

EI209 Computer Organization Final Materials (produced by Galaxies99)

Lecture 01. Introductions

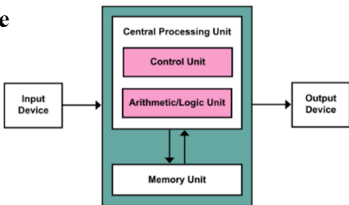
1. What is the Computer Science? Computer science is the study of mathematical algorithms and processes that interact with data and that can be represented as data in the form of programs. It enables the use of algorithms to manipulate, store, and communicate digital information. A computer scientist studies the theory of computation and the practice of designing software systems. (Source: Wikipedia)
2. What is a computer? "All problems in computer science can be solved by another level of indirection." (David Wheeler)
Computer has many abstractions. Here are some examples.
 - a) A Turing machine is a mathematical model of computation that defines an abstract of machine, which manipulates symbols on a strip of tape according to a table of rules;
 - b) Abstraction in OS: files (text files, executable files, etc.) and disk (different drives/directories);
 - c) Abstraction in WWW: website and webpage;
 - d) Abstraction in C Programming language: variables, data types, function calls, etc.

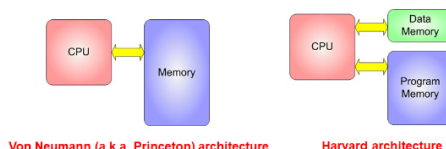
Computer Abstraction:

Other Application Software	IDE (e.g. Visual Studio)
Operating System (OS)	
Computer	

3. Course contains programming interface & computer organization.

4. Von Neumann Architecture

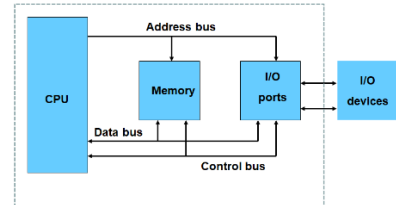
- a) Five components partitioning: Input, Output, Memory, ALU, Control Unit.
 
- b) **Three main components:** I/O, Memory and CPU.
- c) **Three key abstractions:** Data, Instruction and Sequential execution model, corresponding to variables, statement and sequential execution model in C programming language, but C programming language has more abstractions, such as function call, etc. Actually, C is just a higher-level abstraction of Von Neumann Architecture.
- d) Abstraction & Components Interactions:
 - i. Data and instruction are stored in memory;
 - ii. CPU executes instruction in sequential order;
 - iii. Instruction can read and write memory;
 - iv. Instruction can perform Arithmetic/Logic operations.
- e) a.k.a., **Princeton Architecture**.
- f) Comparing with **Harvard architecture**:
Harvard architecture has separate data and inst memories.



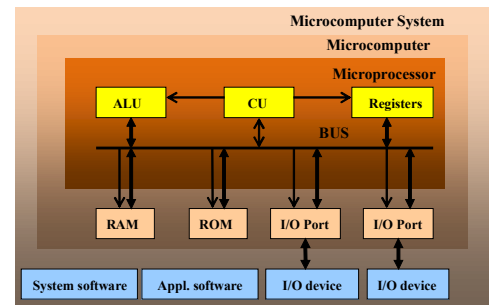
- g) Three key concepts in Von Neumann Architecture:
 - i. Both instructions and data are stored in a single read-write memory;
 - ii. The contents of memory are addressable by location, without regard to the type of data;
 - iii. Execution occurs in a sequential fashion.

5. Microcomputer: including CPU, memory, I/O ports and buses.

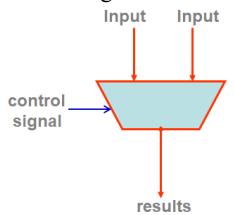
Structure:



6. Microcomputer system: including microcomputer, peripheral I/O devices and software (system software, applications software).



7. CPU

- a) Key concepts: core (including ALU, CU and registers), clock (the CPU circuitry is made by digital circuits), ISA;
- b) Layers: sand, transistor, logical gate, and ISA;
- c) **ALU** (arithmetic logic unit): a multifunctional calculator performs arithmetic functions and logic functions according to the particular control signals; It has two inputs, and the calculation result can be temporarily stored in one of the registers.
 
- d) **Instruction:** An instruction is a pre-defined code which defines a specific operation, processing and exchanging information among CPU, memory and I/O devices.
- e) **CU** (control unit): control unit works under instructions; CU contains an instruction decoder, which can decode an instruction and generates all control signals, coordinating all activities within the computer; and a program counter, which points to the address of the next instruction to be executed.
- f) **Instruction Set (ISA):** All recognizable instructions by the instruction decoder. Important ISAs are CISC and RISC.
 - i. **CISC:** Complex Instruction Set Computers, e.g., x86 family. Variable instruction length, variable execution time of different formats of instructions.

more instruction formats, upwardly compatible (new ISA contains instructions in old ISA).

- ii. **RISC:** Reduced Instruction Set Computers, e.g., MIPS, ARM. Fixed instruction size, fixed execution time of all instructions, easy to pipeline the RISC instruction (fast), fewer formats (simple hardware, shorter design cycle).

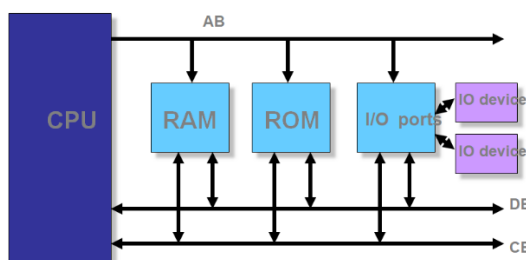
8. Memory

- a) Some definitions
 - i. Bit (b): a binary digit;
 - ii. Byte (B): consists of 8 bits, is the smallest unit that can be addressed in microcomputers;
 - iii. Nibble: is half a byte;
 - iv. Word: the number of bits that a CPU can process at one time. The length of it depends on the width of the CPU's registers and the width of the data bus.

$$|\text{Word}| = \min(|\text{Reg}_{ALU}|, |\text{Bus}_{Data}|)$$
 - v. Double word;
 - vi. Kilo-, Mega-, Giga-, Tera-, Peta-, ...
- b) Memory hierarchy: cache, primary memory (RAM, ROM), secondary memory (disk, tape, optical memory, etc.).

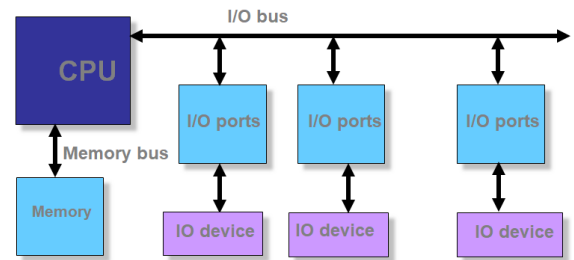
9. Bus

- a) A bus is a communication pathway connecting two or more devices, that is, a shared transmission medium and one device at a time.
- b) **System bus:** connects major computer components (processor, memory, I/O etc.)
- c) **Arbitration:** centralized (bus controller responsible for allocating time on bus, master/slave, master controls the buses and slave passively waits for command), distributed (each module has access logic and collaborate.)
- d) **Type:** dedicated (e.g., physical dedication), multiplexed (time multiplexing).
- e) **Timing:** synchronous (events on the bus is determined by a global clock, a single 1-0 transmission is referred to as a bus cycle), asynchronous (devices have their own clocks and communicate before and after an event).
- f) **Single bus structure:** A bus connects all modules; simple but poor performance in terms of throughput.

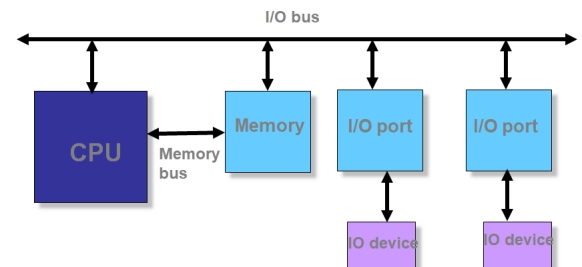


- g) **CPU-Central Dual-Bus Structure:** a dedicated bus between CPU and memory, and a dedicated bus between CPU and I/O devices; efficient in terms of data transfer but

information between memory and I/O devices has to go through CPU and result in poor CPU performance.



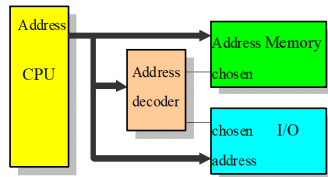
- h) **Memory-Central Dual-Bus Structure:** a dedicated bus between CPU and memory, and a I/O bus connecting I/O ports, memory and CPU; gain both high CPU performance and data transfer throughput.



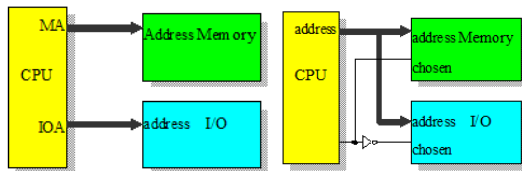
- i) **Data Bus:** used to provide a path from moving data between system modules. Bidirectional (CPU read, memory or I/O device to CPU, and CPU write, CPU to memory or I/O device.) The width of the data bus is often as wide as the registers of a CPU, and both of them define a word of this computer; it also determines how much data the processor can read or write in one memory or I/O cycle.
- j) **Address Bus:** used to designate the source or destination of the data on the data bus that the processor intends to communicate with. Unidirectional (CPU to memory or I/O device). The width of the address bus n determines the total number of memory locations addressable by a given CPU is 2^n .
- k) **Control Bus:** used to control each module and use of data and address buses, sending command and timing information between modules (e.g., memory read/write, I/O read/write and bus grant/request). Two sets of unidirectional control signals (Command signal, from CPU to memory or I/O device; State signal, from memory or I/O device to CPU). In control bus, input/output is defined from the processor's point of view (e.g., when memory read is active, data is input to the processor.).

10. I/O: input/output

- a) **Addressing scheme to accessing I/O:** memory-mapped I/O and isolated I/O.
- b) **Memory-mapped I/O:** one single address space for both memory and I/O, status and data registers of I/O modules are treated as memory locations, and use the same machine instructions to access both, no special commands for I/O.

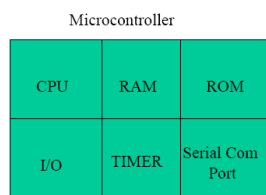


- c) **Isolated I/O:** two separate address spaces for memory and I/O modules, and use different set of accessing I/O devices, which means special commands for I/O, and it may need I/O or memory select lines.



The two figures above are all isolated I/O, but with different implementation (dedicated address lines on the left and multiplexing address lines on the right.).

11. **Microcontrollers (MCS):** a microcontroller has a CPU in addition to a fixed amount of ROM, RAM, I/O ports on one single chip. It is ideal for applications in which cost and space are critical (e.g., TV remote controller, etc.).



12. **Embedded System:** an embedded system uses a microcontroller or a microprocessor to do one task and one task only (e.g., TV remote controller, keyless entry, etc.). Using microcontroller is cheap but sometimes inadequate for the task. Microcontroller differ in terms of their RAM, ROM, I/O sizes and types.

13. (optional) About **Harvard Architecture**.

- Pros: Efficient Pipelining - Operand Fetch and Instruction Fetch can be overlapped; Separate Buses for data and instructions; Tailored towards an FPGA implementation.
- Cons: Not widely used; More difficult to implement; More pins.

Lecture 02. I/O and Memory

1. Memory Basic

- Physical Types of memory: semiconductor (RAM and ROM), magnetic (disk and tape), optical (CD and DVD).
- Location: CPU (registers), internal memory (cache and main memory), external memory (disks, CD and DVD).
- Capacity: determined by word size (not the word size in Lecture 01, it is the word size of the memory) and the number of words. (e.g., 2M words with 8-bit word size memory has the same capacity as 16M words with 1-bit word size memory.)

- Unit of transfer:** the maximum of bits that can be read or written into memory at a time.
 - Internal: usually a word, governed by data bus width;
 - External: usually a block, which is much larger than a word.
 - Example: the CPU can calculate one addition every cycle, but a memory transfer takes two cycles. With the memory interface width to be 4 words, the CPU can be kept with 100% utilization.)
- Addressable unit:** smallest location which can be uniquely addressed, normally a byte for internal memory, and cluster on disks.
- Access methods:**
 - Sequential: start at the beginning and read through in order; access time depends on location of data and previous location (e.g., tape).
 - Direct: individual blocks have unique address; access is by jumping to vicinity plus sequential search; access time depends on location and previous location (e.g., disk).
 - Random: individual addresses identify locations exactly; access time is independent of location or previous access (e.g., ROM and RAM).
 - Associative: data is located based on a portion of its contents rather than its address; access time is independent of location or previous access (e.g., cache).
- Performance**
 - Access time: time between presenting the address and getting the valid data;
 - Memory cycle time: access time, in addition to time may be required for the memory to recover before next access (recovery time).
 - Transfer rate: rate at which data can be moved; unit is transfer per second (e.g., GT/s).
 - Transfer bandwidth: equals transfer rate multiplied by transfer unit size; unit is byte per second (e.g., GB/s).

[Example] A memory transfer takes two cycles and each transfer has 4 bytes. Clock frequency is 1GHz.

$$A.T. = 2 \text{ cycles} = 2 \text{ ns}$$

$$T.R. = 0.5 \text{ T/cycle} = 0.5 \text{ GT/s}$$

$$T.B. = 0.5 \text{ GT/s} \times 4 \text{ B/T} = 2 \text{ GB/s}$$

- Volatility of Memory:** volatile memory loses data over time when power is removed (e.g., RAM); but non-volatile memory stores data even when power is removed (e.g., ROM);
- Static and Dynamic Memory:** static memory holds data as long as power is applied (e.g., SRAM); but dynamic

memory will lose data unless refreshed periodically (e.g., DRAM).

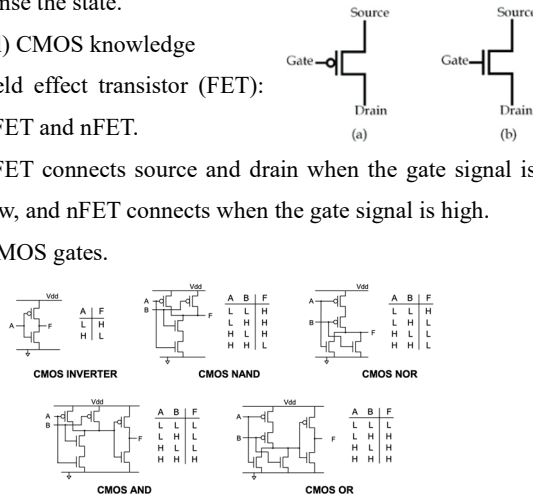
- j) **Semiconductor Memory Basics:** the basic element of a semiconductor memory is the memory cell, which exhibits two stable states, representing binary 1 and 0; it is capable of being written into set the state, and being read from to sense the state.

2. (optional) CMOS knowledge

- a) field effect transistor (FET):
pFET and nFET.

pFET connects source and drain when the gate signal is low, and nFET connects when the gate signal is high.

- b) CMOS gates.

