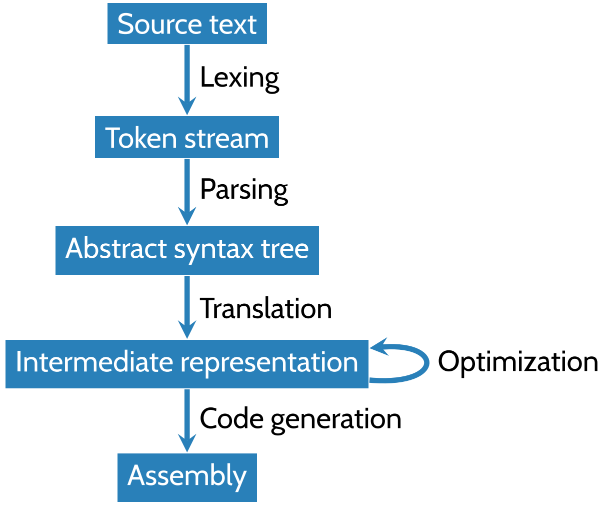
COS320: Compiling Techniques

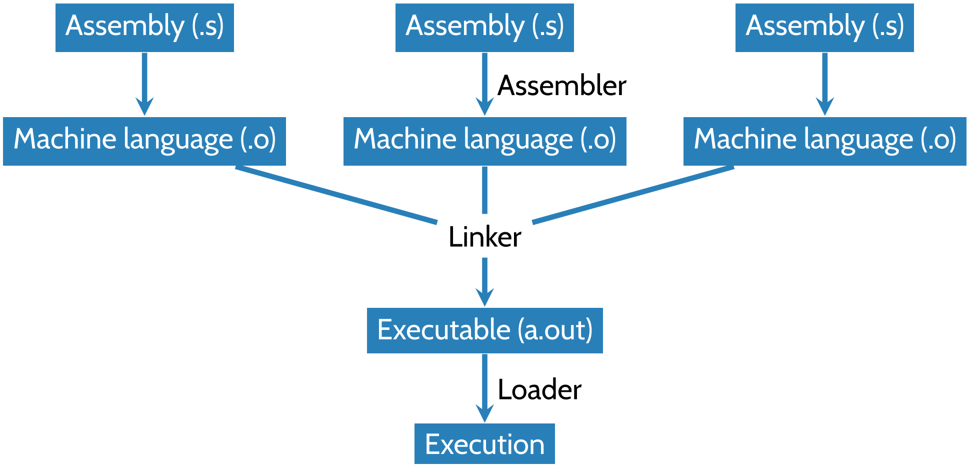
Zak Kincaid

January 29, 2026

Compiler phases (simplified)



HW1: assembly language & below



- Assember (`as`): translate assembly to object file (.o)
  - Re-arrange assembly into text and data segments
  - Encode instructions
  - Resolve symbolic references to (absolute or relative) memory addresses
  - Construct *relocation table* and *symbol table*
- Linker (`ld`): combine object files into an executable
  - Concatenate data and text sections
  - (Partial) *symbol resolution*: replace symbolic references with addresses
  - *Relocation*: fix references to relocated addresses
- Loader (`exec` family): load executable into memory and transfer control
  - *Dynamic* linking

Today: x86Lite

X86

- X86 is *very* complicated
    - 8-, 16-, 32-, 64-bit values, floats, …
    - Hundreds or thousands of instructions (depending on how they're counted)
    - Variable-length encoding for instructions (1-17 bytes)

