6.S096: Assembly

Daniel Kang

Massachusetts Institute of Technology

A typical computer

Fig. 01-06 from Englander, Irv. *The Architecture of Computer Hardware and System Software: An Information Technology pproach*. 2nd edition. John Wiley & Sons, Inc.

Von Neumann Architecture: CPU

Image of Von Neumann architecture removed due to copyright restrictions.

What is assembly?

"An assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions."

What is assembly?

"An assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions."

Why high-level languages?

- Stronger abstractions
  - *e.g.* object oriented programming
  - *e.g.* C, C++, Java, Python
- Increased portability
  - *e.g.* interpreted languages
  - *e.g.* Java, Python
- Faster development cycles
- ...

Why assembly?

- Debugging often requires reading assembly
- Understand how things work at the machine level
- Helps you write faster code (even in high-level languages)

Assembly languages (architectures)

- x86, x64 (desktops, laptop, servers)
- ARM (phones)
- SPARC (Sun)
- MIPS
- ...

Registers

- Storage close to the CPU
- Most instructions manipulate **registers**
  - Manipulation of register content
  - Load and store from registers
- Register are **fast**, memory access is **slow**

x86 syntax (AT&T / GAS)

- Registers
- 8 registers in x86 (32-bit)
- 16 registers in x64 (64-bit)
  - 16-bit: ax, bx, ...
  - 32-bit: eax, ebx, ...
  - 64-bit: rax, rbx, ..., r8, r9, ...
  - Referenced by %REGISTER
- Constants: $0, $1, $0x20 (32), ...

x86 syntax (AT&T / GAS)

- Arithmetic instructions
  - OP a, b $\rightarrow$ b = b OP a
  - *e.g.* add %edx, %eax $\rightarrow$ %eax = %eax + %edx
- Assignment instructions
  - OP a, b $\rightarrow$ b = a
  - *e.g.* mov %edx, %eax $\rightarrow$ %eax = %edx
- Condition testing
  - OP a, b
  - *e.g.* test %eax, %eax
- Control flow
  - OP address
  - *e.g.* jmp 0x420e80 $\rightarrow$ jump unconditionally to 0x420e80
- ...

Live demos!

Hailstone sequences: code

```c
#include <stdio.h>

int main(void) {
  int n = 32;
  int count = 0;
  while (n > 1) {
    count++;
    if (n & 1)
      n = n * 3 + 1;
    else
      n /= 2;
  }

  printf("%d\n", count);

  return 0;
}
```

Hailstone sequences: unoptimized disassembly

- Compiled with gcc version 4.8.1
- CPU: `Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz`
- `gcc -O0 hailstone.c -o hailstone`
- `gdb ./hailstone`

