CSC2/452 Computer Organization
Addresses and Instructions

Sreepathi Pai

URCS

September 18, 2019

Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

Announcements

- Homework #2 due today IN CLASS.
- Assignment #1 out
  - Due next week, Friday Sep 27.
- Homework #3 will be out Thursday
  - Due next Wed as usual

Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

Bits

- Bits: 0 and 1
  - Bit operations: AND, OR, NOT, XOR, etc.
- Data structures made of bits
  - Bitsets (also called Bitmaps, Bitstrings, Bitvectors, etc.)
  - Bitfields
  - Bitwise operations: AND, OR, NOT, XOR, ...
  - Primary operations: setting bits, extracting bits, shifting bits
- Applications
  - Representing integers (bitfield of sign + value)
  - Representing floats: single-precision, etc.

What is this bitstring?

1100 0000 1000 1001 0000 1111 1101 1100

- Hexadecimal value: `0xc0890fdb`
- A: The 32-bit unsigned integer: $3,226,013,659$
- B: The 32-bit signed two's complement integer: $-1,068,953,637$
- C: The 32-bit IEEE754 single-precision float: -3.141593
- D: All of the above
- E: None of the above

Bits are in the eye of the beholder

- If the ALU consumes the bits, it is interpreted an integer
  - either signed or unsigned depending on the context (more later)
- If the FPU consumes the bits, it is interpreted as a float
- Bits themselves do not encode what they mean
  - Powerful idea
  - Good sometimes, mostly bad?
- High-level languages introduce the notion of *type safety* to maintain logical sanity
  - Roughly, track what the bits mean and prevent invalid operations

Basic Machine Level Data “Types”

- Integers
- Floats
- Addresses
- Instructions

Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

Addresses

- An address is ultimately an unsigned integer
- It represents a location in memory (i.e. RAM)
- Addresses on a machine are usually of a fixed size
  - 16-bit (a long time ago, and in some very small machines these days)
  - 32-bit (remarkably common until about a decade ago)
  - 64-bit (most common today)

The Size of an Address

- Addresses run from 0 to $2^n - 1$ where n is 16, 32, or 64-bit
  - 16: 65,536
  - 32: 4,294,967,296
  - 64: 18,446,744,073,709,551,616
- If machines are byte addressable (as most are), this means the *maximum* size of addressable memory is:
  - 16: approx. 64 Kilobytes
  - 32: approx. 4 Gigabytes
  - 64: approx. 18 Exabytes
- Most *single* computers these days can accommodate upto a few terabytes
  - Intel and AMD 64-bit hardware only supports 48 bits (about 256 Terabytes)

Creating Addresses

- Addresses are just integers *at the machine level*
  - High-level languages try very hard to prevent integers and addresses from mixing
- Two primary uses of addresses
  - Address of code
  - Address of data
  - No way to distinguish between these uses by looking at only the address without additional info
- Three primary address consuming units inside a CPU
  - The *instruction fetch* unit treats address as address of code
  - The *load/store* unit treats address as address of data
  - The *address generation* unit performs simple operations on addresses

Addresses in Assembly: Absolute Addresses

```
call    printf        /* printf is an absolute address
                      of the printf code */
```

- (Recall that labels are just human-readable addresses)
- Here the address of `printf` would be filled in by the linker
- It is a full 64-bit address
- This is calling the `printf` function by providing its address
- In machine code, this is represented as e8 00 00 00 00, where the zeroes are filled in with the 64-bit address

Addresses in Assembly: Indirect Addresses

```
call    *%rdx
```

- Here, `%rdx` is a 64-bit register containing an address
  - We'll talk about registers later, but assume it is a memory location
  - Or, in high-level language terms, it is a variable
- This is calling *whatever* is at the address contained in `%rdx`
- In machine code, this is represented as `ff d2`, note no address
- The `*%rdx` syntax is AT&T, you may also see `call [rdx]` in Intel syntax

Operations on Indirect Address Operands

- An *effective address* (EA) is the final address formed from an indirect address and (optional) various indexing and scaling operations.

