## Chapter 1

1. Convert from 1 base to any other base

* **Convert to base 10**
* **Divide by new base until 0 coefficient**
* **Write remainders starting from bottom to top**
* **Ex: Convert 1304 from base-7 to base-12**
  + **First, convert to base-10: 4\*7^0 + 0\*7^1 + 3\*7^2 + 1\*7^3 = 494**
  + **Divide by new base until 0 quotient:**
    - **494 / 12 = 41 R 2**
    - **41 / 12 = 3 R 5**
    - **3 / 12 = 0 R 3**
  + **Write remainders from bottom to top: 352**
  + **So, 1304 base-7 = 352 base-12**
  + **See** [**http://www.wolframalpha.com/input/?i=1304+base+7+to+base+12**](http://www.wolframalpha.com/input/?i=1304+base+7+to+base+12)

1. What is used to represent everything inside the computer?

* **Bits**

1. Why do we like using hex numbers to represent bits?

* **Hexadecimal is a human friendly representation of bits in which each hexadecimal digit represents four bits.**
* **We like to use hex as programmers because it is very easy to convert from binary to hex and vice versa, as both bases are powers of 2. It also is often easier to read than binary, because more information is contained in each digit (so numbers are shorter when written). (Information Representation slide page 22)**

1. How space is made for multidimensional arrays
   1. Array of arrays

* **Create an array of pointers and point to each row**
  + **Ex: Ar[i][j] = \*(\*(ar + i) + j)**
  + **Ex: Ar[i][j][k] = \*(\*(\*(ar + i) + j) + k)**
  1. One big chunk
* **Maps elements of a multidimensional array into a single array**
* **Column Major: Maps columns of a matrix end to end**
  + **Ex: int ar[I][J][K]; ar[i][j][k] = \*(Ar + k\*J\*I + j\*I + i)**
  + **The distance between two consecutive ‘cols’ (ex. [0][0] and [0][1]) in memory is larger (~~’major’) than the distance between two consecutive rows (ex. [0][0] and [1][0])**
* **Row Major: Maps rows of a matrix end to end**
  + **Ex: ar[I][J][K]; ar[i][j][k] = \*(ar + i\*J\*K + j\*K + k)**
  + **The distance between two consecutive ‘rows’ (ex. [0][0] and [1][0]) in memory is larger than the distance between two consecutive cols (ex. [0][0] and [0][1])**
  1. How does C decide which to use?
* **By the code that you write**
  + **One big chunk: int ar[3][4]**
  + **Array of arrays: int\*\* ar;**
  + **If we know the size of the array at compile time we would want to use one big chunk (row major), because we declare it statically. If the size is unknown, we dynamically allocate space for the array as an array of arrays.**
    - **You can also dynamically allocate a one-big-chunk array, but you have to manually calculate the offsets when accessing elements.**
      * **Ex. int ar[m][n] could be approximated by int \*ar = malloc(sizeof(int) \* m \* n);**
      * **So make sure to actually read the code to be sure which one is being used.**
  1. Be able to write expressions that access elements of an array without using [ ] operator
     1. If ar was declared as int ar[3][4][7] how would you access element ar[i][j][k]
* **Row Major: \*(ar + i\*4\*7 + j\*7 + k)**
* **Column Major: \*(ar + k\*4\*3 + j \*3 + i)**
  + 1. Repeat the above but for ar declared as int\*\*\*. (Assume sufficient space has been already allocated for it)
* **\*(\*(\*(ar + i) + j) + k)**
  + 1. Be able to write the assembly code for these accesses as well
* **Look at matAdd.s (one Big Chunk & Array of arrays)**

1. What does it mean if a machine is little endian?

* **The least significant byte of a word is stored at the lowest address in memory, the word is stored “backwards”**
  1. Big endian?
* **The most significant byte is stored at the lowest part of an address, the word is stored “forwards”** 
  1. Which is Intel?
* **Little Endian**
  1. If the word size of the machine and the addressability of a machine are the same are there any endianness issues? For example the size of a word is a byte and memory is byte addressable

**No. Since the word size and addressability are the same, an entire address would not have to be separated into chunks by word size, meaning that endianness would not matter.**

* 1. What would the value in eax be after the performing movl 100, %eax

(movl 100, %eax means "copy the four bytes starting at location 100 to eax")

* + 1. Memory looks like

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Addr | 100 | 101 | 102 | 103 | 104 | 105 |
| Value | 0x11 | 0x22 | 0x33 | 0x44 | 0x55 | 0x66 |