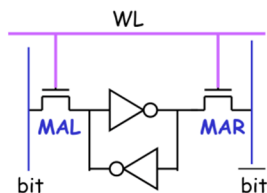


3. **RAM:** Random Access Memory, (misnamed) refers to memory with both read and write capabilities.

- a) Properties: read/write, volatile (temporary storage);
b) **SRAM** (Static-): bits stored as on/off switches; no charges to leak and no refreshing needed when powered.

Property: complex, large space, expansive, fast.

Usage: registers & cache.



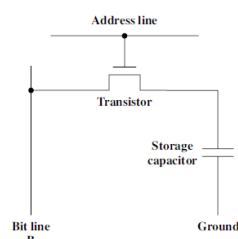
Operations: When the WL is 1, then the cell is chosen, both MAL and MAR are connected. Therefore, we can perform r/w operations by getting the bits from the lines or sending the bits to the lines (and wait some time for it to finish writing). When the WL is 0, the cell is not chosen, and it will hold the data in the infinite loop (2 inverters).

Totally: 6 semiconductors (2 for each inverter and 2 inverters, MAL, MAR). (a.k.a., 6T-SRAM).

- c) **DRAM** (Dynamic-): bits stored as charges in capacitors; charges leak and need refreshing even when powered.

Property: simple, small space, cheap, slow.

Usage: main memory.



Operations: When the Address Line is 1, then the cell is chosen, and we need to get the data from the bit line. At the same time, the charges will leak, and we need to refresh it by recharging it after reading/writing. When the Address Line is 0, then the cell is not chosen, but the charges will leak as time goes by.

Totally: 1 transistor (r/w enable) and 1 capacitor.

4. **ROM:** Read Only Memory, non-volatile and needs no power supply; no capabilities for “online” memory write operations, and its write typically requires high voltages or erasing by UVlight.

- a) Normal Applications (firmware): library subroutine, BIOS, function tables, etc.

- b) Types of ROM

- Written during manufacture: very expensive for small runs;
- Programmable (once): PROM, needs special equipment to program;
- Read “mostly”: erasable programmable (EPROM), which can be erased by ultraviolet radiation; electrically erasable programmable (EEPROM), which takes much longer to write than read; flash memory, which only supports to erase whole memory electrically.

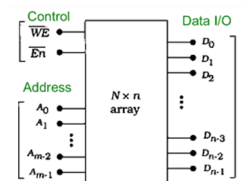
5. **Memory Hierarchy** in computer system: Registers, L1 cache, L2 cache, main memory, disk cache, disk, optical, tape, ...

Why? For memories, bigger is slower, faster is more expensive (both dollars and chip area); other techs have their places as well.

How? Move what we use to the fast and small memories, and backup everything in the big and slow memories. With good locality of reference, memory appears to be fast and big.

6. **RAM Organization:** $N \times n$ memory chip (N is the number of words, and n is the chip word length).

- a) Chip I/O: data in/out in $D_{n-1} \dots D_0$ pins, address in $A_{m-1} \dots A_0$ pins, and control pins includes \overline{WE} , which means write enable (assert low), and \overline{En} , which means block enable (assert low).



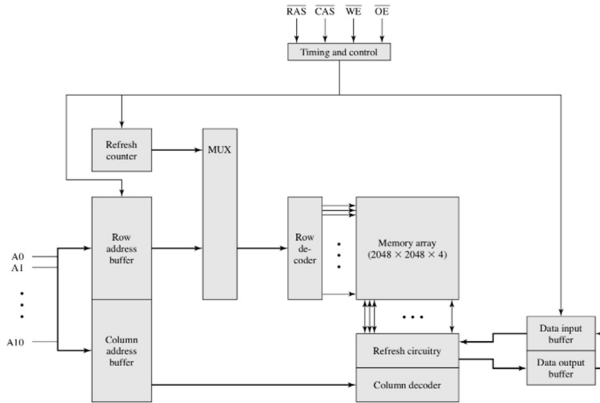
[Example] 4M*4 RAM, totally 16Mbit RAM chip; 4M (2^{22}) words, each word has 4bits, therefore, 22 address lines (no multiplexing) and 4 data lines.

- b) **Memory Array:** $N_R \times N_C$ array of 1-bit cells, where N_R representing the number of rows, and N_C representing the number of columns. Organizing RAM in this way allows **multiplexing** row address and column address. If not multiplexed, we usually regard higher part as row address and lower part as column address.

- c) **Multiplexing:** uses the address pins to represent the row number and the column number in different time

(according to \overline{RAS} and \overline{CAS}). Therefore, if we want to read the data in a cell, we just need to send the address twice. Therefore, less address pins are needed. Moreover if we add an additional address pin, we can actually multiply the capacity by 4 (2x line, 2x row).

- d) **Line/row decoder:** the line/row decoder is actually an n to 2^n decoder, it turns the bits in binary to the decimal value and find out the corresponding line.
- e) **Structure (with multiplexing)**



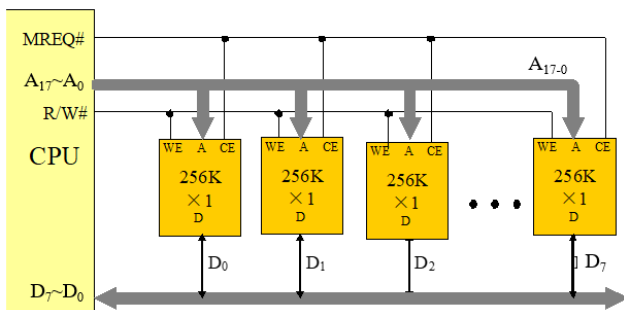
- f) **Pins explanations:** *WE*: write enable; *OE*: output enable (if output is disabled, then will block the output but the read operation will still be performed); *CE/CS*: chip enable or chip select (disable it will save ~80% power).

7. **Bit extension:** If the module needs bigger unit of transfer than that of given memory chips, then we need bit extension. In bit extension, every chip has the same address space. We just need to send the same address to the chips and gather the results.

Whether we need bit extension: can the data bus be occupied by a single chip? No – bit extension!

Notes: If the data bus width is larger than the addressable unit length, then we need to ignore the lower few bits, because these bits represents the position in the whole data word.

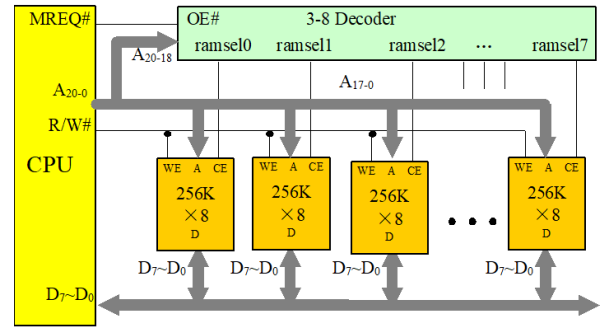
[Example] You have 256K*1bit RAM chips, and we need 256K*1byte memory. Here are the constructions.



8. **Word extension:** If the module needs larger number of words than that of given memory chips, then we need word extension. Chips in different group has different address range. We use the higher bits and a decoder to choose the chips.

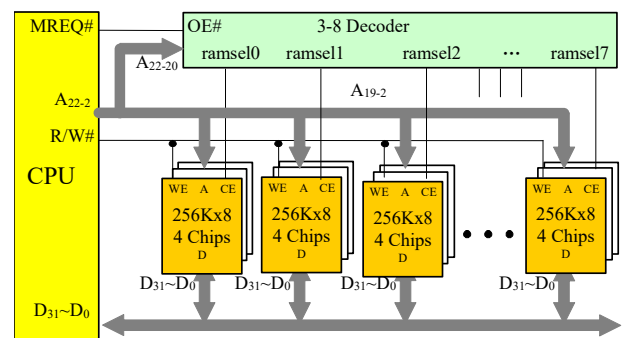
Whether we need word extension: can we use a single chip to represent the whole memory? No – word extension!

[Example] You have 256K*8bits RAM chips, and we need 2M*8bits memory. Here are the constructions.



Combined with bit extension, we can have the following example.

[Example] You have 256K*8bits RAM chips, and we need 2M*32bits memory. The addressable unit is byte. Here are the constructions.



Notice that we discard the last two address lines because the addressable unit is byte and our data bus width is 32bits, which is 2^2 bytes. Therefore, we can make the final conclusions.

Conclusion. How to perform bit/word extension?

Suppose CPU wants $N_{CPU} \times n_{CPU}$, where n_{CPU} is usually determined by the data bus width; and the memory chips we have are $N_{mem} \times n_{mem}$. If $N_{CPU} > N_{mem}$, then we need word extension, and we need to put the first few address lines into an extra decoder, and we connect data bus of CPU and all memory chips directly; If $n_{CPU} > n_{mem}$, then we need bit extension, and we may need to discard the last few address lines if the addressable unit length is less than the data bus length, that is, we need to discard:

$$\max(\log n_{CPU} - \log n_{addressable\ unit}, 0)$$

lines, and we also need to assemble memory chips' data buses to connect with CPU.

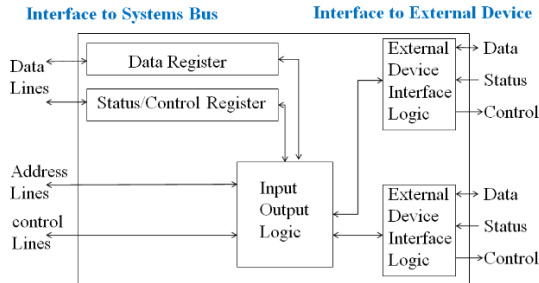
Notes: default addressable unit: 8 bits (1 byte).

9. **I/O Problems (why I/O modules?):** Wide variety of peripherals.
- Different operation logic, so impractical for CPU to control all kinds of devices.
 - Speak different "languages", that is, delivering different amount of data (serial/parallel) at different speed in different formats (analog, digital), so impractical for CPU to understand;
 - Slower than CPU and RAM, so impractical to directly connect devices with highspeed system bus.

10. External Devices

- Human readable: Screen, Printer, Keyboard;
- Machine readable: Monitoring and control;
- Communication: Modem, Network Interface Card (NIC).

11. **I/O Modules:** has an interface to CPU and memory and an interface to one or more peripherals. It's like a bridge, an interpreter, a buffer, ...



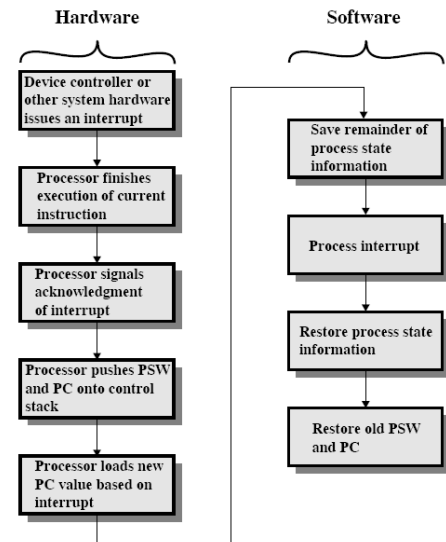
- Functions:** Control & Timing, CPU communication, device communication, data buffering, error detection.
 - Steps:** e.g. transfer of data from an external device to the processor: CPU first checks I/O module for device status; then I/O module returns the device status; if the device status is ready, CPU requests data transfer by means of a command to I/O module; I/O module gets a unit of data from device and transfer it to CPU.
 - Design Decisions:** Hide / reveal device properties to CPU; support multiple or single device; control device functions or leave for CPU.
 - Techniques:** Programmed, Interrupt-driven, DMA.
12. **Programmed I/O:** CPU executes a program that gives it direct control of the I/O operation, including sensing device status, sending read/write commands to the I/O modules and transferring data; and CPU will wait for I/O module to complete operation.
- Detailed Steps:** CPU requests I/O operation, and I/O modules performs operation and sets the status bits, which will be checked by CPU periodically; I/O modules does not inform CPU or interrupt CPU; CPU may wait or come back later again.
 - CPU viewpoint:** CPU issues address and identify modules if there are more than one devices; then CPU issues commands, such as control commands (telling module what to do, e.g., spin up disk), test commands (check status, e.g., power and error) and read/write commands (module will transfer data via buffer from/to device).
 - Addressing Scheme:** very like memory access, each device is given a unique identifier, and CPU commands contain the identifier (address) of the corresponding module (and device).
 - Problems:** simple, but if CPU is faster, it is a huge waste of CPU time.
13. **Interrupt driven I/O:** I/O modules will interrupt if they are ready, therefore it can overcome CPU waiting and do not need repeated

CPU checking of device.

- Detailed Steps:** CPU request I/O operation, and I/O module performs operation while CPU is doing other work; I/O module will inform CPU when something come up by interrupting CPU, and CPU will deal with the event.

- Interrupt:** new events needs CPU to handle first but CPU needs to go back to previous work after that.

Interrupt handler: save the PCB, deal with the interrupt, restore the PCB and go back to work.



- CPU viewpoint:** CPU issues read command and do other work, and CPU will check for interrupt at the end of each instruction cycle; and if interrupted, then save context (registers, etc.), handle interrupt (fetch data and store) and recover from the saved context (restore original registers, etc.).
- Design Issues:**
 - How can CPU know which module is issuing the interrupt? (interrupt identification scheme)
 - Connect a **dedicated line** for each module, but limits the number of devices that we can use;
 - Software poll:** all the devices share one common interrupt request line to interrupt CPU. Once get an interrupt, CPU will ask each module in turn to identify the device.
 - Hardware poll:** all the devices share one common interrupt request line to interrupt CPU. Interrupt ack signal is sent down a daisy chain (hardware circuits).
 - Bus master:** module must claim the bus before raising interrupt. (e.g., PCI & SCSI bus protocols)
 - Interrupt controller** (e.g., 8259)
 - How to locate the corresponding handler program when interrupted?
 - General handler program:** CPU enters this

handler every time it gets interrupted, and looks for the module responsible and gets the address of the corresponding handler program. (this method increases software complexity, because it must search for the module, but performance may be poor since we have to check each I/O module)

2. **Interrupt vectors:** pointers are used to link the handler programs, such pointers are **interrupt vectors**. We can use the pointer to find the handler program. (this method may increase hardware complexity, because it must transfer the pointer to CPU, but high-performance)

iii. How to deal with multiple interrupts?

1. Assign priority for interrupts: high-priority interrupts first.
2. Nesting of interrupts: high-priority interrupts can further interrupt low-priority interrupts.

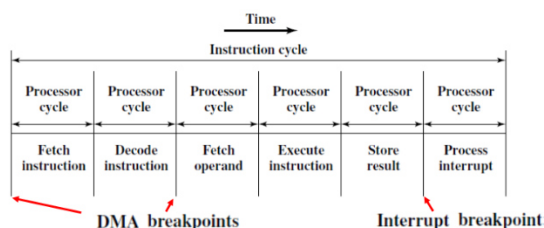
e) **Problems:** although it does not need CPU to check periodically, it still needs CPU involvement.

14. **DMA:** Direct Memory Access. DMA needs additional module (hardware) on bus; DMA controller will do the I/O work for CPU.

