CSci 2021: Review Lecture 2

Stephen McCamant

University of Minnesota, Computer Science & Engineering

Midterm 2 topics (in one slide)

- Machine-level code representation
  - Instructions, operands, flags
  - Branches, conditions, and loops
  - Procedures and calling conventions
  - Arrays, structs, unions
  - Buffer overflow attacks
- Dynamic memory allocation
  - Implementation techniques
  - Pitfalls
- CPU architecture
  - Y86 instructions
  - Control logic and HCL
  - Sequential Y86-64
  - Pipelined Y86-64

Outline

Topics in machine code

Announcements break

Topics in dynamic allocation

Topics in CPU architecture

Review questions

Instructions and operands

- Assembly language $\leftrightarrow$ machine code
- Sequence of instructions, encoded in bytes
- An instruction reads from or writes to operands
  - x86: usually at most one memory operand
  - AT&T: destination is last operand
  - AT&T shows operand size with b/w/l/q suffix

Addressing modes

- General form: disp(base,index,scale)
  - Displacement is any constant, scale is 1, 2, 4 or 8
  - Base and index are registers
  - Formula: mem[disp $+$ base $+$ index $\cdot$ scale]
- All but base are optional
  - Missing displacement or index: 0
  - Missing scale: 1
  - Drop trailing (but not leading) commas
- Do same computation, just put address in register: `lea`

Flags and branches

- Flags (aka condition codes) are set based on results of arithmetic
  - ZF: result is zero
  - SF: result is negative (highest bit set)
  - OF: signed overflow occurred
  - CF: unsigned overflow (“carry”) occurred
- Used for condition in:
  - `setCC`: store 1 or 0
  - `cmovCC`: copy or don’t copy
  - `jCC`: jump or don’t jump
- Just for setting flags: `cmp` (like `sub`), `test` (like `and`)

Loops

- Simplest structure: conditional jump "at the bottom", like a C `do-while`
- C `while` also checks at beginning
- C `for` e.g. initializes a variable and updates it on each iteration
- Assembly most like C with `goto`

Stack and frames

- "The" stack is used for data with a function lifetime
- `%rsp` points at the most recent in-use element ("top")
- Convenient instructions: `push` and `pop`
- Section for one run of a function: stack frame

Calling conventions

- Function arguments go in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`
- Return value is in `%rax`
- Handle that both *caller* and *callee* want to use registers
- Caller-saved: callee might modify, caller must save if using
  - `%rax`, `%rdi`, ..., `%r10`, `%r11`, flags
- Callee-saved: caller might be using, callee must save before using
  - `%rbx`, `%r12`, ..., `%rbp`, (`%rsp`)

Arrays

- Sequence of values of same size and type, next to each other
- Numbered starting from 0 in C
- To find location: start with base, add index times size
- C's pointer arithmetic is basically the same operation
- Multi-dimensional array
  - Needs more multiplying
- Array of pointers to arrays
  - Different, more flexible layout
  - Each access needs more loads

Structs and unions

- Struct groups objects of different types and sizes, in order
- Fields often accessed using displacement from a pointer
- Alignment requirements $\rightarrow$ padding
  - Primitive values aligned to their size
  - Pad between elements, when next needs more alignment
  - Pad at end, to round off total size
- Unions: "like structs where every offset is 0"
  - Used to save space if only one needed at a time
  - Can also reveal storage details

Buffer overflows

- Local arrays stored on the stack
- C compilers usually do not check limits of array accesses
- Too much buffer data can overwrite a return address
  - Changes what code will execute
  - Various nefarious uses
- Various partial defenses:
  - Randomize stack location
  - Non-executable stack
  - Stack canary checking

Outline

- Topics in machine code
- Announcements break
- Topics in dynamic allocation
- Topics in CPU architecture
- Review questions

Announcements

- Solution set for Exercise Set 3 is up now
  - First draft: some expansions/corrections later
  - Also open for Moodle forum discussion
- HA4 is due Monday night

