Lecture 3

COMPILER DESIGN
Announcements

- HW1: Hellocaml
  - Due *Tuesday, 1 Oct. at 23:59*

- HW2: X86lite
  - Will be available soon on our course Moodle
  - Pair-programming project
  - Simulator / Loader for x86 assembly subset

- Web site: [https://people.inf.ethz.ch/suz/teaching/252-0210.html](https://people.inf.ethz.ch/suz/teaching/252-0210.html)
- E-mail for teaching staff: [cd-tas-f19@lists.inf.ethz.ch](mailto:cd-tas-f19@lists.inf.ethz.ch)
  - Will reach all the TAs and me for private questions/concerns
  - Please use Moodle for other course related questions
The target architecture

X86LITE
Intel’s X86 Architecture

- 1978: Intel introduces 8086
- 1982: 80186, 80286
- 1985: 80386
- 1989: 80486 (100MHz, 1µm)
- 1993: Pentium
- 1995: Pentium Pro
- 1997: Pentium II/III
- 2000: Pentium 4
- 2003: Pentium M, Intel Core
- 2006: Intel Core 2
- 2008: Intel Core i3/i5/i7
- 2011: SandyBridge / IvyBridge
- 2013: Haswell
- 2014: Broadwell
- 2015: Skylake (4.2GHz, 14nm)
- AMD has a parallel line of processors
X86 Evolution & Moore’s Law

Intel Processor Transistor Count

Transistor Count (log scale)

Intel Processor Transistor Count

10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
100
10

X86 vs. X86lite

• X86 assembly is very complicated!
  – 8-, 16-, 32-, 64-bit values + floating point, etc.
  – Intel 64 and IA 32 architectures have a huge number of functions
  – “CISC” complex instructions
  – Machine code: instructions range in size from 1 byte to 17 bytes
  – Lots of hold-over design decisions for backward compatibility
  – Hard to understand, there is a large book about optimizations at just the instruction-selection level

• X86lite is a very simple subset of X86:
  – Only 64-bit signed integers (no floating point, no 16-bit, no …)
  – Only about 20 instructions
  – Sufficient as a target language for general-purpose computing
X86lite Machine State: Registers

- **Register File**: 16 64-bit registers
  - `rax` general purpose accumulator
  - `rbx` base register, 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

- **rip**: a “virtual” register, points to the current instruction
  - `rip` is manipulated only indirectly via jumps and return
Simplest instruction: **mov**

- **movq SRC, DEST**  
  copy SRC into DEST

- Here, DEST and SRC are *operands*
- DEST is treated as a *location*
  - A location can be a register or a memory address
- SRC is treated as a *value*
  - A value is the *contents* of a register or memory address
  - A value can also be an *immediate* (constant) or a label

- **movq $4, %rax**  // move the 64-bit immediate value 4 into rax
- **movq %rbx, %rax**  // move the content of rbx into rax
A Note About Instruction Syntax

- X86 presented in two common syntax formats

- AT&T notation: source before destination
  - Prevalent in the Unix/Mac ecosystems
  - Immediate values prefixed with ‘$’
  - Registers prefixed with ‘%’
  - Mnemonic suffixes: movq vs. mov
    - q = quadword (4 words)
    - l = long (2 words)
    - w = word (16-bit)
    - b = byte (8-bit)

- Intel notation: destination before source
  - Used in the Intel specification / manuals
  - Prevalent in the Windows ecosystem
  - Instruction variant determined by register name

Note: X86lite uses the AT&T notation and the 64-bit only version of the instructions and registers
X86lite Arithmetic instructions

- `negq DEST` two’s complement negation
- `addq SRC, DEST` DEST ← DEST + SRC
- `subq SRC, DEST` DEST ← DEST – SRC
- `imulq SRC, Reg` Reg ← Reg * SRC (truncated 128-bit mult.)

Examples as written in:

```
addq %rbx, %rax // rax ← rax + rbx
subq $4, %rsp // rsp ← rsp - 4
```

- Note: Reg (in `imulq`) must be a register, not a memory address
X86lite Logic/Bit manipulation Operations

- **notq DEST**  
  logical negation  
  (bitwise)

- **andq SRC, DEST**  
  DEST ← DEST & SRC  
  (bitwise)

- **orq SRC, DEST**  
  DEST ← DEST | SRC  
  (bitwise)

- **xorq SRC, DEST**  
  DEST ← DEST xor SRC  
  (bitwise)

- **sarq Amt, DEST**  
  DEST ← DEST >> Amt  
  (arithmetic shift right)

- **shlq Amt, DEST**  
  DEST ← DEST << Amt  
  (bitwise shift left)

- **shrq Amt, DEST**  
  DEST ← DEST >>> Amt  
  (bitwise shift right)

- **salq Amt, DEST**  
  (the same as shlq Amt, DEST)
X86 Operands

- Operands are the values operated on by the assembly instructions

- **Imm**
  - 64-bit literal signed integer “immediate”

- **Lbl**
  - a “label” representing a machine address
  - the assembler/linker/loader resolve labels