- a) **Detailed Steps:** CPU first tells DMA controller: read/write, device address, starting address of memory block for data, amount of data to be transferred, etc.; then CPU carries on with other work, and DMA controller deals with transfer; DMA controller sends interrupt when finished.

Notes: the *interrupt* is different from the *interrupt* in interrupt driven I/O. The *interrupt* here is the sign that the process is finished, and the *interrupt* of interrupt-driven I/O is the ready sign of the device.

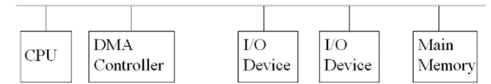
- b) **Cycle Stealing:** a perfect cycle stealing is that DMA transfers data when CPU is not accessing data bus; but most of the time, the processor may be suspended due to DMA operation in a processor cycle. That is, CPU suspended just before DMA controller accesses bus, and DMA controller takes over bus for a cycle, transfers of one word of data. This is not an interrupt because CPU does not switch context, therefore, it slows down CPU but not as much as CPU doing transfers.



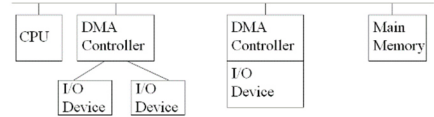
c) **DMA Configurations:**

- i. Single bus, detached DMA controller: each transfer uses bus twice (I/O to DMA, and DMA to memory);

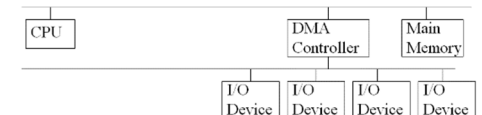
for one transfer, CPU is suspended twice.



- ii. Single bus, integrated DMA controller: each controller may support more than one device; each transfer uses bus once (DMA to memory); for one transfer, CPU is suspended once.



- iii. Separate I/O bus supports all DMA enabled devices: each transfer uses bus once (DMA to memory); for one transfer, CPU is suspend once.



15. **Summary of different I/O techniques:** increase the design complexity to improve performance.

Types of I/O techniques	Waiting for device	Transfer data from device to memory
Programmed I/O	Software (CPU)	Software (CPU)
Interrupt driven I/O	Hardware	Software
DMA	Hardware	Hardware

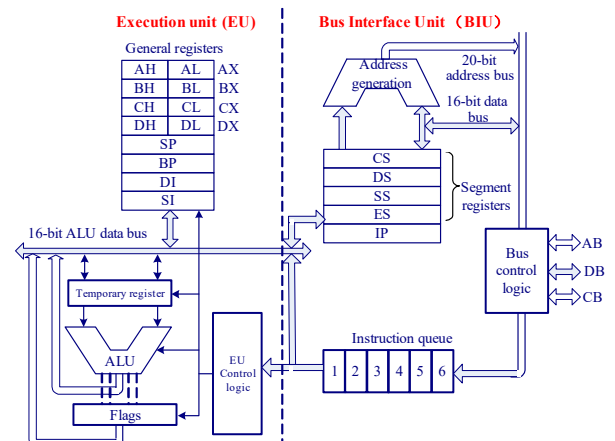
Lecture 03. 80x86 Microprocessor

1. Evolution of x86 family.

- a) 8086 (1978): first 16-bit microprocessor; 20-bit address bus, that is, $2^{20} = 1 \text{ MB}$ memory; segment-based memory manangement; first pipelined microprocessor.
- b) 8088: 16-bit internal, 8-bit external data bus; fit in the 8-bit world; adopted in the IBM PC + MS-DOS open system.
- c) 80286, 80386, 80486: add more features such as real/protected modes and virtual memory.

2. **Internal Structure of 8086**

- a) Two sections: Bus Interface Unit (BIU) and Execution Unit (EU).



- b) **Bus Interface Unit (BIU):** takes in charge of accesses to memory and peripheral I/O devices, including instruction

fetch, instruction queueing, operand fetch and storage, address relocation and bus control. It consists of:

- i. four 16-bit segment registers: CS, DS, ES, SS;
- ii. one 16-bit instruction pointer: IP;
- iii. one 20-bit address adder:

$$\text{Address} = \text{CS} \times 16 + \text{IP}$$

- iv. one 6-byte instruction queue

While EU is executing an instruction, the BIU will fetch the next several instructions from the memory and store them in the instruction queue.

- c) **Execution Unit (EU):** takes in charge of instruction execution. It consists of:
 - i. four 16-bit general registers: AX, BX, CX, DX;
 - ii. two 16-bit pointer registers: SP (stack pointer), BP (base pointer);
 - iii. two 16-bit index registers: SI (source index), DI (destination index);
 - iv. one 16-bit flag register: FR (9 of the 16 bits are used);
 - v. ALU.

3. **Registers of 8086:** on-chip storage, superfast but expensive; can only store information temporarily. Divided into 6 groups.

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note:
The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

4. **Pipelining of 8086:** two stages, fetch (by BIU) and execute (by EU); BIU fetches and stores instructions once the queue has more than 2 empty bytes; EU consumes instructions pre-fetched and stored in the queue at the same time. Pipelining increases the efficiency of CPU; In 8085 (non-pipelining), n instructions takes $2n$ cycles; but in 8086 (pipelining), n instructions only takes $(n + 1)$ cycles if perfectly pipelined. It only works under sequential instruction execution, and has branch penalty, that is, when jump instruction executed, all pre-fetched instructions are discarded.

5. **Flag Registers of 8086:** 16-bit flag register, a.k.a., status register, processor status word (PSW). It consists of three control flags (DF, IF, TF) and six conditional flags (CF, PF, AF, ZF, SF, OF).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag
SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

- a) **DF (direction flag):** a flag that controls the left-to-right or right-to-left direction of string processing;
- b) **IF (interrupt flag):** a flag set or cleared to enable or disable

only the external maskable interrupt requests (after reset, all flags are cleared, which means you have to set IF in your program if allow INTR);

- c) **TF (trap flag):** a flag that permits operation of a processor in single-step debugging mode. If such a flag is available, debuggers can use it to step through the execution of a computer program;
- d) **CF (carry flag):** a flag set whenever there is a carry out, from the 7th digit after a 8-bit operation, or from the 15th digit after a 16-bit operation;
- e) **PF (parity flag):** the parity of the operation result's low-order byte, set when the byte has an even number of 1s
- f) **AF (auxiliary carry flag):** a flag set if there is a carry from the 3rd digit to the 4th digit, used by BCD-related arithmetic;
- g) **ZF (zero flag):** a flag set when the result is zero;
- h) **SF (sign flag):** a flag copied from the sign bit (the most significant bit) after operation;
- i) **OF (overflow flag):** a flag set when the result of a signed number operation is too large, causing the sign bit error.

6. **Signed Number:** original value (cannot be added directly and has two zeros +0/-0); one's complement (can be added directly but has two zeros +0/-0); two's complement (can be added directly and only has one zero, one's complement plus 1).

The most significant bit (MSB) as sign bit, the rest of bits as magnitude. For negative number, D7 is 1, but the magnitude is represented in two's complement.

D7	D6	D5	D4	D3	D2	D1	D0
sign	magnitude						

CF is used to detect errors in unsigned arithmetic operations; CF=1 means there is a carry bit from the MSB, usually overflow; **OF** is used to detect errors in signed arithmetic operations; two ways to understand OF: (1) OF=1 when two values are positive but the adding result is negative, or when two values are negative but the adding result is positive; (2) e.g., for 8-bit operations, OF is set to 1 when there is a carry from the 6th digit to the 7th digit, and no carry from the 7th digit, or there is no carry from the 6th digit to the 7th digit, and a carry from the 7th digit.

[Examples]

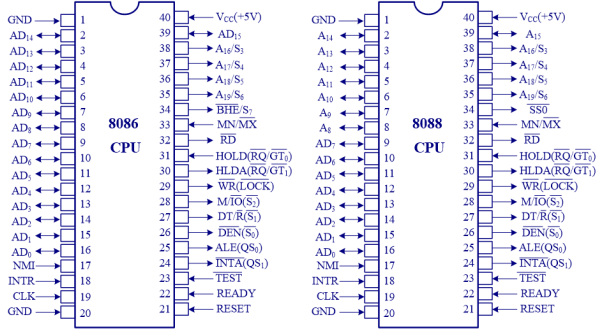
	38	0011	1000
+	2F	0010	1111
	67	0110	0111

CF = 0 since there is no carry beyond d7
PF = 0 since there is an odd number of 1s in the result
AF = 1 since there is a carry from d3 to d4
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero
OF = 0 since there is no carry from d6 to d7 and no carry beyond d7

+	96	0110 0000	-128	1000 0000
+	70	0100 0110	+ - 2	1111 1110
+	166	1010 0110	-130	0111 1110

According to the CPU, this is -90, which is wrong. (OF = 1, SF = 1, CF = 0)
According to the CPU, the result is +126
OF=1, SF=0 (positive), CF=1

7. **8086/8088 Pins.** The input and output are in CPU's point of view.
Something worth noticing: first, 8088 only has AD_0 to AD_7 because it only has 8-bit external data bus; second, AD means time multiplexing by addresses and data.

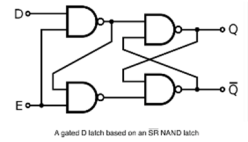


- MN/\overline{MX} pin: input, select **work modes**, 8086/88's two work modes are minimum mode and maximum mode;
 - minimum mode** is selected when $MN/\overline{MX} = 1$, it has single CPU and control signals from CPU;
 - maximum mode** is selected when $MN/\overline{MX} = 0$, it has multiple CPUs (8086+8087) and 8288 control chip supports. (8087 is used for floating operations.)
- \overline{RD} pin: output, CPU is reading from memory or I/O;
- \overline{WR} pin: output, CPU is writing to memory or I/O;
- M/\overline{IO} pin: output, CPU is accessing memory (high level) or I/O (low level);
- $READY$ pin: input, memory or I/O is ready for data transfer;
- \overline{DEN} pin: output, used to enable the data transceivers;
- DT/\overline{R} pin: output, used to inform the data transceivers the direction of data transfer, that is, sending data (high level) or receiving data (low level);
- \overline{BHE} pin: output, when $\overline{BHE} = 0$, AD_8 to AD_{15} are used; otherwise AD_8 to AD_{15} are not used; (for data bus)
- ALE pin: output, used as the latch enable signal of the address latch;
- $HOLD$ pin: input signal, hold the bus request;
- $HLDA$ pin: output signal, hold request ack;
- $INTR$ pin: input, interrupt request from 8259 interrupt controller, maskable by clearing the IF in the flag registers;
- $INTA$ pin: output, interrupt ack;
- NMI pin: input, non-maskable interrupt, CPU is interrupted after finishing the current instruction; cannot be masked by software;
- $RESET$ pin: input signal, reset the CPU; IP, DS, SS, ES and the instruction queue are cleared and $CS = FFFFH$. Therefore, the address of the first instruction the CPU will execute after reset is FFFF0H.

8. Sequential Logic & Combinational Logic

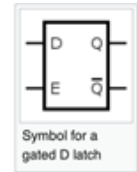
- CMOS gates are combinational logic circuits, the output is only determined by the input;
- D Latch is the sequential logic circuit.

9. **D-Latch:** (similar to SRAM cell) there is a state in D-latch and we output the state in Q and \overline{Q} pins. If the latch is not enabled, then we cannot change the state stored in latch; otherwise we can change the state by providing new data through D pin.

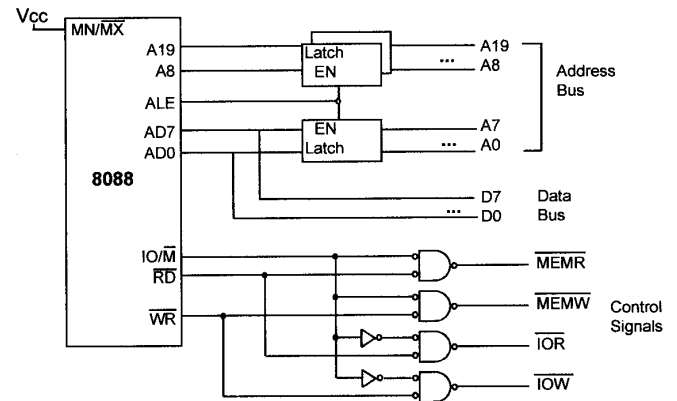


Gated D latch truth table

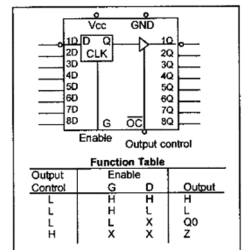
E/C	D	Q	\overline{Q}	Comment
0	X	Q_{prev}	\overline{Q}_{prev}	No change
1	0	0	1	Reset
1	1	1	0	Set



10. **Memory/IO Control Signal:** the memory/IO control signal is generated using the following method.

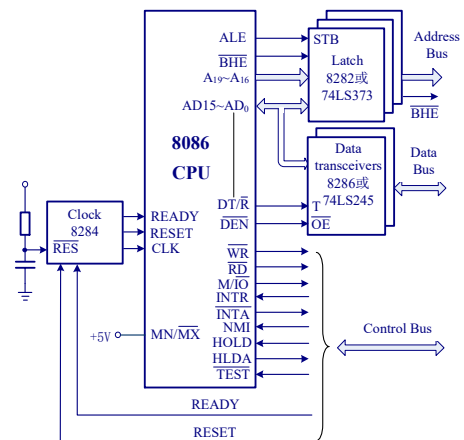


11. **Address/Data Demultiplexing & Address latching:** there are 20 latches corresponding to 20 bits address, ALE pin is connected to the Enable pins in the latches; when ALE is activated and we send the address data to the latches, and after sending the ALE will be deactivated, then the address will be stored in latches for later use; then we can send the data through AD_0 to AD_{15} and there is no need to worry about that we may modify the address.



Function Table

Output Control	Enable	D	Output
L	H	H	L
L	L	X	Q_0
H	X	X	Z



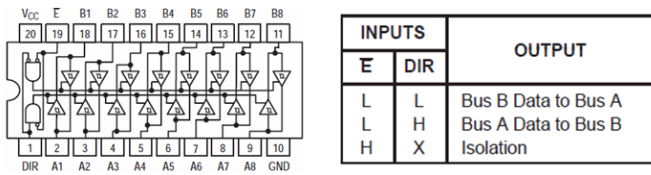
Lines Connection: ALE connects to G ; and \overline{BHE} , AD_{15} to AD_0 and A_{19} to A_{16} connect to latch pins; \overline{OC} connects to the ground (always output).

12. **Data Transceiver:** because the data bus is bidirectional, so we need data transceivers to protect the CPU signals. So the data

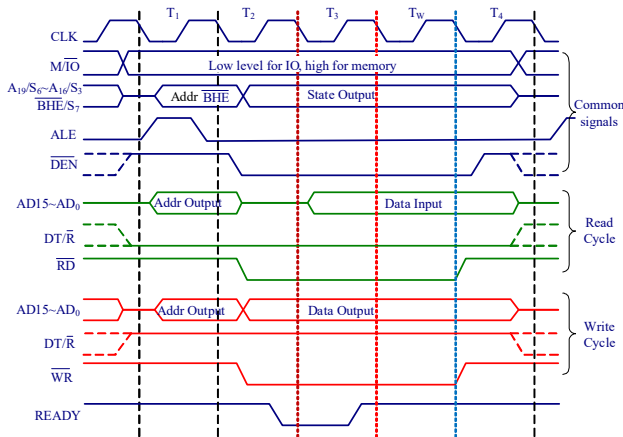
transceiver is basically the bit extension result of some tri-gates.