Expression	Example	Meaning
%reg	%rdx	EA is in %reg
disp(%reg) displacement is a 8/16/32 bit signed integer	-4(%rbp)	EA is %reg + disp
(%basereg, %indexreg, scale) scale can only be 1, 2, 4, or 8	(%rbx,%rsi,4)	EA = %basereg + %indexreg * scale
disp(%basereg, %indexreg, scale)	-8(%rbx,%rsi,4)	EA = %basereg + %indexreg * scale + disp

- These are Intel 64 assembly language *addressing modes*, represented in AT&T syntax

Computing on Addresses

```
.LC1:
    .string    "Hello, the value of pi is %f\n"

    ...
    leaq     .LC1(%rip), %rdi
    ...
```

- `.LC1` is a label (i.e., an address)
- `leaq` is *Load Effective Address Quadword*
  - Quadword is 64-bit
- Compute `%rip + .LC1` and store it in `%rdi`
  - After assembly, `.LC1` is `0x99`
  - So the machine code looks like `leaq 0x99(%rip), %rdi`
  - During execution, `%rdi = %rip + 0x99`
  - This calculation performed by the address generation unit (AGU) [not the ALU]

Base, Index, Scale and Displacement

- The `disp(%basereg, %indexreg, scale)` addressing mode is most helpful for calculating addresses of array elements

```
/* pseudocode to find address of array element 5 */
/* array Arr contains doubles, recall doubles are 64-bit*/
array_base = Arr
index = 5
address_of_fifth = array_base + index * 8
```

- In pseudo-assembly:

```
/* moving absolute address into %rbx */
leaq Arr, %rbx

movq $5, %rsi

/* compute address of fifth element */
leaq (%rbx, %rsi, 8), %rdx

/* (contd. on next slide */
```

Loading data using an effective address

```
/* %rdx has address of fifth element, load it into %rax */

movq (%rdx), %rax
```

- The `mov` instruction is the primary way to load or store data on Intel 64
  - Note: It copies data, does not move it
- `mov` as several forms
  - `mov %REG1, %REG2`, copies data in `%REG1` to `%REG2`
  - `mov IMM, %REG`, copies constant `IMM`ediate to `%REG`
  - `mov EA_EXPR, %REG`, load data from effective address `EA_EXPR` into `%REG`
  - `mov %REG, %EA_EXPR`, store data from `%REG` to effective address computed by `EA_EXPR`
- `EA_EXPR` are all the forms we saw previously: e.g. `0x99(%rip)`, etc.
- Accesses to memory are performed by the load/store unit

Addresses for data: Data size

- Addresses do not contain information about the *size* of data they address
- It is possible to use the same address x to read:
  - 1 byte stored at x
  - 2 bytes at x and $x + 1$
  - 4 bytes at $x, x + 1, x + 2, x + 3$
  - 8 bytes at $x, x + 1, x + 2, x + 3, ..., x + 7$
- In AT&T syntax, the size of data is indicated by a suffix on the instruction:
  - `movq` (quad, 64-bit integer)
  - `movl` (long, 32-bit integer, 64-bit floating point)
  - `movw` (word, 16-bit integer)
  - `movb` (byte, 8-bit byte)
  - `movs` (single, 32-bit float)

Addresses for data: Alignment

- An address is set to be aligned to be x if it is divisible by x
- For example,
  - All even addresses are aligned to 2
  - Address 0x44 is aligned to 4 (lowest two bits are zero)
- On many machines, data of size n can only be read if it stored at address x aligned to n
  - No such general alignment requirement on Intel/AMD machines
  - Some instructions may have these requirements on Intel/AMD
- Attempting to read misaligned data is
  - slower on Intel/AMD
  - can cause errors on other processors

Relative Addresses

```
610:    eb 06                   jmp    618 <main+0x1e>
```

- Recall output of `objdump -S`
  - Columns contain: address, machine code, disassembly
  - `jmp` instruction at address `0x610`
  - encoded as two bytes `eb 06`
  - `jmp` to address 0x618
- Is address 0x618 stored in the machine code?

Relative Addresses (contd.)