X86

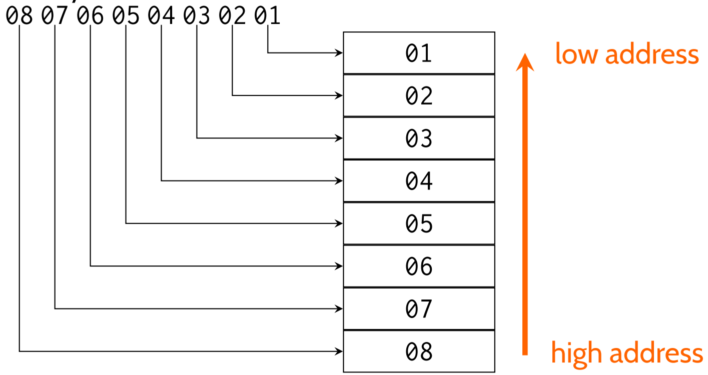
- X86 is *very* complicated
  - 8-, 16-, 32-, 64-bit values, floats, …
  - Hundreds or thousands of instructions (depending on how they're counted)
  - Variable-length encoding for instructions (1-17 bytes)
- X86 vs ARM
  - CISC vs RISC
  - Variable-length encoding vs all instructions are 32 bits
  - Remainder of lecture: purple denotes comparison with ARM

X86

- X86 is *very* complicated
  - 8-, 16-, 32-, 64-bit values, floats, ...
  - Hundreds or thousands of instructions (depending on how they're counted)
  - Variable-length encoding for instructions (1-17 bytes)
- X86 vs ARM
  - CISC vs RISC
  - Variable-length encoding vs all instructions are 32 bits
  - Remainder of lecture: purple denotes comparison with ARM
- X86lite is a simple subset, still suitable as a compilation target
  - Values are 64-bit integers
  - About 20 instructions
  - Fixed-length encoding for instructions

X86lite machine state

- Memory, consisting of 2^{64} bytes
  - Quadword at `addr` is stored little-endian in **Mem**`[addr]` … **Mem**`[addr+7]` (least significant byte least address)



X86lite machine state

- Memory, consisting of 2^{64} bytes
  - Quadword at `addr` is stored little-endian in **Mem**`[addr]` … **Mem**`[addr+7]` (least significant byte least address)
- 16 64-bit registers
  - `rax`: general purpose accumulator
  - `rbx`: base pointer, pointer to data
  - `rcx`: counter register for strings & loops
  - `rdx`: data register for I/O
  - `rsi`: pointer register, string source register
  - `rdi`: pointer register, string destination register
  - `rbp`: base pointer, points to the stack frame
  - `rsp`: stack pointer, points to the top of the stack
  - `r08-r15`: general-purpose registers

X86lite machine state

- Memory, consisting of 2^{64} bytes
  - Quadword at `addr` is stored little-endian in **Mem**`[addr]` … **Mem**`[addr+7]` (least significant byte least address)
- 16 64-bit registers
  - `rax`: general purpose accumulator
  - `rbx`: base pointer, pointer to data
  - `rcx`: counter register for strings & loops
  - `rdx`: data register for I/O
  - `rsi`: pointer register, string source register
  - `rdi`: pointer register, string destination register
  - `rbp`: base pointer, points to the stack frame
  - `rsp`: stack pointer, points to the top of the stack
  - `r08-r15`: general-purpose registers
- 3 flags (bits)
  - `OF`: ("overflow") set when result is too big/small to fit in 64 bits
  - `SF`: ("sign") set to the sign of the result (0=positive, 1=negative)
  - `ZF`: ("zero") set when the result is 0

X86lite machine state

- Memory, consisting of 2^{64} bytes
  - Quadword at `addr` is stored little-endian in **Mem**`[addr]` … **Mem**`[addr+7]` (least significant byte least address)
- 16 64-bit registers
  - `rax`: general purpose accumulator
  - `rbx`: base pointer, pointer to data
  - `rcx`: counter register for strings & loops
  - `rdx`: data register for I/O
  - `rsi`: pointer register, string source register
  - `rdi`: pointer register, string destination register
  - `rbp`: base pointer, points to the stack frame
  - `rsp`: stack pointer, points to the top of the stack
  - `r08-r15`: general-purpose registers
- 3 flags (bits)
  - `OF`: ("overflow") set when result is too big/small to fit in 64 bits
  - `SF`: ("sign") set to the sign of the result (0=positive, 1=negative)
  - `ZF`: ("zero") set when the result is 0
- `rip`: "virtual" register, points to current instruction
  - `rip` is manipulated only by jumps and return

