* Preference towards actual computer, will gladly except simulated computer - more of demonstration of data flow
* 10:30 to 11:00 is a good time for him to talk, if we want to talk in person again we can
* The memory blocks will not be moved while being read, meaning all blocks will need at least 2 copies
* The current suggested plan is for us to draw a rough diagram of the pieces of computer and send it to him to edit and go back and forth with
* Targeting computer to be 8 bit
* For the sake of understanding 3 stacks means his original design and traditional means a layout more commonly seen in computers and on traditional block diagrams
* He wants the machine to last as long as possible
* We need from him examples of code to get a scope for storage size, and a explanation of pointers
* He can write assembly language and explain it
* Blocks become pieces of code?
* We need to know a unit
* Code definition- The instructions to perform the intended actions(telling it to add then divide)
* Data definition- The numbers being acted upon( the 5, 4, and 3 in the equation (5+4)/3)
* Stack definition- Keeping track of the progression through the program(knowing to go first add, then divide)
* Stack I am the least clear on and i understand data however i don't understand when it would be appropriate to put numbers into it, both stack and data could be explained further
* He mentioned the usage of LED's in the memory, although i'm not sure exactly what or how they would be used- potentially to illuminate the path the block is taking, but it didn't sound like that was what he was insinuating
* He mentioned not needing all the registers used in the diagrams, not sure which ones we need
* Thoughts-
* We need to create the first diagram to send to him for review
* The first related design consideration would be how much of 3 stacks vs traditional will we use
* From what i understand the 3 stacks in the 3 stacks design are the different memory storages- clarification would be appreciated
* The next design consideration would be how general purpose would we want to be with the mechanical design
* General purpose relating to the ability of the machinery to put any block any where
* The most general purpose design we could do would be a single gantry moving everything, and being able to reach all parts of the display
* a more efficient design would be having a gantry only removing the blocks from there storage and then having conveyors move the blocks, how ever this design is less able to move any blocks to any place
* The downside of the more efficient design is that it removes flexibility and requires us to have a much more in-depth knowledge of the exact process that will occur
* The biggest concern i would have with the single gantry design would be the speed at which it would work, i don't think it would be sufficient to have the programs complete in an appreciable amount of time
* The mechanical design i am leaning towards would be broken up into different modules, within each module their would be a gantry, this would allow for any block to be placed anywhere , while still having the machine work quickly
* To move forward we are going to want to have him get very specific with what programs he wants to run and how the date storage will be displayed
* During the discussion he mention how he was trying write a bubble sort algorithm that was under 256 bytes, if the code takes anywhere near that many blocks to show we would only be able to display a few programs, and only one large one
* Once we agree on a general layout we need to meet to have him explain what pointers are
* The other big question is where exactly on his 3 stacks design are the computations taking place, for the first iteration of the design i think we should just treat the 3 stacks like memory only and have the traditional designs computation area and have him correct us if we are wrong

## Questions

* Are there any early computers/instruction sets/architectures we could base our design off of?
* What parts of the 3 stacks correspond to the parts of traditional design?
* What exactly do the blocks represent?
* How will the memory be divided up?
* How will results be displayed?
* What portions will be electronic vs mechanical?
* How will the LEDs be used in the design?
* What exactly are pointers?
* What do pointers do?
* How will we physically represent pointers in our design?
* Can we have access to the web page he showed me with the interactive design of his 3 stacks concept
* How fast does the machine need to be?
* What programs are going to be included in the machine
* Will the machine itself be programmable?
* Are we going to have separate memory representing permanent and random access?
* Can we represent assembly in hexadecimal form and then include an easy explanation with it?