Tri-gates guarantee that the signal can only flow at one directions.

Lines Connection: \overline{DEN} connects to \overline{E} ; DT/\overline{R} connects to DIR ; AD_{15} to AD_0 connect to the A pins of transceiver.



13. The bus cycle for data transfers in 8086: at least 4 cycles.



14. Memory Management in 8086: a segment in 8086 is a memory block which includes up to 64 KB and begins on an address evenly divisible by 16, that is, an address looks like XXXX0H. (why? because we need two registers to represent the address, and the offset register has 16 bits, which means the size of a segment is $2^{16} = 64$ KB; since we use segment \times 16 + offset to represent the address, the address of the begin element should be like so.) A typical program on 8086 consists of at least 3 segments.

- Code segment:** contains instructions that accomplish certain task;
- Data segment:** stores information to be processed;
- Stack segment:** stores information temporarily.

15. Logical & Physical Address in 8086:

- Physical Address:** 20-bit address that is actually put on the address bus; a range of 1MB from 00000H to FFFFFH; mapped to the actual physical location in memory;
- Logical Address:** consists of a **segment value** (determines the beginning of a segment) and an **offset value** (a relative location within a 64KB segment).
- Logical Address to Physical Address:** shift the segment value left one hex digit (4 bits), then adding the value to the offset values; one logical address corresponds to only one physical address.

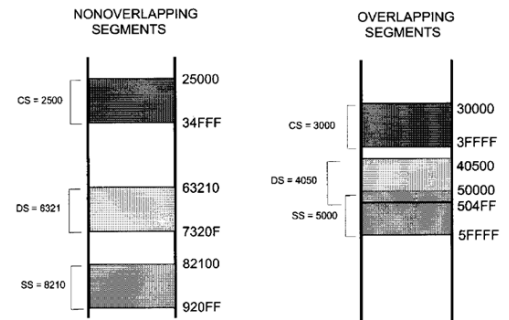
$$PA = \text{segment} \times 16 + \text{offset}$$

- Segment range representation:** maximum 64 KB; logical address ranges from 2500:0000H to 2500:FFFFH, and physical address ranges from 25000H to 34FFFH.
- Physical Address Wrap-around:** When adding the offset

to the shifted segment value results in an address beyond the maximum value FFFFFH, the address will wrap around from 00000H again.

[Example] if the segment value is FF59H, then the physical address range is from FF590H to 0F58FH.

- Physical Address to Logical Address:** one physical address can be derived from different logical addresses, e.g., physical address 15020H can be derived from 1000:5020H, 1500:0020H, etc..
- Segment Overlapping:** according to the address scheme, two segments can overlap because of dynamic behavior of the segment and offset concept, and it may be desirable in some circumstances.



16. Code Segment in 8086: 8086 fetches instructions from the code segment; the logical address of an instruction is $CS:IP$ and we can use it to generate physical address to retrieve the instruction from memory. If desired instructions are physically located beyond the current code segment, then we can change the CS value so that those instruction can be located using new logical address.

17. Data Segment in 8086: information to be processed is stored in the data segment with a logical address of a piece of data $DS:offset$, where *offset* value can be immediate value ranging from 0000H to FFFFH and also can be *offset* registers for data segment (BX, SI and DI); then we can use the logical address to generate physical address to retrieve data (8 bits or 16 bits) from memory. If desired data are physically located beyond the current data segment, then we can change the DS value so that those data can be located using new logical address.

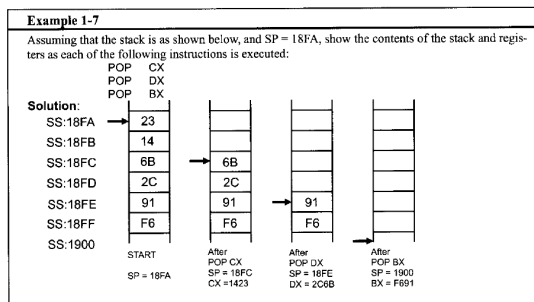
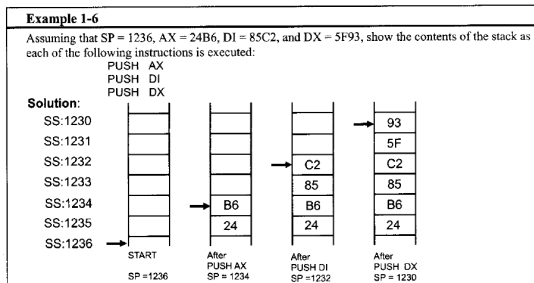
Data representation in memory: Memory can be logically imagined as a consecutive block of bytes. If the data size is larger than a byte, then we can use the little endian or the big endian to store the data;

- Little endian:** the low byte of the data goes to the low memory location;
- Big endian:** the high byte of the data goes to the low memory location.

18. Stack Segment in 8086: a section of RAM memory used by the CPU to store information temporarily with a logical address of a piece of data $SS:SP$ (special applications with BP , that is, $SS:BP$). Most registers (except segment registers and SP) inside

CPU can be stored in stack and brought back into the CPU from the stack using *push* and *pop* respectively. Stack grows downward from upper addresses to lower addresses in the memory allocated for a program in order to protect other programs from destruction, and ensure that the code section and stack section would not write over each other.

[Example] Push and Pop operation. Notice here we use little endian.



19. **Extra Segment:** an extra data segment, essential for string operations within logical address of a piece of data *ES:offset*, where *offset* value can be immediate value ranging from 0000H to FFFFH and also can be *offset* registers for extra segment (*BX*, *SI* and *DI*).
20. [Example] IBM PC memory map: 1MB logical address space; 640K max RAM (MS-DOS, application software; DOS does memory management and you do not set *CS*, *DS* and *SS*); video display RAM; ROM includes 64KB BIOS and various adapter cards.
21. **BIOS Function:** BIOS (basic input-output system) tests all the devices connected to PC when powered on and reports errors if any, and it will load DOS from disk into RAM and hand over control of PC to DOS. Therefore, the first instruction that CPU will execute after being reset is to go to the location of the BIOS program.
22. **Addressing Modes in 8086:** 8086 has seven distinct addressing modes. Take MOV instruction, which has a format of MOV destination, source, as an example. Notice that the destination and source of MOV should have the same size.

- a) **Register Addressing Mode:** Data are held within registers, and no need to access memory. Data can be moved among all registers except CS (can not be set) and IP (can not be accessed by MOV).

```
MOV BX,DX ;copy the contents of DX into BX
MOV ES,AX ;copy the contents of AX into ES
```
- b) **Immediate Addressing Mode:** The source operand is a

constant embedded in instructions, and no need to access memory. Immediate numbers cannot be moved to segment registers. Therefore, if we want to use an immediate value to update a segment register, we may need an intermediate register.

```
MOV AX,2550H ;move 2550H into AX
MOV CX,625 ;load the decimal value 625 into CX
MOV BL,40H ;load 40H into BL
```

- c) **Direct Addressing Mode:** Data is stored in memory and the address is given in instructions, and we need to access memory to gain the data. The default segment is DS; but we can specify the segment we want.


```
MOV DL,[2400] ;move contents of DS:2400H into DL
```
- d) **Register Indirect Addressing Mode:** Data is stored in memory and the address is held by a register, and we need to access memory to gain the data. The default segment is DS, but we can specify the segment we want. Registers for this purpose are SI, DI and BX.

```
MOV AL,[BX] ;moves into AL the contents of the memory location
             ;pointed to by DS:BX.
```

- e) **Based Relative Addressing Mode:** Data is stored in memory and the address can be calculated with base registers BX and BP as well as a displacement value, and we need to access memory to gain the data. The default segment is data segment *DS* for *BX* and stack segment *SS* for *BP*.

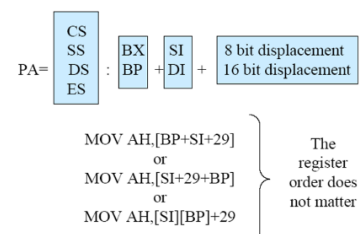

```
MOV CX,[BX]+10 ;move DS:BX+10 and DS:BX+10+1 into CX
                ;PA = DS (shifted left) + BX + 10
MOV AL,[BP]+5 ;PA = SS (shifted left) + BP + 5
```
- f) **Indexed Relative Addressing Mode:** Data is stored in memory and the address can be calculated with index registers *DI* and *SI* as well as a displacement value, and we need to access memory to gain the data. The default segment is data segment *DS* for both.

```
MOV DX,[SI]+5 ;PA = DS (shifted left) + SI + 5
MOV CL,[DI]+20 ;PA = DS (shifted left) + DI + 20
```

- g) **Based Indexed Relative Addressing Mode:** Combines based and indexed addressing modes, one base register and one index register are used, and we need to access memory to gain the data. The default segment is data segment *DS* for *BX* and stack segment *SS* for *BP*.

```
MOV CL,[BX][DI]+8 ;PA = DS (shifted left) + BX + DI + 8
MOV CH,[BX][SI]+20 ;PA = DS (shifted left) + BX + SI + 20
MOV AH,[BP][DI]+12 ;PA = SS (shifted left) + BP + DI + 12
MOV AH,[BP][SI]+29 ;PA = SS (shifted left) + BP + SI + 29
```

We can change the order of registers to get the same result.



Notes: For all instructions involving memory, they will move consecutive places in memory to satisfy the need of the register.

Segment Overrides: offset registers are used with default segment registers, but 8086 allows the program to override the default segment registers by specifying the segment register in the code.

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

Instruction	Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Lecture 04. Assembly Language Programming (1)

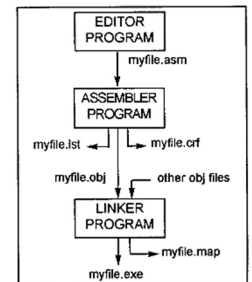
1. Programming Languages

- Machine language:** binary code, for CPU but not human beings;
 - Low-level language:** deals with the internal structure of a CPU, such as assembly language.
 - Assembly language:** mnemonics for machine code instructions; hard to program and poor portability but very efficient.
 - High-level languages:** do not have to be concerned with the internal details of a CPU, such as C, C++, Pascal, Python, Perl, etc. It's easy to program and good portability but less efficient.
- Assembly Language Programs:** includes a series of statements (assembly language instructions perform the real work of the program and are instructions for CPU; directives, a.k.a. pseudo-instructions, give instructions for the assembler program about how to translate the program into machine code) and multiple segments (CPU can access only one data segment, one code segment, one stack segment and one extra segment; but assembly language can define multiple segments and tells CPU which segment to use).
 - Form of a statement:** `[label:] mnemonic [operands] [;comment]`
 - label** is a reference to this statement; each label must be unique; letters, 0-9, (?), (.), (@), (_, and (\$) ; first character cannot be a digit, less than 31 characters;
 - “:” is needed if it is an instruction otherwise omitted;
 - “;” leads to a comment, the assembler omits anything on this line following a semicolon.
 - Segment Definition Types:** full & simplified segment definition.
 - Full segment definition** uses the following directives to define the segments. It is quite old-fashioned so we will use the next one.
`label SEGMENT [options]`
`label ENDS`
 You can name those labels and define as many as needed;

DOS automatically assigns *CS* and *SS* (using *assume CS: ..., ES: ...*), but *DS* (and *ES*) has to be manually specified in the program.

- Simplified segment definition** uses directives “*.STACK*”, “*.CODE*” and “*.DATA*” to define the segments, and these directives mark the beginning of the segments they represent. Only these three segments can be defined and automatically correspond to the CPU's *CS*, *DS* and *SS*. DOS determines the *CS* and *SS* segment registers automatically, but *DS* (and *ES*) has to be manually specified in the program (using *MOV* instructions).

- Model Definition:** use *.MODEL* directive to select the size of the memory model, here are some options.
 - TINY:** code + data < 64KB.
 - SMALL:** code ≤ 64KB, data ≤ 64KB;
 - MEDIUM:** code > 64KB, data ≤ 64KB;
 - COMPACT:** code ≤ 64KB, data > 64KB;
 - LARGE:** code > 64KB, data > 64KB, but single set of data < 64KB;
 - HUGE:** code > 64KB, data > 64KB;
- Procedure Definition:** procedures defined in code segments, and use the following directives:
`label PROC [FAR|NEAR]`
`label ENDP`
 The FAR and NEAR tells the assembler the range of the control transfer. The entrance procedure should be FAR.
- Program Execution:** program starts from the entrance, and program ends whenever calls 21H interruption with AH = 4CH. Assembler will follow the last line (usually *end procedure*) to get the entrance procedure. We call the procedure using “*CALL*” instruction to call the callee. In subroutine, we can use “*RET*” instruction to return to caller. (In ISR, the instructions are different, and will be discussed later.)
- Build up the program:** use editor to write the assembly the program; then use the assembler to assemble the source code and get object code (*.obj), then linker program will link the other files (maybe results of another code) into a single executable file.
- Control Transfer Instructions:** transfer the control of the program, such as “*CALL*” statement and jumps statement. There are three ranges:
 - SHORT:** intra-segment, IP changes in one-byte range (-128~127);
 - NEAR:** intra-segment, IP changes in two-byte range (-32768 to 32767); and the control should be transferred within the same code segment;
 - FAR:** inter-segment, CS and IP all changed; and the



control should be transferred outside the current code segment.

10. **Jumps:** include conditional jumps and unconditional jumps.

- a) **Conditional Jumps:** jump according to the value of the flag register, and all of them are short jumps. Often we need to perform some operations (such as subtraction) to set the flags before we use conditional jumps.

Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

- b) **Unconditional Jumps:** "*JMP [SHORT|NEAR|FAR PTR] label*". Unconditional jumps jump regardless of the flag register, and we will choose NEAR by default. The *label* specifies the location we want to jump to.

11. **CALL instructions:** "*CALL [NEAR|FAR PTR] procedure*", where *procedure* is the callee's label, and NEAR by default.

- a) When we execute "*CALL (NEAR PTR) procedure*" instruction, CPU will automatically put the address of the next instruction IP (IP points to the next instruction we will execute) into the stack and transfer the control to the subroutine. After subroutine finished, we can use "*RET*" instruction to return to the caller, and CPU will pop the value on the stack and assign the value to the IP register, and then we can continue executing.
- b) When we execute "*CALL FAR PTR procedure*" instruction, the situation is similar, except that we need to push both CS and IP into the stack. After subroutine finished and "*RET*" instruction is executed, we will restore both CS and IP from the stack.

12. **Data Types & Definition:** CPU can process either 8-bit or 16-bit operations, but we can define bigger types. Here are some directives.

