CS 33:
Introduction to Computer Organization

Week 2

Agenda

- Lab 1 – datalab news, tips, tricks
- x86 Organization
- x86 Assembly
- x86 Control (part 1)

Signed Overflow: The Final Word

- I apologize for the horrors that you're about to be subjected to: excerpts from the C standard.
- Since the C standard seems to be the final say in “undefined”, etc. what does it say about when left shifting causes overflow?
  - (1 << 31) is overflow?
  - (~0 << 31) is not overflow?

Signed Overflow: The Final Word

- *And now... a selection from:*
  - Committee Draft — April 12, 2011 ISO/IEC 9899:201x
- On shifting by a negative value or a value that is more than the data width:
- “The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.”
- Makes sense. No overflow here. Simply undefined behavior.

Signed Overflow: The Final Word

- On left shifting:
- “The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 × 2^E2, reduced modulo one more than the maximum value representable in the result type. If E1 has a signed type and nonnegative value, and E1 × 2^E2 is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.”
- …?

Signed Overflow: The Final Word

- The standard clearly defines the operation from the bit perspective.
- However, it also defines the operation and potential overflow from the arithmetic perspective for:
  - unsigned integers
  - signed positive integers
- Artfully omitted and therefore technically undefined:
  - signed negative integers
- Yes. Apparently, left shifting of ANY signed negative number is, strictly speaking, undefined.
- Apparently “undefined” means it was a Friday afternoon and they were too lazy to finish.

Signed Overflow: The Final Word

- On right shifting:
- “The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2^E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.”
- Are you kidding me?

Signed Overflow: The Final Word

- Not only does this definition make it clear that right shifting a negative number is implementation defined, it does not even mention under what circumstances the shift is arithmetic or logical.
- Undefined – The implementation is allowed to do anything (crash, wrap, trap, pursue its dream of being an actor)
- Implementation Defined – The implementation can differ by machine, but it must be well documented and cannot crash.

Signed Overflow: The Final Word

- For your lab:
  - We are assuming well defined right shifting. Signed shifting will arithmetically shift regardless of if it's positive or negative.
  - For left shifting, we are currently extending the overflow rule for positive shifting to negative shifting.
    - If E1 has a signed type and negative value, and E1 × 2^E2 is representable in the result type, then that is the resulting value. Otherwise, overflow.

Signed Overflow: The Final Word

- Or at least... we were.
- The left shifting issue is likely to be a problem mostly for rotateRight.
- A solution is possible, but needlessly messy.
- At this point, due to all of the confusion and uncertainty, I am no longer sure how much we will adhere to this rule.
- As an relevant aside, why is the standard so vague and imprecise when it comes to signed shifting?

Signed Overflow: The Final Word

- As an relevant aside, why is the standard so vague and imprecise when it comes to signed shifting?
- The C standard is written to accommodate any type of machine that will want to run C.
- This includes machines that are ones complement, sign magnitude, and any other machine whose underlying organization doesn't use two's complement.
- So blame those guys.

datalab tips: howManyBits

- While the other questions of the datalab require you to operate on the binary level, howManyBits will force you to consider a few principles:
  - Doing work on multiple bits with a single instruction (ie. parallelism)
  - Doing iterations that are logarithmic rather than linear.

datalab tips: xor magic

- 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

datalab tips: xor magic

- 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)

datalab tips: xor magic

- The xor way:
  - x ^ mask

datalab tips: Overflow-less signed shifting

- Consider an arbitrary 16-bit signed vector:
  - ie 1010 1100 1101 1001
  - In this example, any sort of left shifting will cause an overflow by the extension of our relaxed overflow rules.
- What if we wanted to shift the least significant 5 bits into the most significant position.
- How can we perform this shift so that it doesn't cause overflow?

datalab tips: Overflow-less signed shifting

- Goal:
  - 1010 1100 1101 1001 => 1100 1000 0000 0000
- Shifting 1010 1100 1101 1001 left is undefined only because mathematically speaking, this will overflow (ie the bit vector multiplied by 2^anything will be too great to represent)
- What's the property that we're using that ensure we can shift?

datalab tips: Overflow-less signed shifting

- Goal:
  - 1010 1100 1101 1001 => 1100 1000 0000 0000
- Shifting 1010 1100 1101 1001 left is undefined only because mathematically speaking, this will overflow (ie the bit vector multiplied by 2^anything will be too great to represent)
- What's the property that we're using that ensure we can shift?
  - Over the course of the shift, the MSB never changes.