# Design Questions

* What do we want people to learn from each section of the machine?
  + Example: For memory, are we teaching about a more realistic structure of memory and how data is stored, or is the memory going to be representative, not showing how it actually looks, instead having the included data be readable to humans so it is clear what is being stored
* How theoretical vs realistic does the design need to be?
  + A purely theoretical design would act as a demonstration of the theory behind how computers work.The design would be less focused on being able to do traditional calculations and more on pure demonstration of the cause and effect that allow the machine to function. The best example would be a Turing machine.
  + A purely realistic design would be less focused on demonstrating the core theory behind the inner workings in a computer and would be more focused on replicating a modern processor. This design would be more complicated, having more pieces than an older, simpler computer. The best example of this would be a computer that copies the core layout and architecture of a modern processor architecture(Any of the Zen or Lake architectures)
* How in depth vs practical will the design be?
  + An extremely in depth design would show more of the inner workings of the computer, a more in depth design would need to be larger and more complex and would work slower. An increase in the depth would make the computer less capable of demonstrating complex programs.
  + An extremely practical design would show less of the inner workings and would simplify where possible. A design like this would be smaller and would work faster. This style of design would allow for more complex programs to be demonstrated

\*different sections of the design could have different combinations of attributes so long as they are not significantly different

### More General Notes

# Types of Registers

1. **Accumulator**: This is the most common register, used to store data taken out from the memory.
2. **General Purpose Registers**: This is used to store data intermediate results during program execution. It can be accessed via assembly programming.
3. **Special Purpose Registers**: Users do not access these registers. These registers are for Computer system,
   * **MAR:** Memory Address Register are those registers that hold the address for the memory unit.
   * **MBR:** Memory Buffer Register stores instruction and data received from the memory and sent from the memory.
   * **PC:** Program Counter points to the next instruction to be executed.
   * **IR:** Instruction Register holds the instruction to be executed.

# General Notes

High vs low registers

"AH-DH"

The high bit AH, BH, CH, and DH registers.

"AL-DL"

The low bit AL, BL, CL, and DL registers.

It's quoted from AMD64 volume 1, also I have seen that in Intels programmer manual.

I don't clearly understand what does it mean. Does it have anything to do with endian order? Since both amd and intel microprocessors are little-endian order.

Can it be explained as:

AH: 0 0 0 0 0 0 1 0

Is read from the first bit so it returns: 2 (in decimal)

But the same value in

AL: 0 0 0 0 0 0 1 0

Is read from the end and it returns: 64 (in decimal)

AH is the greater half of AX, AL is the lesser half, and similarly for the B,C, and D registers. Because we are directly specifying the high order bits, or the low order bits, rather than just asking for whatever comes first, endianness doesn't really come into play.

AX = 0x288

\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_

/ \

0000 0010 1000 1000

\\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_/

AH=0x2 AL=0x88

TLDR: Maybe no need, but we still could

### Word

The memory stores **binary information(1's and 0's)** in groups of **bits** called **words**. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary coded information.

The fourth is the accumulator. It is a register that is similar to the general purpose register. It holds data or the results of an operation during the processing cycles. The accumulator can hold one of the two operands during any ALU operation. If we want to add two bytes together, this would just be an example here; one byte would go to the accumulator, and the other in memory or general purpose register. When executed the two numbers serve as the input to the ALU. You have two bytes, one would go to the accumulator, one to, probably, the general purpose register and then when executed the two numbers serve as inputs to the ALU, which we talked about up here. The output of the ALU is fed back into the accumulator. The accumulator would receive the results, the two bytes. The specific features of accumulators will vary from processor to processor. The accumulator is the only register that can perform the shift function that we discussed earlier. Remember if we shifted right or shifted left we could effectively divide by two or multiply by two.

What is index addressing?

Determine memory address mode- Immediate, Direct, Indirect, Indexed

Registers used are 16 bit registers

Current set of registers does not address the Bus interface unit

### Segment Registers

The CPU contains four segment registers, used as base locations for program instructions,

data, or the stack. In fact, all references to memory on the IBM PC involve a segment

register as a base location.

