·) Signed magnitude : The 1st bit represents the sign

· 1 : −

· 0 : +

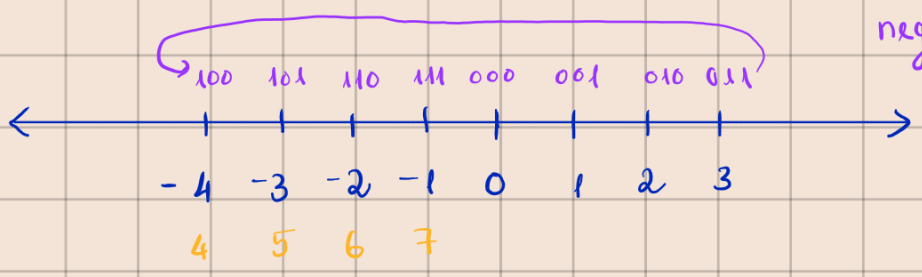
Ex : 011 → 3

111 → −3

- n-bit can represent $[-2^{n-1}, 2^{n-1}]$
- Problematic because addition / substraction are done differently

·) Two complement : n bits cover $[-2^{n-1}, 2^{n-1} - 1]$

Ex : 3 bits repre int ∈ [−4, 3]



negative int is 1st bit is still 1 !

- Addition in sign magitude : translate the last bit (the 1st bit on the left) into sign and the rest to decimal

$$\begin{array}{r} 010 \\ +\ 101 \\ \hline 001 \end{array} \quad \begin{array}{r} 2 \\ +\ -1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 001 \\ +\ 111 \\ \hline 110 \end{array} \quad \begin{array}{r} 1 \\ +\ -3 \\ \hline -2 \end{array}$$

- Addition in 2-complement : Add like normal

Ex :

$$\begin{array}{r} 010 \\ +\ 001 \\ \hline 011 \end{array} \quad \begin{array}{r} 2 \\ 1 \\ \hline 3 \end{array} \qquad \begin{array}{r} 010 \\ +\ 101 \\ \hline 111 \end{array} \quad \begin{array}{r} 2 \\ -3 \\ \hline -1 \end{array} \qquad \begin{array}{r} 001 \\ +\ 111 \\ \hline 000 \end{array} \quad \begin{array}{r} 1 \\ -1 \\ \hline 0 \end{array}$$

·) Conversion : $563_{10} = ?_2$

$$563 \xrightarrow{1} 281 \xrightarrow{1} 140 \xrightarrow{0} 70 \xrightarrow{0} 35 \xrightarrow{1} 17 \xrightarrow{1} 8 \xrightarrow{0} 4 \xrightarrow{0} 2 \xrightarrow{0} 1$$

$\Rightarrow 563_{10} = 1000110011_2$

·) $43_{10} = 101011_2$

$14_{10} = 1110_2$

$$\begin{array}{r} 43_{10} \\ +\ 14_{10} \\ \hline 57_{10} \end{array} \rightarrow \begin{array}{r} 101011_2 \\ +\ 001110_2 \\ \hline 111001_2 \end{array} \quad 32 + 16 + 8 + 0 + 0 + 1 = 57$$

$$\begin{array}{r} 43_{10} \\ -\ 14_{10} \\ \hline 29_{10} \end{array} \rightarrow \begin{array}{r} 101011_2 \\ -\ 001110_2 \\ \hline 011101_2 \end{array} \quad 16 + 8 + 4 + 0 + 1 = 29$$

·) How to convert negative int into 2-complement : "flip & inc"

$14_{10} = 001110_2$

↓ flip

$-14_2 = 110001_2 + \underbrace{1_2}_{\text{inc}} = 110010_2$

• Then we can perform $43_{10} - 14_{10}$ using addition in 2-C :

$$\begin{array}{r} 101011_2 \\ +\ 110010_2 \\ \hline 1\,|\,011101_2 \end{array}$$

excluded
Because the addition is positive

- Overflow : Out of bit to repre numbers
  - Adding 2 too big positive #
  - _____ small negative #

Sign : • A carry in into a sign bit but no carry out
A carry out of a sign bit but no carry in
(carry in ≠ carry out)

- Sign extension : To extend from m-bit → n-bit for 2- comp
  - For positive int, add 0 in front
  - ___ negative int, add 1 _____

- Fractionary Binary Number :
  - $1.1010\ldots = 1 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + \ldots$
  - IEEE-754 : $(-1)^{s} * 1.M * 2^{E-127}$ (floating point numbers)
    - s : sign - 1 bit → sign magnitude sys
    - E : exponent - 8 bits → (E-127) ∈ [-127, 129]
    - M : mantissa - 23 bits
  - Drawbacks : Big value sacrifices precision
    - It is because big value requires big E → shifting the floating point to the right

- Comparing 2 FP numbers :
  - Positive > Negative
  - If both positive, compare from left → right, which has larger bit first is larger

:) Bitwise operations : 1 - true 0 - false

- AND :

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

- OR
- NAND : NOT AND
- NOT
- XOR :

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

:) Given two four bit #s $\Rightarrow$ $2^4 = 16$ boolean functions

- Ex: $13\frac{41}{64}$ → IEEE 754 → sign : 1st bit; power of 2 : 8 bits; mantissa : 23 bits

$13 \xrightarrow{1} 6 \xrightarrow{0} 3 \xrightarrow{1} 1$

1101.101001

$2^{130-127} * 1.\underbrace{10110100}_{m}1$

$130 \xrightarrow{0} 65 \xrightarrow{1} 32 \xrightarrow{0} 16 \xrightarrow{0} 8 \xrightarrow{0} 4 \xrightarrow{0} 2 \xrightarrow{0} 1$

$0 \quad 10000010 \quad 10110100\,1\,0\ldots0$

- Bit shifting : right `>>` , left `<<`

Ex : $7 << 2 = ?$

$7 = 0111_2 \Rightarrow 7 << 2 = \underbrace{01}_{\text{trash}}\,1100 = 1100$

- Bit vectors : packed 1, 0 into a vector

Ex : $v = \underbrace{0b}_{\text{binary}}\,10101010$

$v' = 0b\ 10000000$

$w = 0 = 00000000$

$\Rightarrow \sim w = 1111\,1111$

·) Manipulating Bits using bool funcs :

- CLEAR : zero out a bit → use AND with 0
- SET : make a bit become → use OR with 1
- TOGGLE : flip a bit → use XOR with 1

  1 XOR 1 → 0

  0 XOR 1 → 1

  ⋮

·) Change from base 2^n → base 2^m :