datalab tips: Overflow-less signed shifting

- Thus:
  - 1010 1100 1101 1001 << 11 = undefined
  - 1111 1111 1111 1001 << 11 = 1100 1000 0000 0000, defined
- Similarly:
  - 1010 1100 1101 1001 << 13 = undefined
  - 0000 0000 0000 0001 << 13 = 0010 0000 0000 0000, defined

x86 Organization

- The basic abstraction of memory that is taught in CS 31 and CS 32 is 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).
- It's not like you'll ever see a variable that you can't dereference in C. Therefore variables must always be stored in memory, right?

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, we need virtual memory, but for now, we'll 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 (which is tiny), but 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 Organization: Registers

- 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 Organization: Registers

- _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 (r stands for...? 'rextended'? I don't know, they can't all be winners)

General Purpose Registers (A, B, C and D)

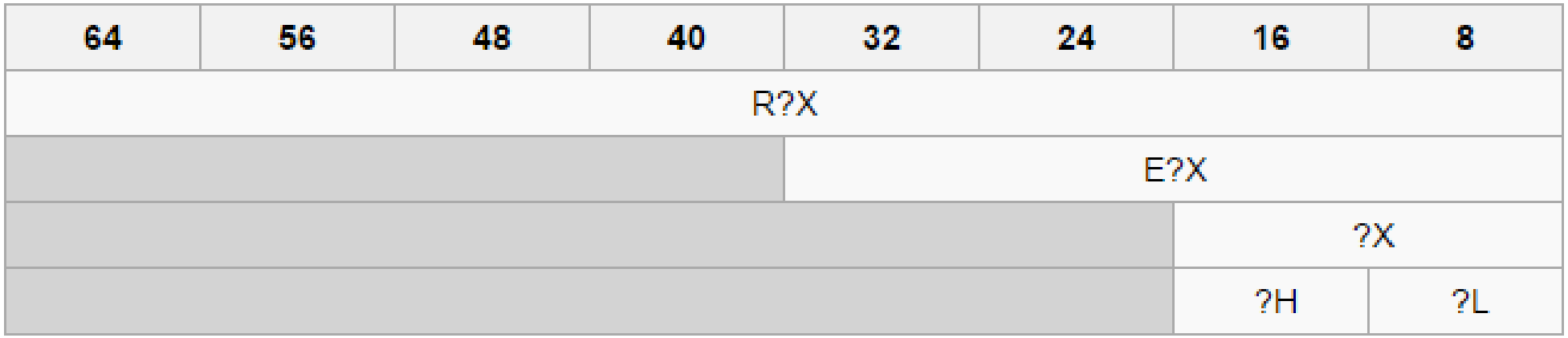


Image courtesy of Wikipedia

x86 Organization: Registers

- Registers are very simple containers that store a bit configuration and nothing more.
- If you know that a 64-bit signed long is stored in a register, there is nothing about this register that indicates that the original intention was for this value to be signed.
- The compiler will simply compile machine code that treats the bit vector in the register as if it were a signed value.

x86 Assembly

- AT&T's (also GNU/GAS) x86 notation (the notation that we will use):
- [op] [src] [dst]
- “Fun” fact: Intel's x86 notation:
- [op] [dst] [src]
- If you need to take CS M151B, you'll be learning about MIPS which has notation similar to Intel's.
- Why do I bring this up?

x86 Assembly

- What happens when we look up a plain old assembly instruction which, as I understand, is something that all the young people are doing.
- Some resources online will quietly show example of GNU/AT&T order while other will show Intel order.
- From my experience as a poor, beleaguered CS33 student, I can tell you that this is confusing as hell if you're not aware of the differences.

x86 Assembly

- What does this (GNU/GAS/AT&T) syntax look like?
- [operation] [source] [destination]
- ex.
- movq %rax 4(%rbx)
- addq %rbx %rcx

x86 Assembly: Ops

- The mov_ family: move data from the source to the destination. The suffix determines how much data to move.
  - movb : move a byte
  - movw : move a word (16 bits (?))
  - movl : move a long/double word (32 bits)
  - movq : move a quad word (64 bits)
- movq %rax, %rbx
- movq %rax, (%rbx)

x86 Assembly: Ops

- A comment regarding “word” size.
- A “word” is just a label of convenience that is used refer to contiguous bytes of memory of a common size.
- On a 32-bit machine, a word is 4 bytes. On a 64-bit machine, a word is 8 bytes.
- But back when registers were only 16-bits and x86 was being developed, “words” were 16-bits.
- Dark days...

