CPET9

PIPELINING

Basic Concepts of Pipelining

Pipelining

- Technique of decomposing a sequential process into a sub –operations/sub – process of about the same complexity.
- Each sub – operations/sub – process has an assigned segments that operates concurrently with all the other segments.
- To be able for these segments to operate simultaneously with each other (even if computation overlaps), a register is associated to each to isolate one segment from the other.

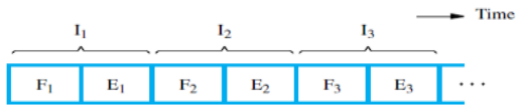
5 Classic Stages of Pipeline

- Instruction Fetch (Ifetch or IF) – obtains the requested instruction from memory
- Register Read (Reg) – read registers while decoding the instruction (DI)
- Decode Instruction (DI) – decodes instruction and sends various control lines to the other parts of the processor
- Execute (EX/ALU) – performs calculations through ALU (Arithmetic Logic Unit).
- Data Memory Access (Dmem) – stores and loads values to and from memory and to the input or output of a processor.
- Register Write (Reg or RW) - writing the result of a calculation, memory access or input into the register file

Traditional Pipeline Concept

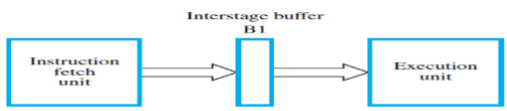
- The basic idea of a pipeline is very simple. It can be compare to a small laundry with one washer, one dryer and one operator.
- The Washer operates at 30 minutes, Dryer at 40 minutes and an Operator at 20 minutes. It takes 90 minutes to finish one load.
- A new task will not start unless one is already done with the previous task.
- The process is sequential. Sequential laundry takes 6 hours for 4 loads.
- While one is already finish in a certain task, another can work on task that he/she needs to do, and yet another can proceed to a task that they need to do also.
- The process is pipelined. Pipelined laundry takes 3.5 hours for 4 loads

Pipelining in a Computer



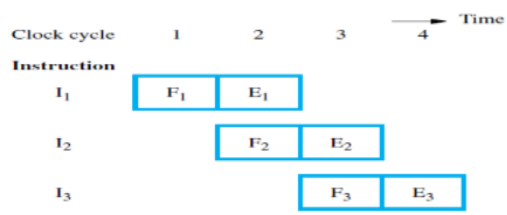
(a) Sequential execution

- The figure above shows an execution of a program which consists of a sequence of fetch and execute steps.
- Fi and Ei refer to the fetch and execute steps for instruction Ii.
- The processor executes a program by fetching and executing instructions, one after the other.



(b) Hardware organization

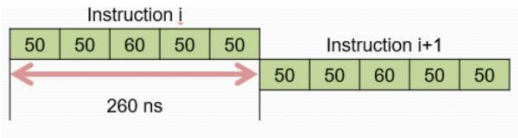
- The figure above shows a computer that has two separate hardware units, one for fetching instructions and another for executing them.
- The figure above shows an operation of a computer that's being controlled by a clock.
- The fetch and execute steps of any instruction completes one clock
- cycle.
- In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle.



(c) Pipelined execution

Consider a nonpipelined machine with 5 execution stages of lengths 50ns, 50ns, 60ns, 50ns, and 50ns.

- Find the instruction latency on this machine.
- How much time does it take to execute 100 instructions?




Instruction Latency

- Also known as the Response time or Execution time
- Time between the start and the completion of a task

• Note: In non-pipelined systems, instructions (n) require (n*k (stages)) cycles to process. Example: 5 instructions require 5 clock cycles

- ***Solution:***
- ***Instruction latency*** *= 50+50+60+50+50=* ***260ns***
- ***Time to execute 100 instructions*** *= 100*300 =* ***26000ns***

Suppose we introduce pipelining on this machine. Assume that when introducing pipelining, the clock skew adds 5ns of overhead to each execution stage.

- What is the instruction latency on the pipelined machine?
- How much time does it take to execute 100 instructions?

***Solution:***

- *Note:* Using a pipelined system with k pipeline stages, instructions (n) require (k (stages)) + (n-1)) cycles to complete. Example for 5 instructions require (5 + (5-1)) = 9 clock cycles
- *Remember that in the pipelined implementation, the length of the pipe stages must all be the same, i.e., the speed of the slowest stage plus overhead. With 5ns overhead it comes to:*

- The length of pipelined stage = MAX(lengths of unpipelined stages) + overhead = 60 + 5 = 65ns
- Instruction latency = 65ns
- Time to execute 100 instructions = 65*5 + 65*99 = 325 + 6435 = 6760ns

Speedup

- the ratio of the average instruction latency without pipelining to the average instruction latency with pipelining.