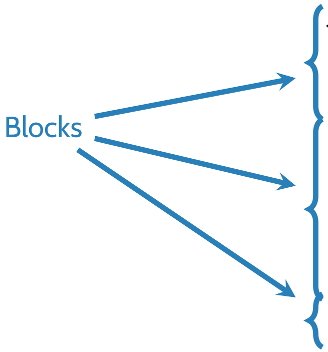
Anatomy of an x86lite program

Text segment

Data segment

```
.text
factorial:
   cmpq   %rdi, $0
   je     .L4
   movq   $1, %rax
   movq   $0, %rdx
.L3:
   imulq  %rdx, %rax
   addq   $1, %rdx
   cmpq   %rdx, %rdi
   jne    .L3
   retq
.L4:
   movq   $1, %rax
   retq
   .data
.str:
   .asciz "Factorial␣is␣%ld\n"
.global_int:
   .quad 42
```

Anatomy of an x86lite program



```
.text
factorial:
   cmpq   %rdi, $0
   je     .L4
   movq   $1, %rax
   movq   $0, %rdx
.L3:
   imulq  %rdx, %rax
   addq   $1, %rdx
   cmpq   %rdx, %rdi
   jne    .L3
   retq
.L4:
   movq   $1, %rax
   retq
   .data
.str:
   .asciz "Factorial␣is␣%ld\n"
.global_int:
   .quad 42
```

Anatomy of an x86lite program

Labels

```
.text
factorial:
   cmpq   %rdi, $0
   je     .L4
   movq   $1, %rax
   movq   $0, %rdx
.L3:
   imulq  %rdx, %rax
   addq   $1, %rdx
   cmpq   %rdx, %rdi
   jne    .L3
   retq
.L4:
   movq   $1, %rax
   retq
   .data
.str:
   .asciz "Factorial␣is␣%ld\n"
.global_int:
   .quad 42
```

X86Lite instructions

- Instruction = opcode + operand list
  - AT&T syntax: `movq $42, %rax` stores the number 42 in `rax`
    - `$` prefix denotes immediate (constant)
    - `%` prefix denotes register
    - `q` suffix denote quadword
  - Intel notation: `mov rax 42`
    - Swap source & destination
    - No prefixes / suffixes

X86Lite instructions

- Instruction = opcode + operand list
  - AT&T syntax: `movq $42, %rax` stores the number 42 in `rax`
    - `$` prefix denotes immediate (constant)
    - `%` prefix denotes register
    - `q` suffix denote quadword
  - Intel notation: `mov rax 42`
    - Swap source & destination
    - No prefixes / suffixes
- Opcodes (**full specification on course webpage**)
  - Arithmetic: `addq`, `imulq`, `subq`, `negq`, `incq`, `decq`
    - `adds x0, x0, 1` ~ `addq $1, %rax`
  - Logic: `andq`, `orq`, `notq`, `xorq`
  - Bit-manipulation: `sarq`, `shlq`, `shrq`, `setb`
  - Data-movement: leaq, `movq`, `pushq`, `popq`
  - Control flow: `cmpq`, `jmp`, `callq`, `retq`, `j CC`

X86Lite Operands

- `Imm` (“immediate”) 64-bit literal signed integer
    - `42`, `0x3de7`
- `Lbl` (“label”) symbolic machine address (to be resolved by assembler/linker/loader)
    - `_factorial`, `.L2`
- `Reg` (“register”)
    - `%rax`, `%r04`
- `Ind` (“indirect”) memory address
    - `(%rax)`, `-8(%rbp)`

X86 Addressing

- Three components of an indirect address: Disp(Base, Index, Scale)
  - Base: a machine address stored in a register
  - Index & Scale: a variable offset from the base (not in x86lite)
  - Disp: displacement/offset (optional)
- Disp(Base, Index, Scale) $\sim$ [Base, Disp, Index lsl 2^{Scale}]
- Refers to the location **Mem**`[Base + Index * Scale + Disp]`
  - `movq (%rsp), %rax` retrieves **Mem**`[rsp]` and stores it in `rax`
  - `movq -8(%rsp), %rax` retrieves **Mem**`[rsp - 8]` and stores it in `rax`
  - `movq %rax, (%rsp)` stores value of `rax` in **Mem**`[rsp]`.