* + 1. If the machine is little Endian?
* **0x44332211** 
  + 1. Big Endian?
* **0x11223344**

1. What does 111 1010 represent?

* **111 1010 could represent many different things depending on the context in which it is used.**

1. Be able to find the unsigned, 2’s complement, sign magnitude, and floating point representation of any number

**a. Unsigned Number**

**Convert to base 10 to get the value**

**b. Signed Magnitude Number**

**If positive just convert to base 10**

**If negative, negate the value, convert to base 10 and put a - sign in front**

**1.** **To negate the value just flip the sign bit**

**c. 2’s Complement Number**

**If positive just convert to base 10**

**If negative, negate the value, convert to base 10 and put a - sign in front**

**1.** **To negate the bit string flip the bits and add 1 to it**

**d. Floating point**

[**https://www.youtube.com/watch?v=8afbTaA-gOQ**](https://www.youtube.com/watch?v=8afbTaA-gOQ)

**Sign: 1 bit (0 for positive, 1 for negative)**

**Exponent: the next 8 bits (Use 0~254 to represent -127~127)**

**Mantissa: the next 23 bits**

**Notes:**

1. **If exponent is all 0s, then the number before the decimal point is 0, not 1**
2. **If exponent is all 1s (255, representing 128):**
   1. **If mantissa is all 0s, it represents positive or negative infinity (depending on sign bit)**
   2. **If mantissa is not all 0s, it represents NaN (not a number)extra**
   3. Be able to find the decimal value of any bit string when interpreted as one of the above
3. Be able to write **valid** C code that uses bitwise operators to extract fields, set bits, clear bits or examine bits of a bit string

[**C Bitwise Operators -**](https://en.wikipedia.org/wiki/Bitwise_operations_in_C)

**To extract bits, AND with a “mask” that has 1’s in the positions you want to extract.**

**ex. If unsigned int B = 0b10101,**

**to extract the three least significant bits of B into A:**

**A = B & 0b111**

**to extract the three most significant bits of B into A:**

**A = (B >> 2) & 0b111**

**To set bits to 1, OR with a “mask” that has 1 in the positions you want to set and 0 in the other bits**

**ex. set bits 1 and 3 in B to 1 and store the result in A:**

**A = B | 0b1010**

**To set bits to 0, AND with a “mask” that has 0 in the positions you want to set and 1 in the other bits**

**ex. set bits 0 and 2 in B to 0 and store the result in A:**

**A = B & 0b11010**

1. Are there types at the machine level? **No**
2. If you have B bits how many unique things can you represent? **2B**
3. If you have S unique states, what is the minimum number of bits you need to represent them all? **⎡log2(S)⎤**

Chapter 2

1. What are the two major components of the computer?

**Hardware and low level software made up of OS and System calls**

1. Give at least 2 reasons why code compiled on one· computer might not run on another computer

**- Different CPUS (Hardware). Each can have its own instruction set and instructions for one won’t necessarily work on the other**

**- Different System Calls. Your program will be making use to system calls which are often different on different operating systems**

1. What is the difference between RAM and ROM?

**RAM is volatile, meaning its memory will be erased once the computer is shut off, ROM is not, so it will keep its memory intact.**

* 1. Why is it that turning off your computer and then turning it back on is likely to fix a problem you have with your computer?

**It will free up the RAM (being volatile) which will erase the current state of the software, including any problems that have developed within that state.**

1. What is the bus?

**Connects the components of the computer together so that they can communicate**

* 1. Control Bus?

**Allows control signals to be sent such as read or write**

* 1. Address Bus?

**Allows addresses to be sent**

* 1. Data Bus?

**Allows data to be sent**

1. What do the following components inside the CPU do
   1. PC? **points to the current instruction being executed**
   2. IR? **holds the current instruction being executed**
   3. Data registers? **small fast memory inside the CPU used to store the output of the ALU and as temporary space for variables being operated on**
   4. MAR? **CPU’s connection to the memory address bus**
   5. MDR? **CPU’s connection to the data bus**
   6. ALU? **executes the arithmetic and logic instructions**
   7. ESP? **points to the top of the stack**
   8. FLAGS? **store various bits of state about the current program being run**
2. What are the steps in the CPU cycle?