- a) **ORG:** indicates the beginning of the offset address, which means the next memory address will start from this location.
[Example] "*ORG 10H*"
- b) **Variable definition.**

- i. **DB:** allocate byte-size chunks;

[Example]

"*x DB 12*": one byte;

"*y DB 23H, 48H*": two one-byte variables;

"*z DB 'Good Morning!'*" string;

"*str DB 'I'm good!'*" string.

In assembly level, no extra character after the string.

- ii. **DW:** allocate word-size chunks (in 8086, 2 bytes).
- iii. **DD:** allocate double-word-size chunks;
- iv. **DQ:** allocate quadruple-word-size chunks.

We can define names for variables. Variable names have three attributes attached to them: segment value, offset address and type (DW/DB/...) (that is, how a variable can be accessed). We can use the following instructions to get some attributes of a variable.

- v. **SEG** directive: get the segment value of a variable.
[Example] "*MOV AX, SEG x*".
- vi. **OFFSET** directive: get the offset value of a variable.
[Example] "*MOV AX, OFFSET x*".
- vii. **LEA** instruction: get the low-effective-address of a variable, that is, the offset value.
[Example] "*LEA AX, x*".

- c) **EQU:** define a constant.

[Example] "*NUM EQU 234*"

- d) **DUP:** duplicate a given number of characters.

[Example]

"*x DB 6 DUP(23H)*" will repeat the value for 6 times;

"*y DW 3 DUP(0FF10H)*" will repeat the value for 3 times.

13. **Labels:** there are two method to define a label, and a label has three attributes: segment value, offset address and type (NEAR/FAR).

- a) **Implicitly:** [Example] "*AGAIN: ADD AX, 03423H*".

- b) **Label** directive: [Example]

AGAIN LABEL FAR

ADD AX, 03423H

14. **Pointer.** Use *PTR* directive, which temporarily changes the type (range) attribute of a variable (or label), to guarantee that both operands in an instruction match, and to guarantee that the jump can reach a label.

[Example]

DATA1 DB 10H, 20H, 30H

DATA2 DW 4023H, 0A845H

MOV BX, WORD PTR DATA1; 2010H -> BX

MOV AL, BYTE PTR DATA2; 23H -> AL

MOV WORD PTR [BX], 10H; [BX], [BX+1] ← 0010H

JMP FAR PTR alabel

15. **.COM Executable:** one segment in total; put the data and code all together, and less than 64KB. We use "JUMP" instruction to skip the data.

```

TITLE  PROG2-4 COM PROGRAM TO ADD TWO WORDS
PAGE   60,132
CODSG  SEGMENT
        ORG 100H
        ASSUME CS:CODSG,DS:CODSG,ES:CODSG
;-----THIS IS THE CODE AREA
PROGCODE PROC NEAR
        MOV AX,DATA1      ;move the first word into AX
        MOV SUM,AX        ;move the sum
        MOV AH,4CH        ;return to DOS
        INT 21H
PROGCODE ENDP
;-----THIS IS THE DATA AREA
DATA1   DW 2390
DATA2   DW 3456
SUM      DW ?
CODSG   ENDS
END     PROGCODE

```

```

TITLE  PROG2-5 COM PROGRAM TO ADD TWO WORDS
PAGE   60,132
CODSG  SEGMENT
        ORG 100H
START:  JMP  PROGCODE      ;go around the data area
;-----THIS IS THE DATA AREA
DATA1   DW 2390
DATA2   DW 3456
SUM      DW ?
;-----THIS IS THE CODE AREA
PROGCODE PROC NEAR
        MOV AX,DATA1      ;move the first word into AX
        ADD SUM,AX        ;add the second word
        MOV AH,4CH        ;move the sum
        INT 21H
PROGCODE ENDP
CODSB   ENDS
START

```


Lecture 05. Assembly Language Programming (2)

1. **Arithmetic Instructions:** In assembly level, all the values are treated as unsigned, just the explanations are different.

- a) **Addition:** CPU will treat all the values as unsigned value. No mem-to-mem operations in 8086, that is, *dest* and *src* cannot be in memory together. Three types of instruction, “*ADD dest, src*”, “*ADC dest, src*” and “*INC dest*”. The first two change all the condition flags, and the last one changes all the condition flags except *CF*.

[Example]

ADD dest, src; $dest = dest + src$, affect all flags;

ADC dest, src; $dest = dest + src + CF$, affect all flags;

INC dest; $dest = dest + 1$, affect all flags except *CF*.

Notes: we use *ADC* in multi-byte adding.

```

TITLE      PROG3-2 (EXE)  MULTIWORD ADDITION
PAGE      60,132
.MODEL    SMALL
.STACK    64

DATA1     DD      DATA
          DQ      548FB9963CE7H
DATA2     DQ      3FCD4FA23B8DH
DATA3     DQ      ?

MAIN      PROC     FAR
          MOV     AX,@DATA
          MOV     DS,AX
          CLC
          MOV     SI,OFFSET DATA1      ;clear carry before first addition
          MOV     DI,OFFSET DATA2      ;SI is pointer for operand1
          MOV     BX,OFFSET DATA3      ;DI is pointer for operand2
          MOV     CX,04                 ;BX is pointer for the sum
          BACK:  MOV     AX,[SI]         ;CX is the loop counter
          ADC     AX,[DI]               ;move the first operand to AX
          MOV     SI,[BX]AX            ;add the second operand to AX
          INC     SI                   ;store the sum
          INC     DI                   ;point to next word of operand1
          INC     DI                   ;point to next word of operand2
          INC     BX                   ;point to next word of sum
          INC     BX                   ;point to next word of sum
          LOOP    BACK                 ;if not finished, continue adding
          MOV     AH,4CH                ;go back to DOS
          INT     21H
          MAIN      ENDP
          END

```

Loop instruction is equivalent to the following instruction.

“*DEC CX*” then “*JNZ*”.

- b) **Subtraction:** CPU will treat all the values as unsigned value. No mem-to-mem operations in 8086, that is, *dest* and *src* cannot be in memory together. Three types of instruction, “*SUB dest, src*”, “*SBB dest, src*” and “*DEC dest*”. The first two change all the condition flags, and the last one changes all the condition flags except *CF*. (method: treat subtraction as addition, then invert carry flag.)

[Example]

SUB dest, src; $dest = dest - src$, affect all flags;

SBB dest, src; $dest = dest - src - CF$, affect all flags;

DEC dest; $dest = dest - 1$, affect all flags except *CF*.

Notes: we use *SBB* in multi-byte subtracting.

```

DATA_A    DD      62562FAH
DATA_B    DD      412963BH
RESULT    DD      ?
...
MOV     AX,WORD PTR DATA_A
SUB     AX,WORD PTR DATA_B
MOV     WORD PTR RESULT,AX
MOV     AX,WORD PTR DATA_A+2
SBB     AX,WORD PTR DATA_B+2
MOV     WORD PTR RESULT+2,AX

```

- c) **Unsigned Multiplication:** using “*MUL operand*” instruction. It will change *OF, CF*; the rest of flags are undetermined. There are several types of multiplications.
- i. **Byte * Byte:** one implicit operand is *AL*, and the other is *operand*; the result is stored in *AX*;

- ii. **word * word:** one implicit operand is *AX*, the other is *operand*; the result is stored in *DX* and *AX*;
- iii. **word * byte:** *AL* hold the implicit byte operand and *AH = 0*, the word is the operand; the result is stored in *DX* and *AX*.
- d) **Unsigned Division:** using “*DIV denominator*” instruction. All the condition flags are undetermined; *denominator* cannot be zero, and quotient cannot be too large for the assigned register. There are several types of divisions.
- i. **byte / byte:** numerator in *AL*, clear *AH*; quotient in *AL*, remainder in *DX*;
- ii. **word / word:** numerator in *AX*, clear *DX*; quotient in *AX*, remainder in *DX*;
- iii. **word / byte:** numerator in *AX*; quotient in *AL* (maximum 0FFH), remainder in *AH*;
- iv. **double-word / word:** numerator in *DX, AX*; quotient is in *AX* (maximum 0FFFFH), remainder in *DX*.

[Example] Unsigned Division Example

MOV AL,DATA7	MOV AX,10050
SUB AH,AH	SUB DX,DX
DIV 10	DIV BX,100
	MOV BX,QOUT2,AX
	MOV REMAIND2,DX

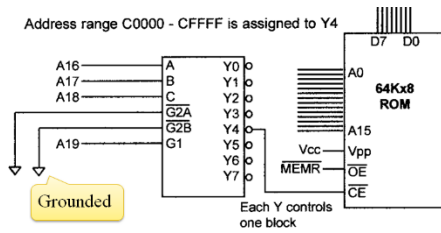
MOV AX,2055	DATA1 DD 105432
MOV CL,100	DATA2 DW 10000
DIV CL	QUOT DW ?
MOV QUO,AL	REMAIN DW ?
MOV REMI,AH	

MOV AX,WORD PTR DATA1	
MOV DX,WORD PTR DATA1+2	
DIV DATA2	
MOV QUOT,AX	
MOV REMAIN,DX	

2. **Logic Instructions:** no mem-to-mem instruction.

- a) **AND:** “*AND dest, src*” will perform $dest = dest \& src$, where $\&$ is logical and; *dest* can be a register or in memory; *src* can be a register, in memory, or immediate. The operation will update *SF, ZF* and *PF*, and will clear *CF* and *OF* (set to zero), but *AF* will be undetermined;
- b) **OR:** “*OR dest, src*” will perform $dest = dest | src$, where $|$ is logical or; *dest* can be a register or in memory; *src* can be a register, in memory, or immediate. The operation will update *SF, ZF* and *PF*, and will clear *CF* and *OF* (set to zero), but *AF* will be undetermined;
- c) **XOR:** “*XOR dest, src*” will perform $dest = dest \wedge src$, where \wedge is logical xor; *dest* can be a register or in memory; *src* can be a register, in memory, or immediate. The operation will update *SF, ZF* and *PF*, and will clear *CF* and *OF* (set to zero), but *AF* will be undetermined;
- d) **NOT:** “*NOT operand*” will perform $operand = !operand$, where $!$ is logical not; *operand* can be a register or in memory. The operation does not change the flag registers.
- e) **Logical SHIFT:** “*SHR dest, times*” and “*SHL dest, times*” will right-shift/left-shift the *dest times* times; *dest* can be a register or in memory; usually *times* should be a 8-bit

[Example]



Notes: Both $\overline{G2A}$ and $\overline{G2B}$ are low-effective, and $G1$ needs to have high voltage; use the number $(CBA)_2$ in decimal to get the corresponding output (low-output).

Notes: for all port, regard as high-effective by default; if the port name has overline or there is a circle on the port, then the port will be low-effective; so $\overline{G2A}$ and $\overline{G2B}$ here are low-effective.

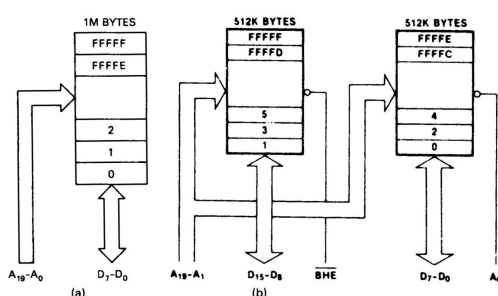
3. Address Decoding Types

- Absolute address decoding:** all address lines are decoded;
- Linear select decoding:** only selected lines are decoded; this method is cheap but with aliases, that is, the same memory unit (I/O port) can have multiple addresses.

4. Data Integrity

- Checksum byte for ROM:** check the integrity of a series of bytes; add all bytes together and drop all carries, then take the two's complement of the sum and we get the checksum byte. Store the checksum byte together with the data, and check the integrity by adding data and checksum together. If the answer is 0 regardless of carries, then the data may be correct; otherwise the data must be incorrect.
- Parity bit for DRAM:** check the integrity of a series of bits (a byte); parity bit has two types:
 - even parity:** if the number of 1s in the series of bits is odd, then the parity bit is set to 1; otherwise, set to 0, making the total number of 1s even (data and parity bit);
 - odd parity:** if the number of 1s in the series of bits is even, then the parity bit is set to 1; otherwise set to 0, making the total number of 1s odd (data and parity bit). 8086 uses odd parity.
- CRC for disks and the Internet.**

- Memory Organization in 8086:** even and odd banks. Address bits A_1 through A_{19} select the storage location that is to be accessed. They are applied to both banks in parallel. A_0 and bank high enable \overline{BHE} are used as **bank-select** signals.

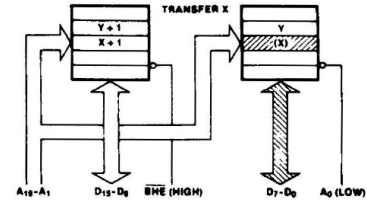


Here are some explanations about A_0 and \overline{BHE} .

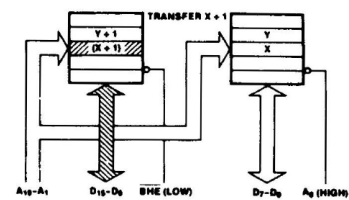
\overline{BHE}	A_0		
0	0	Even word	$D_0 - D_{15}$
0	1	Odd byte	$D_8 - D_{15}$
1	0	Even byte	$D_0 - D_7$
1	1	None	

For memory operations,

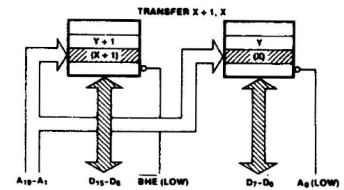
- Byte-memory operation at even address X:** only use the even bank (controlled by A_0 , low-effective);



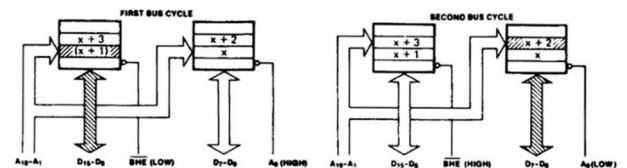
- Byte-memory operation at odd address X+1:** only use the odd bank (controlled by \overline{BHE} , low-effective);



- Aligned word-memory operations** (at even address X): use both banks and completes the transfer in one cycle.



- Misaligned word-memory operation** (at odd address X+1): take two cycles; in each cycle, use one bank. This is because the addresses in two cycles are different, and we need two cycles to access both data.



6. I/O in x86 family

a) Introduction:

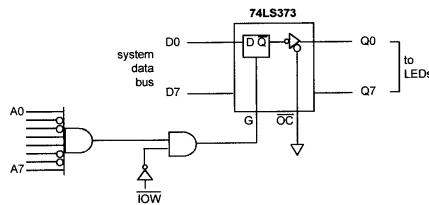
- x86 microprocessors have an I/O space in addition to memory space, which is also called **peripheral I/O** or **isolated I/O**.
- Use special I/O instructions accessing I/O devices at ports (i.e., addresses for I/O).
- Memory can contain machine codes and data, I/O ports only contain data.