Ex : $72124_8 \rightarrow ?_{16}$

7	2	1	2	4
111	010	001	010	100

0111	0100	0101	0100
7	4	5	4

7454_{16}

·) ASCII : Repre words / nums / symbols with bits

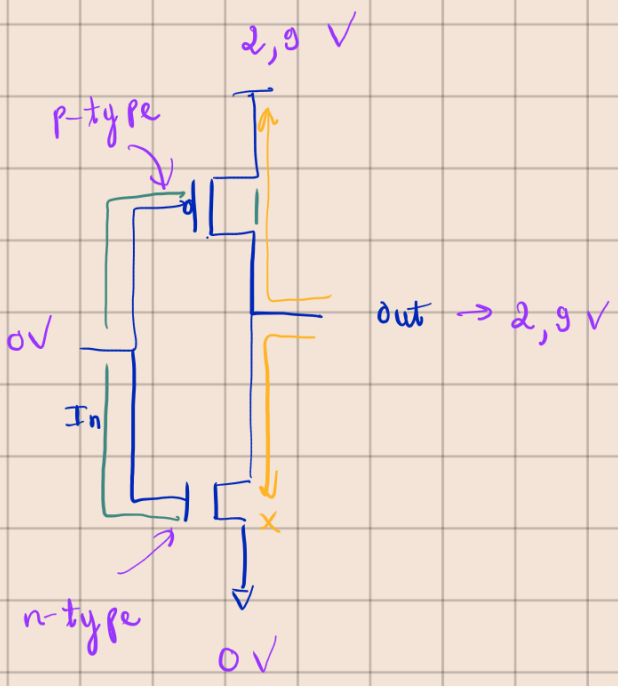
- uppercase → lowercase : SET bit 5

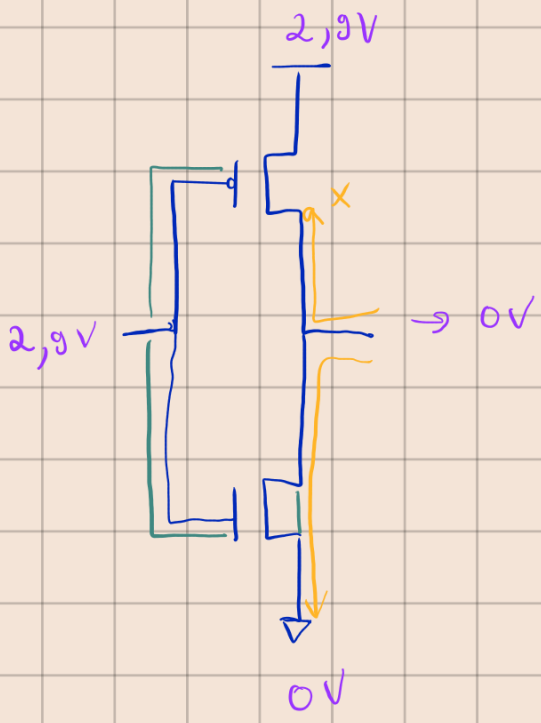
  lowercase → uppercase : CLEAR bit 5

65 536

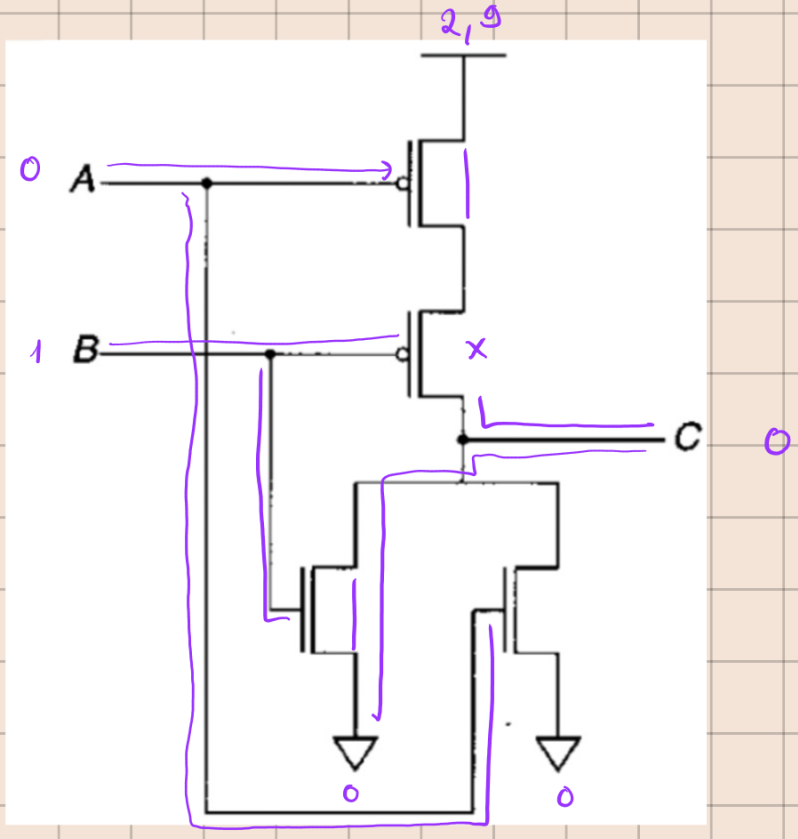
·) Transitor : we want to understand the behavior of transitor or a switch

