#132, AECS Layout, IT Park Road, Kundalahalli, Bangalore – 560 037
T:+9180 28524466 / 77

CMR INSTITUTE OF TECHNOLOGY

Scheme and Solution

Department of Computer Science and Engineering

SEMESTER : VII-A, B
BRANCH : CSE
NAME OF THE FACULTY : Reshma Shet
SUBJECT : Advanced Computer Architecture

Scheme

Q.1 A. Definition of ISA (1 mark)

Explanation of 7 dimensions (7 marks)

B. Explanation of Amdahl's law with fraction enhanced and speedup enhanced (1 mark)

Definition of Speedup and formula (1 mark)

Q.2 A. Formula for MTTF (1 mark)

Calculation of FIT for whole system (2 marks)

Calculation of MTTF of whole system (2 marks)

B. Formula for die yield (1 mark)

Calculation of die yield for die of 1.0 cm (2 marks)

Calculation of die yield for die of 1.5 cm (2 marks)

Q.3 Explanation of trends in power along with formula for static power and dynamic power (5 marks).

Explanation of trends in cost along with formula for Cost of IC, No of dies per wafer and die yield (5 marks).

Q.4 Explanation of measuring the performance (4 marks).

Explanation on reporting the performance (3 marks).

Explanation on summarizing the performance (3 marks).

Q.5. Explanation of classic five stage pipeline for RISC processor without pipeline register and with pipeline registers (8 marks).

Neat diagram for classic five stage pipeline for RISC processor with pipeline register and without pipeline register (2 marks).

Q.6. Explanation of Pipelining (1 mark)

Listing three different type of Hazards (1 mark).

Explanations on any two hazards with neat diagram (8 marks).

Q.7 A. Explanation on five different clock cycles i.e. IF, ID, EX, MEM and WB (5 marks).

B. Explanation on five different categories of exception i.e. Synchronous versus Asynchronous, User requested versus Coerced, within versus between, User maskable versus User non-maskable and resume versus terminate (5 marks).

Q.8 A. Definition of ILP (1 mark).

Listing three types of Dependencies (1 mark)

Explanation on data dependence (2 marks)

Explanation of RAW, WAW and WAR hazard with example (6 marks)

Solution

Q.1 A **Definition of ISA** : Instruction set Architecture is the interface between the software and hardware. ISA defines the instructions, data types, registers, addressing modes which are visible to the programmer.

Seven dimensions of ISA

1. Class of ISA
2. Memory Addressing
3. Addressing modes
4. Type and size of operands
5. Operations
6. Control flow instruction
7. Encoding an ISA

Class of ISA

1. There are two important classes of GPR based ISA.
2. Register-memory ISA such as 80*86: Here one input operand (i.e. data or value) is in register and other input operand in memory and after ALU operation result is stored in register.
3. Load-store ISA such as MIPS: Here one input operand is in register and other input operand is in register and result goes to register. To transfer the value to memory we need to use load store instructions.
4. The 80*86 has16 general purpose registers and 16 floating point registers.
5. MIPS has 32 general purpose registers and 32 floating-point register.

Memory Addressing

1. The 80*86 and MIPS use byte addressing for accessing memory operands.
2. Byte addressing refers to architectures where data can be accessed 8 bits (1 Byte) at a time. In some machines; accesses to objects larger than a byte must be aligned.
3. For MIPS the data is aligned and for 8086 data is misaligned.
4. An access to an object of size s bytes at byte address A is aligned if A mod s = 0. Misalignment causes hardware complications.

Addressing modes

1. Addressing modes specify the address of a memory object. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.
2. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of displacement: no register (absolute or direct addressing mode), two registers (based indexed with displacement), two registers where one register is multiplied by the size of the operand in bytes (based scaled index and displacement)

Type and Size of Operands

1. Like most of the ISAs the 8086 and MIPS support operand size of 8-bit (ASCII Character), 16-bit (Unicode character or half word), 32-bit (Integer or word), 64-bit (Long or double word) and floating point in 32-bit (Single precision) and 64-bit (Double precision).
2. The 80x86 also supports 80-bit floating point (extended double precision).

Operations

1. The general categories of operations are data transfer, arithmetic, logical, control, and floating point operations.
2. The instructions for data transfer are LB,LW,SB,SW,SD etc.
3. The instructions for arithmetic operations are DADD,DADDI,DMUL,DDIV etc.
4. The instructions for control flow operations are BEQ,BNEZ etc.

Control Flow Instructions

1. Virtually all ISAs including 8086 and MIPS support conditional branches, unconditional jumps, procedure calls, and returns.
2. There are some small differences. MIPS conditional branches (BE,BNE, etc.) test the contents of registers, while the 80x86 branches (JE,JNE, etc.) test condition code bits set as side effects of arithmetic/logic operations.
3. MIPS procedure call (JAL) places the return address in a register, while the 80x86 call (CALLF) places the return address on a stack in memory.

Encoding an ISA

1. Encoding means converting the instruction into binary form. There are two types of encoding 1) fixed length encoding 2) variable length encoding.
2. The MIPS has fixed length encoding and all instructions are 32-bit long.
3. While 8086 has Variable length encoding and instructions are ranging from 1-18 bytes, so a program compiled for the 80x86 is usually smaller than the same program compiled for MIPS

Q.1 B **Amdahl's law**

The performance gain that can be obtained by improving some portion of a computer can be calculated by Amdahl’s Law. Amdahl’s law defines the speedup that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement}}{\text{Performance for entire task without using the enhancement}}$$

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement}}$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Q.2 A)

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 SCSI controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 SCSI cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

Answer The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10 + 2 - 5 + 5 + 1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

Q. 2 B)

Example Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.4 per cm^2 and α is 4.