- I/O instructions:** use instruction "*IN dest, port#*" and "*OUT port#, src*" to represent input from *port#* to *dest* and output from *src* to *port#*. I/O instructions have 8-bit

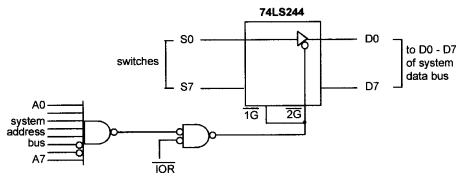
operation and 16-bit operation, and you can choose the *dest* register or the *src* register from AX and AL to choose the mode of different bits. (e.g., AX 16-bit; AL 8-bit.)

- i. **Direct I/O instructions:** immediate number *port#* ranges from 00H to FFH, 256 ports in total;
- ii. **Indirect I/O instructions:** *port#* is stored in special register *DX*, ranging from 0000H to FFFFH, 65536 ports in total.
- iii. **Notes:** no segment concept for port addresses.
- c) **Interfacing 8-bit I/O modules to a 16-bit data bus:** for 8086, data for even-address ports are carried on data bus D_0 to D_7 ; while data for odd-address ports are carried on data bus D_8 to D_{15} .

- i. **Output Port Design:** need a latch to latch the data sent from CPU; (this figure is an 8088 figure, so we do not need to consider odd/even bank)



- ii. **Input Port Design:** need a tri-state buffer to connect to system data bus and the status of switches. (this figure is an 8088 figure, so we do not need to consider odd/even bank)



- iii. **Notes:** In 8086, the data bus in above two figures should be D_8 to D_{15} .

as input or output;

- iii. **Port C (PC_0 to PC_7):** can be split into two separate parts PCU and PCL; any bit can be programmed individually.
- b) **Control Register (CR):** internal register used to setup chip.
- c) **Groups.**
 - i. **Group A** consists of PA and PCU;
 - ii. **Group B** consists of PB and PCL.
- d) **Data bus buffer:** an interface between CPU and 8255; bidirectional, tri-state and 8-bit.
- e) **Read/Write control logic**
 - i. **RESET:** high-active, clear the control register, all ports are set as input port;
 - ii. $\overline{CS}, \overline{RD}, \overline{WR}$ pins, representing chip selection, read and write respectively;
 - iii. A_1, A_0 pins: port selection signals.

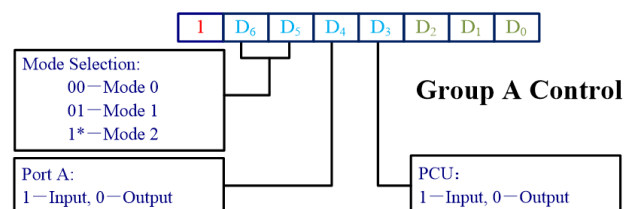
$\sim CS$	A_1	A_0	$\sim RD$	$\sim WR$	Function
0	0	0	0	1	PA \rightarrow Data bus
0	0	1	0	1	PB \rightarrow Data bus
0	1	0	0	1	PC \rightarrow Data bus
0	0	0	1	0	Data bus \rightarrow PA
0	0	1	1	0	Data bus \rightarrow PB
0	1	0	1	0	Data bus \rightarrow PC
0	1	1	1	0	Data bus \rightarrow CR
1	x	x	1	1	$D_0 \sim D_7$ in float

2. Operation modes of 8255.

a) Input / Output (IO) modes

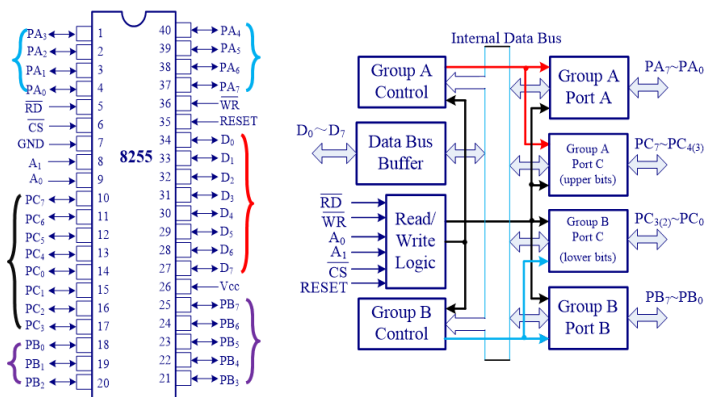
- i. **Mode 0, simple I/O mode:** PC is divided into PCU ($PC_4 \sim PC_7$) and PCL ($PC_0 \sim PC_3$); no handshaking, which is a negotiation between two entities before communication; each port (PA, PB, PCU, PCL) can be programmed as input/output port;
- ii. **Mode 1:** PA and PB can be used as input/output ports with handshaking; and PC is divided into PCU ($PC_3 \sim PC_7$) and PCL ($PC_0 \sim PC_2$), which are used as handshaking lines for PA and PB respectively;
- iii. **Mode 2:** Only PA can be used for bidirectional handshake data transfer; and PCU ($PC_3 \sim PC_7$) are used as handshake lines for PA.

- b) **Bit set/reset (BSR) mode:** only PC can be used as output port and each line of PC can be set/reset individually.
- c) **Control Register:** a 8-bit internal register in 8255; selected when $A_1 = 1$ and $A_0 = 1$. Mode selection word:
 - i. **Select IO modes:**



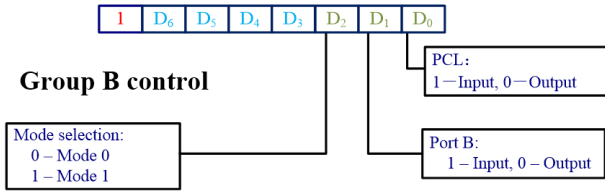
Lecture 07. 8255 PPI Chip

1. 8255 structure and pins

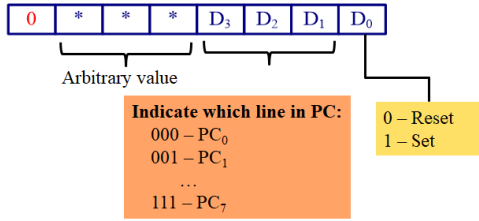


a) Data Ports: A, B and C.

- i. **Port A (PA, PA_0 to PA_7):** can be programmed all as input or output;
- ii. **Port B (PB, PB_0 to PB_7):** can be programmed all

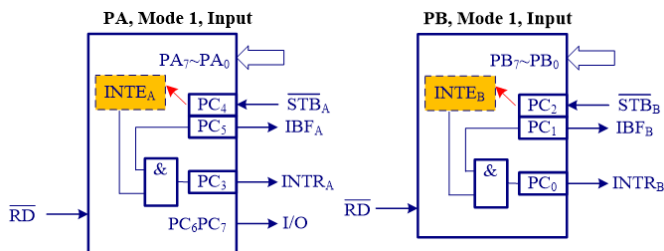


ii. **Select BSR mode:**



3. **Modes Details.**

- Mode 0 (Simple I/O):** for simple input / output scenario. CPU directly read from or write to a port using *IN* or *OUT* instructions; input data are not latched but output data are latched;
- Mode 1 (Strobe I/O):** for handshake input/output scenario. Both input and output data are latched;
- Mode 1 as input ports.** Here are pins introductions.
 - **\overline{STB}** : the strobe input signal from input device loads data into the port latch; (PC_4 in PA and PC_2 in PB);
 - **IBF** : Input Buffer Full output signal to the device indicates that the input latch contains information (for programmed I/O); (PC_5 in PA and PC_1 in PB);
 - **$INTR$** : interrupt request is an output to CPU that requests an interrupt (for interrupted driven I/O); (PC_3 in PA and PC_0 in PB).
 - **PC_6 and PC_7** : can be used as separate I/O lines for any purpose (set via the control bit of PCU);
 - **$INTE$** : the interrupt enable signal is neither an input nor an output; it is an internal bit programmed via PC_4 (port A) or PC_2 (PORT b); 1-allowed, 0-forbidden.

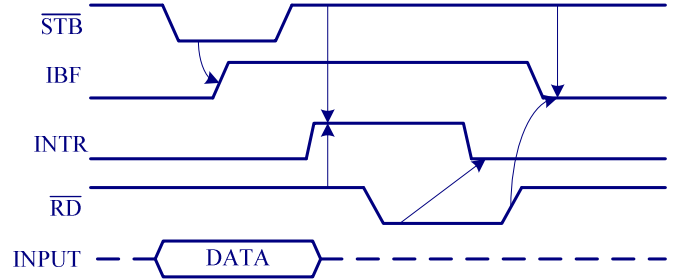


d) **Timing in Mode 1 input:**

- Input device first puts data on $PA_0 \sim PA_7$, then activates \overline{STB}_A , data is latched in Port A;
- 8255 activates IBF_A which indicates the device that the input latch contains information but CPU has not taken it yet, so device cannot send new data until IBF_A is cleared;
- When IBF_A , \overline{STB}_A and $INTE_A$ are all high (notice that \overline{STB}_A is low-effective, and \overline{STB}_A is

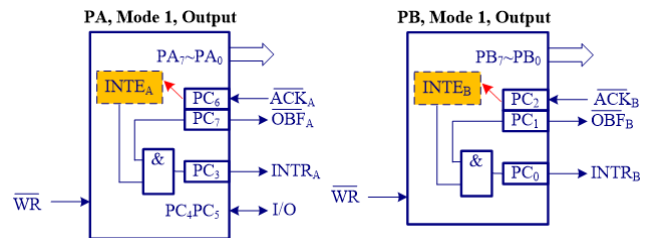
high means \overline{STB}_A is not activated), 8255 activates $INTR_A$ to inform CPU to take data in PA by interruption;

- CPU responds to the interruption and read in data from PA; the \overline{RD} signal will clear $INTR_A$ signal;
- After CPU finishes reading data from PA (i.e., \overline{RD} signal goes high), the IBF_A signal is cleared.



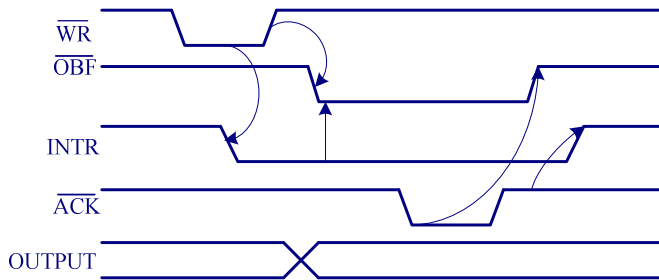
e) **Mode 1 as output ports.** Here are pins introductions.

- **\overline{OBF}** : Output Buffer Full output signal that indicates data has been latched in the port; (PC_7 in PA and PC_1 in PB);
- **\overline{ACK}** : the acknowledge input signal indicates that the external device has taken the data; (PC_6 in PA and PC_2 in PB);
- **$INTR$** : interrupt request is an output to CPU that requests an interrupt; (PC_3 in PA and PC_0 in PB);
- **PC_4 and PC_5** : can be used as separate I/O lines for any purpose (set via the control bit of PCU);
- **$INTE$** : the interrupt enable signal is neither an input nor an output; it is an internal bit programmed via PC_6 (port A) or PC_2 (PORT b); 1-allowed, 0-forbidden.



f) **Timing in Mode 1 Output:**

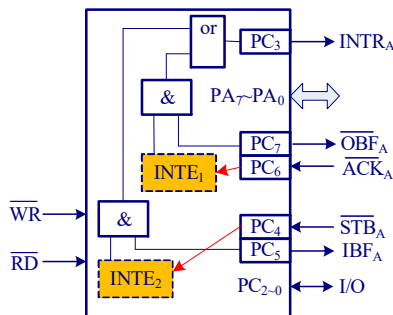
- If $INTR_A$ active, CPU responds to the interruption and writes data to PA and clears the $INTR_A$ signal;
- When data has been latched in PA, 8255 activates \overline{OBF}_A which informs the output device to pick up data;
- After the output device has taken the data, it sends \overline{ACK}_A signal to 8255 which indicates that the device has received the data, and also makes \overline{OBF}_A go high, indicating CPU can write new data to 8255;
- When \overline{OBF}_A , \overline{ACK}_A and $INTE_A$ are all high (notice that \overline{STB}_A and \overline{ACK}_A are low-effective, and \overline{STB}_A and \overline{ACK}_A are high means \overline{STB}_A and \overline{ACK}_A are not activated), 8255 sends an $INTR_A$ to inform CPU to write new data to PA by interruption.



- g) **Mode 2 (Bidirectional Bus):** for bidirectional handshake input/output scenario; both input and output are latched.
- h) **Mode 2 as input & output port:** PC_3 to PC_7 are used as handshake lines for PA.

• $\overline{OBF}_A, \overline{IBF}_A, \overline{ACK}_A, \overline{STB}_A, INTR_A$ has the same meanings as the previous mode.

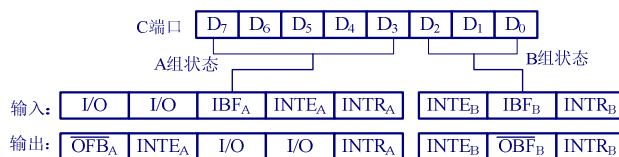
• $PC_0 \sim PC_2$ can be used as separate I/O lines for any purpose, or as handshake lines for PB (set via the control bit of PCL).



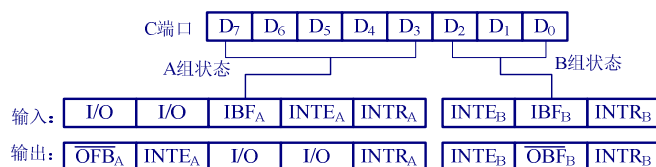
Notes. When CPU responds to an interrupt of 8255 working in Mode 2, the interrupt handler has to check the \overline{OBF}_A and \overline{IBF}_A in order to tell whether the input process or the output process is generating the interrupt.

- i) **Comparison of PC in different modes.**

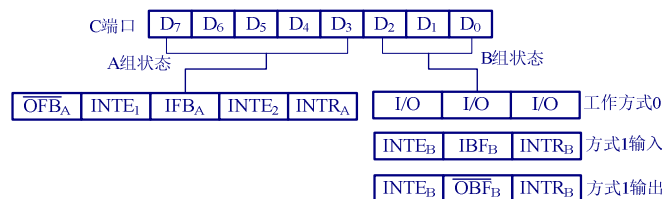
Mode 1 as input port.



Mode 1 as output port.



Mode 2 as input/output port.

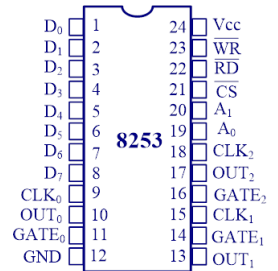


Lecture 08. 8253/4 Timer

1. Timer.

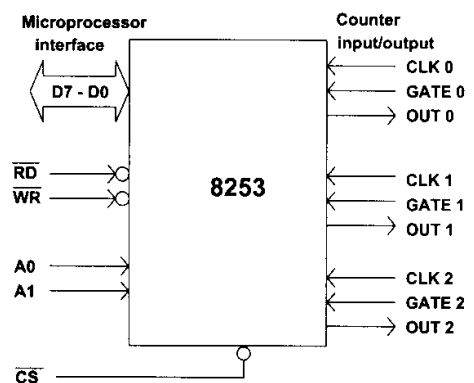
- a) **Software solution:** setting up a timing loop;
pros: simple, does not need extra support
cons: CPU doing useless things - counting, occupy more resources; and cannot control the delay accurately.
- b) **Hardware solution:** using 8253 to count out the delay and interrupt the CPU.
pros: CPU do not involve in counting, and can control the delay accurately;
cons: need extra support and a little complex.