• float : a wire that doesn't connect neither to ground nor power completely

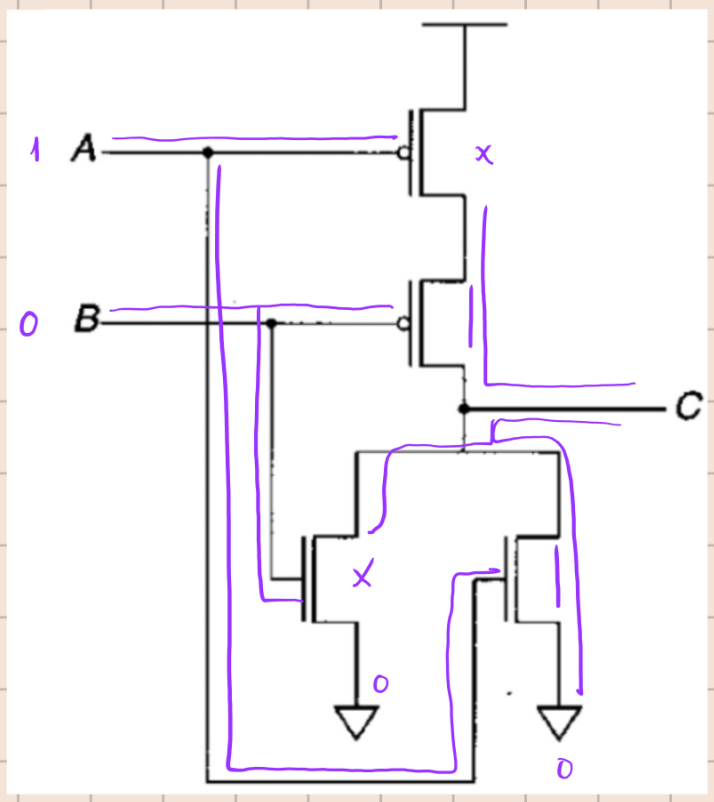




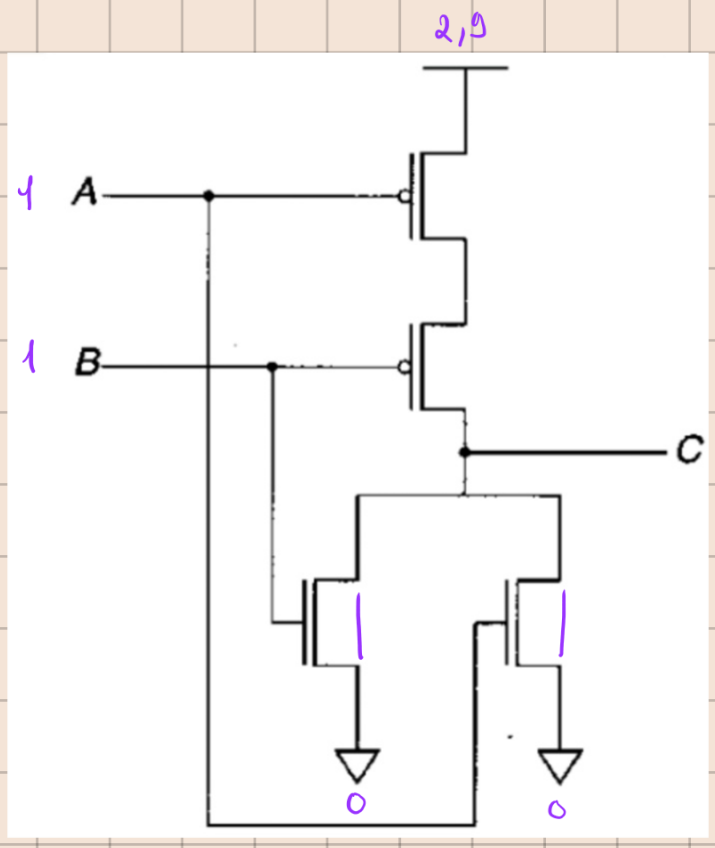
⇒ NOT Gate



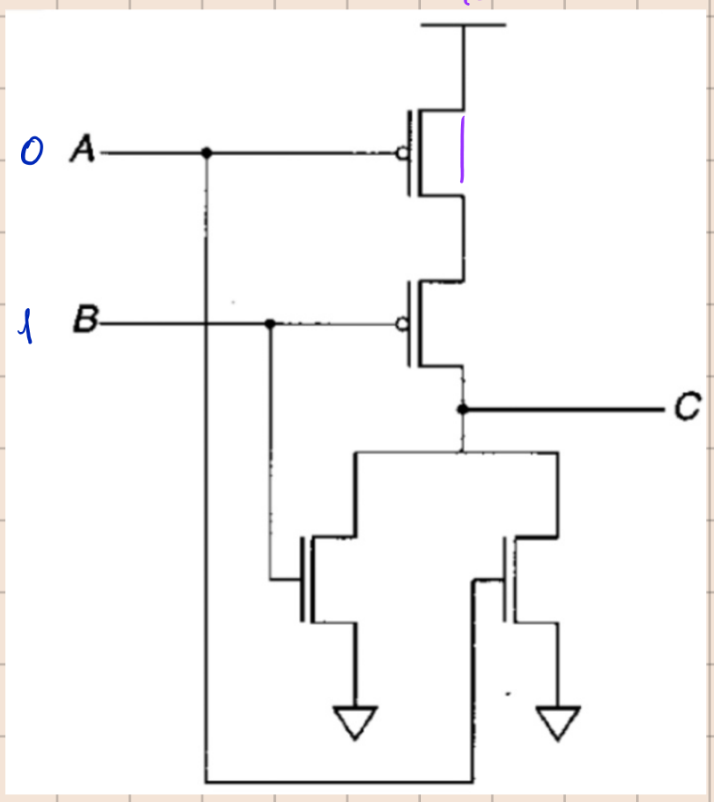
2,3



2,3



2.9

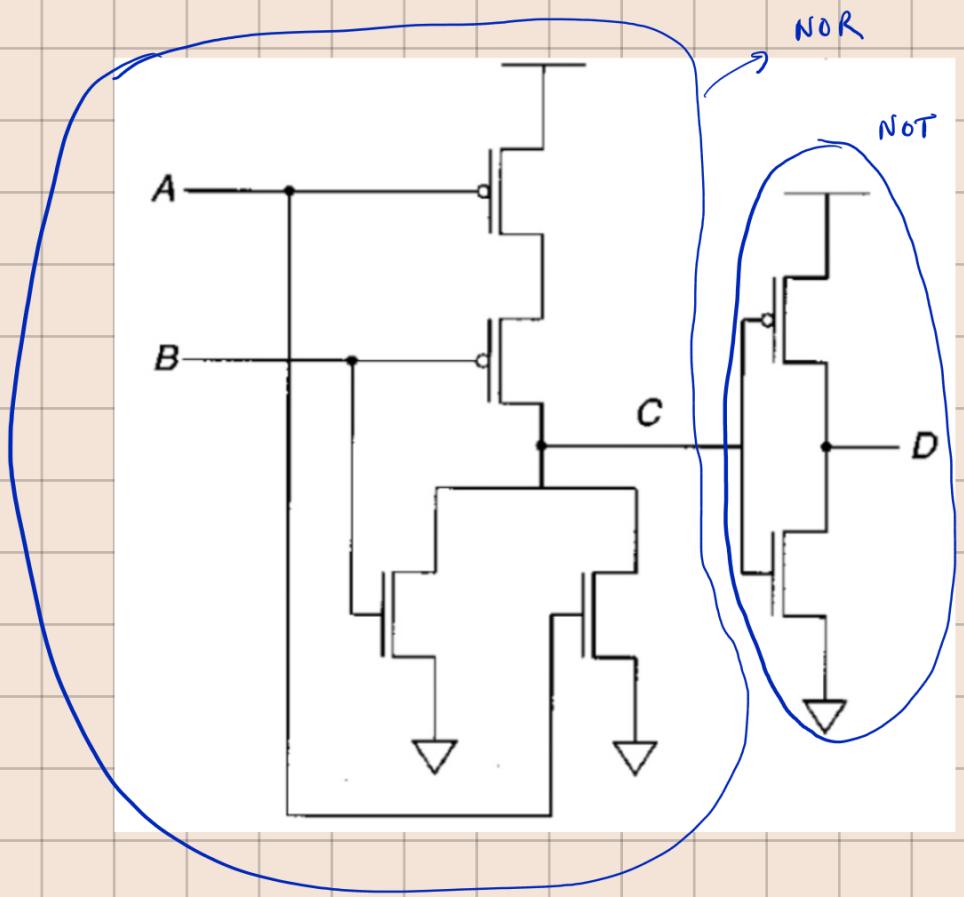


→ 1

(Not OR)

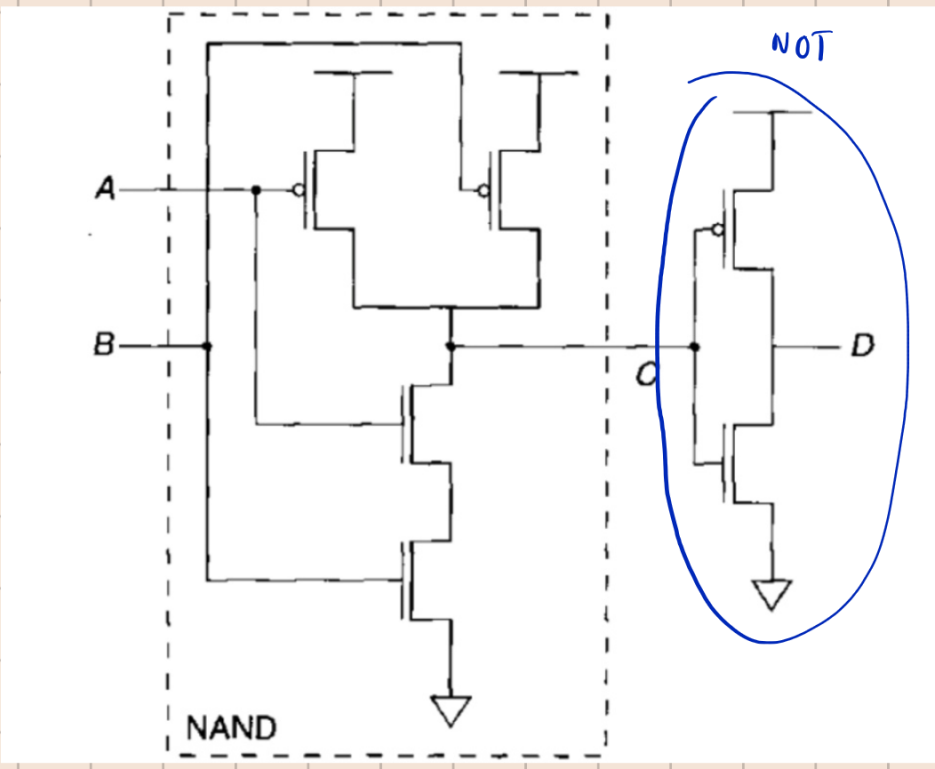
⇒ NOR Gate

A	B	A NOR B
1	1	0
1	0	0
0	1	0
0	0	1



NOT(NOR)

