CSC2/452 Computer Organization
Assembly Language

Sreepathi Pai

URCS

September 26, 2022
Outline

Administrivia

Recap

x86-64 assembly

Programming in Assembly
Outline

Administrivia

Recap

x86-64 assembly

Programming in Assembly
Assignments and Homeworks

- Assignment #1 due today
- Homework #4 out Wed
- Review class Mon Oct 3.
- Midterm next Wed, Oct 5 (in class), all material up to this Wed.
Outline

Administrivia

Recap

x86-64 assembly

Programming in Assembly
Previously

- Integers
- Floats
- Addresses
- Instructions
- The Instruction Set Architecture
  - The programmer’s interface to the processor
Addresses: Summary

- Unsigned integers from $0$ to $2^n - 1$ where $n$ is size of address in bits
  - Usually $n = 64$ on modern systems
- Labels are addresses
- Addresses can be loaded into registers
  - leaq instruction on Intel 64 machines
- Effective addresses (the final address after any computations used to access memory) may be specified:
  - Directly (Absolute) e.g., 0x7f08e678d000 (but not in x86)
  - Indirect e.g., mov (%rbx, %rsi, 1), 1
  - Relative e.g., jmp
  - Implicit e.g., push
Instruction Encoding

- Instructions are encoded as multiword bitfields
  - On Intel 64, they can occupy more than 64 bits
  - Instruction encodings vary by processor
- They convey to the processor:
  - What operation to perform
  - What the operands (i.e. inputs and outputs) to that operation are
    - Operands can be registers, memory or constants
- In the Intel ISA, not all combinations of operands are valid
  - It is not fully orthogonal
Outline

Administrivia

Recap

x86-64 assembly

Programming in Assembly
Intel Processors have traditionally been known as x86
- 8086 (their first 16-bit processor)
- 80186, 80286
- 80386, 80486 (their 32-bit processors)
- 80586 became the Pentium, and Intel dropped the numbering scheme
  - also 32-bit
  - Courts said you couldn’t trademark numbers
- The ISA for this was usually called ‘x86’ by everybody or IA-32 (by Intel)
Going to 64-bits

- The first 64-bit version of the x86 was made by AMD
  - Was a new ISA based on x86 (much nicer!)
  - Therefore sometimes called ‘amd64’
  - Also referred to as ‘x86-64’
  - Intel calls their version (which is not exactly the same) ‘Intel 64’
  - Sometimes you will see ’x64’ to refer to this architecture
- Intel’s original proposal for a 64-bit processor was called Itanium
  - ‘IA-64’, but the ISA was not widely adopted (sold about 250K each year, EOLed 2021)
  - Don’t confuse IA-32 and IA-64 – they’re not related at all!
An overview of the Intel Manuals

- **Volume 1: Basic Architecture**
  - Overview of all the data types, instructions, etc. that a programmer needs to know (500 pages)
  - Recommend reading this

- **Volume 2: Instruction Set Reference**
  - Describes every instruction, its operands, its encoding, and semantics (2356 pages)
  - Look this up when you have to

- **Volume 3: System Programming Guide**
  - If you’re writing an OS or compiler or linker or assembler (1700 pages)

- **Volume 4: Model-specific registers**
  - MSR allows you to control processors (500 pages)
  - Put them to sleep, set their operating mode, etc.

- **You could order paper copies from Intel for free in the past**
  - You still can, though not for free
B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

![Diagram of General Machine Instruction Format]

**Figure B-1. General Machine Instruction Format**

The `pushq` instruction

```
11 0000 55 pushq  %rbp
```

- Line 11, address 0000 of `main`
- `pushq  %rbp` is encoded as 0x55
The PUSH instruction

- \%rbp is a 64-bit register
- That has Opcode 50+\(rd\)
- That's 0x50, and the 0 format
- Page 4-521 (i.e., 1235) in Vol 2

### PUSH—Push Word, Doubleword or Quadword Onto the Stack

<table>
<thead>
<tr>
<th>Opcode(^*)</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 66</td>
<td>PUSH r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF 66</td>
<td>PUSH r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF 66</td>
<td>PUSH r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64.</td>
</tr>
<tr>
<td>50+rw</td>
<td>PUSH r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r16.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r32</td>
<td>0</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r32.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r64</td>
<td>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64.</td>
</tr>
<tr>
<td>6B lw</td>
<td>PUSH imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm16.</td>
</tr>
<tr>
<td>6B id</td>
<td>PUSH imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm32.</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH CS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push CS.</td>
</tr>
<tr>
<td>16</td>
<td>PUSH SS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push SS.</td>
</tr>
<tr>
<td>1E</td>
<td>PUSH DS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push DS.</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH ES</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push ES.</td>
</tr>
<tr>
<td>0F A0</td>
<td>PUSH FS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push FS.</td>
</tr>
<tr>
<td>0F A8</td>
<td>PUSH GS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push GS.</td>
</tr>
</tbody>
</table>

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>20</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Decrement the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:
What's +rd?

