CS 33 Discussion 2

April 14, 2017

Agenda

Datalab

x86 Organization

x86 Assembly

Datalab: Setup

Recommended to use SEASnet servers: 07/09

Check gcc version:

usr/local/cs/bin

PATH=$PATH:/usr/local/cs/bin

Connecting to linux server

Copying files to linux server

Datalab

Environment variables? Variables known by OS or instance of terminal

Path defines the directories in which to look for programs

Example

Datalab

Setting up the datalab:

– Download datalab zip file from CCLE.

– Copy it to a directory of your choosing.

– .tgz is a compressed file type.

– Uncompress it with the following:

– “tar -zxvf datalab.tgz”

Datalab

- Running the datalab:
  - make compile
  - ./btest check correctness
  - ./dlc -e bits.c check rules
- Check the README for instructions.

x86

x86 Organization

- We know that data is stored in memory.
- For example, if you have a 32-bit addressable space, then the addresses that are in memory range from 0x00000000 to 0xFFFFFFFF(2^31-1).
- Where are variables stored?

x86 Organization

- The variables will be stored in memory, which is a physical construct (RAM).
- However, RAM is too slow to keep up with the demands of a processor.
- Accessing RAM takes approximately 200 times the amount of time as it takes to execute a standard instruction.

x86 Organization

- Since nearly everything involves doing some operation on a variable, we need some way of accessing memory at a speed that is comparable to the speed that it takes to execute the average instruction.
- We need caches and virtual memory.
- For now, let us focus on “registers”.

x86 Organization

- Registers are extremely small physical containers that each store a number of bits.
- A 64-bit addressable machine will have registers that are 64-bits.
- In such a case, each register holds 8 bytes and the access time is extremely quick.
- When a program needs to work on a piece of data, it will bring it into a register first.

x86 Register Basics

- x86-64 contains 16 general purpose registers.
- They are identified by a short name such as (%rax, %rsi, %r8, etc.)
- In the interest of being able to access each register at a finer grain, there are multiple ways of accessing the data within each register.
- Ex. %rax refers to the full 64-bits stored in the register while %eax refers to the lower 32-bits.

x86 Register Basics

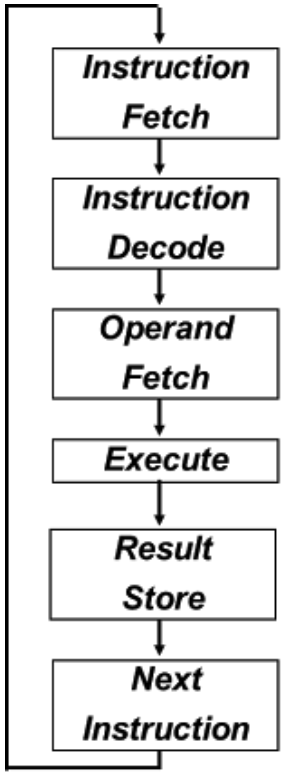
- _h: upper 8-bits of lower 16-bits
- _l: lower 8-bits
- _x :lower 16-bits.
- e_x : lower 32-bits (e stands for 'extended').
- r_x : full 64-bit register

x86 Basics

[operation] [source] [destination]