* **Fetch, decode, execute, write (and check for interrupt)**

## Chapter 3 & 4

1. Be able to convert any C code into assembly code.

**if statements-**

**if\_start:**

**check condition**

**jmp else\_if (if condition is false)**

**code**

**jmp if\_end**

**else\_if:**

**check condition**

**jmp else(if condition is false)**

**code**

**jmp if\_end**

**else:**

**code**

**if\_end:**

**for loops-**

**initialize counter**

**for\_loop\_start:**

**check condition**

**jmp for\_loop\_end (if condition is false)**

**code**

**update counter**

**jmp for\_loop\_start**

**for\_loop\_end:**

1. What is the difference between jae and jge?

**jae: jump if above/equal to zero (unsigned)**

**jge: jump if greater/equal to zero (signed)**

* 1. How do you determine which you should use?
     1. For example if we had if(i > 7), which would you use?
     2. Do you need more information? If so what?

**If i was declared as a signed variable, it would need a signed jump (jge)**

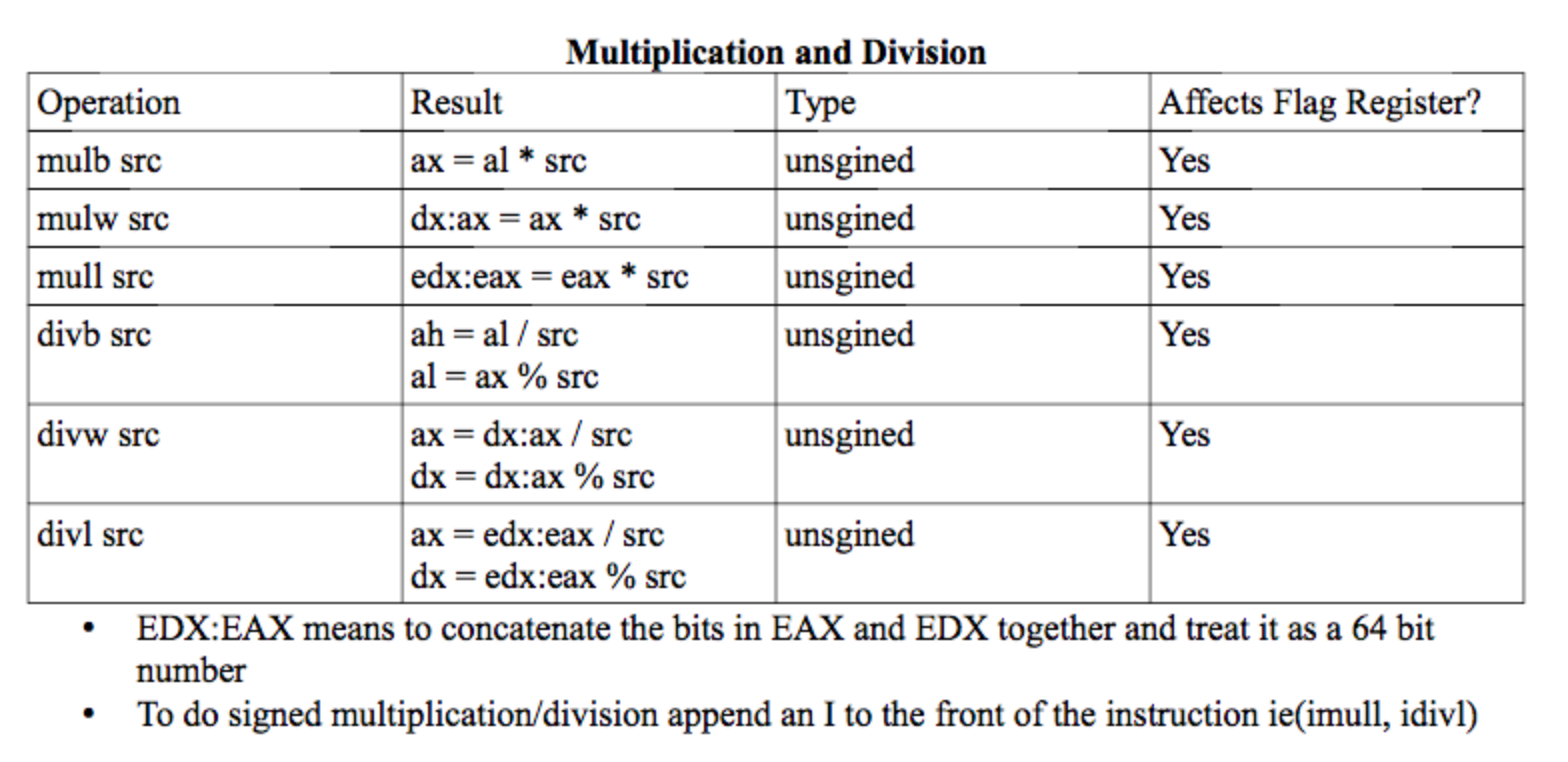
**If we knew it was an unsigned variable, it would be an unsigned jump (jae)**