The registers are:

CS – Code Segment, base location of program code

DS – Data Segment, base location for variables

SS – Stack Segment. Base location of the stack

ES – Extra Segment. Additional base location for variables in memory.

### Index Registers

The index registers contain offsets from a segment register for information we are

interested about. There are four index registers:

BP – Base Pointer, offset from SS register to locate variables on the stack

SP – Stack Pointer, offset from SS register as to the location of the stack’s top

SI – Source Index, used for copying strings, segment register varies

DI – Destination Index, used for destination for copying strings\

## Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments −

* Code Segment − It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
* Data Segment − It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
* Stack Segment − It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Apart from the DS, CS and SS registers, there are other extra segment registers - ES (extra segment), FS and GS, which provide additional segments for storing data.

In assembly programming, a program needs to access the memory locations. All memory locations within a segment are relative to the starting address of the segment. A segment begins in an address evenly divisible by 16 or hexadecimal 10. So, the rightmost hex digit in all such memory addresses is 0, which is not generally stored in the segment registers.

The segment registers stores the starting addresses of a segment. To get the exact location of data or instruction within a segment, an offset value (or displacement) is required. To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location.

The stack is a Last In First Out (LIFO) data structure; data is pushed onto it and popped off of it in the reverse order.

mov ax, 006Ah

mov bx, F79Ah

mov cx, 1124h

push ax *; push the value in AX onto the top of the stack, which now holds the value 0x006A.*

push bx *; do the same thing to the value in BX; the stack now has 0x006A and 0xF79A.*

push cx *; now the stack has 0x006A, 0xF79A, and 0x1124.*

call do\_stuff *; do some stuff. The function is not forced to save the registers it uses, hence us saving them.*

pop cx *; pop the element on top of the stack, 0x1124, into CX; the stack now has 0x006A and 0xF79A.*

pop bx *; pop the element on top of the stack, 0xF79A, into BX; the stack now has just 0x006A.*

pop ax *; pop the element on top of the stack, 0x006A, into AX; the stack is now empty.*

The Stack is usually used to pass arguments to functions or procedures and also to keep track of control flow when the call instruction is used. The other common use of the Stack is temporarily saving registers.

Memory addressing schemes:

1. An Absolute Address, such as 04A26H, is a 20 bit value that directly references

a specific location.

2. A Segment Offset Address combines the starting address of a segment with an

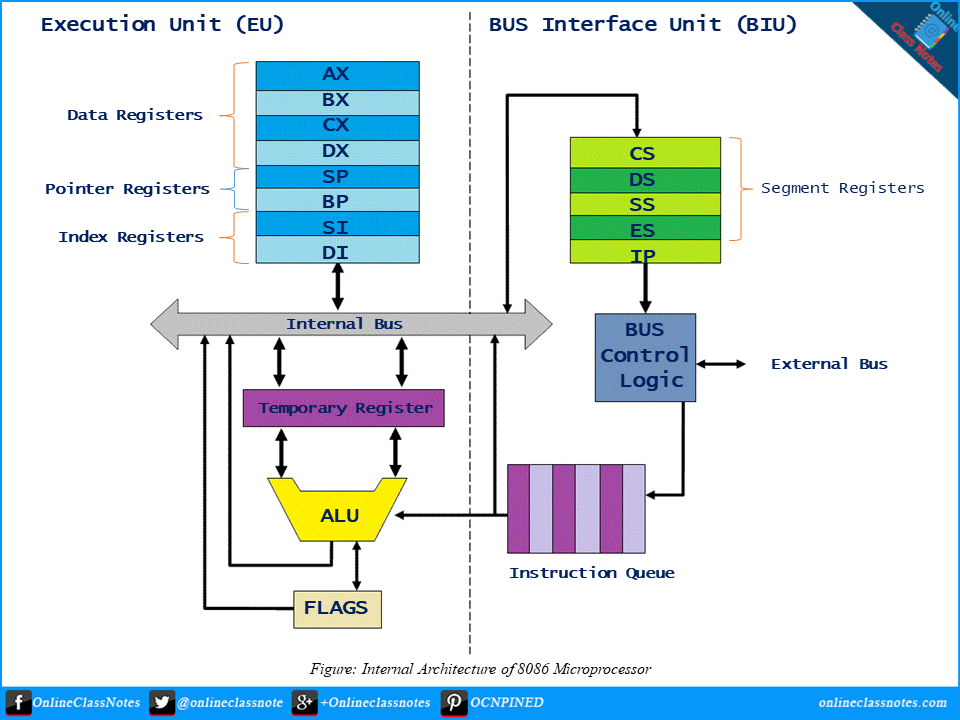
offset value.