Control flow

- Three condition flags:
  - `OF`: ("overflow") set when result is to big/small to fit in 64 bits
  - `SF`: ("sign") set to the sign of the result (0=positive,1=negative)
  - `ZF`: ("zero") set when the result is 0
  - OF $\sim$ V, SF $\sim$ N , ZF $\sim$ Z

Control flow

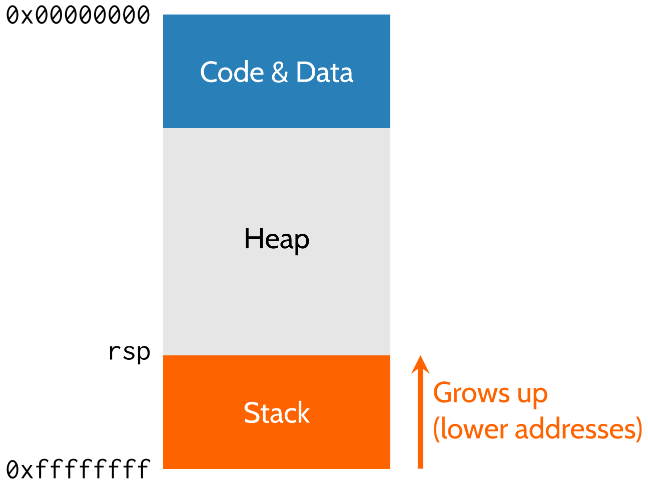
- Three condition flags:
  - `OF`: (“overflow”) set when result is to big/small to fit in 64 bits
  - `SF`: (“sign”) set to the sign of the result (0=positive,1=negative)
  - `ZF`: (“zero”) set when the result is 0
  - OF $\sim$ V, SF $\sim$ N , ZF $\sim$ Z
- Instruction `cmpq SRC1, SRC2`: compute `SRC2-SRC1` and set flags

Control flow

- Three condition flags:
  - `OF`: ("overflow") set when result is to big/small to fit in 64 bits
  - `SF`: ("sign") set to the sign of the result (0=positive,1=negative)
  - `ZF`: ("zero") set when the result is 0
  - OF $\sim$ V, SF $\sim$ N , ZF $\sim$ Z
- Instruction `cmpq SRC1, SRC2`: compute `SRC2-SRC1` and set flags
- Instruction `j CC SRC`: jump if to `SRC` if condition code `CC` is set
  - `e` ("equality"): `ZF` set
  - `ne` ("inequality"): `ZF` clear
  - `g` ("greater than"): `SF` clear and `ZF` clear
  - `l` ("less than"): `SF` not equal to `OF`
  - `ge` ("greater than or equal"): `SF` clear
  - `le` ("less than or equal"): `SF` not equal to `OF` or `ZF` set

Conventions

Memory layout



Stack operations

- `%rsp`: pointer to the top of the stack
- `pushq SRC`
  ```
  rsp := rsp - 8
  Mem[rsp] := SRC
  ```
- `popq DEST`
  ```
  DEST := Mem[rsp]
  rsp := rsp + 8
  ```
- `callq SRC`
  ```
  pushq rip
  rip := SRC
  ```
- `retq`
  ```
  popq rip
  ```

Calling conventions

- Implementation of function calls is up to the compiler
  - How are parameters passed?
  - How is return value passed back?
  - How is the return address stored?
  - Which registers is a function allowed to change?
    - caller save: freely usable by called code
    - callee save: must be restored by called code

Calling conventions

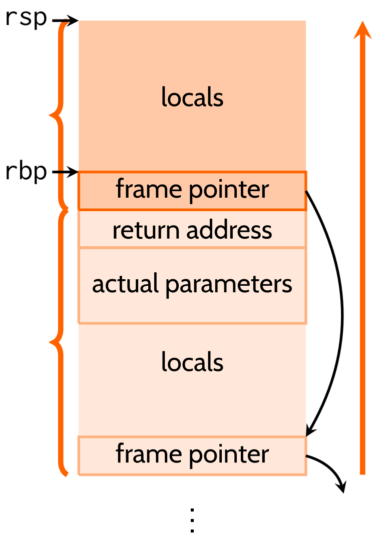
- Implementation of function calls is up to the compiler
  - How are parameters passed?
  - How is return value passed back?
  - How is the return address stored?
  - Which registers is a function allowed to change?
    - caller save: freely usable by called code
    - callee save: must be restored by called code
- A *calling convention* is a contract that specifies the structure of the stack and the interface between function *caller* and *callee*