1. What determines which suffix (b, w, or l) you should use for an instruction when translating from C to assembly?  
   **The size of the unit that the instruction is operating on:**

**l for int, w for short, b for char  
 b: byte, uses register portion ?L (AL, BL, CL, etc)**

**w: 2 bytes, uses register portion ?X (AX, BX, CX, etc)  
 l: 4 bytes, uses register portion E?X (EAX, EBX, ECX, etc)**

1. Be able to use the mul, imul, and div instructions



* 1. **mul - unsigned multiplication, one argument, assumes multiplication with eax or corresponding smaller bit portions**
  2. **imul - signed multiplication, can use one argument like standard mul, however, can also use two arguments (storing result in second arg)**
  3. **div - unsigned division, src is divisor**

1. Be able to use the advanced indexing mode
   1. What is the difference between
      1. Movl 5(%eax, %ebx, 2),%ecx**:Moves the value stored at address 5+eax+2\*ebx to ecx**
      2. Leal 5(%eax, %ebx, 2), %ecx:**Moves the address 5+eax+2\*ebx to ecx**

* Leal doesn’t touch memory!

1. Know which bytes of memory are accessed by an instruction. For example if eax = 10 and ebx = 5 which bytes of memory are accessed by
   1. Movl (%eax), %ecx : **10 11 12 13**
   2. Movw (%eax), %cx: **10 11**
   3. Movb (%eax), %cl: **10**
   4. Movl (%eax,%ebx, 4), %ecx: start from 10 + (5\*4) → **30 31 32 33**
   5. Movw (%eax, %ebx, 2), %cx: start from 10 + (5\*2) → **20 21**
   6. Movb (%eax, %ebx, 1), %cl: start from 10 + (5\*1) → **15**
2. Know what each of the following preprocessor directives do
   1. .global **makes a label visible to linker**
   2. .byte **makes space for one byte**
   3. .word **makes space for two bytes**
   4. .long **makes space for four bytes**
   5. .space **makes space for N bytes(uninitialized)**
   6. .rept **loop, repeats code between .rept and .endr N times**
   7. .equ **allows a label to be substituted with a value**

## Chapter 7

1. What is the stack?

**Wherever ESP points and all addresses higher than that**

1. What is the current stack frame?

**Between EBP and ESP**

What does it mean that the stack frames are chained?

**Any stack frame can access previous stack frame**

* + 1. How is chaining accomplished?

**- The first (i.e. highest-address) element will contain a pointer to the beginning of the caller’s stack frame**

**(in prologue: push ebp, movl esp to ebp)**

1. What is the purpose of the prologue?

- **establishes current stack frame**

**- saves registers that may have live values that will be used by pushing their contents to the stack**

* 1. The epilogue?

-  **undoes the prologue, cleans up the stack and restores EBP and ESP to values before prologue was called, also removes locals**

**- restores values of registers used that may had live values that were used in the function**

1. What are the gcc C calling conventions?

**- arguments are pushed onto the stack in reverse order of appearance**

**- return value (if exists) placed in eax**

**- when calling a function, eax, ecx and edx must not have live values. All other registers are assumed to have live values.**

1. Where is space for variables made?
   1. Global? **Data section**
   2. Static? **Data section**
   3. Local? **Stack**

## Chapter 8

1. How does I/O mapped I/O work?

**- separate address space for I/O devices and memory**

**- separate instructions to access I/O and memory**

**- in, out for I/O**

**- mov, and any instructions that uses advanced indexing for memory**

**- additional wires on the control bus to specify I/O or memory address**

* 1. Memory Mapped?

**- unified address space but we reserve some of the addresses for I/O devices**

**- Same instruction to access both memory and I/O devices**

1. How does polling (wait loop) I/O work?

**- keep asking I/O device if it has input**

**- once it does, we get input from device**

**- use input**

**- go back to asking**

* 1. What are its advantages?

**· Low latency (faster)**

**· Can all be done in software**

* 1. Disadvantages?

**· Inefficient use of CPU**

1. How does driven I/O work?

**CPU does not wait for response, instead receives an interrupt once something has been inputted from the I/O device**

* 1. What are its advantages?