- **Reg**
  - One of the 16 registers, the value of a register is its contents

- **Ind**
  - \([\text{base:Reg}][\text{index:Reg,scale:int32}][\text{disp}]\)
  - machine address (see next slide)
• In general, there are three components of an indirect address
  – Base: a machine address stored in a register
  – Index * scale: a variable offset from the base
  – Disp: a constant offset (displacement) from the base

• \( \text{addr(\text{ind})} = \text{Base} + \left[ \text{Index} \times \text{scale} \right] + \text{Disp} \)
  – When used as a \textit{location}, \text{ind} denotes the address \text{addr(\text{ind})}
  – When used as a \textit{value}, \text{ind} denotes Mem[\text{addr(\text{ind})}], the contents of the memory address

• Example: \(-4(\%\text{rsp})\) denotes address: \(\text{rsp} - 4\)
• Example: \((\%\text{rax}, \%\text{rcx}, 4)\) denotes address: \(\text{rax} + 4\times\text{rcx}\)
• Example: \(12(\%\text{rax}, \%\text{rcx}, 4)\) denotes address: \(\text{rax} + 4\times\text{rcx} + 12\)

• Note: Index cannot be \(\text{rsp}\)

\textit{Note: X86lite does not need this full generality. It does not use index \times scale.}
X86lite Memory Model

- The X86lite memory consists of $2^{64}$ bytes numbered 0x00000000 through 0xffffffff.
- X86lite treats the memory as consisting of 64-bit (8-byte) quadwords.
- Therefore: legal X86lite memory addresses consist of 64-bit, quadword-aligned pointers.
  - All memory addresses are evenly divisible by 8

- `leaq Ind, DEST` \hspace{6pt} DEST $\leftarrow$ addr(Ind) \hspace{6pt} loads a pointer into DEST

- By convention, the stack grows from high addresses to low addresses
- The register `rsp` points to the top of the stack
  - `pushq SRC` \hspace{6pt} `rsp $\leftarrow$ rsp - 8; Mem[rsp] $\leftarrow$ SRC
  - `popq DEST` \hspace{6pt} DEST $\leftarrow$ Mem[rsp]; `rsp $\leftarrow$ rsp + 8`
• X86 instructions set flags as a side effect
• X86lite has only 3 flags:
  – **OF**: “overflow” set when the result is too big/small to fit in 64-bit reg.
  – **SF**: “sign” set to the sign of the result (0=positive, 1 = negative)
  – **ZF**: “zero” set when the result is 0

• From these flags, we can define *Condition Codes*
  – To compare SRC1 and SRC2, compute SRC1 – SRC2 to set the flags
  – **e** equality holds when **ZF** is set
  – **ne** inequality holds when (not **ZF**)
  – **g** greater than holds when (not **ZF**) and (not **SF**) (SF = OF)
  – **l** less than holds when **SF <> OF**
    • Equivalently: ((SF && not OF) || (not SF && OF))
  – **ge** greater or equal holds when (not **SF**) (SF = OF)
  – **le** than or equal holds when **SF <> OF** or **ZF**
Code Blocks & Labels

• X86 assembly code is organized into labeled blocks

```
label1:
  <instruction>
  <instruction>
  ...
  <instruction>

label2:
  <instruction>
  <instruction>
  ...
  <instruction>
```

• Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls).
• Labels are translated away by the linker and loader – instructions live in the heap in the “code segment”
• An X86 program begins executing at a designated code label (usually “main”).
Conditional Instructions

• `cmpq SRC1, SRC2`  Compute SRC2 – SRC1, set condition flags

• `setbCC DEST`  DEST’s lower byte ← if CC then 1 else 0

• `jCC SRC`  rip ← if CC then SRC else fallthrough

• Examples

```
cmpq %rcx, %rax  // Compare rax to ecx
je __truelbl  // If rax = rcx then jump to __truelbl

movq $1, %rbx  // %rbx = ... 0001
movq $2, %rcx  // %rcx = ... 0010
cmpq %rbx, %rcx  // OF=0, ZF=0, SF=0
setbg %rax  // %rax = ... 0001
```
Jumps, Call and Return

• **jmp SRC** \(\text{rip} \leftarrow \text{SRC}\) \(\) Jump to location in SRC

• **callq SRC** \(\text{Push rip; rip} \leftarrow \text{SRC}\)
  – Call a procedure
    • Push the program counter to the stack (decrementing \(r_{sp}\)), and then
    • Jump to the machine instruction at the address given by SRC

• **retq** \(\) Pop into rip
  – Return from a procedure
    • Pop the current top of the stack into rip (incrementing \(r_{sp}\))
    – This instruction effectively jumps to the address at the top of the stack
See file: x86.ml

IMPLEMENTING X86LITE
See: runtime.c

**DEMO: HANDCODYING X86LITE**
Compiling, Linking, Running

• To use hand-coded X86:
  1. Compile main.ml (or something like it) to either native or bytecode
  2. Run it, redirecting the output to some .s file, e.g.:
     ./main.native >> test.s
  3. Use gcc to compile & link with runtime.c:
     gcc -o test runtime.c test.s
  4. You should be able to run the resulting executable:
     ./test

• If you want to debug in gdb:
  – Call gcc with the –g flag too