- +rb, +rw, +rd, +ro — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. “+ro” is applicable only in 64-bit mode. See Table 3-1 for the codes.

- +i — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

<table>
<thead>
<tr>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
<th>Register</th>
<th>REX.B</th>
<th>Reg Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>None</td>
<td>0</td>
<td>AX</td>
<td>None</td>
<td>0</td>
<td>EAX</td>
<td>None</td>
<td>0</td>
<td>RAX</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>CL</td>
<td>None</td>
<td>1</td>
<td>CX</td>
<td>None</td>
<td>1</td>
<td>ECX</td>
<td>None</td>
<td>1</td>
<td>RCX</td>
<td>None</td>
<td>1</td>
</tr>
<tr>
<td>DL</td>
<td>None</td>
<td>2</td>
<td>DX</td>
<td>None</td>
<td>2</td>
<td>EDX</td>
<td>None</td>
<td>2</td>
<td>RDX</td>
<td>None</td>
<td>2</td>
</tr>
<tr>
<td>BL</td>
<td>None</td>
<td>3</td>
<td>BX</td>
<td>None</td>
<td>3</td>
<td>EBX</td>
<td>None</td>
<td>3</td>
<td>RBX</td>
<td>None</td>
<td>3</td>
</tr>
<tr>
<td>AH</td>
<td>Not encodeable (N.E.)</td>
<td>4</td>
<td>SP</td>
<td>None</td>
<td>4</td>
<td>ESP</td>
<td>None</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CH</td>
<td>N.E.</td>
<td>5</td>
<td>BP</td>
<td>None</td>
<td>5</td>
<td>EBP</td>
<td>None</td>
<td>5</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>DH</td>
<td>N.E.</td>
<td>6</td>
<td>SI</td>
<td>None</td>
<td>6</td>
<td>ESI</td>
<td>None</td>
<td>6</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BH</td>
<td>N.E.</td>
<td>7</td>
<td>DI</td>
<td>None</td>
<td>7</td>
<td>EDI</td>
<td>None</td>
<td>7</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SPL</td>
<td>Yes</td>
<td>4</td>
<td>SP</td>
<td>None</td>
<td>4</td>
<td>ESP</td>
<td>None</td>
<td>4</td>
<td>RSP</td>
<td>None</td>
<td>4</td>
</tr>
<tr>
<td>BPL</td>
<td>Yes</td>
<td>5</td>
<td>BP</td>
<td>None</td>
<td>5</td>
<td>EBP</td>
<td>None</td>
<td>5</td>
<td>RBP</td>
<td>None</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

- +rd indicates EBP is 5
- But EBP is 32-bits!
- Page 3-2 (i.e. 106) of Vol 2, see Table 3-1
2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

- Since `PUSH` implicitly references register RSP, this makes 0x55 reference `%rbp`, not `%ebp`
  - In 64-bit mode, you can’t push EBP.
- The REX prefix of 0x48 tells the processor to use 64-bit registers
  - 12 0001 4889E5 `movq %rsp, %rbp`
  - It is not needed for `PUSH`

- Page 2-B (i.e. 44) of Vol 2 (REX prefixes)
CPU: The Decode Unit

Fetch  Decode
  AGU  L/SU
  ALU  FPU

RAM
Outline

Administrivia

Recap

x86-64 assembly

Programming in Assembly
Features of High-level Languages (HLL)

- Variables
- Arrays
- Complex Expressions (large number of operands)
  - Arithmetic Operations, Logical Operations, etc.
- Block structure - { and } in C-like languages
- Conditionals if-then-else
- Loops while, for
- Functions
Features of Assembly Languages

- Memory
- Registers
- Expressions are very simple, maybe up to 3 operands
  - Need to break up complex expressions into simple expressions
- No block structure
- No direct equivalent for conditionals
- No direct equivalent for loops
- Limited support for functions
Translating Variables

- Typically variables in programs like C end up in either:
  - Memory
  - Registers

- Memory can be seen as two logical regions – heap, and stack
  - Heap usually stores global variables and data
  - Stack usually stores (function) local variables

