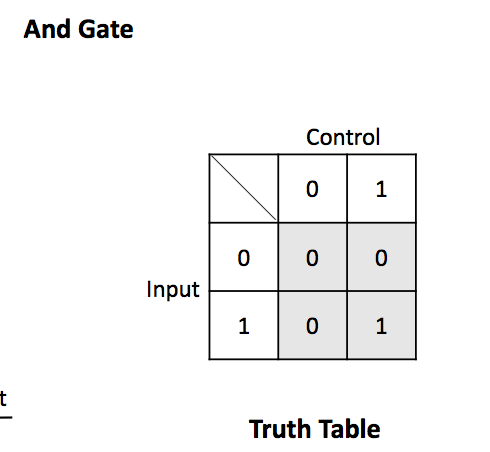
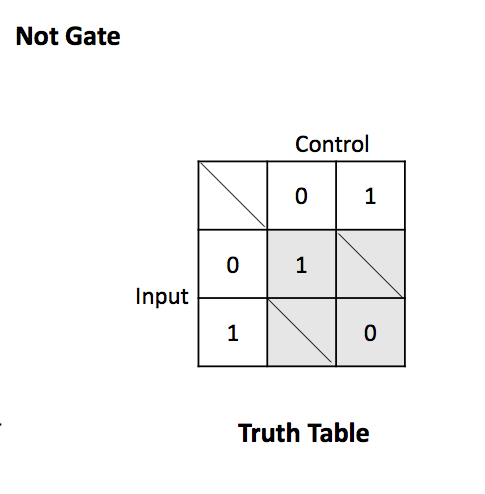
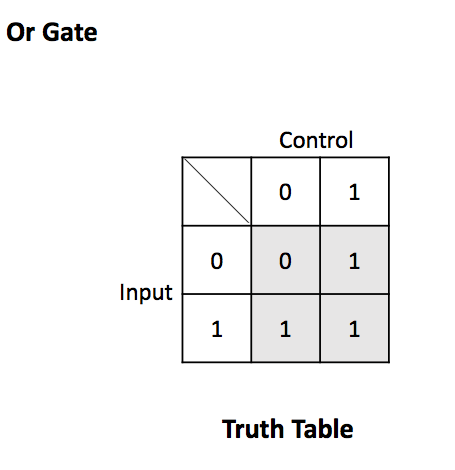
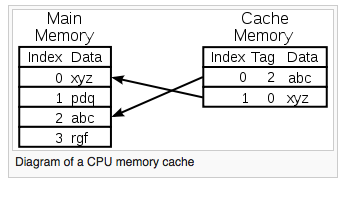
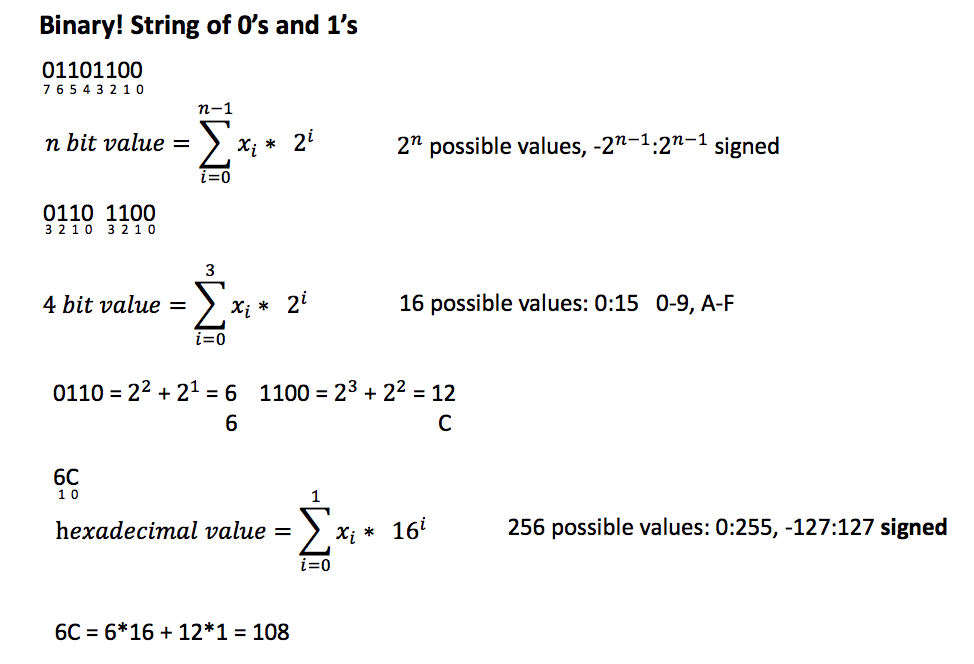
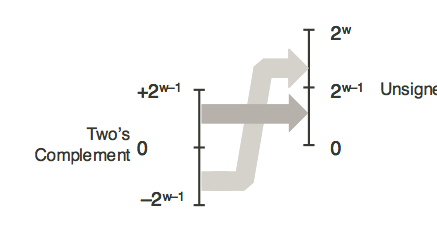
* A list of relevant chapters: 1,2,3,5,6,7,8,9,12
* 2 mid-term exams (open notes: 15 points each)
* final exam (open notes, 30 points )
* 4 labs (15 point each)
* ?? homework only passing grade, +1 point for pass extra credit.
* Using spreadsheet
* An **analog computer** is a form of **computer** that uses the continuously changeable aspects of physical phenomena such as electrical, mechanical, or hydraulic quantities to model the problem being solved. In contrast, digital **computers** represent varying quantities symbolically, as their numerical values change.
* Turing machine:
* <http://en.wikipedia.org/wiki/Turing_machine>
* an analog: voltage below:0 voltage above :1
* Three types of gate: and gate, or gate not gate.
* 
* 
* in CPU: Program counter: keeps track where is it to be executed.
* ALU: <http://en.wikipedia.org/wiki/Arithmetic_logic_unit>
* The bus travels back and forth between the CPU and memory.
* Use cache to speed up the program.
* There’s various combination of cache.
* The main memory contain a lot of stuff.
* There’s a cache between ALU and main memory.
* 
* There’s a certain area in the memory cannot be access because they are reserved to store information on what can or can not be accessed.