x86 Assembly: Addressing Modes

- Consider:
  - movq %rax, (%rbx) <--- ?
- The parentheses indicate a memory operation.
- That is, the source and destination operands are able to refer to values that are located in memory, rather than just registers.
- The parentheses () means:
  - Treat the bit vector within as a memory address.
  - Go follow that address into memory and get the value at that address.

x86 Assembly: 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: Addressing Modes

- The '$' symbol prefix indicates an “immediate” which is constant number value.
- If %rax = 0xb1ab
- movl $0xdea1, %rax
  - Result: %rax = 0xdea1

x86 Assembly: Basic insns

- Other instructions:
- add_ src, dst # dst = dst + src
- sub_ src, dst # dst = dst - src
- neg_ dst #dst = -dst
- not_ dst #dst = ~dst
- sar imm, dst # dst = dst >> imm (shift arithmetic right)
- Etc... (pg. 192)

x86 Assembly: Advanced Addressing Modes

- IMM(R1, R2, S) : Scaled and displaced array access.
  - Intended usage:
  - R1 : Base array address
  - R2 : Index into array
  - S : Size of array data type in bytes
  - IMM : Displacement
- movq A(B, C, D), %rax
  - %rax = *(A + B + C*D)
- See pg 181 for full list

x86 Assembly: Advanced Addressing Modes

- D(R1) : Base + displacement addressing
  - If %rax = 0x10.
  - movq 8(%rax), %rbx
    - Result: %rbx gets the value at memory address 0x10 + 8 = 0x18, *not* the number 0x18
    - %rbx = *(%rax + 8).

x86 Assembly: Advanced Addressing Modes

- (,R2,S) : Scaled indexing, s must be 1, 2, 4, or 8.
  - If %rax = 0x01 and S = 2.
  - movl (, %rax, 2), %rcx
    - Result: The value at memory address 0x01 * 2 is saved to %rcx.
- IMM(R1, R2, S) : Scaled and displaced array access.
  - If %rax = 0x400, %rbx = 2, S = 2, and D = 20.
  - movl 0x20(%rax, %rbx, 2), %rcx
    - Result: The value at memory address 0x400 + 0x2*2 + 0x20 is placed in %rcx.

x86 Assembly: lea_

- “lea_” or the equally-as-confusing full name “load effective address” is an unusual instruction that slightly violates the normal behavior.
- movq (%rax), %rbx => The value at memory address contained by %rax is moved to %rbx.
- leaq (%rax), %rbx => The value in %rax itself is moved to %rbx.
- What's the point of this?

x86 Assembly: lea_

- Consider:
- movq 4(%rax,%rbx, 2), %rcx.
  - This means %rcx = MEM[%rax + %rbx * 2 + 4]
- leaq 4(%rax,%rbx, 2), %rcx.
  - This means %rcx = %rax + %rbx * 2 + 4
- It's just a shortcut. This is a faster way to do address calculation or just to do arithmetic.

x86 Assembly nuances: 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?

x86 Assembly nuances: 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
- However, you may want to extend the value in the source (ie %eax = 0xCD).

x86 Assembly nuances: mov_

- movsXY : move and sign extend from size X to size Y.
  - Ex: movsbl : move a byte from the source and sign extend it to a long (4 bytes)
- movzXY : move and zero extend from size X to size Y.
  - Ex: movzbw : move a byte from the source and zero extend it to a word (2 bytes)

x86 Assembly nuances: mov_

- %dh = 0xCD, %eax = 0x98765432
- movsbl %dh, %eax
  - Result: %eax = 0xFFFFFFCD
- movzbl %dh, %eax
  - Result: %eax = 0x000000CD
- movzbw %dh, %eax?
  - Result: Not allowed. Sign extending to w (16-bits) but to %eax (32-bit container).

x86 Assembly nuances: mov_

- As a final note about mov, the size of the prefix must match the operands. You cannot have:
  - movl %ax, (%esp) // Can't move a 32-bit quantity from a 16-bit register
  - movl %eax, %dx // Can't move a 32-bit quantity into a 16-bit register.

x86 Assembly nuances: mov_

- 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
- However:
  - movb %al, (%ebx)
  - Is not meaningful on x86-64 because the %ebx register is only the lower 32-bits while addresses are 48-bits.