= OR



→ AND Gate

Note : Don't connect N - type transitor to power because they

P - type _____ ground

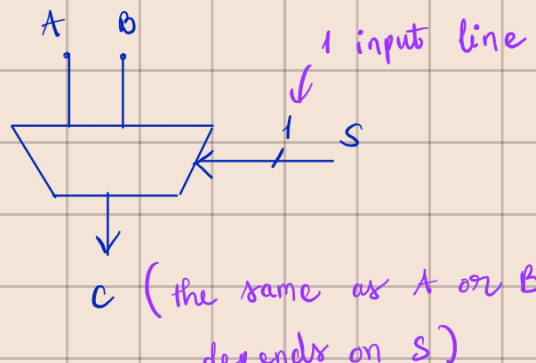
will cause floating voltage across the transitor

(CLC)

) Combinational Logic Circuits : output depends only on the inputs

- Decoder : The output has exactly one 1 and the rest is 0
  - n inputs
  - 2^n outputs

- Mux : select one of the inputs and connect it to the output



- n inputs line
- 2^n inputs

- One Bit Adder : Perform A + B where A, B are n-bits numbers
  one bit / circuit

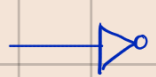
- Input : 3 n-bits numbers A, B, and C (carry)
- Output : 2 n-bits ——— C′, S
  C′ : carry to next column
  S : result

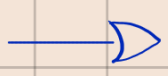
) Circuit Sim :
- Wire color :
  - Dark green : wire with value 0
  - Light green : ——— 1
  - Blue : unitialized wire

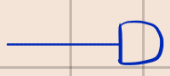
Connected to power is considered initialized

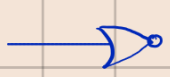
- For boolean circuit ✓ three 1s and one 0 [P tran : parallel, N tran : sequential
  three 0s and one 1 [P tran : sequential, N tran : parallel

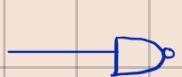
) Symbols :

NOT 

OR 

AND 

NOR 

NAND 

⋅) De Morgan's Law :

- Note : Notation are diff

$$A + B \equiv A \vee B$$
$$A' \equiv \neg A$$
$$AB \equiv A \wedge B$$

⋅) Boolean algebra :

- Additive : $A + B = B + A$

- Multiplicative : $AB = BA$
  $A(BC) = (AB)C$

- Basic Identities :

$A + 0 = A$	$0A = 0$
$A + 1 = 1$	$1A = A$
$A + A = A$	$AA = A$
$A + \bar{A} = 1$	$A\bar{A} = 0$

- Useful equation : $A + AB = A$
  $A + \bar{A}B = A + B$
  $(A + B)(A + C) = A + BC$

⋅) Truth table to circuit :

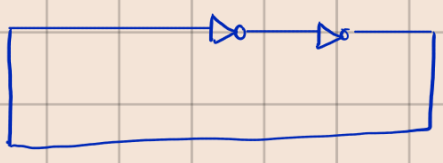
- Each AND gate repre one row of the true table
- Each OR gate repre one col of the true table

k-map :

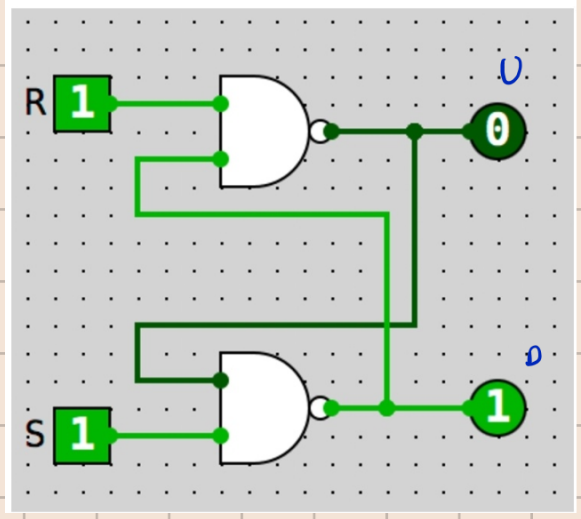
	B'	B
A'	What is the output if both A and B are false?	What is the output if A is false and B is true?
A	What is the output if A is true and B is false?	What is the output if both A and B are true?

- Valid grey code : • one entity changes from one state to the next

Basic storage :



R/S Latch :



- R = 1 → U = 0, D = 1 (regardless of S)
  S = 1 → U = 1, D = 0 (_____ R)
  → Stable → The circuit has memory

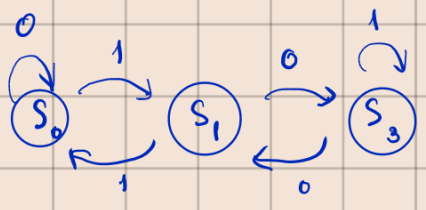
- R = S = 0 → U = D = 1
  → Not stable

  If we want stable and don't want an input = 1 all the time

→ D - latch

) Clock : control the pace the circuit does thing

) State machine : control system with a finite # states



- the state we gonna ended up at is N°1. 3
- Maintaining state is memory
- Make sure we have sufficient states for our machine
- Make sure at a state, the machine knows what state to go to next.

) Moore machine : output determined solely by current state

) One Hot : a method to encode state

0001 → state 1

0010 _____ 2

0100 _____ 3

1000 _____ 4

8/4/20

- Binary encoder : another method to encode state

- Reading state machine true table : pay attention to the state bit combination

  - State : input bits & state bits

- VonNeumann Model : A fundamental model of a computer for processing comp programs, consisting of :

  - Memory
  - A processing unit
  - Input
  - Output
  - A control unit

  → We w

- Comp program : a set of instructions, contained in memory

  - The data the program needs is either in input or mem

- Addressibility & Address space :

  - Address space : # distinct memory locations
    - Ex : 2^{34} (each location has a 34-bit address)

  - Addressibility : capability of each location

- Reading from a mem location :

addressing ind locations

- Place the addr of that location in the mem addr reg (MAR)

→ The info in the location then put into mem data reg (MDR)

hold the content of a mem location

) Writing to a mem location:

- Place the addr
- Place the value need to write into MDR
- Turn on Write Enable signal

GPR

) Processing Unit : ALU + 8 general purpose registers (R0 → R7)

) Input, Output : keyboard / monitor

) Control Unit : keep tracks of both where we are in the process of executing the program and executing each instruction

) Instruction Processing : The computer processes one instruction at a time

- Instruction : opcode + operands

(what instruction is it)

- 3 kinds of instructions:
  - Operates: operate on data (ADD, AND, NOT)
  - Data Movement: move info from processing unit to and from memory and to and from I/O devices
  - Control: Altering order of instructions (normally the instruction in the next mem location will be processed)

- ADD : 2 source operands + 1 destination operand
  - 2 source operands: requires ≥ 1 operand stored in the processing unit
  - 1 destination operand: stored in the processing unit

Note : 3 bits is required to identify a register

Ex : `0 0 0 1 | 1 1 0 | 0 1 0 | 0 | 0 0 | 1 1 0`

- `0 0 0 1` : opcode for add
- `1 1 0` : R6 → destination operand
- `0 1 0` : R2
- `0` :
- `0 0` :
- `1 1 0` : R6

→ $R6 \leftarrow$ "R2" + "R6" ⇒ content in R6

`0 0 0 1 | 1 1 0 | 0 1 0 | 1 | 0 0 1 1 0`

- `1 1 0` : R6
- `0 1 0` : R2
- `1` :
- `0 0 1 1 0` : 6 → immediate val

→ $R6 \leftarrow$ "R2" + 6

- AND : Similar to ADD

) Addressing mode : A formula to calculate the address of a mem location to be read