* IBM has a decimal data type.
* Pointer is to someplace in the memory.
* In 64- the pointer is 64 bits.
* The reason the booting steps takes so long is because that every time you boot the computer, it goes back to the booting record and read it all over again.
* Device controller, keyboard controller runs kind of autonomously.
* The ALU starts an instruction , and what it does, is it goes along the command control, the CPU send information along.
* GNU on linux debugger.
* Convert program to memory, set the program counter-> do it.
* first digit represent whether it’s negative or not.
* Brandon :
* OH: 9:30-10:30 M/Th
* ISA (instruction set architecture)
* Representing data:
* 4 bits (0 and 1) represent 16 numbers.
* Max is 2^n-1
* For example: 10100=2^0\*0+2^1\*0+2^2+2^5=20
* Convert decimal to binary:
* 126-64=62 ….1
* 62-32=30 …. 1
* 30-16=14 … 1
* 14-8=6…1
* 6-4=2 …. 1
* 2-2=0…. 1
* 0 <1 …. 0
* 1111110
* 101011110010 ---split to 4.
* 1010, 1111, 0010
* 1010 -> 10+2
* 1111-15
* 0010- 2
* 2^32 address
* 4 GB for address on a 32-bit machine.
* 2^30 bytes -> 1 GB
* **get to negative: flip the bits and add 1**
* maximum for 5 bits: 2^(n-1)-1
* Tmin=-2^(n-1)
* Not symmetric because the positive side has one reserved for zero.
* **In project, test for overflow.**
* Because 10000
* Flip to 011111 add one you get overflow.