• What is the speedup obtained from pipelining?

**Solution:**

*Speedup = Instruction Latency (Unpipelined) / Instruction Latency (Pipelined)*

- Speedup=**260**/**65**
- Speedup = **4**

Pipelining Hazards

Hazard- a situation that prevents the next instruction in the instruction stream from executing during its designated clock cycle.

Three classes of hazards

1. Data hazard – Arise when either of the source or the destination operands of an instruction are not available at the time expected in the pipeline.

Solutions:

- Stall – involves always waiting for the PC to be updated with the correct address
- before moving on or stop loading instructions until result is available.
- Forwarding – connect new value directly to next stage before the completion of another instruction
- Reordering

2. Structural hazard – Arise when two instructions attempt to use a given hardware resource at the same time.

Solutions:

- Delay the second access by one clock cycle
- Provide separate memories for instructions & data

3. Instruction (control) hazard - Arise when there is an attempt to make a decision before condition is evaluated.

Solutions

- Stall – involves always waiting for the PC to be updated with the correct address

before moving on or stop loading instructions until result is available.
- Predict - assume whether the branch is taken or not, and acting on that guess (If correct, then proceed with normal pipeline execution, undo if prediction is wrong); lose cycles only on mis-prediction
- Delayed branch – involves executing the next sequential instruction with the branch taking place after that delayed branch slot.

---

CPET9L
EMU 8086

DIFFERENCES BETWEEN 8085, 8086 and 8088 Microprocessors

| 8085 | 8086 | 8088 |
|---|---|---|
| 8085 is an 8-bit microprocessor | 8086 is a 16 bit microprocessor | 8088 is a 16-bit microprocessor |
| It has 8-bit data bus | It has 16-bit data bus | It has 8-bit data bus |
| It has 8-bit ALU | It has 16-bit ALU | It has 16-bit ALU |
| 8085 does not require memory banking as it has an 8-bit data bus | 8086 requires memory banking to transfer 16-bit data at a time | 8088 does not require memory banking as it has an 8-bit data bus |
| 8085 performs slower memory operations as it can transfer only 8 bits in one cycle | 8086 performs faster memory operations as it can transfer 16 bits in one cycle | 8088 performs slower memory operations as it can transfer only 8 bits in one cycle |

| 8085 | 8086 | 8088 |
|---|---|---|
| 8085 does not support pipeline architecture | 8086 supports pipeline architecture | 8088 supports pipeline architecture |
| 8085 has no prefetch queue as it does not support pipelining | 8086 has a 6-byte pre-fetch queue for pipelining | 8088 has a 4-byte pre-fetch queue for pipelining |
| 8085 has an IO/pin to differentiate between memory and I/O operations | 8086 has an M/pin to differentiate between memory and I/O operations | 8088 has an IO/pin to differentiate between memory and I/O operations |
| 8085 has no prefetch queue | 8086 BIU will fetch new bytes into the pipelining queue when 2 bytes of the queue are empty | 8088 BIU will fetch new byte into the pipelining queue when 1 byte of the queue is empty |
| 8085 has 5 flags | 8086 has 9 flags | 8088 has 9 flags |

Functional Units of 8086- 8086 contains two independent functional units: a Bus Interface Unit (BIU) and an Execution Unit (EU).

A. Bus Interface Unit (BIU)- The segment registers, instruction pointer and 6-byte instruction queue are associated with the bus interface unit (BIU).

- Handles transfer of data and addresses,
- Fetches instruction codes, stores fetched instruction codes in first-in-first-out register set called a queue,
- Reads data from memory and I/O devices,
- Writes data to memory and I/O devices,

It has the following functional parts:

1.) Instruction Queue: When EU executes instructions, the BIU gets 6-bytes of the next instruction and stores them in the instruction queue and this process is known as instruction pre fetch. This process increases the speed of the processor.
2.) Segment Registers: A segment register contains the addresses of instructions and data in memory which are used by the processor to access memory locations. It points to the starting address of a memory segment currently being used.

There are 4 segment registers in 8086 as given below:

- Code Segment Register (CS): Code segment of the memory holds instruction codes of a program.
- Data Segment Register (DS): The data, variables and constants given in the program are held in the data segment of the memory.
- Stack Segment Register (SS): Stack segment holds addresses and data of subroutines. It also holds the contents of registers and memory locations given in PUSH instruction.
- Extra Segment Register (ES): Extra segment holds the destination addresses of some data of certain string instructions.