- LD (load) : go into a mem loction, read the value in there, and store it in one of the register

Ex : `0 0 1 0 | 0 1 0 | 0 1 1 0 0 0 1 1 0`

- `0 0 1 0` : LD
- `0 1 0` : R2 → destination operand
- `0 1 1 0 0 0 1 1 0` : 198 → operand used to calc the mem addr to read from

- Use PC-relative addressing mode : mem = [PC + offset] ↑ (w/ sign extension)

- Instruction Cycle : 6 phases

  - Fetch : obtain the next instruction from memory and loads it into the instruction register (IR). At the same time, increment PC
  - Decode : 4-to-16 decoder, taking in op code, output the line corr to that opcode
  - Evaluate address : computes the address of mem location that is needed for the instruction
  - Fetch operands : loading MAR w/ the addr calculated in "Evaluate Address"
  - Execute : executing the instruction
  - Store result : Storing the result

  ⊛ Note : Not all instructions require 6 phases

- GPR : temporary storage locations that can be accessed in a single clock cycle. Used by ADD, AND, NOT, LD, LDI, LDR

- Instruction set : The LC-3 ISA instruction set has 15 instructions (not 16 since the code 1101 is reserved for future need)

- Operand : An operand can be found in 3 places
  - Part of the instruction → immediate operand
  - Register
  - Memory

There are 5 addr modes : immediate, register, and 3 memory addr

- Immediate
- Register
- PC-relative
- Indirect
- Base + offset

) Memory

) Condition Codes : There are 3 single-bit reg N, Z, P that are individually set each time one of the 8 GPR is written :

3 cases 100
010
001

- 100 : negative data written into GPR
- 010 : zero ____
- 001 : positive ____

Load Effective Addr

) LEA instruction : load a GPR with an address

Ex : 1 1 1 0 | 1 0 1 | 1 1 1 1 1 1 1 0 1

LEA R5 -3

→ R5 ← PC* + (-3)

PC* : incremented PC

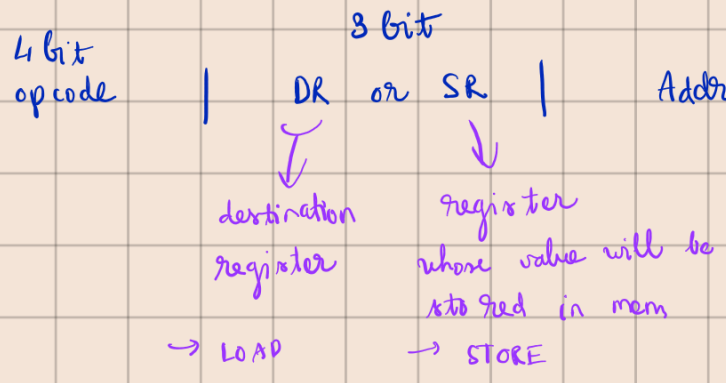
) Load : the process of moving info from mem to a register

- The info in mem is still there

) Store : ____ a register to memory

- The info in register is still there

Format for Load / Store :



·) PC-relative : mem = PC* + offset (8bit)

operand = [PC* + offset] → mem location

- LD, ST

·) Indirect Mode : mem = [PC* + offset] (8bit), operand = [[PC* + offset]]

- The addr of the operand can be anywhere in mem
- LDI, STI

·) Base + offset Mode : mem = content of base register + offset (6bit)

base register → a GLR

- LDR, STR

·) BR : If (condition ...) → do this Branch condition

Format : 0 0 0 0 | n z p | PC offset

- mem location of the next instruction
- PC Relative

- n, z, p cmpd to N, Z, P set by the previous instruction

If n / z / p = 1 : check the bit N / Z / P, if it is 1 → change the incremented PC ← address from EVALUATE ADDRESS phase

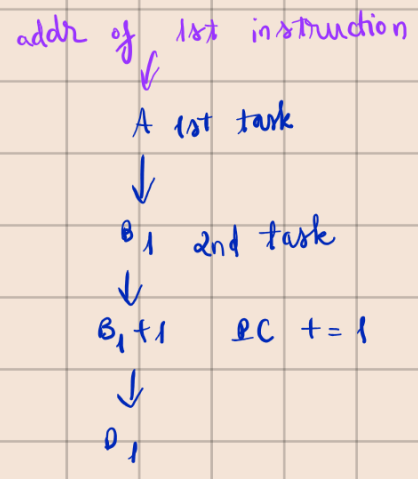
- JMP : Change the content of PC → Content in Base R (a GPR)
  - This helps reaching mem location outside of offset's range
  - Format : 1 1 0 0 | 0 0 0 | 0 1 0 | 0 0 0 0 0 0
    BaseR

- Systematic decomposition : decompose a complex program into small managable unit

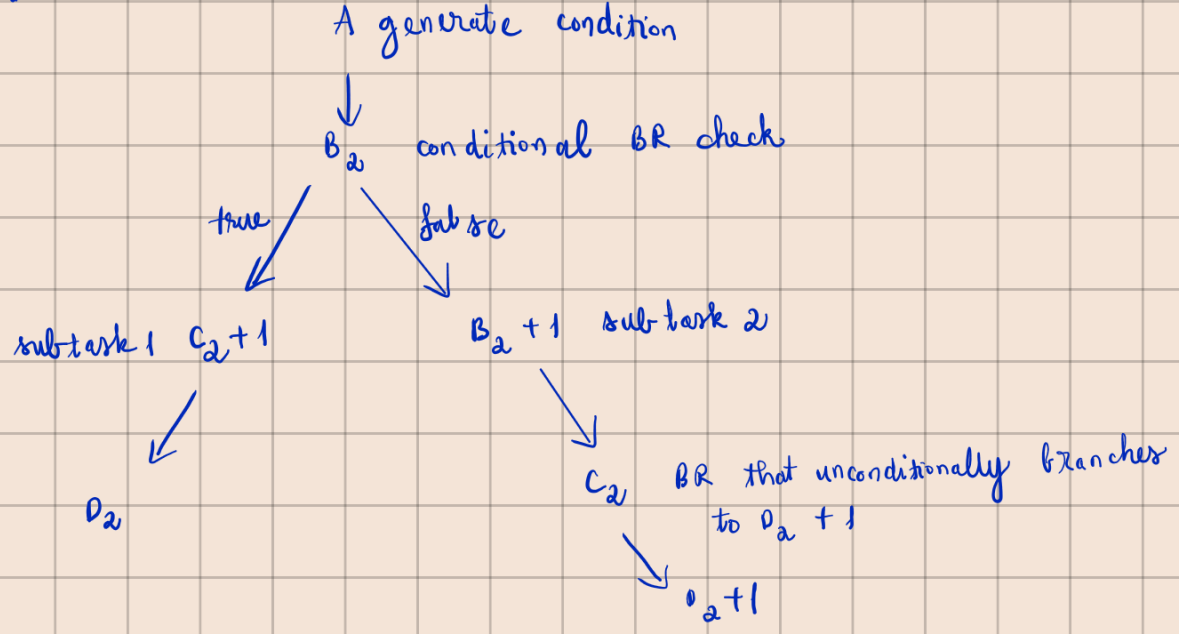
- Three construct to build the large unit of work are sequential, conditional, and iterative