**·**   **Efficient use of CPU**

* 1. Disadvantages?

**· Increased complexity**

**· Slower (slightly) in handling input**

* 1. Explain in detail how an interrupt is serviced once the interrupt assert line is set to 1

**I/O device produces an interrupt**

**IRQ line will be set to 1 to signify an interrupt has happened**

**- CPU checks the IRQ (interrupt request) at the end of every instruction**

**- CPU asserts the INTA (interrupt acknowledged) line**

**- I/O device replies back with its interrupt number**

**- CPU needs to save its current state**

**o Push eflags**

**o Push cs (code segment register)operating system**

**o Push EIP**

**- Service the interrupt**

**o When your computer started, the OS filled out the shared drive and set register IDT to point to beginning of table. This table is full of interrupt descriptor vectors.**

**o Each vector contains info about the interrupt as well as a pointer to the interrupt handler**

**o Pick the vector that corresponds to the interrupt number we received earlier, and jump to the handler**

**- IRET instruction which returns from the interrupt**

**o Pop EIP**

**o Pop cs**

**o Pop eflags**

## Chapter 9

1. What are the 4 major services provided by the operating system?

**- memory management**

**- process management**

**- I/O device management**

**- file system management**

**(try to ensure that all programs get fair access to these resources)**

1. What does it mean for a program to be running?

**If EIP points to instructions/memory that belong to the program**

1. Assuming only a single CPU with a single core, if your program is running is the Operating System as well?

**No. If your program is being pointed to by EIP and is within a single core, then your program is running and the OS isn’t, only one program can be run at a time**

1. Can the OS stop your program from running if your program is currently running?  
   **A program can only be stopped by:  
    -a system call FROM the program to some function in the OS  
    -a hardware interrupt from an I/O device to the CPU  
    -a (software/hardware)? interrupt from the CPU (dividing by 0 / seg faults)**
2. Explain how the program a.out gets loaded after you type ./a.out on the terminal

**- Shell will run execve (system call to OS)**

**- OS locate a.out on the disk**

**- Read the header of a.out**

**· Info about size of program**

**· Location of the sections (.data, .text, etc.)**

**· Size of stack**

**· Location of \_start in our code**

**- OS looks at its memory allocation table for a space in memory big enough to fit a.out**

**- OS copies a.out from disk into memory and updates memory allocation tables**

**- Save OS state (registers)**

**- Initialise a.out’s registers**

**· Zeroing the data registers, set ESP**

**- Start a.out by jmp to a.out’s \_start**

**- a.out is running**

1. If the OS is the program that loads programs, how does the OS get loaded?

* **Boot loader loads the OS**
  + **Stored in ROM**
  + **ROM is nonvolatile**

**Loading the OS**

* When you turn the CPU on it is pointing to 0xFFFFFFF0
* Put the ROM that holds the boot loader at this address

**Boot Loader’**

* Located in ROM
* ROM is at address 0xFFFFFFF0
* Read the first sector of the disk into memory and then jump to the beginning of that location
* The OS put a portion of itself at the first sector of the disk
* This is what causes the OS to be run first
* This program loads the rest of the OS into memory and then jumps to it

1. What is time sharing?

**Multiple programs taking turns so quickly that it looks like they’re running simultaneously.**

* 1. On what hardware components does it depend on?
* **Timer (emits an interrupt at regular intervals)**
* **Interrupt (switch to other program) + interrupt support**

1. What is a context switch?

* **Switch between two programs running**
  1. Explain the steps taken in performing a context switch