Outline

- Topics in machine code
- Announcements break
- Topics in dynamic allocation
- Topics in CPU architecture
- Review questions

Implementing `malloc`

- Data structures to represent the heap
  - Boundary tags and the implicit list
  - Explicit free list(s)
- Algorithms for heap management
  - First fit vs. best fit
  - Size segregation

Dynamic allocation pitfalls

- Allocating too big or too small
- Freeing too soon or more than once
- Mixing up local and dynamic objects
- Uninitialized memory
- Memory leaks (failing to free)
- Debug with GDB or Valgrind
- Alternative: garbage collection

Outline

- Topics in machine code
- Announcements break
- Topics in dynamic allocation
- Topics in CPU architecture
- Review questions

Y86-64 instructions

- Simplified subset of x86-64, simpler encoding
- 64-bit only, 15 registers
- Four kinds of moves, only one addressing mode
- Add, subtract, bitwise and, bitwise xor
- Conditional jump and move based on equality and signed comparison
- Call, return, push, pop
- Halt and two fatal errors, no exceptions

Logic design for control

- Combinational circuits:
  - Compute a function of bits, no memory
  - Acyclic network of AND, OR, and NOT gates
  - Also includes word-sized comparison, multiplexors, and ALU
- Stateful elements:
  - (Clocked) registers
  - Random-access memory
  - State updates occur on rising clock edge only

Hardware design in HCL

- Simple language for specifying control circuits
- Two types: Boolean and word
- Comparison and logic operators (no side-effects or "short circuiting")
- Core construct: sequential conditional
  - $[C_1 : V_1; C_2 : V_2; \ldots 1 : V_n]$
  - "Else" case written 1

Sequential Y86-64

- Whole state update function is one big combinational circuit
- Express behavior of each instruction using smaller computations
- Processing split into stages for organization:
  - Fetch, decode, execute, memory, write back, PC update
- Simplest, but requires long cycle time (slow)

Pipelining basics

- Split processing into stages, and work on multiple instructions at once
- Reduces cycle time and increases hardware utilization
- Pipeline registers hold data between stages
- Performance concerns: balanced stages, and not too many
- Correctness concerns: must have same final behavior

Pipelining techniques

- *Hazards*: dependencies introduce danger of incorrect results
- Branch prediction: guesses result of conditional jumps
- Stalling: hold up instructions until data ready
  - Simple, but introduces a lot of delay
  - Used for return instruction in Y86-64
- Cancelling: kill incorrect instructions
  - Must happen before they have side-effects
  - Used for branch mis-predictions
- Forwarding: copy data to a different stage right as needed

Outline

- Topics in machine code
- Announcements break
- Topics in dynamic allocation
- Topics in CPU architecture
- Review questions

Calling conventions

According to the standard x86-64 calling convention, which of these registers would your function need to save before modifying it?

A. `%rdi`
B. `%rsi`
C. `%r10`
D. `%rbx`
E. `%rax`

x86-64 instructions

Which two instructions can be used to compare `%rax` to zero?

A. `cmp $0, %rax` and `test $0, %rax`
B. `cmp $0, %rax` and `test %rax, %rax`
C. `cmp %rax, %rax` and `test $0, %rax`
D. `cmp %rax, %rax` and `test %rax, %rax`

`for` loops

Which of these while loop patterns is equivalent to the loop `for (A; B; C) { D; }`?

A. `A; while (B && C) { D; }`
B. `B; while (A) {D; C}`
C. `A; while (B) {C; D}`
D. `A; while (B) {C; D; C}`
E. `A; while (B) {D; C}`

Structure padding

Because of padding, which of these `struct`s would not be the same size as the others?

A. `struct { short s; long l; }`
B. `struct { float f; double d; }`
C. `struct { char c; long l; }`
D. `struct { long l1; long l2; }`
E. `struct { int i1; int i2; }`

Y86-64 instructions

Which of these Y86-64 instructions is an indirect jump?

A. call
B. ret
C. jmp
D. jle
E. jne