2. **8253 PIT structure & pins:** the 8253/54 programmable interval timer is used to generate a lower frequency for various uses (e.g., event counter, accurate time delays).

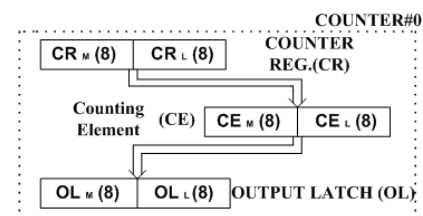


- a) **Counters:** three independent pre-settable counters in 8253;

the input frequency can be divided from 1 to 65536 (binary), or from 1 to 10000 (BCD) (can be operated either in binary or BCD); and each of them can generate square-wave, or one-shot, or square-wave with various duty cycles; gate is used to enable (high) or disable (low) the counter; input, gate and output are configured by selection of modes; reading from a counter does not disturb the actual count in process.



- b) **Counter Internal Structure.**



Method to operate a 16-bit down counter:

- a 16-bit count is loaded in the counter;
 - begins to decrement the count until it reaches 0;
 - generates a pulse that can be used to interrupt CPU.
- c) **Data bus buffer:** interface the 8253/4 to the system data bus; bi-directional, tri-state, 8-bit;

d) **Control Logic:**

- \overline{CS} pins: tied to a decoded address;
- $\overline{RD}, \overline{WR}$ pins: In isolated I/O, connect with \overline{IOR} and \overline{IOW} ; in memory-mapped I/O, connect with \overline{MEMR} and \overline{MEMW} ;
- A_1, A_0 pins: counter select signal.

$\sim CS$	$\sim RD$	$\sim WR$	A1A0	FUNCTION
0	1	0	00	Write counter0 (to CR0)
0	1	0	01	Write counter1 (to CR1)
0	1	0	10	Write counter2 (to CR2)
0	1	0	11	Write control port
0	0	1	00	Read counter0 (from OL0)
0	0	1	01	Read counter1 (from OL1)
0	0	1	10	Read counter2 (from OL2)
0	0	1	11	Read control port (for 8254)
1	X	X	XX	Not available

- e) **Control word register:** selected when $A_1A_0 = 11$, used to specify which counter to be used, its mode, and a read or write operation.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
SC ₁	SC ₀	RW ₁	RW ₀	M ₂	M ₁	M ₀	BCD

SC₁ SC₀ SC - Select counter

0	0	Select counter 0
0	1	Select counter 1
1	0	Select counter 2
1	1	Illegal for 8253 Read -Back command for 8254 (See Read operations)

RW₁ RW₀ RW - Read /Write

0	0	Counter latch command (See Read operations)
0	1	Read / Write least significant byte only
1	0	Read / Write most significant byte only
1	1	Read / write least significant byte first, then most significant byte

M₂ M₁ M₀ M - Mode

0	0	0	Mode 0
0	0	1	Mode 1
x	1	0	Mode 2
x	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD :

0	Binary counter 16 - bits
1	Binary coded decimal (BCD) Counter (4 Decades)

3. **Features**

- Three independent 16-bit down counter;
- 8254 can handle inputs from DC to 10MHz (5MHz 8254-5; 8MHz 8254; 10MHz 8254-2) whereas 8253 can operate up to 2.6MHz;
- Three counters are identical and pre-settable, and can be programmed for either binary or BCD count;
- Counter can be programmed in six different modes;
- Compatible with all Intel and most other processors;
- 8254 has powerful command called *READ BACK* command which allows the user to check the count value, programmed mode and current mode and current status of the counter.

- 8253 takes one CLK pulse to convey the count from CR (counter register) to CE (counting element);
- CE (counting element) will start to count only when GATE is high. On every CLK pulse's rising (0-to-1) edge we will check the GATE; and on every CLK pulse's falling (1-to-0) edge we will count down.

4. **Read/Write Operations.**

a) **Read.**

- Simple Read:** two I/O read operations, first one for low-order byte and last one for high-order byte;
- Counter Latch Command:** one I/O write operation used to write a control word to the control register to latch a count in the output latch, then two I/O read operations are used to read the latched count as in Simple Read.
- Read-Back Command:** for 8254 only.

- Write:** write a control word into control register; then load the low-order byte of a count in the counter register; then load the high-order byte of a count in the count register.

5. **Modes Details**

- Mode 0 Interrupt on Terminal Count:** the output will be initially low after the mode set operation; After the count is loaded into the selection CR, the output will remain low; When the terminal count is reached, the output will go high and remain high until the selected counter is reloaded.

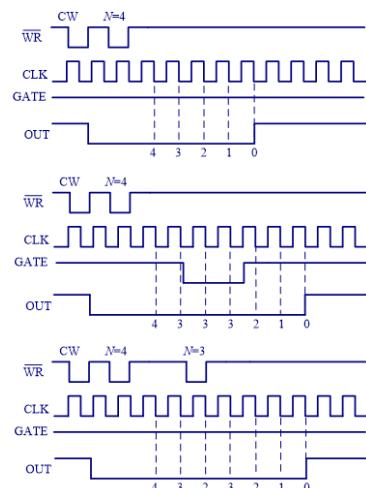
Output: N clock pulses low and one clock pulse high afterwards after writing a count.

Gate disable: Gate=1 enables counting; Gate=0 disables.

New count: If a new count is written to the counter, it will be loaded on the next CLK pulse and counting will continue from the new count. In case of two-byte count, writing the first byte disables the current counting, then writing the second byte loads the new count on the next CLK pulse and counting will continue from the new count.

Repeat: does not automatically repeat.

New load effect: when loading a new count N, the actual number of CLK pulse in OUT is $N+1$.



- b) **Mode 1 Hardware Re-triggerable One-shot:** the output will be initially high after the mode set operation; the output will go low on the CLK pulse following the rising (0-to-1) edge of the gate input; the output will go high on the terminal count and remain high until the next rising edge of the gate input.

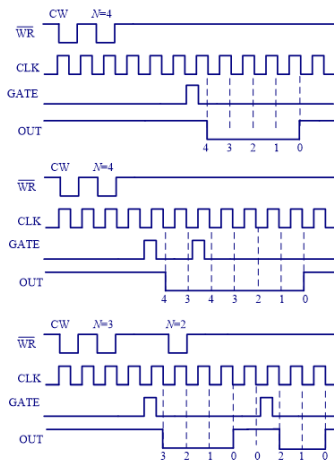
Output: one-shot of N clock pulses on every trigger.

Retriggering: re-triggerable, hence the output will remain low for the full count after any rising edge of the gate input.

New count: if the counter is loaded during one shot pulse, the current one shot is not affected unless the counter is retriggered; if retriggered, the counter is loaded with the new count and the one-shot pulse continues until the new count expires.

Repeat: does not automatically repeat.

New load effect: when loading a new count N , the current counting will not be affected.



- c) **Mode 2 Rate Generator:** the output will be initially high after the mode set operation; the output will go low for one clock pulse before the terminal count; then the output will go high, the counter reloads the initial count and the process is repeated.

Output: periodical signal with a period of $(N-1)$ clock pulses high and 1 clock pulse low.

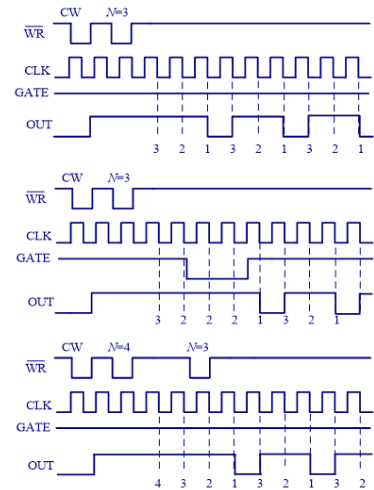
Gate disable: If $\text{Gate}=1$ it enables a counting otherwise it disables counting; if gate goes low during a low output pulse, output is set immediately high.

New count: the current counting sequence is not affected when the new count is written; if a trigger (a rising edge of GATE) is received after writing a new count but before the end of the current period, the counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the counter will be loaded with the new count at the end of the current counting cycle.

Notes: In mode 2, a count of 1 is illegal.

Repeat: Automatically repeat on terminal count.

New load effect: when loading a new count N , the current counting will not be affected.



- d) **Mode 3 Square Wave Rate Generator:** the output will be initially high. For even count, the counter is decremented by 2 on the falling edge of each clock pulse; when reaches terminal count, the state of the output is changed and the counter is reloaded with the full count and the whole process is repeated. For odd count, the first clock pulse decrements the count by 1, and subsequent clock pulses decrement the clock by 2; after timeout, the output goes low and the full count is reloaded; the first clock pulse (following the reload) decrements the count by 3 and subsequent clock pulse decrement the count by two. Then the whole process is repeated.

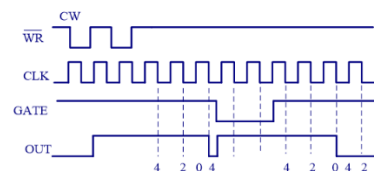
Output: the output will be high for $\lfloor N/2 \rfloor$ cycles and low for $\lfloor N/2 \rfloor$ cycles.

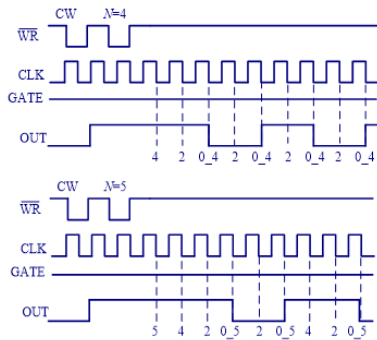
Gate disable: If $\text{Gate}=1$ then counting is enabled otherwise it is disabled; if gate goes low while output is low, output is set high immediately. After this, when gate goes high, the counter is loaded with the initial count on the next block pulse and the sequence is repeated.

New count: The current counting sequence does not affect when the new count is written. If a trigger is received after writing a new count but before the end of the current half-cycle of the square wave, the counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at end of the current half-cycle.

Repeat: Automatically repeat on terminal count.

New load effect: when loading a new count N , the current half cycle will not be affected.





- e) **Mode 4 Software Triggered Strobe:** the output will be initially high; the output will go low for one CLK pulse after the terminal count.

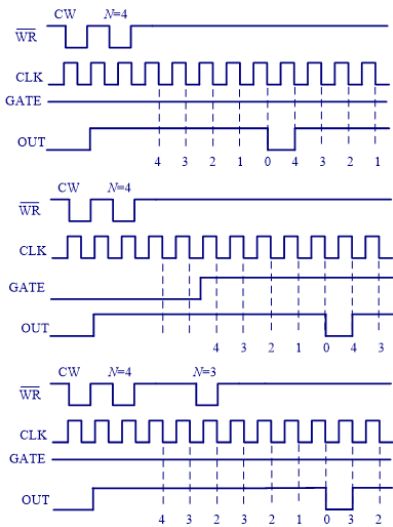
Gate disable: Gate=1 enables counting; Gate=0 disables.

New count: If a new count is written during counting, it will be loaded on the next CLK pulse and counting will continue from the new count. If the count is two bytes then: writing the first byte has no effect on counting, and writing the second byte allows the new count to be loaded on the next CLK pulse.

Output: N clock pulses high and 1 clock pulse low afterwards after writing a count.

Repeat: Automatically repeat on terminal count.

New load effect: when loading a new count N , the actual number of CLK pulse in OUT is $N+1$.



- f) **Mode 5 Hardware Triggered Strobe:** the output will be initially high, and the counting is triggered by the rising edge of the gate; the output will go low for one CLK pulse after the terminal count.

Retriggering: If the triggering occurs during the counting, the initial count is loaded on the next CLK pulse and the counting will be continued until the terminal count is reached.

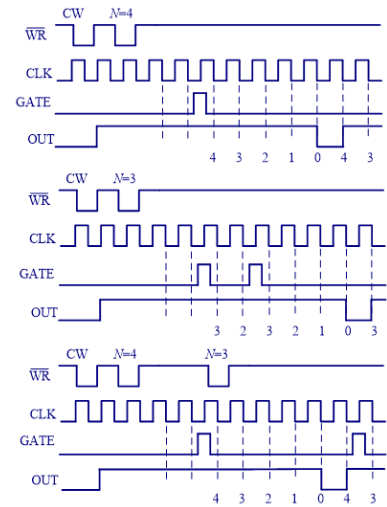
Output: N clock pulses high and 1 clock pulse low afterwards after (re-)triggering.

New count: the current counting sequence will not be affected. If the trigger occurs after the new count but

before the terminal count, the counter will be loaded with the new count on the next CLK pulse and counting will continue from there.

Repeat: Automatically repeat on terminal count.

New load affect: When loading a new count N , the current counting will not be affected.



6. Modes Summary:

- a) When does the count start?

Mode 0: immediate after we write initial value to CR.

Mode 1: triggering: the rising edge of GATE.

Mode 2: immediate after we write initial value to CR.

Mode 3: immediate after we write initial value to CR.

Mode 4: immediate after we write initial value to CR.

Mode 5: triggering: the rising edge of Gate

- b) Does the count automatically repeat?

Mode 0: No.

Mode 1: No.

Mode 2: Yes.

Mode 3: Yes.

Mode 4: Yes.

Mode 5: Yes.

- c) What's impact of GATE?

Mode 0: enable/disable count.

Mode 1: trigger.

Mode 2: enable/disable count.

Mode 3: enable/disable count. also trigger.

Mode 4: enable/disable count.

Mode 5: trigger.

- d) What would happen if we write new initial value?

Mode 0: start new count on next CLK pulse.

Mode 1: start new count after current count ends / trigger.

Mode 2: start new count after current count ends.

Mode 3: start new count after half-cycle, sometimes on next CLK pulse (depends on the 0-1 trigger of GATE).

Mode 4: start new count on next CLK pulse.

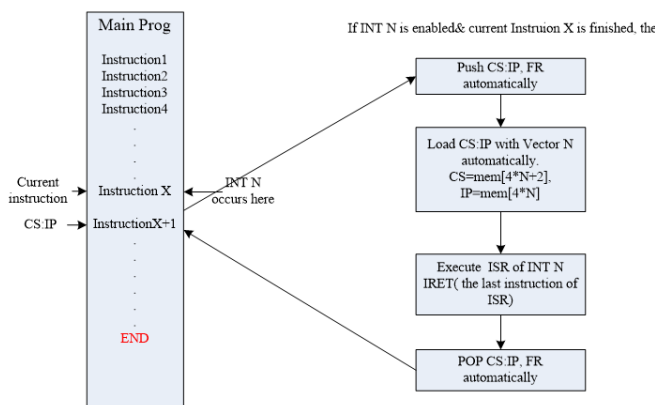
Mode 5: start new count after current count ends / trigger.