x86 Assembly nuances: mov_

- Consider the following assembly snippet:
  - movq %rsi, %rax
- What can we say about the data type stored in %rsi (ie, pointer? value? signed? etc.)

x86 Assembly nuances: mov_

- Consider the following assembly snippet:
  - movq %rsi, %rax
- What can we say about the data type stored in %rsi?
  - From this, not much.
  - Because each register simply holds a bit vector, it is difficult to make assumptions about what the original value is without some context.
- For example, if the previous line was:
  - movq (%rsi), %rbx
- It's probably safe to assume that %rsi contains a pointer

x86 Assembly nuances

- As a curiosity, let's look at what potentially undefined behavior is implemented in the SEASnet x86-64.
- Consider:

```
int undef0(int x)
{
  return x << 31;
}
```

- In our relaxed overflow definition, if x is anything other than 0 or -1, this is an overflow.
- What does gcc undef.c (no optimization, no wrapv, etc)?

x86 Assembly nuances

```
Dump of assembler code for function undef0:
   0x0000000000400747 <+0>:     push   %rbp
   0x0000000000400748 <+1>:     mov    %rsp,%rbp
   0x000000000040074b <+4>:     mov    %edi,-0x4(%rbp)
   0x000000000040074e <+7>:     mov    -0x4(%rbp),%eax
   0x0000000000400751 <+10>:    shl    $0x1f,%eax
   0x0000000000400754 <+13>:    pop    %rbp
   0x0000000000400755 <+14>:    retq
End of assembler dump.
```

- What's happening here?

x86 Assembly nuances

```
1. push   %rbp
2. mov    %rsp,%rbp
3. mov    %edi,-0x4(%rbp)
4. mov    -0x4(%rbp),%eax
5. shl    $0x1f,%eax
6. pop    %rbp
7. retq
```

- What's happening here?
  1/2: Stack maintenance
  3. %edi contains x. x is moved to memory
  4. x is moved from memory into eax
  5. The value in eax is shifted left by 31 and stored in eax

  The end.

x86 Assembly nuances

- The effects of x86 instructions are well defined and thus, once you look at code from the assembly level, you can know exactly what's going on.
- Here, x86 behaves the way we'd think that the shift operation should behave. It simply shifts without worrying about the mathematical value.
- This is the behavior that the gcc/x86 chooses to implement during this type of overflow.

x86 Assembly nuances

- Consider something simpler:

```
int undef1()
{
  return (1 << 31);
}
```

- It's compiled into this:

x86 Assembly nuances

```
Dump of assembler code for function undef1:
   0x0000000000400756 <+0>:     push   %rbp
   0x0000000000400757 <+1>:     mov    %rsp,%rbp
   0x000000000040075a <+4>:     mov    $0x80000000,%eax
   0x000000000040075f <+9>:     pop    %rbp
   0x0000000000400760 <+10>:    retq
End of assembler dump.
```

- Even without optimization, the compiler doesn't doesn't hesitate to implement 1 << 31 without concern for overflow.
- It goes so far as to simply return the constant that it knows the shift will produce.

x86 Assembly nuances

- For the following behaviors that are undefined by the C standard, x86 seems to implement:
  - Arithmetic right shifting for signed values
  - Logical right shifting for unsigned values
  - Bitwise left shifting that simply performs a bitwise operation, regardless of the sign of the number

x86 Assembly: Special Ops

- Certain operations are expected to work with data types that are larger than a single register can contain.
- Consider the x86-64 case. We have 64-bit registers, but in order to store a 128-bit value, we can use two registers.
- Ex: %rdx contains the upper 64-bits of the value, %rax contains the lower 64-bits of the value.
- Value = %rdx : %rax, the colon means concatenation.

x86 Assembly: Special Ops

- imulq S :
  - The register %rdx contains the upper half of R[%rax] * S.
  - The register %rax contains the lower half of R[%rax] * S.
  - This is signed multiplication (the 'i' stands for '***i***nteger' as opposed to 'unsigned integer'? 's***i***gned'? Whatever, I don't care anymore.)
- clto :
  - The register %rdx contains the upper half of %rax sign extended to 128-bits.
  - The register %rax contains the lower half of %rax sign extended to 128-bits.