- Some addresses can be expressed as relative to an *implicit* register
  - usually `%rip`
- In the previous slide, *after* reading the `jmp` instruction
  - `%rip` is `0x612`
- The instruction contains an offset (`0x6`) that is added to `%rip`
- Sometimes called a short jump or a near jump

Implicit Addressing

```
pushq %rbp
```

- Some instructions use *implicit* indirect addresses
- On x86, notably `pushq`, `popq`, `leave`
  - These store values on the function stack in memory
- The address of the function stack is given by register `%rsp`
- Executing `pushq %rbp` is equivalent to:

```
%rsp = %rsp - 8     /* note stack grows towards lower address */
(%rsp) = %rbp       /* note (%rsp) is indirect memory address */
```

Hello Pi, again

```
        .file   "hellopi.c"
        .text
        .section        .rodata
 .LC1:
        .string "Hello, the value of pi is %f\n"    LC1 is address of this string
        .text
        .globl  main
        .type   main, @function
main:                                               main is address of the next instruction
 .LFB0:                                             so is .LFB0
        pushq   %rbp                                store the value of %rbp at (%rsp)
        movq    %rsp, %rbp                          copy (NOT move) the value of %rsp into %rbp
        subq    $16, %rsp                           subtract 16 from %rsp
        movss   .LC0(%rip), %xmm0                   load single at .LC0(%rip) into %xmm0
        movss   %xmm0, -4(%rbp)                     store single in %xmm0 into -4(%rbp)
        cvtss2sd -4(%rbp), %xmm0                    convert single at -4(%rbp) into double in %xmm0
        leaq    .LC1(%rip), %rdi                    load address of string into %rdi
        movl    $1, %eax                            set %eax to 1
        call    printf@PLT                          call printf
        movl    $0, %eax                            set %eax to 0
        leave                                       set %rsp to %rbp, then popq %rbp
        ret                                         exit the function
        .section        .rodata
        .align 4                                    make sure .LC0 is aligned to 4
 .LC0:
        .long   1078530011                          this is 0x40490fdb, i.e. 3.141593
```

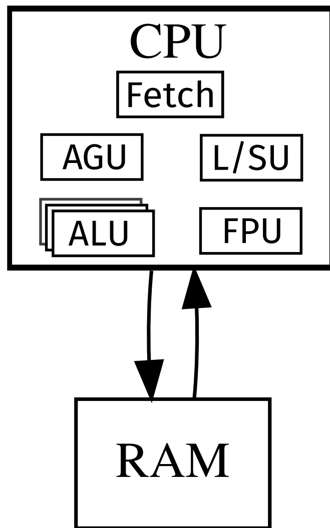
Unexplained stuff (for now)

- Standard x86 function prologue and epilogue
  - `push %rbp, %rsp` (we know what it does, but why?)
  - `subq $16, %rsp`
  - `leave`
- C translation details
  - `cvtss2sd -4(%rbp), %xmm0` (why convert to double?)
  - `movl $1, %eax`
- Details on these as we study translating C to assembly in later classes

Food for thought

Why do we need so many ways to specify an address?

The Computer (Updated)



Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

Instructions

- "Commands" to the processor (e.g. CPU)
- Part of Instruction Set Architecture
  - *Programmer's interface to a processor*
  - Instructions processor understands
  - Data types processor understands
  - Other processor services and functionality [later in the course]

Design Questions

- Which instructions should a processor support?
- Where are inputs and outputs for instructions stored?

Functionality

- Bitwise manipulation instructions
- Integer arithmetic
- Floating point arithmetic
- Address manipulation instructions
- Memory load/store instructions
- Function call instructions
- Stack manipulation instructions
- Cryptography instructions?
- Video encoding/decoding instructions?
- ...

CISC vs RISC

- Complex Instruction Set Computer (CISC)
  - Supports many instructions
- Reduced Instruction Set Computer (RISC)
  - Supports a limited number of instructions

CISC vs RISC (contd.)

CISC	RISC
Easier to program by hand	Easier for compilers to program
Complex circuit implementation	Simpler circuit implementation
Can be made fast	Can be made fast (usually easier)
VAX, Intel x86	MIPS (most famous), ARM, RISC-V, ...

- Distinction is somewhat meaningless now
  - Unlikely it ever was
- Intel x86 is internally RISC
  - Instructions you use are translated into RISC-like micro-instructions
- ARM (Acorn RISC Machines) is hardly RISC anymore
  - Lots of instructions supported

Operands

- Instructions operate on *operands*
- Operands can be stored in:
  - Registers
  - Memory
- Should all instructions be allowed to access both kinds of operands?