- In assembly language, a variable is in:
  - a register, if instruction uses a register operand, e.g. `%rax` (or any other register),
  - heap, if instruction uses an indirect memory operand, e.g. `(%rbx)`
  - stack, if an instruction uses an indirect memory operand relative to `%rbp` or `%rsp`, e.g. `-4(%rbp)`

- Accessing memory is slow, so a variable may be loaded from memory into a register before operating on it
  - In which case, for some time, it exists in both memory and registers
Example

```c
int sum(int a, int b) {
    int c = 0;
    c = a + b;
    return c;
}
```

0000000000000000 <sum>:

```
0:  55 push %rbp
1:  48 89 e5 mov %rsp,%rbp
4:  89 7d ec mov %edi,-0x14(%rbp) # A r->s
7:  89 75 e8 mov %esi,-0x18(%rbp) # B r->s
11: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) # C s
14: 8b 55 ec mov -0x14(%rbp),%edx # A s->r
17: 8b 45 e8 mov -0x18(%rbp),%eax # B s->r
19: 01 d0 add %edx,%eax # A + B
1c: 89 45 fc mov %eax,-0x4(%rbp) # store into C
1f: 5d pop %rbp
20: c3 retq
```

- **r->s**, register to stack. **s->r**, stack to register
- **rval**, return value (must be stored in %eax)
Most instructions accept only 1, 2 or 3 operands

Example: ADD instruction takes two operands

- \texttt{ADD %r1, %r2}
- \texttt{\%r2 = \%r1 + \%r2}

An expression like \( s = a + b + c \) needs to be broken down to fit this instruction

- Do this by introducing temporaries

\begin{verbatim}
  \texttt{tmp = a + b;}
  \texttt{tmp = tmp + c;}
  \texttt{s = tmp;}
\end{verbatim}
Translating Expressions: Example

0000000000000000 <sum3>:
int sum3(int a, int b, int c) {
    push %rbp
    55  # 0: 55
    48 89 e5  # 1: 48 89 e5
    mov %rsp,%rbp
    89 7d ec  # 4: 89 7d ec
    mov %edi,-0x14(%rbp) # A
    89 75 e8  # 7: 89 75 e8
    mov %esi,-0x18(%rbp) # B
    89 55 e4  # a: 89 55 e4
    mov %edx,-0x1c(%rbp) # C

    int s = 0;
    c7 45 fc 00 00 00 00  # d: c7 45 fc 00 00 00 00
    movl $0x0,-0x4(%rbp) # S

    s = (a + b + c);
    8b 55 ec  # 14: 8b 55 ec
    mov -0x14(%rbp),%edx # %edx = A
    8b 45 e8  # 17: 8b 45 e8
    mov -0x18(%rbp),%eax # %eax = B
    01 c2  # 1a: 01 c2
    add %eax,%edx # %edx = A + B
    8b 45 e4  # 1c: 8b 45 e4
    mov -0x1c(%rbp),%eax # %eax = C
    01 d0  # 1f: 01 d0
    add %edx,%eax # %eax = C + %edx
    89 45 fc  # 21: 89 45 fc
    mov %eax,-0x4(%rbp) # store into S

    return s;
    8b 45 fc  # 24: 8b 45 fc
    mov -0x4(%rbp),%eax
}

5d  # 27: 5d
c3  # 28: c3
int max(int a, int b) {
    if(a > b) {
        return a;
    } else {
        return b;
    }
}

Assembly Language Conditionals: Three components

- Comparison functions
- Conditional jumps
- Unconditional jumps
Disassembly of max

0000000000000000 <max>:  
  0:  55  push %rbp  
  1: 48 89 e5 mov %rsp,%rbp  
  4: 89 7d fc mov %edi,-0x4(%rbp)  
  7: 89 75 f8 mov %esi,-0x8(%rbp)  
 a: 8b 45 fc mov -0x4(%rbp),%eax  
 d: 3b 45 f8 cmp -0x8(%rbp),%eax  
 10: 7e 05 jle 17 <max+0x17>  
 12: 8b 45 fc mov -0x4(%rbp),%eax  
 15: eb 03 jmp 1a <max+0x1a>  
 17: 8b 45 f8 mov -0x8(%rbp),%eax  
1a:  5d pop %rbp  
1b:  c3 retq
Comparison

- We know `a` is on the stack at `-0x4(%rbp)`
- We know `b` is on the stack at `-0x8(%rbp)`
- The `cmp` instruction compares `b` with `%eax` (which contains `a`)
  - For `cmp x, y`, the `cmp` instruction calculates `y - x`
- The results of the `cmp` operation are stored in the EFLAGS register. Of relevance to `jle`:
  - ZF: Zero flag: set to 1 if `y - x == 0`
  - OF: Overflow flag: set to 1 if `y - x` underflowed or overflowed
  - SF: Sign flag: set to sign bit of `y - x`
Part of the family of \texttt{jcc} instructions
- \texttt{cc} is conditional code