* **Cast 4 bits to 8 bits:**
* 0011 --- 00000011
* If negative, add all one.
* 1101---11111101
* **Cast 8 bit to 4 bit:**
* X= 00011000
* X2=1000 (take last 4)
* **Bitwise operation:**
* ~(1011)=0111
* 1010 &1100 = 1000
* 0111 << 1 =1110
* 1011 >> 1 0101 (logical)
* 1011 >> 1 =1101 (arithmetic) shift and add 1
* count number 1 digit:
* 1000000..0000
* while (x!=0)
* {
* x=x&x-1;
* count ++;
* **(getting rid of the right most 1)**
* }
* swap:
* (x>>1) & 010101
* |
* (x<<1) & 101010
* Endian:
* Long \*x=0X100
* \*x=0X01234567
* for (int i=0; i<sizeof(long);i++)
* {
* printf(“%lx\n, (long),\*((unsigned char)x+1”)
* }
* LE: 67 45 23 01
* BE: 01 23 45 67
* Homework: little endian
* 1. write a function to convert integer to bit level representation (need to determine overflow)
* set o to 1 if overflow, set o to 0 if good.
* If overflow is detected, we still write to the thing.
* 2.
* 4 sizes: 8 , 16 ,32, and 64 bits
* 
* printf("%9.6f", myFloat) specifies a format with 9 total characters: 2 digits before the dot, the dot itself, and six digits after the dot.
* Fix binary.
* Trading off the size of the number we can put in against the accuracy.
* There are different format for floating point:
* Normalized: e is not all zeros and not all ones.
* With overflow, the outcome would become infinity.
* <http://en.wikipedia.org/wiki/Floating_point>
* an excerpt from wiki:
* A floating-point number consists of two [fixed-point](http://en.wikipedia.org/wiki/Fixed-point_arithmetic) components, whose range depends exclusively on the number of bits or digits in their representation. Whereas components linearly depend on their range, the floating-point range linearly depends on the significant range and exponentially on the range of exponent component, which attaches outstandingly wider range to the number.
* On a typical computer system, a 'double precision' (64-bit) binary floating-point number has a coefficient of 53 bits (one of which is implied), an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10−308 to 10308, because the range of the exponent is [−1022,1023] and 308 is approximately log10(21023). The complete range of the format is from about −10308 through +10308 (see [IEEE 754](http://en.wikipedia.org/wiki/IEEE_754)).
* The number of normalized floating-point numbers in a system F (*B*, *P*, *L*, *U*) (where *B* is the base of the system, *P* is the precision of the system to *P* numbers, *L* is the smallest exponent representable in the system, and *U* is the largest exponent used in the system) is: .
* There is a smallest positive normalized floating-point number, Underflow level = UFL =  which has a 1 as the leading digit and 0 for the remaining digits of the significand, and the smallest possible value for the exponent.
* There is a largest floating-point number, Overflow level = OFL =  which has *B* − 1 as the value for each digit of the significand and the largest possible value for the exponent.
* In addition there are representable values strictly between −UFL and UFL. Namely, [positive and negative zeros](http://en.wikipedia.org/wiki/Signed_zero), as well as [denormalized numbers](http://en.wikipedia.org/wiki/Denormal_numbers).
* Can store approximately 7 digits of accuracy.
* Main register:
* If do unsigned char, we add 1111111…. To the binary array
* If don’t do unsign, we add 00000…. To the binary array.
* Overflow:
* How many bits required to represent decimal (base-10) number x?
* --Signed
* --unsigned
* sum:
* z=x^y^c
* c=(x&y) | (X&C) | (Y&C)
* \*\*\*how to check for overflow for unsigned addition
* \*\*\* does adding two number of opposite sign ever overflow?
* \*\*\* how to check overflow for negative +negative or positive+positive
* 1. convert to positive.
* 2. for loop on each ai
  + accumulative.
* //write a helper function for shifting.
* If shift, and the most right is a one then there’s a overflow.