Load/Store architectures

- Should all instructions be allowed to access both kinds of operands?
- Yes
  - All instructions can access both memory and register operands
  - Mostly CISC
- No
  - Only load/store instructions can access memory and registers
  - All other instructions can access only registers
  - Mostly RISC
- VAX
  - Fully orthogonal architecture
  - Instructions and operand accesses are independent
  - Makes it much easier to program
- x86
  - Most instructions can access both memory and registers
  - But not orthogonal

Designing a Instruction Encoding

- Consider a load/store architecture processor
- 128 instructions
- Supports 64-bit addresses
- Has 32 registers

Encoding

OP RS1, RS2, RD

- 128 different instructions (i.e. `OP` can be 0 to 127)
  - How many bits?
- Has 32 registers
  - `RS1`, `RS2` and `RD` are all registers
  - How many bits?

Encoding (Solutions)

OP RS1, RS2, RD

- 128 different instructions (i.e. `OP` can be 0 to 127)
  - 7 bits
- Has 32 registers
  - `RS1`, `RS2` and `RD` are all registers
  - 5 bits + 5 bits + 5 bits = 15 bits
- Can store all the information we need in 22 bits
  - Maybe use a 32-bit bitfield?

Instructions as a bitfield

```
3322222222 22111 11111 11
1098765432|10987|65432|10987|6543210
------------------------------------
 unused   |  RD | RS2 | RS1 | opcode
------------------------------------
```

- Top two lines indicate bit positions
- Here is what `ADD R0, R1, R2` looks like
  - assume `ADD` is given opcode 0010001_2
  - assume `R`x is indicated by x in binary
  - i.e. `R0` is 00000_2, `R20` is 10100_2
  - set unused bits to 0

```
3322222222 22111 11111 11
1098765432|10987|65432|10987|6543210
------------------------------------
0000000000|00010|00001|00000|0010001
------------------------------------
```

Our Instruction Encoding

- Fixed-width instructions
  - All instructions 32-bit
- Not very flexible
  - Only register operands
  - Does not support advanced addressing modes
  - Can't supply constants (or *immediate* operands)

Open Design Questions

- How do we encode instructions that take memory addresses?
  - Won't fit in 32 bits
- If an operation takes a memory address, what do we do with RS1, RS2 and RD fields?
- How do we specify data sizes?

Intel x86 instruction coding

```
  7                                 .globl  main
  9                         main:
 10                         .LFB0:
 11 0000 55                         pushq   %rbp
 12 0001 4889E5                     movq    %rsp, %rbp
 13 0004 4883EC10                   subq    $16, %rsp
 14 0008 F30F1005                   movss   .LC0(%rip), %xmm0
 14      00000000
 15 0010 F30F1145                   movss   %xmm0, -4(%rbp)
 15      FC
 16 0015 F30F5A45                   cvtss2sd -4(%rbp), %xmm0
 16      FC
 17 001a 488D3D00                   leaq    .LC1(%rip), %rdi
 17      000000
```

- Variable-sized
- Registers, Memory, Constants as operands
- Advanced addressing modes
- But also implemented as a bitfield!

Intel Instruction Format

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)

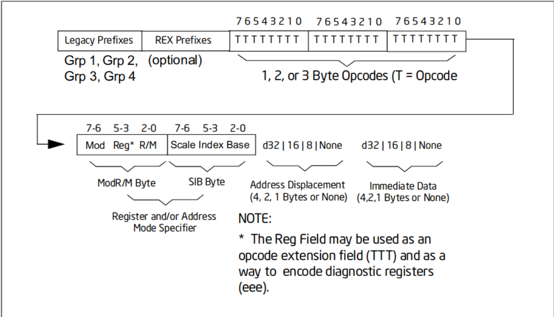


Figure B-1. General Machine Instruction Format

Source: Intel 64 and IA-32 Architectures: Software Developers Manual, Volume 2, Instruction Set Reference (A–Z), pg. 2095

References

- Chapter 3
- Intel 64 and IA-32 Architectures Software Developer's Manuals
  - approx. 6000 pages, be sure to read over weekend, quiz next class :)