x86 Assembly: Special Ops

- idivq S
  - %rax contains the quotient of R[%rdx]:R[%rax]/S
  - %rdx contains the remainder of R[%rdx]:R[%rax]/S
- Note: none of these instructions technically specified %rax or %rdx. mulq and divq are single operand and clto took no operands. By convention %rax and %rdx are the registers that are used in this case.
- If you look it up, there are about million other versions of mull and div which *do* take more operands. You can look them up for “fun”.

x86 Assembly: Some Context

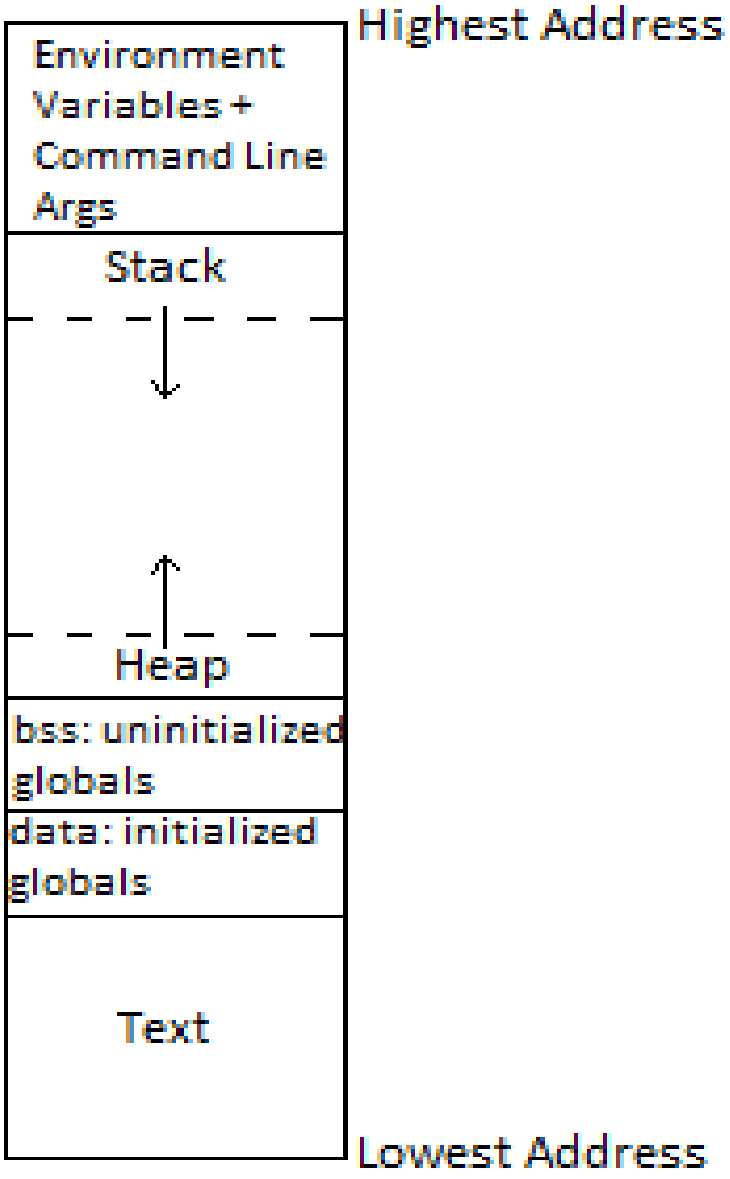
- Hopefully at this point, the basics of assembly are clear. That is, if you see a snippet of assembly, you should be able to describe the operations that are happening.
- Now, let's establish some context as to how a program is run.
- Namely, memory

x86 Assembly: Some Context

- What we know:
  - Data is stored in memory (and in registers when we need to operate on them)
- int i = 100;
- If you do &i, you may get something like 0x7FFF4980, which is it's address in memory.
- This treats memory as if it were a single construct when in fact, it can be subdivided further.

x86 Assembly: Some Context

- We'll talk about “bss” and “data” later.
- Stack: storage for local variables
  - Ex. int i = 0xb1ab.
- Heap: storage for dynamically allocated data.
- Text: The executable code of the running program.



x86 Assembly: Some Context

- The main reason I brought this up:
- The program code is stored in memory. So how does the processor keep track of where it's executing?
- Naturally, with a pointer that points to the instructions location in memory.