Instruction Pointer (IP): The instruction pointer in the 8086 microprocessor acts as a program counter. It indicates to the address of the next instruction to be executed.

B. Execution Unit (EU)

- The EU receives opcode of an instruction from the
- queue, decodes it and then executes it. While Execution, unit decodes or executes an instruction, then the BIU fetches instruction codes from the memory and stores them in the queue.

- The BIU and EU operate in parallel independently. This makes processing faster.
- General purpose registers, stack pointer, base pointer and index registers, ALU, flag registers (FLAGS), instruction 8 decoder and timing and control unit constitute execution unit (EU). Let's discuss them:
- General Purpose Registers: There are four 16-bit general purpose registers: AX (Accumulator Register), BX (Base Register), CX (Counter) and DX. Each of these 16-bit registers are further subdivided into 8-bit registers as shown below:

| 16 Bits Register | 8-bit high order registers | 8-bit low order registers |
|---|---|---|
| AX | AH | AL |
| BX | BH | BL |
| CX | CH | CL |
| DX | DH | DL |

- Index Register: The following four registers are in the group of pointer and index registers:
  1.) Stack Pointer (SP)
  2.) Base Pointer (BP)
  3.) Source Index (SI)
  4.) Destination Index (DI)

- ALU: It handles all arithmetic and logical operations. Such as addition, subtraction, multiplication, division, AND, OR, NOT operations.
- Flag Register: It is a 16-bit register which exactly behaves like a flip-flop, means it changes states according to the result stored in the accumulator. It has 9 flags, and they are divided into 2 groups i.e., conditional and control flags.

8086 Instruction Sets

1.) Data Transfer Instructions- These instructions are used to transfer the data from the source operand to the destination operand.

Instruction to transfer a word

- MOV – Used to copy the byte or word from the provided source to the provided destination.
- PPUSH – Used to put a word at the top of the stack.
- POP – Used to get a word from the top of the stack to the provided location.
- PUSHA – Used to put all the registers into the stack.
- POPA – Used to get words from the stack to all registers.
- XCHG – Used to exchange the data from two locations.
- XLAT – Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- IN – Used to read a byte or word from the provided port to the accumulator.
- OUT – Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- LEA – Used to load the address of operand into the provided register.
- LDS – Used to load DS register and other provided register from the memory.
- LES – Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- LAHF – Used to load AH with the low byte of the flag register.
- SAHF – Used to store AH register to low byte of the flag register.
- PUSHF – Used to copy the flag register at the top of the stack.
- POPF – Used to copy a word at the top of the stack to the flag register.

2.) Arithmetic Instructions- These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Instructions to perform addition

- ADD – Used to add the provided byte to byte/word to word.
- ADC – Used to add with carry.

- INC − Used to increment the provided byte/word by 1.
- AAA − Used to adjust ASCII after addition.
- DAA − Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- SUB − Used to subtract the byte from byte/word from word.
- SBB − Used to perform subtraction with borrow.
- DEC − Used to decrement the provided byte/word by 1.
- NPG − Used to negate each bit of the provided byte/word and add 1/2's complement.
- CMP − Used to compare 2 provided byte/word.
- AAS − Used to adjust ASCII codes after subtraction.
- DAS − Used to adjust decimal after subtraction.

Instruction to perform multiplication

- MUL − Used to multiply unsigned byte by byte/word by word.
- IMUL − Used to multiply signed byte by byte/word by word.
- AAM − Used to adjust ASCII codes after multiplication.

Instruction to perform division

- DIV − Used to divide the unsigned word by byte or unsigned double word by word.
- IDIV − Used to divide the signed word by byte or signed double word by word.
- AAD − Used to adjust ASCII codes after division.
- CBW − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- CWD − Used to fill the upper word of the double word with the sign bit of the lower word.
-

3.) Bit Manipulation Instructions- These instructions are used to perform operations where data bits are involved, i.e., operations like logical, shift, etc.

Instructions to perform logical operation

- NOT − Used to invert each bit of a byte or word.
- AND − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- OR − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- XOR − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- TEST − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- SHL/SAL − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- SHR − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- SAR − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

4.) String Instructions- String is a group of bytes/words, and their memory is always allocated in a sequential order.

