COS320: Compiling Techniques

Zak Kincaid

March 26, 2019
Compiler phases (simplified)

Source text
  ↓ Lexing
Token stream
  ↓ Parsing
Abstract syntax tree
  ↓ Translation
Intermediate representation
Optimization
  ← Code generation
Assembly
Today: x86Lite
• 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)
- 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 $\text{addr}$ is stored little-endian in $\text{Mem}[\text{addr}]$ ... $\text{Mem}[\text{addr}+7]$
X86lite machine state

- Memory, consisting of $2^{64}$ bytes
  - Quadword at addr is stored little-endian in $\text{Mem}[\text{addr}] \ldots \text{Mem}[\text{addr}+7]$
- 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 $\text{addr}$ is stored little-endian in $\text{Mem}[\text{addr}] \ldots \text{Mem}[\text{addr}+7]$
- 16 64-bit registers
  - $\text{rax}$: general purpose accumulator
  - $\text{rbx}$: base pointer, pointer to data
  - $\text{rcx}$: counter register for strings & loops
  - $\text{rdx}$: data register for I/O
  - $\text{rsi}$: pointer register, string source register
  - $\text{rdi}$: pointer register, string destination register
  - $\text{rbp}$: base pointer, points to the stack frame
  - $\text{rsp}$: stack pointer, points to the top of the stack
  - $\text{r08-r15}$: general-purpose registers
- 3 flags (bits)
  - $\text{OF}$: ("overflow") set when result is too big/small to fit in 64 bits
  - $\text{SF}$: ("sign") set to the sign of the result ($0=\text{positive}$, $1=\text{negative}$)
  - $\text{ZF}$: ("zero") set when the result is 0
  - $\text{rip}$: "virtual" register, points to current instruction
  - $\text{rip}$ is manipulated only by indirect jumps and return
X86lite machine state

• Memory, consisting of $2^{64}$ bytes
  • Quadword at addr is stored little-endian in Mem[addr] ... Mem[addr+7]
• 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 indirect jumps and return
Anatomy of an x86lite program

```
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
```
Anatomy of an x86lite program:

```
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
```
Anatomy of an x86lite program

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
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
  - 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: \textbf{Disp}(Base, Index, Scale)
  - Base: a machine address stored in a register
  - Index & Scale: a variable offset from the base (not in x86lite)
  - \textbf{Disp}: displacement/offset \textit{(optional)}

- Refers to the location \textbf{Mem}[Base + Index \ast Scale + Disp]
  - \texttt{movq (}%rsp\texttt{), %rax} retrieves \textbf{Mem}[%rsp] and stores it in \texttt{rax}
  - \texttt{movq -8(}%rsp\texttt{), %rax} retrieves \textbf{Mem}[%rsp - 8] and stores it in \texttt{rax}
  - \texttt{movq %rax, (}%rsp\texttt{)} stores value of \texttt{rax} in \textbf{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
Control flow

- Three condition flags:
  - 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

- Instruction `cmpq SRC1, SRC2`: compute SRC2−SRC1 and set flags
Control flow

• Three condition flags:
  • 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

• Instruction `cmpq SRC1, SRC2`: compute SRC2−SRC1 and set flags
  • 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

• Instruction `j CC SRC`: jump if to SRC if condition code CC is set
Conventions
Memory layout

- Code & Data
- Heap
- Stack

 rsp

0x00000000

0xffffffff

Grows up (lower addresses)
Stack operations

• %rsp: pointer to the top of the stack

• pushq SRC
  
  \[ \text{rsp} := \text{rsp} - 8 \]
  \[ \text{Mem}[\text{rsp}] := \text{SRC} \]

• popq DEST
  
  \[ \text{DEST} := \text{Mem}[\text{rsp}] \]
  \[ \text{rsp} := \text{rsp} + 8 \]

• callq SRC
  
  \begin{align*}
  & \text{pushq rip} \\
  & \text{rip} := \text{SRC}
  \end{align*}

• retq
  
  \begin{align*}
  & \text{popq rip}
  \end{align*}
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?
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?

- 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?

• 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
  • cdecl (“C declaration”) is the standard on 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
cdecl caller protocol

Suppose we call function with parameters $v_1, \ldots, v_n$

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

Suppose we call function with parameters \( v_1, \ldots, v_n \)

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

After call:

1. De-allocate pushed actual parameters
2. Restore caller-save registers, if needed (\( rax, rcx, rdx \))
cdecl callee protocol

On entry:

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

On exit:

1. Store return value in \texttt{rax}
2. Deallocate local storage
3. Restore previous \texttt{rbp}
cdecl callee protocol

On entry:
1. Save old frame pointer (\textit{rbp} is callee-save)
2. Set \textit{rbp} to point to current frame
3. Allocate local storage

On exit:
1. Store return value in \textit{rax}
2. Deallocate local storage
3. Restore previous \textit{rbp}
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