- movq %rax 4(%rbx)
- addq %rbx %rcx

x86 Instruction Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

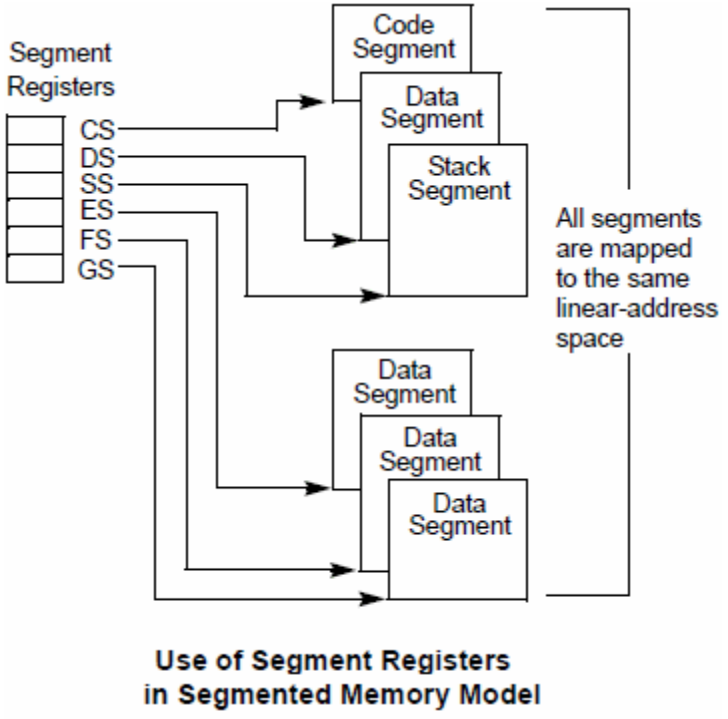
Compute result value or status

Deposit results in storage for later use

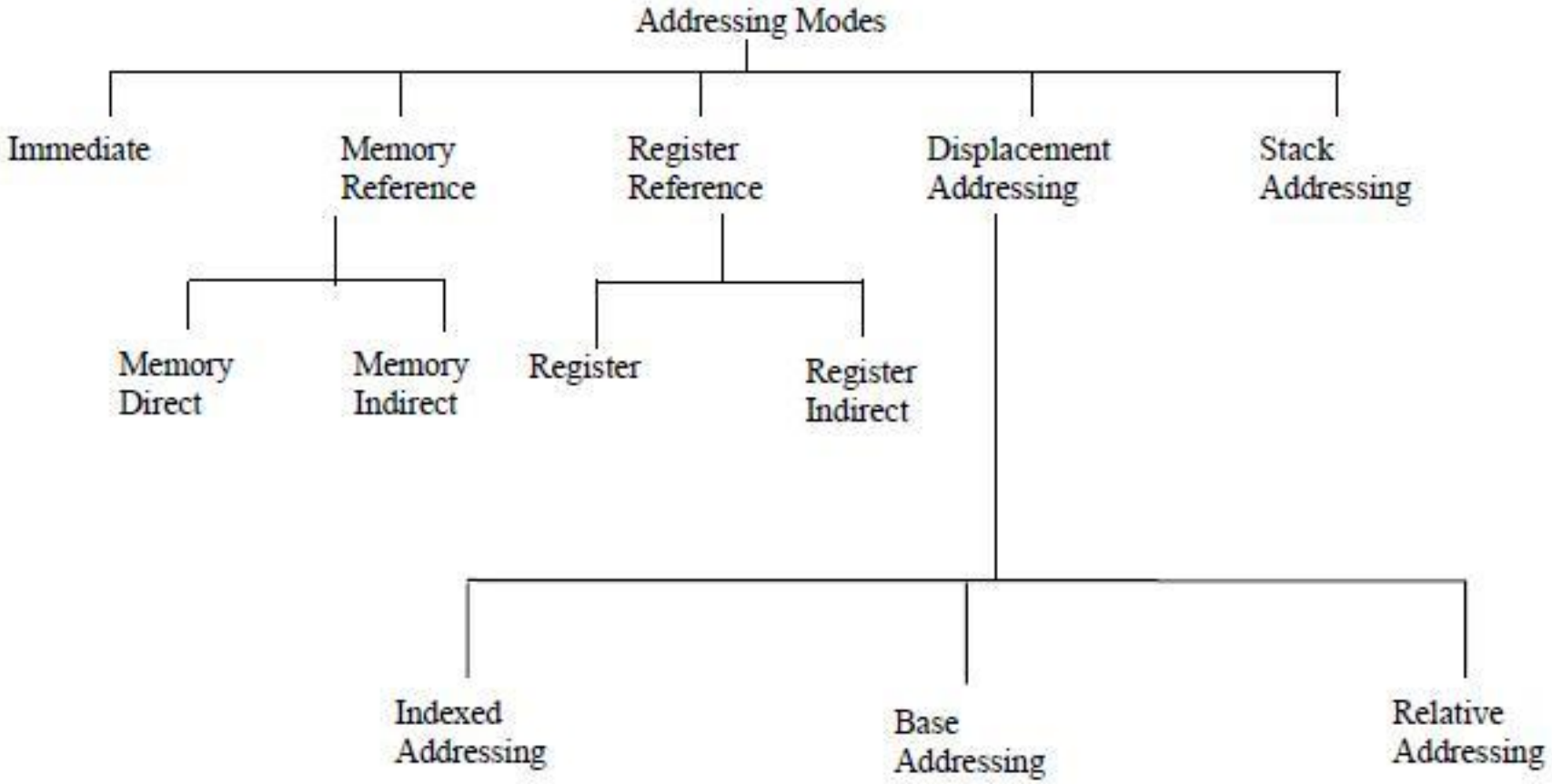
Determine successor instruction; can generally be combined w/ Decode