- subtasks : components of a large task

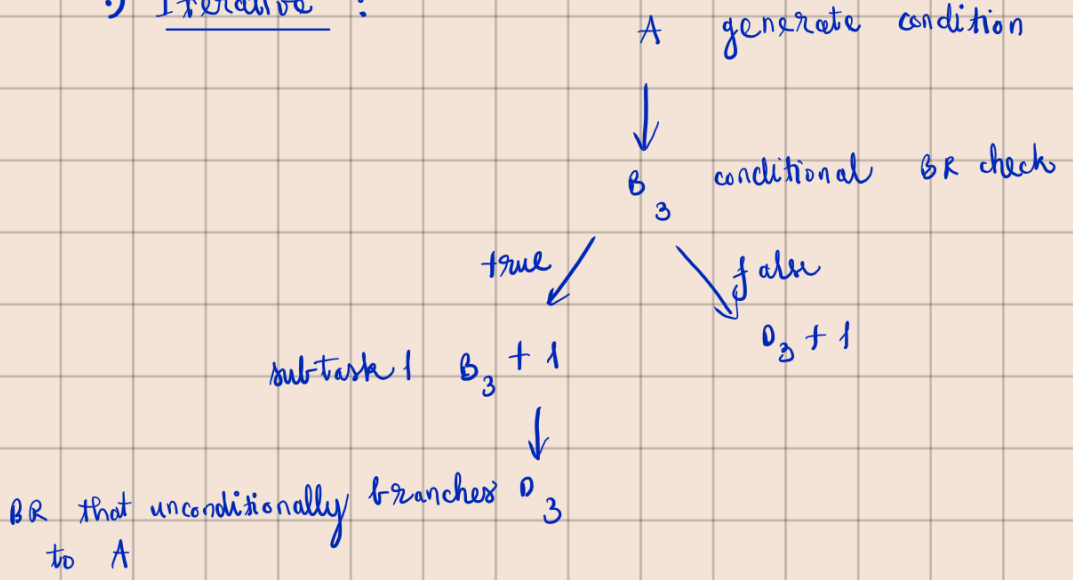
- Control flow :
  - Sequential :



  - Conditional :



·) Iterative :



·) Assembly language : low level

·) HALT : stop running the program

·) Instruction : Label Opcode Operands ; Comment
optional optional

·) Labels : symbolic names assigned to the memory locations
• A location can contain an instruction / value

·) Pseudo-Ops (Assembler Directives) : a message from programmer to the assembler to help it process
translate program in assemby → ISA of the LC-3

① .ORIG : tell the assembler where in the mem to place the program

Ex : .ORIG x3050

② · FILL : set aside the next location in the program and initialize it

with a value / label

③ .BLKW : set aside a number of sequential mem locations

Ex : LOC .BLKW 2

④ .STRINGZ : initialize a sequence of n+1 mem locations, where a sequence of n characters is the operand . Each location holds a zero-extended ASCII code for the crpd char

↑ last loc contains x0000

Ex : .ORIG x3010
.STRINGZ "Hello. World!"

⑤ .END : tell assembler that it reached the end of the program, don't process any after

) Assembly Progress : 2 complete passes

① Identify the actual binary addrs crpd to the symbolic labels
→ This forms a symbol table

② Translate assembly instructions into crpd machine language instructions

) 1st pass : examines instructions in sequence, and increment the LC once for each inst

If an inst has a label → a symbol entry is made in the table

) 2nd pass : translate the program into LC-3 machine language inst

- Ex : LD R3, LTR (LTR = x3013) → load R3 with content in x3013

Have $PC^* = x3002$

We need offset to create binary string repre the inst

$\Rightarrow$ offset = x3013 − x3002 = x0011

Note : because offset is 8-bit $\Rightarrow$ LTR $\in PC + [-255, 256]$

- Executable image : the entity being executed by a computer, created from modules
  - Each module is translated into an object file containing the instructions in the computer used for that module

- Multiple Object Files : Ex : the program as one module and the input data file as another
  - .EXTERNAL (pseudo-op) : identify the START of FILE as not known yet, will be known when translating the other modules (START of FILE = beginning addr of the input file)

- Subroutines : functions for Assembly

- Call/return mechanism : the user makes a call instruction to code A, and after the computer execute code A, it makes a return instruction to the next instruction in the program
  - Caller : the program that contains the call inst
  - Callee : the function

- JSR (R) : loads PC with the starting addr of the subroutine, and loads R7 with the addr immediately after the JSR (R) inst

where to come back after finishing the subroutine
→ return linkage

- JSR : use PC-relative addr mode
  Format: opcode | 1 | offset 11

- JSRR : use Base Register addr mode
  Format opcode | 0 | 0 0 | BaseR | 0 0 0 0 0 0

) Saving and Restoring Registers: the subroutines may return values by writing into MER that already had some values in them
→ The subroutines have to store those existed values into memories before overwrite them
↳ Callee Save

- Caller Save : the caller does the storing for R7 which will be destroyed by JSR (R)

) Stack : last thing we stored in a stack is the first thing we removed from it
- push (), pop ()
- top :

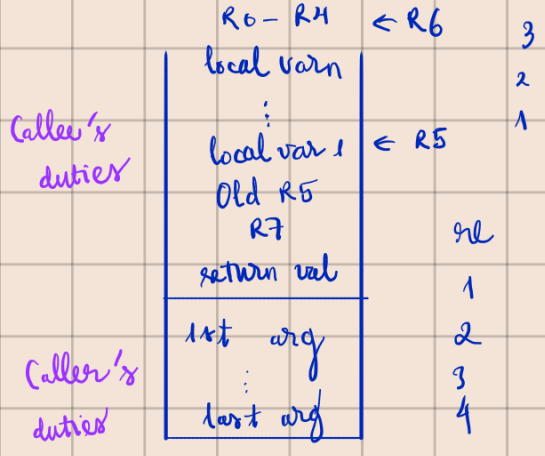
) Memory's implementation : we change the top pos instead of moving around the contents of mem locations
- R6 stores the addr of the top stack → stack pointer
- data are stored in mem locations whose addr is decreasing

) Under flow : pop() when the stack is empty
→ R5 = 1 if underflow
0 otherwise

- Overflow : push 1) _____ full
  - → R5 = 1 if overflow
    0 otherwise

- How a subroutine can get arguments, return value, as well as restore registers' values (including R7) ? → Stack!

- Frame pointer : a ptr used for
  - Accessing local vars from the stack
  - Saving the return value
  - Accessing arguments from the stack



- C-language : Overall
  - Designed for writing OS, other languages, low-level hard drivers, cryptography, ...
  - Easily compiled and to produce compact, efficient code
  - Don't check runtime error → need to be careful
  - Language compiled directly into machine
  - C provides some features with C library
  - C is procedural, no objects / classes
    use C struct instead
  - Pointers are used in place of object ref

- Each func must have a unique name

Data types :

- It depends on your platform
  - char – exactly 8 bits
  - short int – at least 16 bits
  - int – at least 16 bits
  - long int – at least 32 bits
- You can tell with **sizeof**
- **sizeof** is a **compile-time** constant reflecting the number of bytes held by a data type **or** instance
- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
- **sizeof** char is 1.