How will commands that differentiate between the high and low portions of the registers function

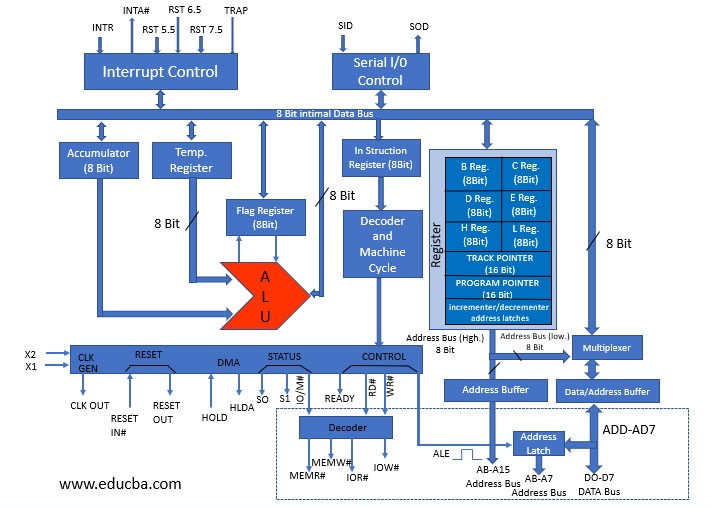
Need in depth explanation for how this will be handled

8086 processor has a 20bit address bus meaning it can sore about a 1mb of memory

8086 architecture



8085 architecture



Code segment: contains the program code (instructions)

Data segment: used to store data (information) to be processed by the program

Stack segment: used to store information temporarily

Figure – Format of flag register

There are total 9 flags in 8086 and the flag register is divided into two types:

(a) Status Flags – There are 6 flag registers in 8086 microprocessor which become set(1) or reset(0) depending upon condition after either 8-bit or 16-bit operation. These flags are conditional/status flags. 5 of these flags are same as in case of 8085 microprocessor and their working is also same as in 8085 microprocessor. The sixth one is the overflow flag.

The 6 status flags are:

Sign Flag (S)

Zero Flag (Z)

Auxiliary Cary Flag (AC)

Parity Flag (P)

Carry Flag (CY)

These first five flags are defined here

Overflow Flag (O) – This flag will be set (1) if the result of a signed operation is too large to fit in the number of bits available to represent it, otherwise reset (0). After any operation, if D[6] generates any carry and passes to D[7] OR if D[6] does not generates carry but D[7] generates, overflow flag becomes set, i.e., 1. If D[6] and D[7] both generate carry or both do not generate any carry, then overflow flag becomes reset, i.e., 0.

Example: On adding bytes 100 + 50 (result is not in range -128…127), so overflow flag will set.

MOV AL, 50 (50 is 01010000 which is positive)

MOV BL, 32 (32 is 00110010 which is positive)

ADD AL, BL (82 is 10000010 which is negative)

Overflow flag became set as we added 2 +ve numbers and we got a -ve number.

(b) Control Flags – The control flags enable or disable certain operations of the microprocessor. There are 3 control flags in 8086 microprocessor and these are:

Directional Flag (D) – This flag is specifically used in string instructions.

If directional flag is set (1), then access the string data from higher memory location towards lower memory location.