Hailstone sequences: unoptimized disassembly

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x10,%rsp
   0x0000000000400535 <+8>:     movl   $0x20,-0x8(%rbp)
   0x000000000040053c <+15>:    movl   $0x0,-0x4(%rbp)
   0x0000000000400543 <+22>:    jmp    0x400573 <main+70>
   0x0000000000400545 <+24>:    addl   $0x1,-0x4(%rbp)
   0x0000000000400549 <+28>:    mov    -0x8(%rbp),%eax
   0x000000000040054c <+31>:    and    $0x1,%eax
   0x000000000040054f <+34>:    test   %eax,%eax
   0x0000000000400551 <+36>:    je     0x400564 <main+55>
   0x0000000000400553 <+38>:    mov    -0x8(%rbp),%edx
   0x0000000000400556 <+41>:    mov    %edx,%eax
   0x0000000000400558 <+43>:    add    %eax,%eax
   0x000000000040055a <+45>:    add    %edx,%eax
   0x000000000040055c <+47>:    add    $0x1,%eax
   0x000000000040055f <+50>:    mov    %eax,-0x8(%rbp)
   0x0000000000400562 <+53>:    jmp    0x400573 <main+70>
   0x0000000000400564 <+55>:    mov    -0x8(%rbp),%eax
   0x0000000000400567 <+58>:    mov    %eax,%edx
   0x0000000000400569 <+60>:    shr    $0x1f,%edx
   0x000000000040056c <+63>:    add    %edx,%eax
   0x000000000040056e <+65>:    sar    %eax
   0x0000000000400570 <+67>:    mov    %eax,-0x8(%rbp)
   0x0000000000400573 <+70>:    cmpl   $0x1,-0x8(%rbp)
   0x0000000000400577 <+74>:    jg     0x400545 <main+24>
   0x0000000000400579 <+76>:    mov    -0x4(%rbp),%eax
   0x000000000040057c <+79>:    mov    %eax,%esi
   0x000000000040057e <+81>:    mov    $0x400644,%edi
   0x0000000000400583 <+86>:    mov    $0x0,%eax
   0x0000000000400588 <+91>:    callq  0x400410 <printf@plt>
   0x000000000040058d <+96>:    mov    $0x0,%eax
   0x0000000000400592 <+101>:   leaveq
   0x0000000000400593 <+102>:   retq
End of assembler dump.
```

Hailstone sequences: optimzed disassembly

- `gcc -O3 hailstone.c -o hailstone`
- `gdb ./hailstone`

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040055d <+0>:     mov    $0x0,%edx
   0x0000000000400562 <+5>:     mov    $0x20,%eax
   0x0000000000400567 <+10>:    add    $0x1,%edx
   0x000000000040056a <+13>:    test   $0x1,%al
   0x000000000040056c <+15>:    je     0x400574 <main+23>
   0x000000000040056e <+17>:    lea    0x1(%rax,%rax,2),%eax
   0x0000000000400572 <+21>:    jmp    0x40057d <main+32>
   0x0000000000400574 <+23>:    mov    %eax,%ecx
   0x0000000000400576 <+25>:    shr    $0x1f,%ecx
   0x0000000000400579 <+28>:    add    %ecx,%eax
   0x000000000040057b <+30>:    sar    %eax
   0x000000000040057d <+32>:    cmp    $0x1,%eax
   0x0000000000400580 <+35>:    jg     0x400567 <main+10>
   0x0000000000400582 <+37>:    sub    $0x8,%rsp
   0x0000000000400586 <+41>:    mov    $0x400654,%esi
   0x000000000040058b <+46>:    mov    $0x1,%edi
   0x0000000000400590 <+51>:    mov    $0x0,%eax
   0x0000000000400595 <+56>:    callq  0x400460 <__printf_chk@plt>
   0x000000000040059a <+61>:    mov    $0x0,%eax
   0x000000000040059f <+66>:    add    $0x8,%rsp
   0x00000000004005a3 <+70>:    retq
End of assembler dump.
```

Intrinsics: code

```c
#include <xmmintrin.h>

__attribute__((noinline)) void vadd(float *a, float *b, size_t size) {
  size_t i;
  for (i = 0; i < size; i += 4) {
    __m128 v1 = _mm_load_ps(a + i);
    __m128 v2 = _mm_load_ps(b + i);
    __m128 v3 = _mm_add_ps(v1, v2);
    _mm_store_ps(a + i, v3);
  }
}

__attribute__((noinline)) void sadd(float *a, float *b, size_t size) {
  size_t i;
  for (i = 0; i < size; i++)
    a[i] += b[i];
}

int main(void) {
  size_t size = 300000000;
  float *a = (float *) calloc(size, sizeof(float));
  float *b = (float *) calloc(size, sizeof(float));

  vadd(a, b, size);
  sadd(a, b, size);

  return 0;
}
```

Intrinsics: vadd optimized disassembly

NOTE: THIS CODE IS UNSAFE (why?)

- `gcc -O3 intrin.c -o intrin`
- `gdb ./intrin`

```
(gdb) disassemble vadd
Dump of assembler code for function vadd:
   0x0000000000400580 <+0>:     xor    %eax,%eax
   0x0000000000400582 <+2>:     test   %rdx,%rdx
   0x0000000000400585 <+5>:     je     0x4005a5 <vadd+37>
   0x0000000000400587 <+7>:     nopw   0x0(%rax,%rax,1)
   0x0000000000400590 <+16>:    movaps (%rdi,%rax,4),%xmm0
   0x0000000000400594 <+20>:    addps  (%rsi,%rax,4),%xmm0
   0x0000000000400598 <+24>:    movaps %xmm0,(%rdi,%rax,4)
   0x000000000040059c <+28>:    add    $0x4,%rax
   0x00000000004005a0 <+32>:    cmp    %rax,%rdx
   0x00000000004005a3 <+35>:    ja     0x400590 <vadd+16>
   0x00000000004005a5 <+37>:    repz retq
End of assembler dump.
```

Intrinsics: sadd disassembly

- Try it yourself!
- Too large to fit on the screen! (why?)

Further questions

- How are parameters passed to functions?
- How are values returned from functions?
- Do all instructions take the same amount of time?
- How does caching work?
- What are the differences between ARM and x86?

Further material

- Intel syntax
- External assembly: yasm, masm, etc.
- Intel manuals: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Classes: 6.004, 6.033, 6.172

6.S096 Effective Programming in C and C++

IAP 2014