* 2 source of overflow for multiplier:
* 1. shift too much
* 2. add overflow.
* **If over flow, continue.**
* Binary floating point
* 21.75
* -16
* 5.75
* -4
* 1.75
* -1
* 0.75
* -0.5
* 0.25
* -0.25
* floating point representation:
* bias for negative power of two.
* Exponent: 8 bits
* Mantissa: 23 bits.
* **Convert this:**
* 10101.11=x
* Step1: s=0;
* Step2: make x std form: 10101.11=1.010111\*2^4
* Step 3: find E.
* E=4;
* E’=E+bias
* Bias=2^(8-1)-1
* E’=131
* Step 4: represent E in binary:
* 131=128+2+1
* 10000011
* Step 5:
* M=010111000000 (delete the integer part.) seventeen bits of zeros.
* 0 10000011 0101110000000000000
* S E M
* Denormalized floating point:
* Occurs when E=000000
* Value of eponent =1-bias
* Other cases:
* Infinity : S=1 for negative infinity
* E=1111111
* F=0000000000000(23 zeros)
* NaN: not a number:
* E=111111111
* F!=0000000(23 zeros)
* F=M
* Normalized/denormalized: whether have a leading one in the mantissa.
* Normalized: E can’t be all zeros or all ones.

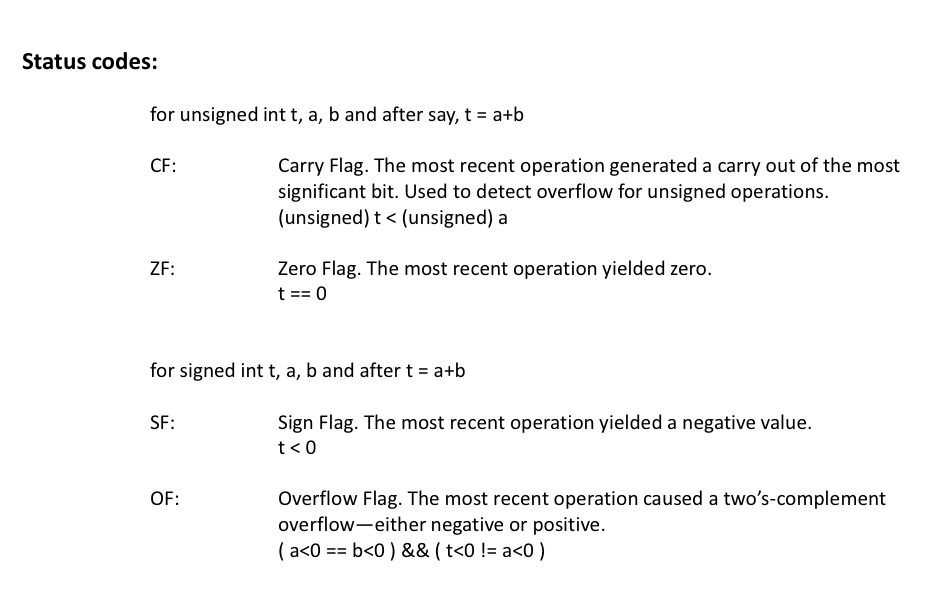
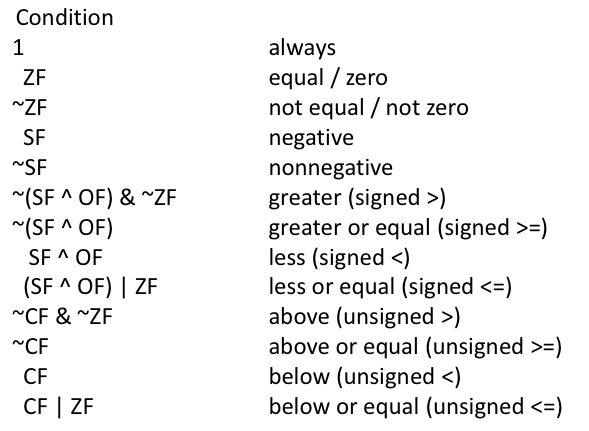
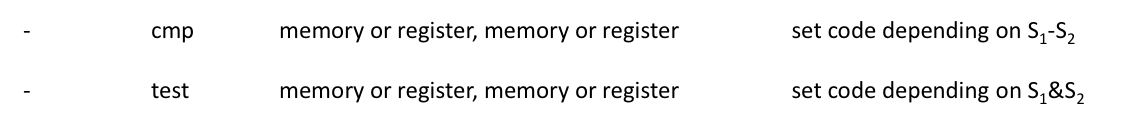
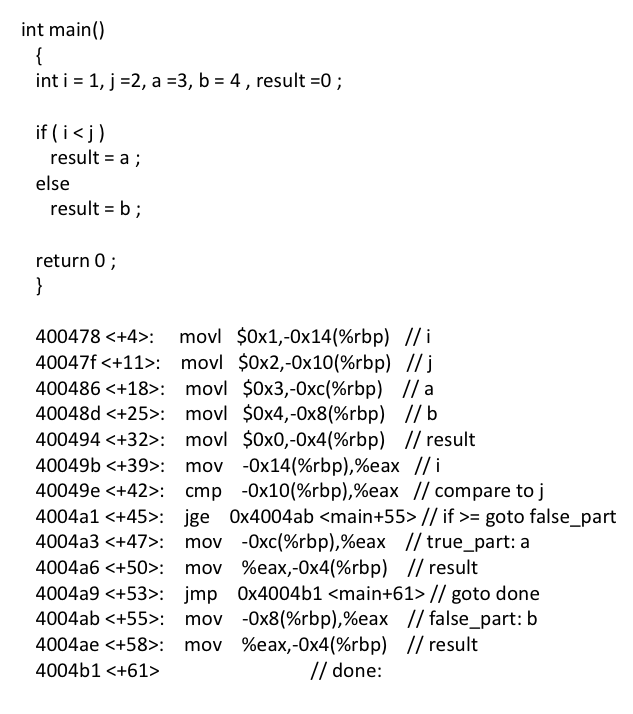
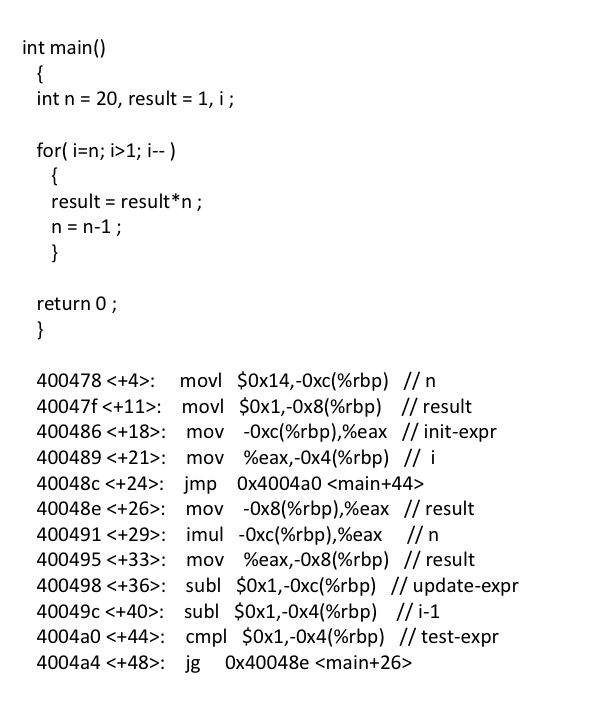
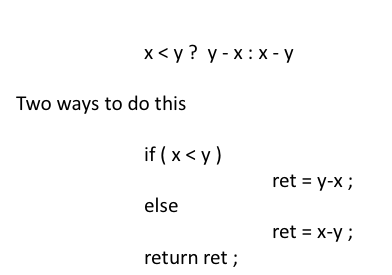
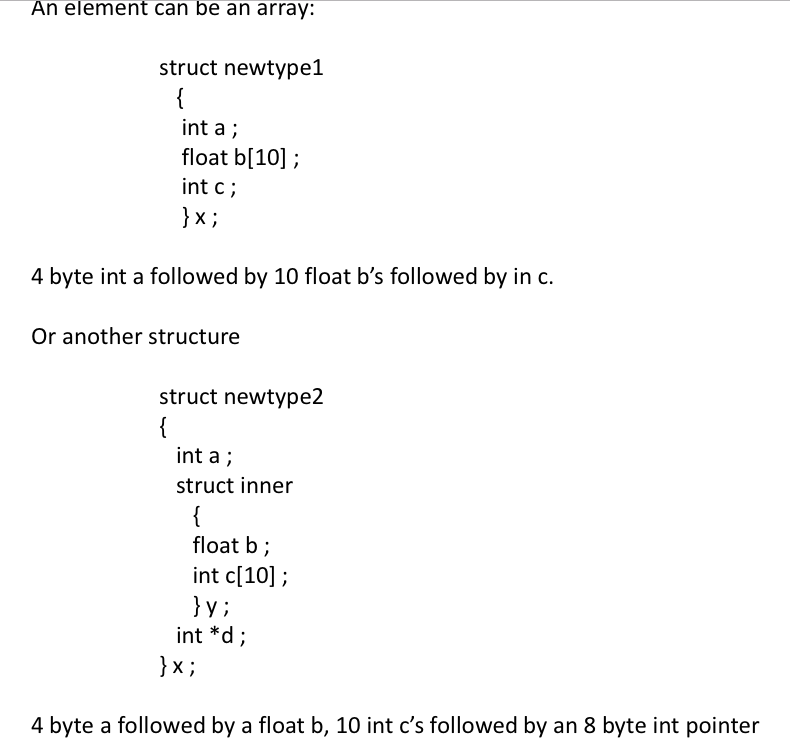
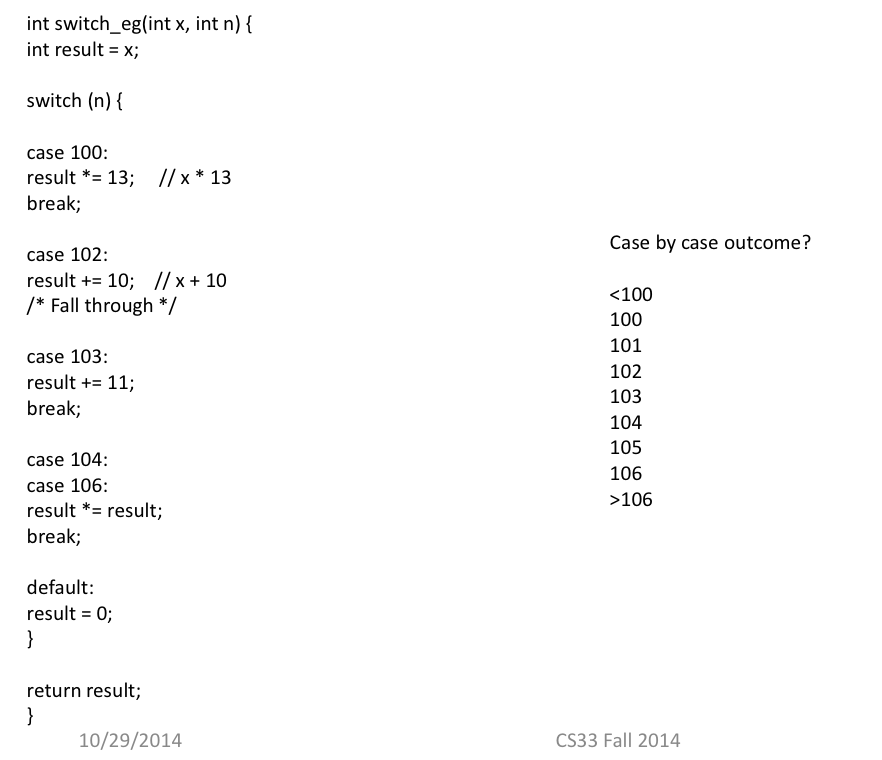
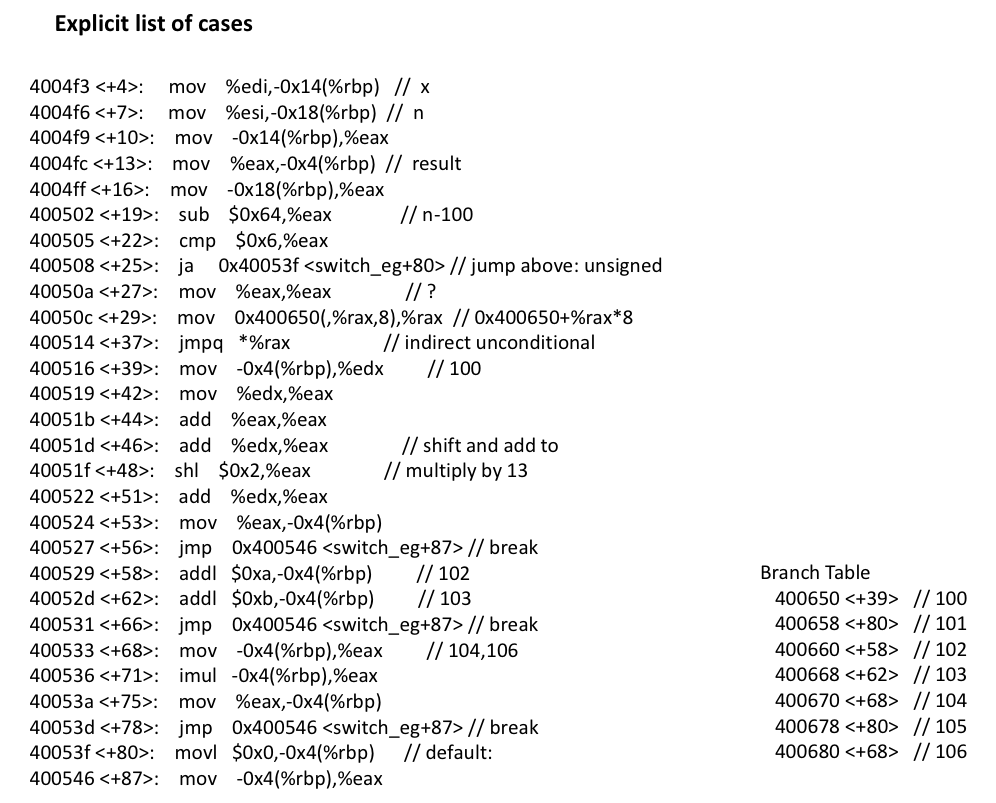
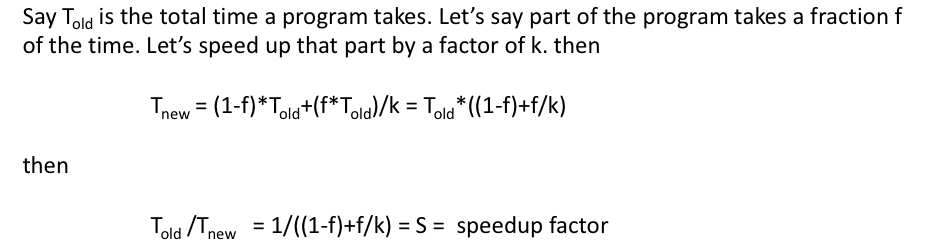
Largest denorm: M=1111111111111111 <1 , so x< 2^-126

Smallest Normalized: 1\*2^-126

**Introduction to machine code:**

* OP-Code.