Lecture 09. Interrupt and the 8259 Chip

- Interrupts in 8086/88:** 256 interrupt types in total, from INT 00H to INT 0FFH; the physical address of the **interrupt vector** is interrupt type multiplied by 4. Therefore, the first 1KB in memory is used to store interrupt vectors, called **interrupt vector table (IVT)**. Interrupt vector points the entrance address of the corresponding **interrupt service routine (ISR)**.

0003FC	CS	} INT FF
	IP	
00018	CS	} INT 06
	IP	
00014	CS	} INT 05
	IP	
00010	CS	} INT 04 signed number overflow
	IP	
0000C	CS	} INT 03 breakpoint
	IP	
00008	CS	} INT 02 NMI
	IP	
00004	CS	} INT 01 single-step
	IP	
00000	CS	} INT 00 divide error
	IP	

- Interrupt handling:** an ISR is launched by an interrupt event.



- Categories of interrupts:**

- Hardware (external) interrupt:** maskable interrupt (from *INTR*) and non-maskable interrupt (from *NMI*);
- Software (internal) interrupt:** using *INT* instruction, predefined conditional (exception) interrupts.

- Non-maskable interrupts:**

- Trigger:** *NMI* pin, input signal, rising edge and two-cycle high active;
- Type:** *INT 02H*;
- Not affected by *IF* (interrupt flag):**
- Reasons:** e.g., RAM parity check error, interrupt request from co-CPU 8087, etc.
- Processing procedure:** CPU checks *NMI*, generates *INT 02H* interrupt automatically regardless of *IF* and executes the corresponding ISR.

- Maskable interrupts:**

- Trigger:** *INTR* pin, input-signal, high-active;
- Type:** no pre-defined type;
- IF* = 1, enable; *IF* = 0, disable; and we can use *STI* to set *IF*, use *CLI* to clear *IF*;**

- Reasons:** input request of external I/O devices.

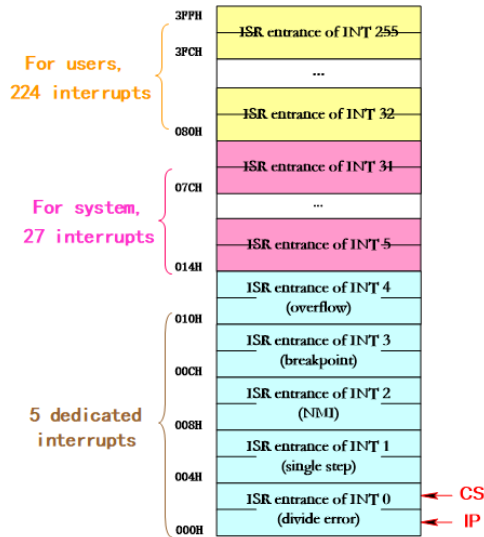
- Processing procedure:**

- CPU responds to *INTR* input requests:** external I/O devices send interrupt request through *INTR* pin to CPU; CPU will check *INTR* pin on the last cycle of an instruction: if *INTR* is high and *IF* = 1, CPU responds to the interrupt request; CPU then sends two negative pulses on the *INTA* pin to the I/O device. After receiving the second *INTA*, I/O device sends the interrupt type *N* on data bus.
- CPU executes the ISR of *INT N*:** CPU reads *N* from data bus, and push *FR* (flag register) in stack; then CPU clear *IF* and *TF*, and push CS and IP of the next instruction in stack. CPU load the ISR entrance address according to interrupt vector table and *N* and then moves to ISR. At the end of ISR, *IRET* instruction will pop IP, CS and FR in turn, and CPU returns to previous program and proceeds.

- Software interrupt:**

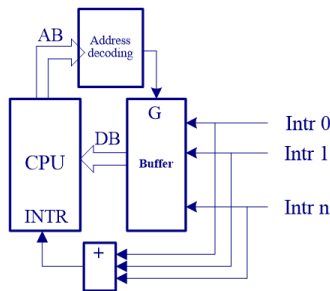
- Instruction:** *INT x*, and the corresponding ISR is called to handle the interrupt; by using *INT* instruction, you can "CALL" any ISR.
- CPU always responds and goes executing the corresponding ISR; not affected by *IF*;
- Difference between *INT* and *CALL*:**
 - CALL FAR* can jump anywhere within 1MB, but *INT* only jumps to a fixed location (corresponding ISR).
 - CALL FAR* is in sequence of instructions, but an external interrupt can come in at any time;
 - CALL FAR* cannot be masked, but an external interrupt can be masked;
 - CALL FAR* saves CS:IP of next instruction, but *INT* saves FR + CS:IP of the next instruction.
 - CALL FAR* uses *RETf* to return from the subroutine, but *INT* uses *IRET* to return from the ISR.
- Predefined conditional interrupts:**
 - INT 00*:** (divide error) reason: dividing a number by zero, or quotient is too large;
 - INT 01*:** (single step) if *TF* = 1, CPU will generate an *INT 1* interrupt after executing each instruction for debugging;
 - INT 03*:** (breakpoint) When CPU hits the breakpoint set in the program, CPU generates *INT 3* interrupt for debugging.
 - INT 04*:** (signed number overflow) using *INTO* instruction and check the *OF* after arithmetic instructions.
- Processing procedure:** CPU generates *INT N* interrupt automatically and executes to the corresponding ISR.

7. Interrupt Vector Table of 8086/88

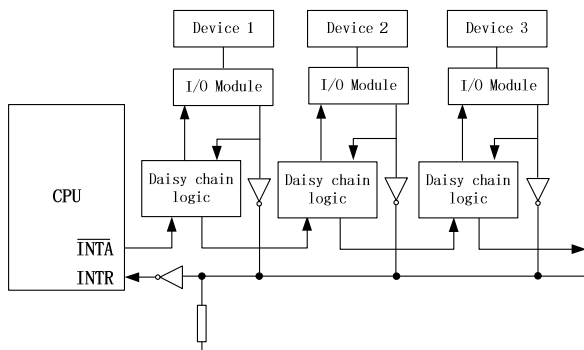


8. **Interrupt priority:** *INT* instruction has higher priority than *INTR* and *NMI*; *NMI* has higher priority than *INTR*; for different external interrupt requests, different strategies can be used to determine their priorities.

a) **Software polling:** the sequence of checking determines the priority.



b) **Hardware checking:** location in the daisy chain counts.

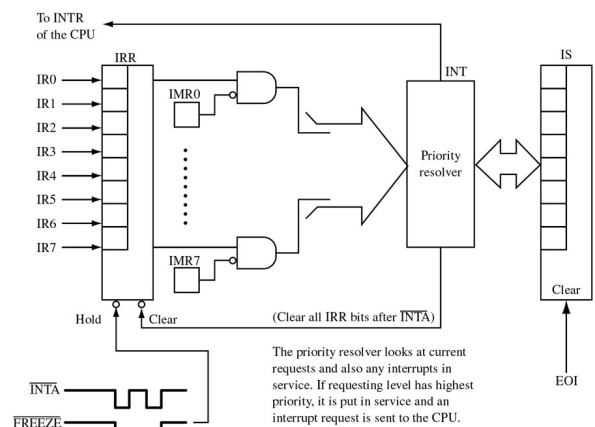
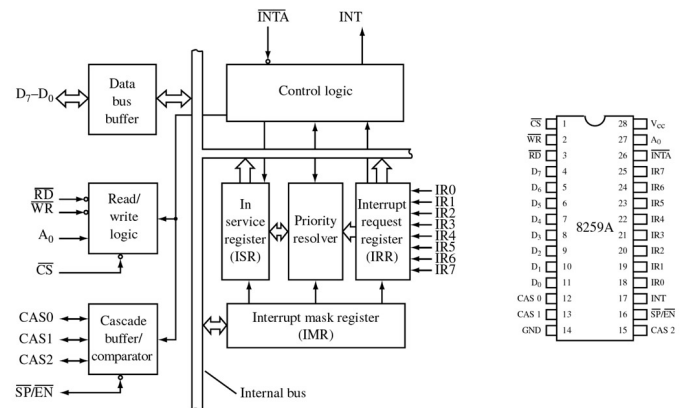


c) **Vectored interrupt controller:** for example, 8259.

9. **8259 Programmable Interrupt Controller (PIC):** a tool for managing the interrupt requests; and it is very flexible peripheral controller chip. because

- PIC can deal with up to 64 interrupt inputs;
- interrupts can be masked;
- various priority schemes can also be programmed.

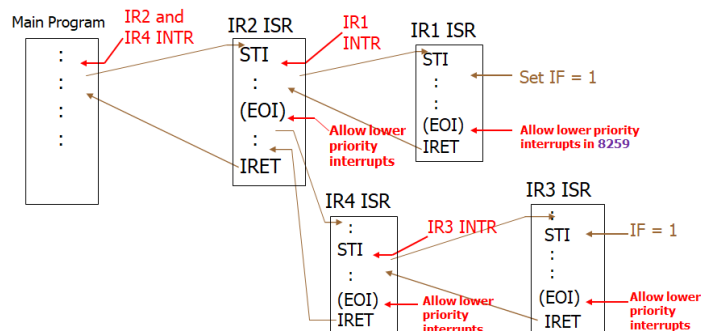
Originally (in PC XT) it is available as a separate IC (interrupt controller). Later the functionality of two PICs is in the motherboard's chipset. In some of the modern processor, the functionality of the PIC is built in.



Explanations.

- **IRR** (Interrupt Request Register): store the Interrupt Request.
- **IMR** (Interrupt Mask Register): $IMR = 1$, 8259 masks this signal; $IMR = 0$, 8259 allows the signal.
- **Priority Resolver:** define different priorities to different interrupts to handle the interrupt.
- **ISR** (Interrupt Service / In-Service Register): the result of priority resolver, the final interrupt controller result.
- ISR will respond to CPU, and CPU will know the priorities.

10. **Interrupt Nesting:** higher priority interrupts can interrupt lower interrupts.



where, STI is used to re-open the interrupt flag which is closed by ISR (Interrupt Service Routine). And EOI will clear the flags in the ISR (Interrupt Service Register), which allows lower priority interrupts to come in.

Lecture 10. BIOS and MS-DOS Programming (not-required)

1. **BIOS and DOS interrupts:** you can use those useful subroutines within BIOS and DOS to implement your applications. You can “CALL” those subroutines by explicitly embedding BIOS and DOS interrupt instructions in your program

- INT 10H:** BIOS interrupt;
- INT 21H:** DOS interrupt.

2. **BIOS INT 10H Programming:** INT 10H subroutines are burned into ROM BIOS (in 80x86-based IBM PCs), used to communicate with the computer’s screen video. By setting AH with different values, you can “call” these functions.

- Scrolling window:** up – 06H, down – 07H.

Additional Call Registers
AL = number of lines to scroll
BH = display attribute
CH = y coordinate of top left
CL = x coordinate of top left
DH = y coordinate of lower right
DL = x coordinate of lower right

[Example] clear the screen

```
MOV AX,0600H ;scroll entire screen
MOV BH,07 ;normal attribute
MOV CX,0000 ;start at 00,00
MOV DX,184FH ;end at 24,79 (hex = 18,4F)
INT 10H ;invoke the interrupt
```

- Set cursor position:** 02H.

Additional Call Registers
BH = page number
DH = row
DL = column

- Set video mode:** 00H. In text mode, the screen is viewed as a matrix of rows and columns of characters, but in graphics mode, the screen is viewed as a matrix of horizontal and vertical pixels (each pixel can have different color, and the size of video memory decides the number of pixels and colors.

- Draw Pixel:** 0CH.

Additional Call Registers
BH = page number
DH = row
DL = column

Example 4-5

Write a program to:

- Clear the screen.
- Set the mode to CGA of 640 × 200 resolution.
- Draw a horizontal line starting at column = 100, row = 50, and ending at column 2

Solution:

```
MOV AX,0600H ;SCROLL THE SCREEN
MOV BH,07 ;NORMAL ATTRIBUTE
MOV CX,0000 ;FROM ROW=00,COLUMN=00
MOV DX,184FH ;TO ROW=18H,COLUMN=4FH
INT 10H ;INVOKE INTERRUPT TO CLEAR SCREEN
MOV AH,00 ;SET MODE
MOV AL,06 ;MODE = 06 (CGA HIGH RESOLUTION)
INT 10H ;INVOKE INTERRUPT TO CHANGE MODE
MOV CX,100 ;START LINE AT COLUMN = 100 AND
MOV DX,50 ;ROW = 50
MOV AH,0CH ;AH=0CH TO DRAW A LINE
MOV AL,01 ;PIXELS = WHITE
BACK: INC CX ;INVOKE INTERRUPT TO DRAW LINE
CMP CX,200 ;INCREMENT HORIZONTAL POSITION
JNZ BACK ;DRAW LINE UNTIL COLUMN = 200
```

3. **DOS Interrupt:** 21H; provided by MS-DOS; based on BIOS-ROM. After the DOS is loaded into the memory, you can invoke INT 21H to perform some extremely useful functions; by setting AH with different values, you can invoke these functions.

- Output string on screen:** 09H. Can be used to send a set of ASCII data to the monitor. DX is set to the offset

address of the ASCII string to be displayed (DS is assumed to be the data segment). All characters will be displayed until it encounters the dollar sign ‘\$’.

[Example]

```
DATA_ASC DB 'The earth is but one country','$'
MOV AH,09 ;Option 09 to display string of data
MOV DX,OFFSET DATA_ASC ;DX= offset address of data
INT 21H ;invoke the interrupt
```

- Exit to DOS:** 4CH. (AL = 00H).

Lecture 11. Serial Data Communication and 8251

1. **Data transmission:** is the transfer of data from point to point often represented as an electromagnetic signal over a physical communication channel

2. **A communication channel:** refers to the medium used to convey information from a sender (or transmitter) to a receiver. (e.g., copper wires, optical fibers of wireless communication channel).

3. **Two ways:** parallel and serial.

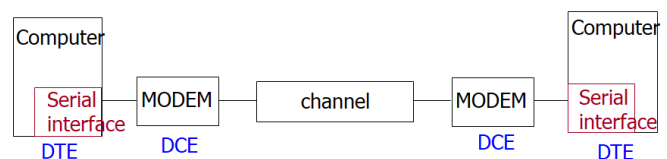
- Parallel data transfers:** (e.g., data bus)

- Each bit uses a separate line (wire);
- Often 8 or more lines are used;
- Control signals in addition;
- Fast, expensive, for short-distance communication

- Serial data transfers:** (e.g. Internet)

- One single data line;
- Bits are sent over the line one by one;
- No dedicated lines for control signals;
- Cheap & slow & for long-distance communication;

4. **Serial Communication:** the sender and receiver need a protocol to make sense of data (e.g., how the data is packed, how many bits constitute a character, what the data begin and end, etc.)



where, DTE is Data Terminal Equipment, usually a computer; DCE is Data Communication Equipment, usually a *modem* (modulize or demodulize the signals); serial interface is ICs such as 8251A, 16550 and 8250, connecting DTE and DCE.

5. **Main concern of serial communication:**

- Data transfer rate;
- Synchronization methods;
- Communication modes;
- Error detection;
- Modulation and Demodulation;