jump to else1
  jge else1:

  # y = 7
  movq $7, -16(%rbp)   # need suffix to set size of "7"

else1:
  ...
```

If statements 2



```
// Case 2
if (x < y) {
  y = 7;
}
else {
  y = 9;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

If statements 2

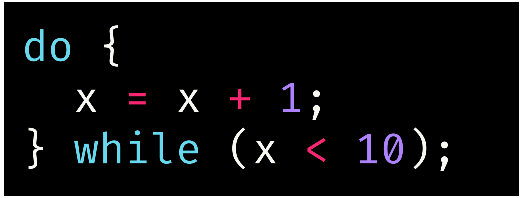
```
// Case 2
if (x < y) {
   y = 7;
}
else {
   y = 9;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

```
# if (x < y)
  mov -16(%rbp), %r10
  cmp %r10, -8(%rbp)
  jge else1:
  # then {
  # y = 7
  movq $7, -16(%rbp) # need suffix to set size of "7"

  jmp done1           # skip else

  # } else {
else1:
  # y = 9
  movq $9, -16(%rbp)

  # }
done1:
   ...
```

Do-while loops



```
do {
  x = x + 1;
} while (x < 10);
```

Variables:

- x is `-8(%rbp)`

Do-while loops

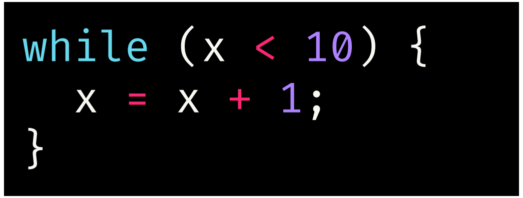
```
do {
  x = x + 1;
} while (x < 10);
```

```
loop:
  add $1, -8(%rbp)

  cmp $10, -8(%rbp) # reversed for cmp arg order
  jl loop          # sense not reversed

  # ...
```

Variables:

- x is `-8(%rbp)`

While loops



```
while (x < 10) {
  x = x + 1;
}
```

Variables:

- x is `-8(%rbp)`

While loops

```
while (x < 10) {
  x = x + 1;
}
```

```
loop_test:
  cmp $10, -8(%rbp) # reversed for cmp
  jge loop_done      # jump out if greater than

  add $1, -8(%rbp)
  jmp loop_test

loop_done:
  ...
```

Variables:

- x is `-8(%rbp)`