

Introduction to Computer Organization

DIS 1A – WEEK 2

SLIDES MODIFIED FROM UT WANG

Today's Schedule

- x86 Organization
- x86 Assembly
- Lecture material review
- Lab 1 overview

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

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]

x86 Assembly

[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)
- `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.

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 Instructions

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...

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
 - $\%rax = *(A + B + C * D)$

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 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 **movb** expects only to move a byte.

- movb %dh, %al

– Result: %eax = 0x987654CD

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 = 0xFFFFFCD
- 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

x86 Assembly nuances

- 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 nuances

- `imulq S` :

- The register `%rdx` contains the upper half of $R[\%rax] * S$.
- The register `%rax` contains the lower half of $R[\%rax] * S$.

- `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. By convention `%rax` and `%rdx` are the registers that are used in this case.

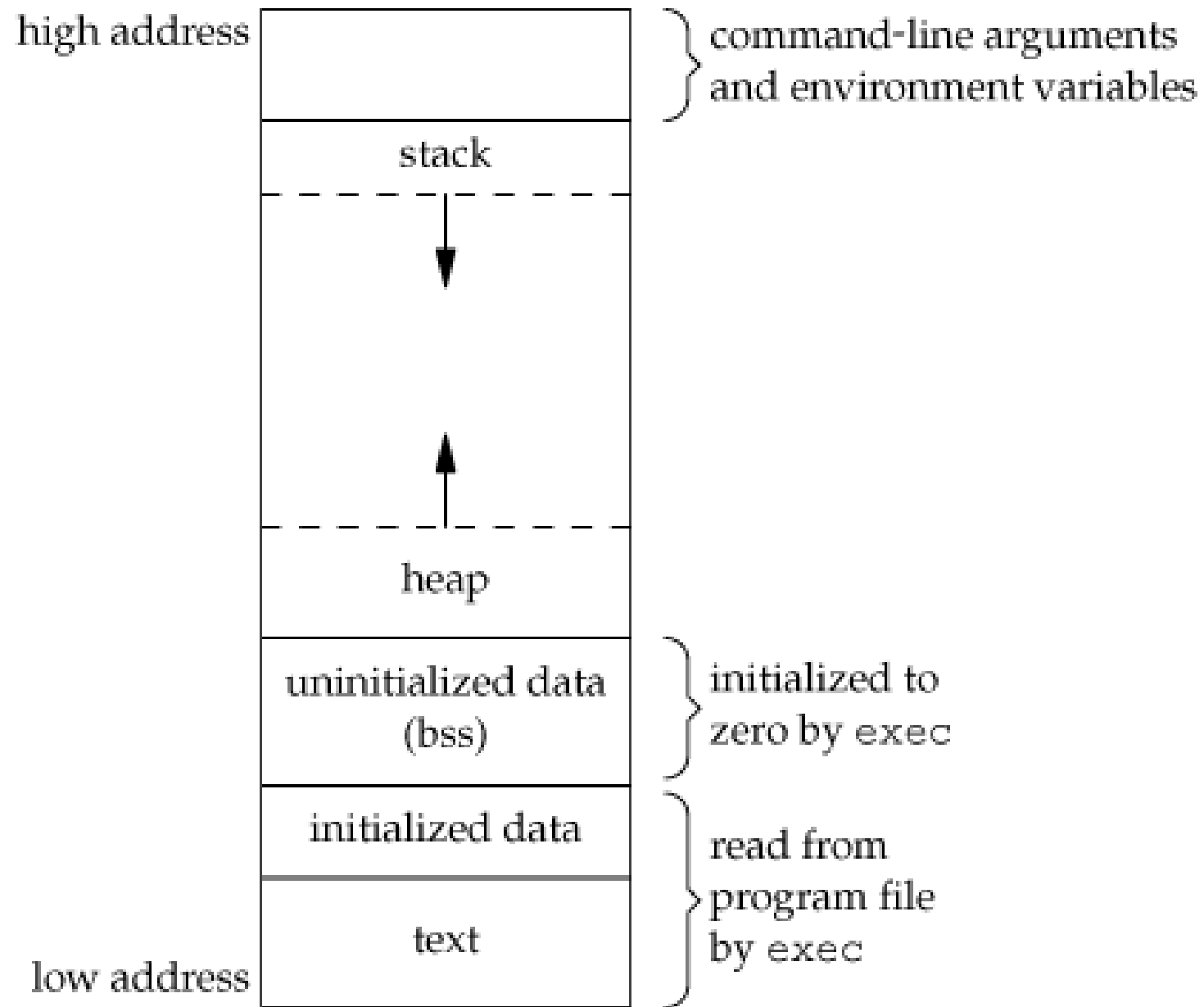
x86 Assembly: Some Context

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 its 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

- The main reason I brought this up:
- The program code is stored in memory. So how does the processor keep track of where it is 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.
- This register simply contains the address of the instruction that is to be executed (note: not the instruction itself).