x86 Addressing

$$\left\{\begin{array}{l}\texttt{CS}:\\ \texttt{DS}:\\ \texttt{SS}:\\ \texttt{ES}:\\ \texttt{FS}:\\ \texttt{GS}:\end{array}\right\}\left[\left\{\begin{array}{l}\texttt{EAX}\\ \texttt{EBX}\\ \texttt{ECX}\\ \texttt{EDX}\\ \texttt{ESP}\\ \texttt{EBP}\\ \texttt{ESI}\\ \texttt{EDI}\end{array}\right\}+\left[\left\{\begin{array}{l}\texttt{EAX}\\ \texttt{EBX}\\ \texttt{ECX}\\ \texttt{EDX}\\ \texttt{EBP}\\ \texttt{ESI}\\ \texttt{EDI}\end{array}\right\}*\left\{\begin{array}{l}1\\ 2\\ 4\\ 8\end{array}\right\}+[\text{displacement}]\right]\right]$$



Use of Segment Registers in Segmented Memory Model



Common Addressing Modes

x86 Indirect Addressing Modes

Mode	Intel	AT&T
Immediate	MOV EAX, [0100]	movl 0x0100, %eax
Register	MOV EAX, [ESI]	movl (%esi), %eax
Reg + Off	MOV EAX, [EBP-8]	movl -8(%ebp), %eax
R*W + Off	MOV EAX, [EBX*4 + 0100]	movl 0x100(,%ebx,4), %eax
B + R*W + O	MOV EAX, [EDX + EBX*4 + 8]	movl 0x8(%edx,%ebx,4), %eax

x86 Addressing Modes

- Say %rax = 0xFEEDABBA and %rbx = 0x80.
- movq %rax, %rbx

– Result: %rax = 0xFEEDABBA, %rbx =0xFEEDABBA

- movq %rax, (%rbx)

– Result: the value that is located in memory address 0x80 is set as 0xFEEDABBA.

– In a more C-like form, this is essentially:

– MEM[0x80] = 0xFEEDABBA; or

– *(0x80) = 0xFEEDABBA;

x86 Assembly - Data Movement

Given assembly code:

```
1    movabsq    $0x0011223344556677, %rax
2    movb       $-1, %al
3    movw       $-1, %ax
4    movl       $-1, %eax
5    movq       $-1, %rax
```

Contents of rax at each step?

x86 Assembly - Data Movement

Given assembly code:

```
1    movabsq    $0x0011223344556677, %rax
2    movb       $0xAA, %al
3    movsbq     $0xAA, %rax
4    movzbq     $0xAA, %rax
```

Contents of rax at each step?

x86 mov_

- It is possible to move from a smaller container to a larger container.
- Assume that %dh = 0xCD, %eax = 0x98765432
- movb %dh, %eax

– What's the result? It's not allowed (suffix mismatch).

The destination has a 32-bit length while the mov**b** expects only to move a byte.

- movb %dh, %al

– Result: %eax = 0x987654CD

x86 mov_

The size of the prefix must match the operands. You cannot have:

– movl %ax, (%esp) // Cannot move a 32-bit quantity from a 16-bit register

– movl %eax, %dx // Cannot move a 32-bit quantity into a 16-bit register.

Additionally memory references match all sizes:

– movb %al, (%rbx)

– movw %ax, (%rbx)

– movq %rax, (%rbx)

– All allowed. The data will be moved to memory starting at that address based on the data type size.

x86 Address Resolution

Given assembly code:

```
1       leaq (%rdi, %rsi, 4) , %rax
2       leaq (%rdx, %rdx, 2) , %rdx
3       leaq (%rax, %rdx, 4) , %rax
```

Address given to rax at the end?