Calling conventions

- Implementation of function calls is up to the compiler
  - How are parameters passed?
  - How is return value passed back?
  - How is the return address stored?
  - Which registers is a function allowed to change?
    - caller save: freely usable by called code
    - callee save: must be restored by called code
- A *calling convention* is a contract that specifies the structure of the stack and the interface between function *caller* and *callee*
- Useful to standardize on a single convention across the whole system
  - x86-64 AMD System V ABI on 64-bit x86
  - AARCH64 System V ABI on 64-bit ARM
  - cdecl ("C declaration") on 32-bit x86

The call stack

- Function calls are implemented using a *stack* of **activation records** (aka **stack frames**)
- Each activation record contains:
    - Frame pointer (start address of previous frame)
    - Local variables
- Except for current frame, also contains:
    - Actual parameters (arguments)
    - Return address



Caller protocol

Suppose we call function with parameters v_1, ..., v_n

1. Save caller-save registers, if needed
2. Store first six actual parameters $v_1, \ldots, v_6$ in `rdi`, `rsi`, `rdx`, `rcx`, `r08`, `r09`
3. Push v_n,...,v_7
   - nth actual parameter is located at **Mem**`[rbp + 8*(`*n*`-5)]`
4. Use `callq` to jump to the code for called function (& push return address)

Caller protocol

Suppose we call function with parameters v_1, ..., v_n

1. Save caller-save registers, if needed
2. Store first six actual parameters $v_1, \ldots, v_6$ in `rdi`, `rsi`, `rdx`, `rcx`, `r08`, `r09`
3. Push v_n,...,v_7
   - nth actual parameter is located at **Mem**`[rbp + 8*(n-5)]`
4. Use `callq` to jump to the code for called function (& push return address)

After call:

1. De-allocate pushed actual parameters
2. Restore caller-save registers, if needed

Callee protocol

On entry:

1. Save old frame pointer (`rbp` is callee-save)
2. Set `rbp` to point to current frame
3. Allocate local storage

Callee protocol

On entry:

1. Save old frame pointer (`rbp` is callee-save)
2. Set `rbp` to point to current frame
3. Allocate local storage

On exit:

1. Store return value in `rax`
2. Deallocate local storage
3. Restore previous `rbp`

Exercise: how do variables map to registers?

```
long factorial(long n) {
  long i;
  long result = 1;
  for (i = 1; i < n; i++) {
    result *= i;
  }
  return result;
}
```

```
factorial:
   cmpq   %rdi, $0
   jle    .L4
   movq   $1, %rax
   movq   $1, %rdx
.L3:
   imulq  %rdx, %rax
   addq   $1, %rdx
   cmpq   %rdx, %rdi
   jne    .L3
   retq
.L4:
   movq   $1, %rax
   retq
```

x86-64 System V AMD 64 ABI

- Callee-save: `rbp`, `rbx`, `r12-r15`
- Caller-save: all others
- Store return value in `rax` (second return value in `rdx`)
- Parameters:
  - Parameters 1-6 in `rdi`, `rsi`, `rdx`, `rcx`, `r08`, `r09`
  - Parameters 7-n in `16(rbp)`, `24(rbp)`, … `(8*(`*n*`-5))(rbp)`
- `rsp` is 16-byte aligned *immediately before* `callq`
- 128 byte "red zone" below `rsp`
  - Not modified by signal / interrupt handlers
  - Useful for storing local data of leaf functions

HW1: X86lite

- In HW1, you will implement an x86 (machine code) simulator, assembler, and loader.
    - Get started!
- Next lecture: Intermediate representations & code generation