**- save program A’s state**

**· all the registers (EIP and EFLAGS are on the stack)**

**· these values are saved in A’s entry in the process table**

**- load B’s state**

**· info is in B’s entry in the process table**

**· overwrite A’s EIP and EFLAGS w/ B’s**

**- do IRET (interrupt return)**

1. How can control of the CPU switch between programs?

**- interrupts**

**- voluntarily relinquish control**

**· generally happens through a system call**

1. What are system calls?

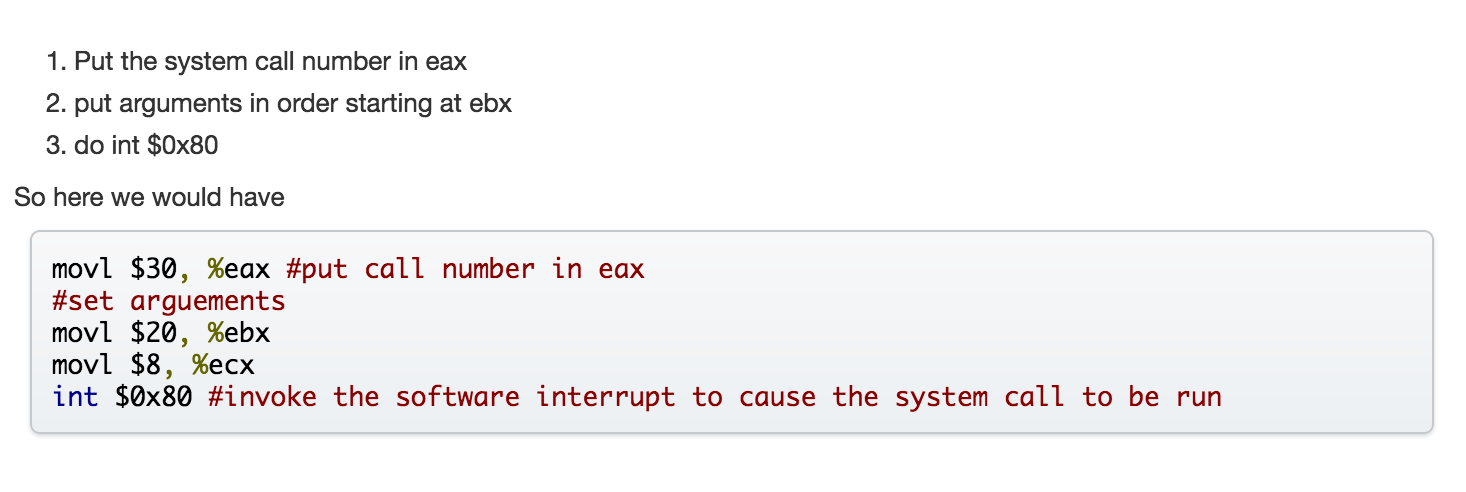
* **System calls are calls by software running on the OS to services provided by the OS.**

## Chapter 11

1. How are system calls implemented?

**Via software interrupts**

* 1. If we had a system call named bob that had call number 30 and we wanted to call bob as bob(20,8) how would we do it in assembly.



## Inline Assembly

1. Be able to translate C code into inline assembly
   1. What do each of the following constraint modifiers mean? When should you use them?
      1. + **read and write, associated register will serve as both input and output**
      2. = **write only, c variable will be overwritten with value in associated register**
      3. & **operand is an early clobber operand, variable will be written before all inputs**

## Debugging

1. Be able to do the following in GDB
   1. Set a breakpoint on a line number

**b line number**

* 1. Print the value of a variable

**p variable**

* 1. Print the value of a register

**p $register**

* 1. Print the value of a variable on the stack assuming the prologue has already been completed

**p ((type\*)$ebp)[number]**

* 1. Print all the arguments on the stack as integers

**p ((int\*)$ebp)[2]@num\_args**

* 1. Print all the local variables for a function.

**p ((type\*)$ebp)[-num\_locals]@num\_locals**

* 1. Print the contents of an array

**p ((type\*\*)$ebp)[distance\_from\_ebp][0]@num\_elements**

1. Print out the elements in an array if the pointer to an array is in the register
   1. If EAX contains the pointer to an array of integers named ar, print out the first 10 elements of ar.

* **p ((int\*)$eax)[0]@10**

1. Print out the elements in an array if the pointer to the array is on the stack and the prologue has been run
   1. Print the first 5 elements of ar in: foo(int a, int b, int\* ar)

**p ((int\*\*)$ebp)[4][0]@5**

* 1. Print the first 5 elements of ar in: foo(int a, int b, char\* ar)

**p ((char\*\*)$ebp)[4][0]@5**

* 1. Print the first len elements of ar in: foo(int a, int\* ar, int len)

**p ((int\*\*)$ebp)[3][0]@((int\*)$ebp)[4]**

1. Print out a row of elements of a 2D array
   1. int\*\* ar is stored in esi. Print out the first 4 elements of row at index 7

**p ((int\*\*)$esi)[7][0]@4**

* 1. foo(int\*\* ar, int num\_rows, int num\_cols). Print out the first 10 elements of ar at row index 2

**p ((int\*\*\*)$ebp)[2][2][0]@10**

* 1. foo(int\*\* ar, int num\_rows, int num\_cols). Print out num\_cols elements of ar at row index 5

**p ((int\*\*\*) $ebp)[2][5][0] @ ((int\*) $ebp)[4]**