Jump If Less Or Equal
- If \texttt{ZF} is 0, then $y = x$.
- If \texttt{OF} \neq \texttt{SF} then, $y < x$
  - To understand this, work out all cases of $y < x$ where they are $+/-, -/+,$ $+/-, -/+$
  - Note that at machine level, two’s complement integers “wrap-around” on overflow and underflow

Here, the conditional jump to $\texttt{max}+0\times17$ occurs if $a \leq b$
- Otherwise control “falls through” to the next instruction
Conditionals: Full translation

- Evaluate the condition and jump to the Else part
  - Or fall through to the Then part

a: 8b 45 fc  
mov -0x4(%rbp),%eax

d: 3b 45 f8  
cmp -0x8(%rbp),%eax  # COMPARISON

10: 7e 05  
jle 17 <max+0x17>  # COND. JUMP

- The Then Part

12: 8b 45 fc  
mov -0x4(%rbp),%eax

15: eb 03  
jmp 1a <max+0x1a>  # UNCOND. JUMP

- The Else Part

17: 8b 45 f8  
mov -0x8(%rbp),%eax

- Code immediately after Else

1a: 5d  
pop %rbp
Loops

int div(int a, int b) {
    int q = 0;

    while(a - b > 0) {
        a = a - b;
        q = q + 1;
    }

    return q;
}

(Ignore what this function is trying to do)
Loops: Removing Structure

```c
int div2(int a, int b) {
    int q = 0;
    goto loop_test;

    loop_body:
    a = a - b;
    q = q + 1;

    loop_test:
    if((a - b > 0))
        goto loop_body;

    loop_exit: /* not required, for clarity only */

        return q;
}
```

We can convert a while loop into *unstructured* form using goto and if
Unstructured Loop Translation

- `goto` is an unconditional `jmp`
- The `if(cond) goto` form is just a conditional jump
Translation

000000000000002c <div2>:

... 3d:  eb 0b    jmp  4a <div2+0x1e> # goto loop_test
   3f:  90    nop # loop_body:
40:  8b 45 e8  mov  -0x18(%rbp),%eax
43:  29 45 ec  sub  %eax,-0x14(%rbp)
46:  83 45 fc 01 addl  $0x1,-0x4(%rbp)
4a:  8b 45 ec  mov  -0x14(%rbp),%eax # loop_test: a - b
4d:  2b 45 e8  sub  -0x14(%rbp),%eax
50:  85 c0    test  %eax,%eax  # is a - b > 0?
52:  7f eb    jg   3f <div2+0x13> # jump if greater
54:  90    nop
55:  8b 45 fc  mov  -0x4(%rbp),%eax
58:  5d    pop  %rbp
59:  c3    retq

▶ test performs a logical and of its two operands and sets ZF and SF
▶ sub sets the OF flag
▶ jg jumps if ZF=0 and SF=OF
int iter(int a, int b) {
    int i;

    for(i = 0; i < 10; i++) {
        a = a + b;
    }

    return i;
}

De-structuring for loops

```c
int iter2(int a, int b) {
    int i;

    loop_head:
        i = 0;
        goto loop_test;

    loop_body:
        a = a + b;

        i++;  /* loop update */

    loop_test:
        if(i < 10)
            goto loop_body;

    loop_exit:
        return i;
}
```
Translating for loops

0000000000000028 <iter2>:

... 32: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) # i = 0
    39: eb 0b jmp 46 <iter2+0x1e> # goto loop_test
3b: 90 nop # loop_body:
3c: 8b 45 e8 movl -0x18(%rbp),%eax
3f: 01 45 ec addl %eax,-0x14(%rbp) # a = a + b
42: 83 45 fc 01 addl $0x1,-0x4(%rbp) # i++
46: 83 7d fc 09 cmpl $0x9,-0x4(%rbp) # loop_test:
4a: 7e ef jle 3b <iter2+0x13> # if(i < 10) goto loop_body
4c: 90 nop
...
Basics of translating HLLs to Assembly (so far)

- Simplify expressions
- Find locations for variables
- Destructure loops
  - Use conditional and unconditional jumps
Handling Function Calls

- How to pass arguments to function?
- How to jump to a function?
- How to come back to just after call location?
  - How does `ret` know where to return to?
- How to receive the return value from a function?
References

▶ Read Chapter 3 of the textbook
  ▶ Esp. the Figure detailing all the registers