- Doesn't have boolean type : 0 ≡ false (meaning '0' ≡ false
  any non-zero integer ≡ true

×00
↓

String : array of chars, ends with '0'

Ex : char s[] = "abc"
char *s = "abc"
cannot modify content of s

- Use strlen to get array's length
  #include <string.h>

Escaped characters :

- \n : new line
- \t : tab
- \r : carriage return
- \000 : ASCII 0
- \\ : \

C preprocessor (PP) : a separated entity (not in C), but can be executed by C-compiler

- Macros: reusable snippets of code, created by `#define MACRO_NAME code`
  - → Take the code snippet defined applied to the macro name
  - → Include the content of one source file into another file
  - → Provide additional direction to the compiler for compiling code

) #include: Convention: we only include files that end in ".h"

- Surrounding the file name with " ": It will look in the curr directory → sys directory
- < >: It only looks in sys dir

(~ Classes)

) Struct: a composite data type to group related vars into one place

- Contain no methods, and all members are publicly visible
- Names following the struct tag define instances of the struct
- Identifier / function names and struct tags occupy diff name spaces
  - → An identifier / function name can be the same as a struct tag

Ex:
```
struct car {
    char mfg[30];
    char model[30];
    int year;
};
```
type is "struct car"

```
struct car thinks_car;
```
type name (not just car) — struct tag

- Ref a struct member: Ex: thinks_car.model

- How to "assign" an array in C: copy the contents to the target var

Ex: strcpy (miker_car.model , "Camry-2024");
↓
should have enough room

- Pointer: contains memory addr of another var

Ex: int b = 29 mem location "b" holds 29
int *baddr = &b mem location "baddr" holds addr of b

- '&' := addr of
- '*' has 2 meanings:
  1. In type declaration: Ex: int *px = &x
     → '*' := pointer to
  2. In an expression: Ex: *px = i
     → '*' := the value in this addr (called dereference)

→ can be used to read and change the value stored at a known addr of a mem location

- For an array, the identifier is the pointer to the first element

Ex: int a[10]
a ≡ (int *p = a[0])

Note: a is a const ptr so we can't modify it

- We can also do arithmetic with pointer to get new pointers

Ex: int *p = &i
p = p+1 ≡ p = p + 1 x size of (*p)
↓ deref

→ If p is an int pointer, p + 1 is the address of the next int

Ex : a[5] ≡ * (a + 5)

- Operator "→" : ptr to a struct → a struct member

Ex :

```
struct myStruct {
      int a, b;
} *p; // p is a pointer to struct myStruct

(*p).a = (*p).b;
p->a = p->b;
```

Same meaning

- Typedef : a shortcut to create a new alias for a type

Ex : struct a b[5] → b is an array of 5 struct a

typedef struct a sa5[5]

sa5 c ≡ struct a c[5]

- function : type_returned function_name (para_type para)
  - Define function prototypes before main()
  - type_returned := void means the function doesn't return
  - para_type := void _____ take arguments
  - main() return an 8-bit value:
    - 0 means okay, everything else means problems

- Any function can exit and return value using exit(ret_val)

  Ex : exit(99)

- We can see the value on command line with echo $?

- Local var : var inside a func, where another func changes its value won't affect its original value in the host func

- Global var : var outside functions, where another func changes its value will also change it for other funcs

- main() :
```
int main( int argc, char *argv[] ) {
    return 0
}
```

  - The first function invoked when running the program
  - int argc := argument counter
  - char *argv[] := array of pointer to a char

- const : a type qualifier indicate that a var after initialized cannot be changed

  Ex : `const int n = 5`

- sizeof : display the number of bytes held by a data type or an instance

- GDB : an utility for debugging and executing programs

- C preprocessor : programs that modifies before compilation begins

- C Compiler : translate C code into machine code
  - Including : Source Code Analysis, Target Code Synthesis, and Symbol Table

- C Linker : Combine various object modules into a single executable

image

- Storage Class : tells where the data will be stored and who can see it

	Outside a function definition	Inside a function definition
<none>	**scope:** external definition **storage:** static address	**scope:** within the function **storage:** on the stack
auto	N/A	**scope:** within the function **storage:** on the stack
static	**scope:** within the file only **storage:** static address	**scope:** within the function **storage:** static address
extern	**scope:** external reference **storage:** static address; location determined by file containing the external definition (can't have initializer either)	**scope:** external reference **storage:** static address; location determined by file containing the external definition (can't have initializer either)
register	N/A	**scope:** within the function **storage:** register or stack (hint to compiler; use of & not allowed; seldom used)

same

a promise that the var will be initialized when the program runs

an extern var can also takes data from other files

Note : static var only initialized once at the program load

- Type qualifiers : it is part of the object's type

① Const : the value of this var can't be changed after initialization

② Volatile : the compiler may not optimize ref to this var

③ Restrict : for the life time of a ptr, only it or a value directly derived from it can be used to access the object pointed by the ptr

- void : unspecified type

- Memory Layout : 4 major memory regions : data, stack, heap, code or text

  heap: related to mem allocating function

- extern (storage class) : definition
  - Another C file allocate storage, not current file
  - A typical way to link global variables between C files

- Declaration : introduce an identifier and its type (scalar / array / struct / function)

- Definition : instantiate the identifier

- Base Type : the type (optionally a storage class and/or a type qualifier
  - Applies to all names up until the ';'

  Ex : `static volatile long int i, *j, k[10];`

  Base type

  Declarators

- Reading type : ① Start at the identifier
  ② Go right until meeting a (/) / end (a function)
  ③ Go left

Ex : `int *(**f)()`

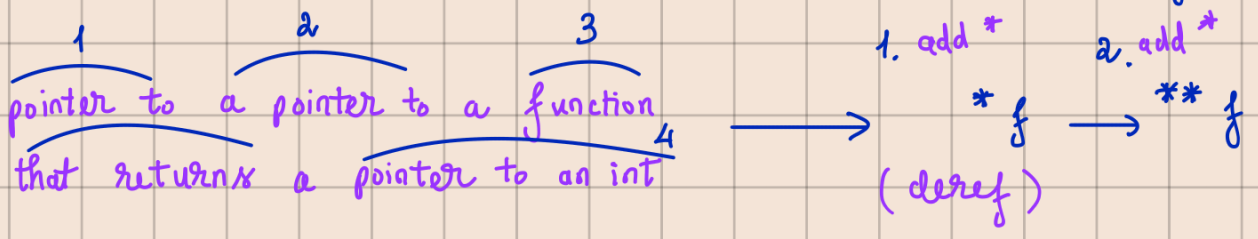
1. Go right : `f)` just f

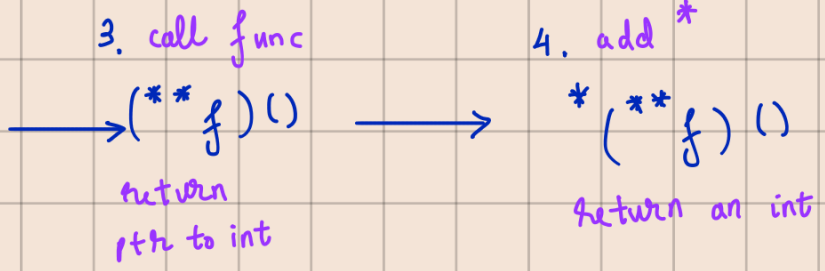
2. Go left : (** f) pointer to a pointer

3. Go right : (** f)() pointer to a pointer to a function

4. Go left : int* (** f)() pointer to a pointer to a function that returns a pointer to an int

:) Unwind a type : just apply the operators in the order they are named





:) GBA Programming : Programming on bare metals, no operating system

:) Data types :

- Integers (signed / default / unsigned)
  - char (1 byte)
  - short int OR short (2 bytes)
  - int (4 bytes)
  - long (8 bytes)

- Floating point (to be avoided because they're software emulated)
  - Float
  - double

- Address space : 32 bit

- Display screen : 240 x 160 pixel color video display screen

  - Hardware supports : video memory → video controller → display

- Device register : specific memory locations that control various hardware functionalities like the display, input, and sound

- REG_DISPCTL : a device register that control many video modes on the GBA, its address is at 0x04000000

  - Access it by using *(unsigned short *) 0x04000000

  - 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

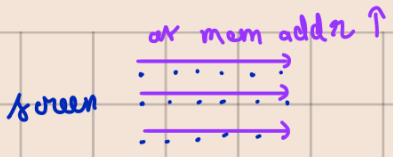
    (0B 0A 09 08: Bit mapped graphic (BG); 03 02 01 00: Mode)