Initial values:

%rdi - x

%rsi - y

%rdx - z

x86 Control Flow

Basics:

CF: carry flag
ZF: zero flag
SF: sign flag
OF: overflow flag

x86 Control Flow

cmpq %rsi, %rax

- Compares values in rax and rsi
- Plays with status register bits (flags)
- Current state of flags used to determine flow of control

Control flow ops:

- sete, sets, setg, setl, seta, setae, setb, setbe
- je, jne, js, jns, jg, jge, jl, jle, ja, jae, jb, jbe

X86-64 cheat sheet

https://www.systems.ethz.ch/sites/default/files/file/COURSES/2014%20FALL%20COURSES/2014_Fall_SPCA/lectures/x86_64_asm_cheat_sheet.pdf

Tips and Tricks

Access specific bits

Can also be used for hex values. 0xF is 1111.

```
      0x11223344
&     0x00FF0000
    ----------
      0x00220000
```

How you can use XOR

- Parity Determiner:
  - b0, b1, b2, … bn are bits
  - b0^b1^b2^...^bn
    = 1 if there are an odd number of bits in the input
    = 0 if there are an even number of bits in the input.
- Equality Comparator:
  - x and y are bit vectors
  - x^y
    = 0 if x == y
    = non-zero if x != y

How you can use XOR

- Conditional Inverter
  - x is a bit vector
  - Say you have bit mask cleverly called “mask” that is either all 1's or all 0's
  - if(mask == 111...111)
    return ~x;
  - else
    return x;
- One way:
  - (mask & (~x)) | ((~mask) & x)

Signed Overflow

Signed overflow is considered “undefined”. This includes overflow caused by shifts.

Consider 4-bit signed:

1010 (-6) << 1 ?= 0100 (4)

This may be the result, but technically it's undefined.

Undefined Behavior

– When behavior is undefined, that means a compiler writer or implementation of a language is free to do whatever it wants if that behavior occurs.
– If you write code that expects an undefined behavior to act in some way (for example, if you assume that signed overflow wraps), then if you take this code and compile it with a compiler that implements different behavior, the program is wrong.

Variables in Memory

- Variables have memory addresses.
- Hence, pointers.
- char c = 0xBA;
- &c would be the address that c is stored in.
- Typically, memory is byte-addressable, which means that for each memory address, a single byte is stored there.
- Say &c = 0x100. If you looked into memory address 0x100, you'd find 0xBA.

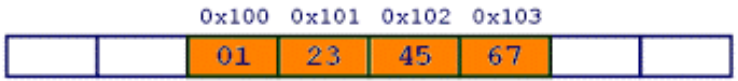
Variables in Memory

- Say &i is at address 0x100.
- Address 0x100 can only fit one byte. How can int i be stored at that address?
- In order to contain the four bytes of the int, we require four different addresses.
- Since &i begins at address 0x100, we will also need addresses 0x101, 0x102, and 0x103.
- But which byte belongs where?

Little and Big Endian Mystery

In little endian machines, last byte of binary representation of the multibyte data-type is stored first.
On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes
(For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes)
then a variable x with value 0x01234567 will be stored as following.



Big Endian

Little Endian

- Big Endian

Addr	0x100	0x101	0x102	0x103
Value	FE	ED	BA	CC

- Little Endian

Addr	0x100	0x101	0x102	0x103
Value	CC	BA	ED	FE

Variables in Memory

- Say you have the following:
- int i = 0x11223344;
- int j = i & 0xFF;
- What is j if we're on a Little Endian machine?

– 0x44

- What is j if we're on a Big Endian machine?

– 0x44

Variables in Memory

- When you use operations at the C level, you're asking the compiler to do a set of bitwise operations on numerical values.

- The endianness does NOT influence the operations. Endianness only affects how the values are stored in memory.

- As a result, when you do a non-memory operation such as the one in the previous slide, the endianness is already sorted out.

Problem

Given an unsigned integer of 32 bits, swap all odd and even bits.

Eg: x = 23 (00010111)

After swapping:

X <- 43 (00101011)

Solution

```
unsigned int swapBits(unsigned int x)

{   // Get all even bits of x

    unsigned int even_bits = x & 0xAAAAAAAA;

    // Get all odd bits of x

    unsigned int odd_bits  = x & 0x55555555;

    even_bits >>= 1;  // Right shift even bits

    odd_bits <<= 1;   // Left shift odd bits

    return (even_bits | odd_bits); // Combine even and odd bits

}
```

References

Slides modified from DJ Kim, UT Wang and Shikhar Malhotra

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You