- REP − Used to repeat the given instruction till CX ≠ 0.
- REPE/REPZ − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- MOVS/MOVSB/MOVSW − Used to move the byte/word from one string to another
- COMS/COMPSB/COMPSW − Used to compare two string bytes/words.
- INS/INSB/INSW − Used as an input string/byte/word from the I/O port to the provided memory location.
- OUTS/OUTSB/OUTSW − Used as an output string/byte/word from the provided memory location to the I/O port
- SCAS/SCASB/SCASW − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- LODS/LODSB/LODSW − Used to store the string byte into AL or string word into AX.

5.) Program Execution Transfer Instructions (Branch and Loop Instructions)- These instructions are used to transfer/branch the instructions during an execution.

Instructions to transfer the instruction during an execution without any condition

- CALL − Used to call a procedure and save their return address to the stack.
- RET − Used to return from the procedure to the main program.
- JMP − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions

- JA/JNBE − Used to jump if above/not below/equal instruction satisfies.
- JAE/JNB − Used to jump if above/not below instruction satisfies.
- JBE/JNA − Used to jump if below/equal/not above instruction satisfies.
- JC − Used to jump if carry flag CF = 1
- JE/JZ − Used to jump if equal/zero flag ZF = 1
- JG/JNLE − Used to jump if greater/not less than/equal instruction satisfies.
- JGE/JNL − Used to jump if greater than/equal/not less than instruction satisfies.
- JL/JNGE − Used to jump if less than/not greater than/equal instruction satisfies.
- JLE/JNG − Used to jump if less than/equal/if not greater than instruction satisfies.
- JNC − Used to jump if no carry flag (CF = 0)
- JNE/JNZ − Used to jump if not equal/zero flag ZF = 0
- JNO − Used to jump if no overflow flag OF = 0
- JNP/JPO − Used to jump if not parity/parity odd PF = 0
- JNS − Used to jump if not sign SF = 0
- JO − Used to jump if overflow flag OF = 1
- JP/JPE − Used to jump if parity/parity even PF = 1
- JS − Used to jump if sign flag SF = 1

6.) Processor Control Instructions- These instructions are used to control the processor action by setting/resetting the flag values.

- STC − Used to set carry flag CF to 1
- CLC − Used to clear/reset carry flag CF to 0
- CMC − Used to put complement at the state of carry flag CF.
- STD − Used to set the direction flag DF to 1
- CLD − Used to clear/reset the direction flag DF to 0 16
- STI − Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- CLI − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

7.) Iteration Control Instructions- These instructions are used to execute the given instructions for number of times.

- LOOP − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- LOOPE/LOOPZ − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- LOOPNE/LOOPNZ − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- JCXZ − Used to jump to the provided address if CX = 0

8.) Interrupt Instructions- These instructions are used to call the interrupt during program execution.

- INT − Used to interrupt the program during execution and calling service specified.
- INTO − Used to interrupt the program during execution if OF = 1
- IRET − Used to return from interrupt service to the main program

8086 Variables

Allocating Storage Space for Initialized Data

Where, variable name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in data segment.

| Directive | Purpose | Storage Space |
|---|---|---|
| DB | Define Byte | allocates 1 byte |
| DW | Define Word | allocates 2 bytes |
| DD | Define Doubleword | allocates 4 bytes |
| DQ | Define Quadword | allocates 8 bytes |
| DT | Define Ten Bytes | allocates 10 bytes |

Following are examples of using define directives

| | | |
|---|---|---|
| choice | DB | 'y' |
| number | DW | 12345 |
| neg_number | DW | -12345 |
| big_number | DQ | 123456789 |
| real_number1 | DD | 1.234 |
| real_number2 | DQ | 123.456 |

Allocating Storage Space for Uninitialized Data

The reserve directives are used for reserving space for uninitialized data

| Directive | Purpose |
|---|---|
| RESB | Reserve Byte |
| RESW | Reserve Word |
| RESD | Reserve Doubleword |
| RESQ | Reserve Quadword |
| REST | Reserve Ten Bytes |

EMU8086- The Microprocessor Emulator

• Emu8086 is the emulator of 8086 (Intel and AMD compatible) microprocessor and integrated assemblers

• The emulator runs programs like the real microprocessor in step- by-step mode.

• It shows registers, memory, stack, variables and flags.

• All memory values can be investigated and edited by a double click.

• The instructions can be executed back and forward.

• Emu8086 can create a tiny operating system and write its binary code to a bootable floppy disk.

• The software package includes several external virtual devices: robot, stepper motor, led display and traffic lights

intersection. Additional devices can be created.