  - Mode : Tile Modes: 0, 1, 2
    Bitmap Modes: 3, 4, 5

  - Use BG2 for bitmapped graphics

- Video Memory : Use mode 3, and to show the screen, use mode 2

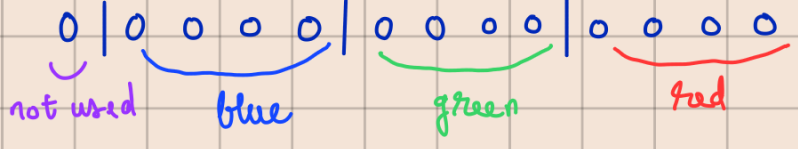
  - Starts at 0x06000000
  - Consists of 160 x 240 16-bit unsigned shorts (represent colors)
  - Mem locs store colors of pixels going left-to-right up-to-down



addr (row , col) = row * 240 + col

- Screen buffer : a portion of video memory that store the data representing all the pixels of a complete video frame

- Colors : color of pixel is represented by a 16-bits unsigned short



- Blanking :
  - HBlank : 68 pixels
  - VBlank : 68 scanlines

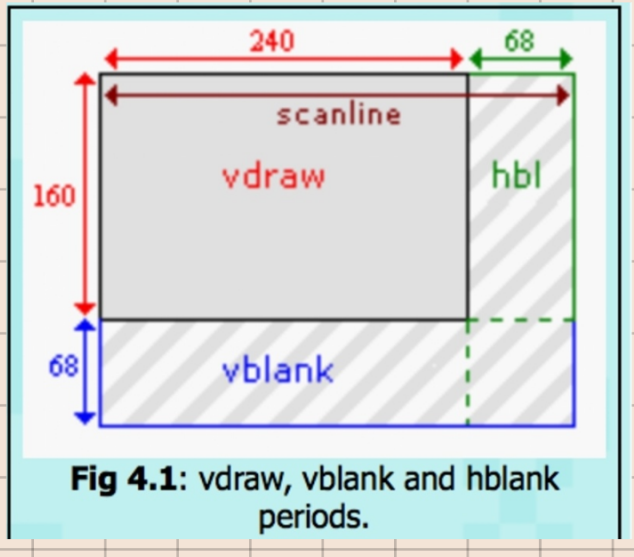


Fig 4.1: vdraw, vblank and hblank periods.

- Display timing :

subject	length	cycles
pixel	1	4
HDraw	240px	960
HBlank	68px	272
scanline	Hdraw+Hbl	1232
VDraw	160*scanline	197120
VBlank	68*scanline	83776
refresh	VDraw+Vbl	280896

~ 60 Hz

Table 4.1: Display timing details

- Draw and blank periods : After a scanline has been drawn (the

HDraw period - 240 px), there is a pause (HBlank, 68 pixel) before it starts drawing another scanline

- After 160 scanlines (VDraw) have been draw, there will be 68 scanline blank (VBlank) before starting over again

:) Tearing: change pixel's input in the middle of VDraw
=> Output: half this screen, half another screen

- To avoid tearing, update data during VBlank

:) Palette: 2 palettes, one for sprites (objects), one for backgrounds

- Both contains 256 entries of 16 bit colors
- There are 2 ways to use palettes:
  - Consider it as a single palette with 256 colors
  - Consider it as a collection of 16 sub-palettes of 16 colors
- Index 0 is the transparency index. In palette mode, pixels w/ value 0 is transparent

:) DMA (Direct Memory Access): A fast way of copying data from one place to another

:) DMA Channels: 4 channels

- 0: highest priority, used for time-critical operations and can only be used with internal RAM

- 1, 2: used to transfer sound data

· 3 : general-purpose copier, can be used for loading in new bitmap or tile data

·) Back to C

·) One dim array : ex : int ia [6]

Address : ia ≡ & ia [0]

·) Two dim array : ex : int ia [3][6]

Address : ia ≡ & ia [0][0]

ia [0] ≡ & ia [0][0] Adr of row 0

ia [1] ≡ & ia [1][0] Adr of row 1

·) Pointer calculations : let char arr [r][c] size of (char) = 1

Calculate pointer to arr [x][y]

int offset = (x * r + y) * sizeof (int)

int * p = (int *) ((char *) arr + offset)

Note : for 3D array and i, j, k be the indices,

offset = i * rows * cols + j * cols + k

·) Why declaring arr [][2] works, but arr [][] doesn't ?

· Because we to calculate arr [i][j] we only need to know # cols

· Declaring arr [] is also fine since we don't need to move to

another row (multiplying with # rows is used to move to different rows)

•)

	Can ptr value be changed?	Can "Hello" be changed?
char * cp = "Hello"	Yes	No
char ca[] = "Hello"	No	Yes

•) <u>sizeof</u> : char c_1 [] = "hello"
char * c_2 = "hello"

- sizeof (c_1) = 6 (including null terminator '\0')
- size of (c_2) = 8 (size of a pointer on your system)
- strlen (c_1) = strlen (c_2) = 5

<u>Fun fact</u> : ia [3] ≡ * (ia +3) ≡ * (3 + ia) ≡ 3 [ia]

•) Array of pointers : <u>ex</u> : static char * name [] = { "Illegal name", "name 1", "name 2", ... , "name 10" }

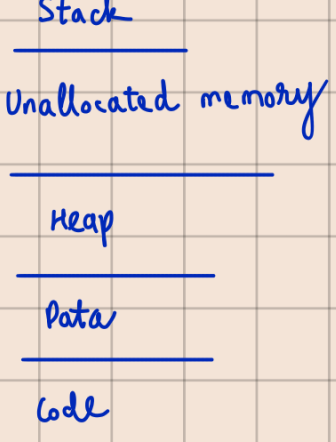
- Also called Marginally Indexed Array

•) <u>Size of struct</u> : size of struct = Σ size of element + filler
- Filler helps "align" elements in memory
- Align means data types / memory addr has to be a multiple of their size

<u>Tips</u> : always use sizeof

- Memory allocation : what if we need some space for user to enter data

- Malloc(size) : reserve a block of memory of size and return a pointer to it
  - return a generic ptr of type void * → ptr to generic type
  - share the heap with other library functions that request storage directly from OS



  - Based on brk() and sbrk()

- Program break : the mem location that represents the end of heap memory and the start of unallocated memory

  - Default is on top of the heap
  - We can move the program break into unallocated mem to get some extra mem

- brk(void * end_data_segment) : Sets the program break to the location specified by end_data_segment

- sbrk(int * increment) : increments the size of the program break by increment

- xp = malloc (size of struct x)

if (rp == NULL)) wrong

- Using rp before checking → crash
- Dereferencing a NULL pointer throws an exception

Correct: ① if ((rp = malloc(sizeof struct r)) == NULL) {
// handle error here
}

or

② if (!(rp = malloc (sizeof struct r))) {
// handle error here
}

·) Should cast the returned generic-type ptr of malloc to its correct type by casting
- If not, the compiler will silently cast it to any type of ptr

·) free(): free the allocated memory back to the heap
- After free, garbage data will be in the place even though the ptr still exists

·) void * calloc (size_t num, size_t size) : allocates ≥ num * size byte of the memory on the heap, zero-out them, and returns a ptr to it

·) void * realloc (void * ptr, size_t n) : reallocates ≥ n byte of mem on the heap, return a ptr to it
- Copies the data starting at ptr that was previously allocated
- Often used to expand the mem size for an existing object on the heap

·) **Memory leaks**: programmer loses track of memory allocated by malloc or other functions that call malloc

·) **Realloc implementation:**

- Find space for new allocation
- Copy original data into new space
- Free old space
- Return ptr to new space

Note: realloc(NULL, n) ≡ malloc(n)

(no need to copy existed data)

realloc(cp, 0) = free(cp)

(new space = NULL)