x86 Assembly: Some Context

- This pointer is stored in register %eip (in 32-bit) or %rip (in 64-bit).
- “ip” stands for instruction pointer. You will come to know this in CS M151B as the “PC” or program counter (which is probably confusing for multiple reasons).
- This register simply contains the address of the instruction that is to be executed (note: not the instruction itself).

x86 Assembly: Control Flags

- Let's pretend I had a segue, but we're talking about Control flags now.
- As the datalab has probably demonstrated (or will demonstrate), it would often be convenient if we could extract some information about certain values (is negative, is equal to zero, etc)

x86 Assembly: Control Flags

- Control flags do just that.
- Most operations will do the main operation but as a side effect, will also change the control flags accordingly.
- Some instructions will *only* change the control flags.

x86 Assembly: Control Flags

- Carry Flag (CF)
  - Checks if the operation caused the most significant bits to have a carry out.
  - The purpose of this is to check for unsigned overflow.
  - If t = a+b, then CF = 1 if
    - (unsigned) t < (unsigned) a
  - Set by most arithmetic instructions and some bitwise instructions, but not inc or dec.
  - Why not? Let's consult our esteemed book.

x86 Assembly: Control Flags

- “For reasons that we will not delve into, the INC and DEC instructions set the overflow and zero flag, but they leave the carry flag unchanged.”
  - Computer Systems: A Programmer's Perspective, 3nd Ed., pg. 201
- Thanks, book.

x86 Assembly: Control Flags

- Zero Flag (ZF)
  - Checks if the result of the operation is zero.
  - If t = a+b, then ZF = 1 if:
    - t == 0

x86 Assembly: Control Flags

- Sign Flag (SF)
  - Checks if the result of the operation has the most significant bit = 1.
  - This sets the flag if the number is negative
  - If t = a+b, then SF = 1 if:
    - t < 0
  - Set by arithmetic (except for mul and div), boolean/bitwise, cmp, and test.

x86 Assembly: Control Flags

- Overflow Flag (OF)
  - If t = a+b, then SF = 1 if:
    - (a < 0) == (b < 0) && (t < 0) != (a < 0)
  - OF is set if the above expression is 1.
  - This effectively checks for signed overflow.

x86 Assembly: Control Flags

- The Carry Flag detects unsigned overflow and the Overflow Flag detects signed overflow.
- Which flag should be set in the following operation:
- addl %eax, %ebx

x86 Assembly: Control Flags

- From the machine perspective, it does not do anything to distinguish between signed and unsigned add (remember, it's just a set of bits).
- Regardless of what the programmer's original intent was, both the Carry and Overflow flags will be set.
- ...it's just that depending on whether it was supposed to be signed/unsigned, one of the flags isn't going to mean a whole lot.

x86 Assembly: Control Flags

- cmp S2, S1 : Sets the flags based on S1 – S2, but doesn't change S1.
- test S2, S1 : Sets the flags based on S1&S2 but does not change S1.
  - Most often used as test %eax, %eax in order to just get the flag info of a particular value.

x86 Assembly: Control Flags

- After setting the flags, you can use them in two ways:
  1. Manually set a register based on the state of the flags.
  2. Use a conditional instruction which does something based on the state of the flags.

x86 Assembly: Control Flags

- Manually set a register based on flags with the set_ family of ops.
- sete D : D = ZF
- sets D : D = SF
- ...and etc.
- However, there are also ones that set the register based on combinations of flags that indicate other information.

x86 Assembly: Control Flags

- setg D : D = ~(SF^OF)&~ZF
  - Sets if greater than. Used in conjunction with something like: cmp %eax, %ebx. If %ebx is greater than %eax, D will be 1.
- setge D : D = ~(SF^OF)
  - Sets if greater than or equal to
- ...and so on (pg. 203)

x86 Assembly: Control Flags

- Use conditional operations:
- Conditional move (cmov_)
  - cmove S, D : move only if ZF is 1.
  - cmovs S, D : move only is SF is 1.
  - ...and so on (pg. 217). The suffixes are the same as in set_

x86 Assembly: Control Flags

- In C, we can call functions or use conditional statements to jump to other parts of code rather than executing the next instruction.
- This is accomplished by setting the %rip/%eip register to the target address.
- With assembly, this is done using the jump instruction:
- jmp Label // Unconditionally jump to the Label

x86 Assembly: Control Flags

- There are also conditional jumps:
- je, jne, etc. (pg. 206) which jumps only if the flag configuration is met.
- The suffixes also match that of set_.

End of

The Second Week

-Eight Weeks Remain-