Machine-level representation of programs: ISAs

Yipeng Huang

Rutgers University

March 8, 2022
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Quiz, PA3, PA4

Quiz 5: The limits of floating point numbers
Out now, due Friday night before spring break officially starts.

PA3
Poll about progress.

PA4
PA4 (Bomblab) is an assignment lab offered by the textbook. A CS211 classic. Will be launched before we go on break. Due two weeks after we get back from break.
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Computer organization
Layer cake: remember the first day of class, we discussed what are parts of a computer?

- Society
- Human beings
- Applications
- Algorithms
- High-level programming languages
- Interpreters
- Low-level programming languages
- Compilers
- Architectures
- Microarchitectures
- Sequential/combinational logic
- Transistors
- Semiconductors
- Materials science
- Physics
- Mathematics
Stored program computer

Stored program:
Instructions reside in memory, loaded as needed.

von Neumann architecture:
Data and instructions share same connection to memory.

Assembly/Machine Code View

Programmer-Visible State
- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Memory
- Byte addressable array
- Code and user data
- Stack to support procedures

Carnegie Mellon

# Memory hierarchy

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Access speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tape</td>
<td>250Pb</td>
<td></td>
</tr>
<tr>
<td>Hard drives</td>
<td>16TB</td>
<td>2Mb/s</td>
</tr>
<tr>
<td>Solid state drives</td>
<td>4TB</td>
<td>2Gb/s</td>
</tr>
<tr>
<td>DRAM</td>
<td>8Gb - 1Tb+</td>
<td>8Gb/s</td>
</tr>
<tr>
<td>Last-level cache</td>
<td>64Mb</td>
<td></td>
</tr>
<tr>
<td>Level-1 cache</td>
<td>1Mb</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>1Kb</td>
<td></td>
</tr>
</tbody>
</table>

- Registers (.25ns; 4GHz => 0.25e-9s)
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Unraveling the compilation chain

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting binary in file `p`

```
C program (p1.c p2.c) → Compiler (gcc -Og -S) → Asm program (p1.s p2.s) → Assembler (gcc or as) → Object program (p1.o p2.o) → Linker (gcc or ld) → Executable program (p) → Static libraries (.a)
```

Figure: Stages of compilation. Image credit CS:APP
Assembly

Human readable machine code

- Very limited
- Not much control flow
- Any more complex functionality is built up
- for loops, while loops, turn into assembly sequence

Choice of what assembly to experiment with

- MIPS
- ARM
- x86 / x86-64 (not ideal for teaching, but it allows us to experiment on ilab)
Assembly instructions

Instructions for the microarchitecture

▶ Binary streams that tell an electronic circuit what to do
▶ Fetch, decode, execute, memory, writeback
A preview of microarchitecture

Figure: Stages of compilation. Image credit Wikimedia
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Why are instruction set architectures important

Interface between computer science and electrical and computer engineering

- Software is varied, changes
- Hardware is standardized, static

Computer architect Fred Brooks and the IBM 360

- IBM was selling computers with different capacities,
- Compile once, and can run software on all IBM machines.
- Backward compatibility.
- An influential idea.
CISC vs. RISC

Complex instruction set computer

- Intel and AMD
- Have an extensive and complex set of instructions
- For example: x86’s extensions: x87, IA-32, x86-64, MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.2, SSE5, AES-NI, CLMUL, RDRAND, SHA, MPX, SGX, XOP, F16C, ADX, BMI, FMA, AVX, AVX2, AVX512, VT-x, VT-d, AMD-V, AMD-Vi, TSX, ASF
- Can license Intel’s compilers to extract performance
- Secret: inside the processor, they break it down to more elementary instructions
CISC vs. RISC

Reduced instruction set computer

- MIPS, ARM, RISC-V (can find Patterson and Hennessy Computer Organization and Design textbook in each of these versions), and PowerPC
- Have a relatively simple set of instructions
- For example: ARM’s extensions: SVE;SVE2;TME; All mandatory: Thumb-2, Neon, VFPv4-D16, VFPv4
- Obsolete: Jazelle
- ARM: smartphones, Apple ARM M1 Mac
Into the future: Post-ISA world

Post-ISA world

- Increasingly, the CPU is not the only character
- It orchestrates among many pieces of hardware
- Smartphone die shot
- GPU, TPU, FPGA, ASIC

Figure: Apple A13 (2019 Apple iPhone 11 CPU). Image credit AnandTech
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

\texttt{swap.s}: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Unraveling the compilation chain

## Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

![Figure: Stages of compilation. Image credit CS:APP](image)

- `gcc -Og -S swap.c`
- `objdump -d swap`

Let’s go to CS:APP textbook lecture slides (05-machine-basics.pdf) slide 28

---

Data movement instructions

Does unsigned / signed matter?

1. void swap_uc ( unsigned char*a, unsigned char*b );
2. void swap_sc ( signed char*a, signed char*b );

Swapping different data sizes

1. void swap_c ( char*a, char*b );
2. void swap_s ( short*a, short*b );
3. void swap_i ( int*a, int*b );
4. void swap_l ( long*a, long*b );
Table of contents

Announcements

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Data size and x86 / x86-64 registers

Assembly syntax

Instruction Source, Dest

```
swap_l:
    movq (%rsi), %rax
    movq (%rdi), %rdx
    movq %rdx, (%rsi)
    movq %rax, (%rdi)
    ret
```

<table>
<thead>
<tr>
<th>swap</th>
<th>data type</th>
<th>mov operation</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap_uc</td>
<td>unsigned char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_sc</td>
<td>signed char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_c</td>
<td>char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_s</td>
<td>short</td>
<td>movw (move word)</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>swap_i</td>
<td>int</td>
<td>movl</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>swap_l</td>
<td>long</td>
<td>movq</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>
Some History: IA32 Registers

<table>
<thead>
<tr>
<th>data type</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>short</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>int</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>long</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>

Note the backward compatibility.
Data size and IA32, x86, and x86-64 registers

<table>
<thead>
<tr>
<th>Data type</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>short</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>int</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>long</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>

Note the backward compatibility.

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Data size and IA32, x86, and x86-64 registers

Figure: x86-64 with SIMD extensions registers. Image credit: https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg