Lec. 2 (3binary):

Bits	U/S	S/M	1C	2C
1111	15	-7	**-0**	-1
1110	14	-6	-1	-2
1101	13	-5	-2	-3
1100	12	-4	-3	-4
1011	11	-3	-4	-5
1010	10	-2	-5	-6
1001	9	-1	-6	-7
1000	8	**-0**	-7	-8

Conversions To:
S/M: Leftmost used for sign. Has (±0).
1'sC: **Flip** 7(0111) to get -7(1000). Has (±0).
2'sC: **Flip+1** 7(0111) to (1000+1)= -7(1001).
Has more neg. than pos.

Conversions From:
S/M: Replace leftmost with a sign.
1'sC: **Flip** -7(1000) to get 7(0111).
2'sC: **Flip except last 1 or 0**
-3(1101) to 3(0011).

OVERFLOW
2'sC: Taking negative of most negative number.
When adding 2#s of same sign: if carry into leftmost ≠ carry out leftmost then it is over flow.
Also, when you go too far from **zero.**

Largest values of any n-bit long number:
Largest positive: (signed) $2^n - 1$
(unsigned) $2^{n-1} - 1$
Largest negative: (S/M) 11…11 = $-2^{n-1} + 1$
(1'sC) 10…00 = $-2^{n-1} + 1$
(2'sC) 10…00 = -2^{n-1}.

Lec. 3 (Oct&Hex):

Octal (Base 8):
Digits: 0 – 7
Used for 3, 6, 9, 12 – bitstring lengths
Usually 3k bit partitions
So, 345 = 011 100 101

Hexadecimal (Base 16):
Digits: 0 – 9 & A – F (for 10 – 15)
Used for 4, 8, 12, 16–bitstring lengths
Usually 4k bit partitions
So, 3FC = 0011 1111 1100

Negation (X to –X):
The 1'sC of X is 15 – X , aka the 15'sC of X.
(15 – 3A = C5)
The 2'sC of X is 16'sC of X (= [15'sC of X] + 1)
([15 – 3A] + 1 = C6)

Oct Conversion (5-bit):
U/S: 31 = 11 001 = 25
2'sC: 31 = -(00 111) = -7
1'sC: 31 = -(00 110) = -6
S/M: 31 = -(1 001) = -9

Hex Conversion (7-bit):
U/S: 5D = 101 1101 = 93
2'sC: 5D = -(101 0011) = -35
1'sC: 5D = -(010 0010) = -34
S/M: 5D = -(01 1101) = -29

Floating point = Sci/notation
Ex. 6.4 = 110.01 = 1.1001*2^2

Lec. 4 (ASCII/Frac/Float/Sci/IEEE):

ASCII (8-bits):
Digit (0 – 9) = Hex (30 – 39) =Dec (48 – 57)
Letter (A – Z) = Hex (41 – 5A) = Dec (65 – 90)
Letter (a – z) = Hex (61 – 7A) = Dec (97 – 122)
Space = Hex 20 = Dec 32

Unicode (16-bits)
ASCII C have null terminator '\0'.

Convert to Binary Whole#:
Divide by 2 w/R then bottom-up
Convert to Binary Fraction#:
Divide by 2 w/R then Top-down

Convert to Whole# Binary:
Mult by 2^n
Convert to Fraction# Binary:
Mult by 2^{-n}

32-bit IEEE:
S = Sign-bit, E = 8-bit exponent, F = 23-bit fraction
Ex. 1100 0101 1011 0100 $0^{(16)}$.
S = 1 (neg), E = 1000 1011 =139 - (127) = 12,
F = 0110 $1000^{(16)}$ = $1.011010^{(18)}$. (prepend 1. to F)
So, Float = - $1.011010^{(18)}*2^{12}$.

IEEE Overflow:
Exp. Too large. E = 254-127=127
IEEE Underflow:
Exp too small. E = 1-127 = -126

Lec. 5 (Boolean Logic):

X	Y	X A D Y	X OR Y	X XOR Y	X A D Y	X OR Y	X IFF Y	X IMPL Y	X	NOT X
0	0	0	0	0	1	1	1	1	0	1
0	1	0	1	1	1	0	0	1	1	0
1	0	0	1	1	1	0	0	0		
1	1	1	1	0	0	0	1	1		

Precedence (high to low):
NOT, AND/NAND, OR/XOR, IMPL(≤)/IFF.

Style 1:

X	Y	NOT Y	X AND NOT Y	X AND NOT Y OR Y	NOT (X AND NOT Y OR Y)
0	0	1	0	0	1
0	1	0	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0

Tautology = always true
Contradiction = always false
Contingency = mix

Operator	Alternatives
AND	$\wedge$, juxtaposition, *
OR	$\vee$, +
XOR	$\oplus$
IMPL	$\rightarrow$, $\Rightarrow$
IFF	$\leftrightarrow$, $\Leftrightarrow$
NOT	~, $\neg$, !, overbar: $\overline{X}$, prime: X'

Lec. 6 (Bit Opr. & shift):

Left-shift (zero-fill):
Mask = ((1<<8) – 1); //last 8bits=1. Zero else.
1<<k = 2^k.
Right: X>>5 //shifts X(of 1s) right 5 bits.

Left Shift Zero Fill: 1110 => 1100
Left Circular Shift: 1110 => 1101
R-shiftw/0(logical R): 0111 => 0011
R-shiftw/sign(arithmetic R):
(positive same as logical R)
(When negative): 10011(-13)=>11001(-7)(2C)
R-circular: same

0…0000 1 000	= 1 << 3
0…0000 0111	= (1 << 3) – 1
0…000 111 00	= ((1 << 3) – 1) << 2
1…111 000 11	= ~(((1 << 3) – 1) << 2)

Lec. 7 (pointers&structs):

Int *p = &x → &x is address of x
*p=7 → value at x is now 7.
P = address. *p = value @address.
"%p" – prints address

```
typedef struct {
double real_part;
double imag_part;
} Complex;
void set_cpx(Complex *x, double a, double b) {
(*x).real_part = a; //or x->real_part = a;
(*x).imag_part = b; //or x->imag_part = b;
}//use printf("%f"); use params (Complex *x)
```

Lec. 8 (pntr&array):

`&b[0] + 1` is an integer-width (4 bytes) larger than `&b[0]`.
`&b[0] + k == &b[k]`. & `&b[k] - j == &b[k-j]`.
`b + i == &b[0] + i` – implies `*(b+i) = *&b[i] = b[i]`.
`p = &b[2]` → `p+1 = &b[3] & p-1 = &b[1]`.
*b = 2 → b[0] = 2; //So, &x[2]=2+x=x+2=&2[x] (same address)
2[x]=x[2] (same value at address)

(a) `p == &b[0]` (b) `q == p+2` (c) `*p == *q-10` (d) `*p == *(q-10)`
(e) `p[0] == p[1]` (f) `q == &p[2]` (g) `*p == *(p+1)`
(h) `p != p+1`
a, b, c, e, f, g, h are true; d may cause a runtime error because that address might be illegal.

Lec. 9 (von Neumann comp):

>3 main parts: CPU, Memory, I/O devices
>Different: programs are stored as data in memory
>Decoding of instruction in instruction register
>PC incremented during fetch instr, after reading intr from memory. Points to next instr.
>Phases of instr. Cycle: Fetch Instruction, Decode Instruction, Evaluate Addresses, Fetch Operands, Execute Instruction, Store Results
>CPU parts: Control Unit & (arith/log)Processing Unit
>MAR: holds location to read/write
>MDR: holds value read from/to write to memory

Opcode	*Meaning*	*Implementation*
0	HALT execution. (Ignore *R* and *MM*.)	*Running* ← *false*
1	LD (Load) Reg[*R*] with the value of memory location *MM*.	Reg[*R*] ← Mem[*MM*]
2	ST (Store) Copy the value of Reg[*R*] into memory location *MM*	Mem[*MM*] ← Reg[*R*]
3	ADD contents of location *MM* to Reg[*R*]	Reg[*R*] += Mem[*MM*]
4	NEG: Set Reg[*R*] to its arithmetic negative (ignore *MM*)	Reg[*R*] ← (– Reg[*R*])
5	LDM (Load immediate): Load Reg[*R*] with *MM*	Reg[*R*] ← *Sign* * *MM*
6	ADDM (Add immediate): Add *MM* to Reg[*R*].	Reg[*R*] += *Sign* * *MM*
7	BR (branch unconditionally): Go to location *MM* (ignore *R*)	PC ← *MM*
8	BRC (branch conditional): If *sign* of Reg[*R*] = *Sign*, go to location *MM*	if *sign*(Reg[*R*]) = *Sign* then PC ← *MM*
90	GETC Read a character and copy its ASCII representation into R0. (A sets R0 = 65, etc.)	*Reg*[0] ← *Keyboard*
91	OUT Print the character whose ASCII representation is in R0. (A for 65, etc.)	*Print Char*(*Reg*[0])
92	PUTS Print a string (i.e., the characters) at locations *MM*, *MM+1*, ... Stop (and don't print) when we get to a location that contains 0. (Don't print the zero.)	temp ← *MM* while Mem[temp] ≠ 0 *Print* Mem[temp]; temp++
93	DMP Print out the values of the control unit registers (the PC, IR, and R0 – R9)	*Dump Control Unit*
94	MEM Print the values in memory in a 10 × 10 table.	*Dump Memory*
95–99	Ignore instruction	

Lec. 10 (simple decimal comp):

5 1 78: LDM R1 <- 78
-5 2 78: LDM R2 <- -78
6 1 89: ADDM R1 <- R1 + 89 = 78 + 89 = 167
-6 2 89: ADDM R2 <- R2 + 89 = -78 + -89 = -167
2 1 45: ST M[45] <- R1 = 167
1 3 45: LD R3 <- M[45] = 167
3 3 45: ADD R3 <- R3 + M[45] = 167 + 167 = 334
4 3 67: NEG R3 <- -R3 = -334
7 8 10: BR 10
8 1 12: BRC 12 if R1 = 167 > 0: Yes
-8 2 14: BRC 14 if R2 = -167 < 0: Yes
9 0 11: I/O 0: Read char