If directional flag is reset (0), then access the string data from lower memory location towards higher memory location.

Interrupt Flag (I) – This flag is for interrupts.

If interrupt flag is set (1), the microprocessor will recognize interrupt requests from the peripherals.

If interrupt flag is reset (0), the microprocessor will not recognize any interrupt requests and will ignore them.

Trap Flag (T) – This flag is used for on-chip debugging. Setting trap flag puts the microprocessor into single step mode for debugging. In single stepping, the microprocessor executes a instruction and enters into single step ISR.

If trap flag is set (1), the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction.

If trap flag is reset (0), no function is performed.

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Since the Instruction Pointer (IP) is 16 bit it means you can only have 64k instructions (2^16), which wasn't much even in the 80s. So to expand the address space you have a second register which addresses 64k blocks. You could consider cs:ip together as one 32 bit register which is then capable of addressing 2^32 bytes...ie 4G which is what you get on a processor which uses 32 bit addresses. The 8086 was using 20 bits of addresses, so you could access 1M of memory.

The physical address is calculated from 2 parts. i) segment address. ii) offset address. The CS(code segment register) is used to address the code segment of the memory i.e a location in the memory where the code is stored. The IP(Instruction pointer) contains the offset within the code segment of the memory. Hence CS:IP is used to point to the location (i.e to calculate the physical address)of the code in the memory.

8086's ALU doesn't have support for multiplication, division, or multiple-bit shifts, even though the 8086 has instructions for these operations. These operations are computed in microcode using simpler ALU operations (shift, add, subtract for multiplication and division, and repeated single-bit shifts for larger shifts).

## Memory Address Mode

Types of addressing modes:

**Register mode –** In this type of addressing mode both the operands are registers.  
Example:  
MOV AX, BX

XOR AX, DX

ADD AL, BL

**Immediate mode –** In this type of addressing mode the source operand is a 8 bit or 16 bit data. Destination operands can never be immediate data.  
Example:  
MOV AX, 2000

MOV CL, 0A

ADD AL, 45

AND AX, 0000  
Note that to initialize the value of segment register an register is required.  
MOV AX, 2000

MOV CS, AX

**Displacement or direct mode –** In this type of addressing mode the effective address is directly given in the instruction as displacement.  
Example:  
MOV AX, [DISP]

MOV AX, [0500]

**Register indirect mode –** In this addressing mode the effective address is in SI, DI or BX.  
Example:  
MOV AX, [DI]

ADD AL, [BX]

MOV AX, [SI]

**Based indexed mode –** In this the effective address is the sum of base register and index register.  
Base register: BX, BP

Index register: SI, DI  
The physical memory address is calculated according to the base register.  
Example:  
MOV AL, [BP+SI]

MOV AX, [BX+DI]

**Indexed mode –** In this type of addressing mode the effective address is sum of index register and displacement.  
Example:  
MOV AX, [SI+2000]

MOV AL, [DI+3000]

**Based mode –** In this the effective address is the sum of base register and displacement.  
Example:  
MOV AL, [BP+ 0100]

**Based indexed displacement mode –** In this type of addressing mode the effective address is the sum of index register, base register and displacement.  
Example:  
MOV AL, [SI+BP+2000]

**String mode –** This addressing mode is related to string instructions. In this the value of SI and DI are auto incremented and decremented depending upon the value of directional flag.  
Example:  
MOVS B

MOVS W

**Input/Output mode –** This addressing mode is related with input output operations.  
Example:  
IN A, 45

OUT A, 50

**Relative mode –**In this the effective address is calculated with reference to the instruction pointer.  
Example:  
JNZ 8 bit address

IP=IP+8 bit address

Instruction Register and Instruction Decoder:

The EU fetches an opcode from the queue into the instruction register. The instruction decoder decodes it and sends the information to the control circuit for execution.

base pointer allows code to independently reference data that have been pushed previously on the stack