* Operands.
* RISC:CISC
* ISA defines the language to talk to hardware.
* RISC: reduced instruction set computer)
* Pros: uniform
* Cons: takes more memory
* CSIC:
* Pros: less code, save memory
* Cons: complicated

IA32: two bottom reserved for stack pointer and frame pointer.

* Instruction pointer==program counter
* &eip
* segment register: point to key areas in memory
* flag register: interrupts over flow
* int do\_math(int x, int y, int s)
* {
* int z=x+(y<<s);
* return z;
* }
* **Memory and stack/heap:**
* %esp stack pointer. Top of the stack is the lowest memory address.
* %ebp -> for push
* %edx-> for pop
* eax:0x0000000000000
* $-4(%eax)🡪add for to the address.
* Moving data:
* Movl <src> <dest>
* -assign dest<=src
* **parentenesse🡪de-reference it**
* shrl-shift
* Register preceded by a percent sign.
* $ means the value.
* If have a parentlthess around, it means the address.
* In the assembler, a suffix that decribes what kind of move is that.
* Types of move:
* Immediate to register
* Register to register
* Memory to register
* Immediate to memory
* Register to memory
* Can’t move to immediate
* Can’t move from memory to memory
* Theres move sign extension and move-zeroe extension.
* Leave: move the base pointer into the stack pointer.
* Ret: takes the location and put it in …
* Rip: instructional register pointer
* Rbp: base pointer
* Rsp: stack pointer
* **%eax** is used to store the return value from **natural\_generator**,
* **%eax** and **%ecx** are general purpose registers, while **%rbp** and **%rsp** are special purpose registers.
* **%rbp** always has a higher value than **%rsp** because the stack starts at a high memory address and grows downwards.
* Midterm: 10 questions
* **sub $0x30,%rsp**
* subtract 30 from rsp
* Problem 3:
* Word << ()(3-bytem)<<3
* Return 1024;
* From the textbook and slides:
* 
* flags are conditions
* 
* 
* **an example of jump:**
* 
* 
* do you terminate things after cpml is not true??
* 
* for char x[20], each box is 1 byte, total size is 20 bytes
* **The address of x[i][j] is &x[0][0]+(i\*M+j) \* sizeof(int). Note that you can still address the array as \*(x+I) but you have to know what you are doing.**
* 
* **Swtich:**
* 
* 
* Memory corruption:
* buffer overflow
* gets/puts example
* pointer goes wild
* pointer error example
* array index goes wild
* **Optimization:**
* 
* CPU is divided into two:
* 1. instruction control
* 2. execution control
* the jump command will change the path of the program.
* The hardware for machine is actually designed like a language
* latency: cycles from start to finish
* issue time from the
* flash memory.
* printf (%x, (unsigned int)\*x)
* spray paint till x+sz-1
* check conditions: if x+sz<max, don’t do anything.
* print rounded address.
* Switch:
* If statement, do while, while🡪direct jump, (jump to this specific address)
* Indirect jump:
* Add $0X80488570,%eax add this address for the jump table.
* There are jump addresses.
* We manipulate the stack pointers using constant offsets.
* Optimization:
* Latency: delay from input to output.
* Writing before reading is a problem. When we use pipline implementation.
* Bad for CPE
* **Unrolled loop**
* For (int i=0; i<100;i=i+2)
* {
* a[i]=1;
* a[i+1]=2;
* }
* spray paint: put token to every bytes
* sub: overwrite things from min ptr to max\_ptr with random stuff. Before we do that, save it and then retore it.
* Instruction cache cannot be written
* A cache miss, on the other hand, means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play — while it’s slower, it’s also much larger.
* miss rate = min (1, (wordsize × k)/B )
* double the word size, double the miss rate.
* For an array this would mean that only some of the elements are present, like just every 10th element. You can then save space by not storing the empty elements in between.
* A dense array would be one where many, if not all, elements are present so there is no empty space between the elements.