Answer The total die areas are 2.25 cm^2 and 1.00 cm^2. For the larger die, the yield is

$$\text{Die yield} = \left(1 + \frac{0.4 \times 2.25}{4.0}\right)^{-4} = 0.44$$

For the smaller die, it is $\text{Die yield} = \left(1 + \frac{0.4 \times 1.00}{4.0}\right)^{-4} = 0.68$

That is, less than half of all the large die are good but more than two-thirds of the small die are good.

Q. 3 **Trends in Power**

1. On IC for switching between multiple transistors there is dynamic power. The power required per transistor is proportional to the product of the load capacitance of the transistor, the square of voltage, and frequency of switching, with watts being the unit.

$$\text{Power}_{\text{dynamic}} = 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

2. But for mobile devices energy is calculated.

$$\text{Energy}_{\text{dynamic}} = \text{Capacitive load} \times \text{Voltage}^2$$

3. Distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges.
4. As a result of this limitation, most microprocessors today turn off the clock of inactive modules to save energy and dynamic power. For example, if no floating-point instructions are executing, the clock of the floating-point unit is disabled.
5. Although dynamic power is the primary source of power dissipation in CMOS, static power is becoming an important issue because leakage current flows even when a transistor is off.

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} \times \text{Voltage}$$

Trends in cost

<u>The Impact of Time, Volume and Commodification on cost</u>

1. The cost of the manufactured computer decreases over time.
2. Manufacturing cost also decreases over time. This is called learning curve.
3. The learning curve is measured by the yield. Yield means the percentage of manufactured devices which survives the testing procedure.
4. Volume also affects the cost in several ways. Volume decreases the cost, since it increases purchasing and manufacturing efficiency.

5. As the rule of thumb, some designers have estimated that costs decreases about 10% for each doubling of volume.
6. Moreover, volume decreases the amount of development cost that must be amortized by each computer, thus allowing selling price and cost price to be closer.
7. Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. For example: DRAMS, Monitors, keyboard, disk, mouse etc.
8. There is lot of competition among the suppliers of the components and thus it decreases the gap between the cost and selling price, but it also decreases the cost.

Typically IC is produced in large batches on a single wafer of EGS (Electronic Grade Silicon) or other semiconductor through process such as photolithography. The wafer is cut into many pieces each containing one copy of the circuit. Each of these pieces is called die

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

The cost of die is calculated by the formula.

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

The cost of die is dependent on the cost of wafer, number of dies which will fit on a wafer and also die yield. Die yield corresponds to percentage of good dies in the wafer. The number of good dies per wafer is approximately the area of the wafer divided by the area of the die. It is given as follows

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \text{Die area}}$$

Q.4. **Measuring Performance**

1. Benchmark is a standard which forms the basis of measurement and through which the performance of others could be judged.
2. The best choice of benchmark to measure the performance is the compiler itself. If we consider smaller programs as benchmarks but it will have many pitfalls. Examples include
3. Kernels which are small, key pieces of real applications. Example: Livermore loops,LINPACK.
4. Toy programs, which are 100 line programs for example: quick sort
5. Synthetic benchmarks, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone.
6. Thus nowadays various benchmark suites are available to measure the performance for variety of systems. Nonetheless, a key advantage of such suites is that the weakness of any one benchmark is lessened by the presence of the other benchmarks.
7. Desktop benchmarks divide into two broad classes: processor-intensive benchmarks and

graphics-intensive benchmarks.

8. SPEC originally created a benchmark set focusing on processor performance (initially called SPEC89), which has evolved into its fifth generation: SPEC CPU2006, which follows SPEC2000, SPEC95, SPEC92, and SPEC89.
9. SPEC benchmarks are real programs modified to be portable and to minimize the effect of I/O on performance.
10. The simplest benchmark used here is processor throughput benchmark. For example : SPECCPU2000 is a processor throughput benchmark which is used to measure the processing rate by running the multiple copies of each SPEC CPU benchmark on multiple processors.
11. Transaction-processing (TP) benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. Airline reservation systems and bank ATM systems are typical simple examples of TP systems in which this benchmark is needed.

Reporting Performance

1. SPEC report contains description of computer, Software and hardware requirements, and compiler flags, publication of baseline and optimized results, actual performance times shown in tabular form or graph.
2. PC benchmark report contains description of benchmarking audit, and cost. These reports are excellent source of finding the cost of the computing system.

Summarizing Performance

1. Consider the two computers A and B. The SPEC benchmark is run on both the computers A and B to measure the performance.
2. For summarizing results we estimate the SPECRatio for both computers. SPECRatio is obtained by dividing the execution time of reference computer to time on the computer being rated, yielding a ratio proportional to performance.
3. For example, suppose that the SPECRatio of computer A on a benchmark was 1.25 times higher than computer B; then you would know.

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

Because a SPECRatio is a ratio rather than an absolute execution time, the mean must be computed using the geometric mean. The formula is

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

Q.5 Classic five stage pipeline for RISC processor

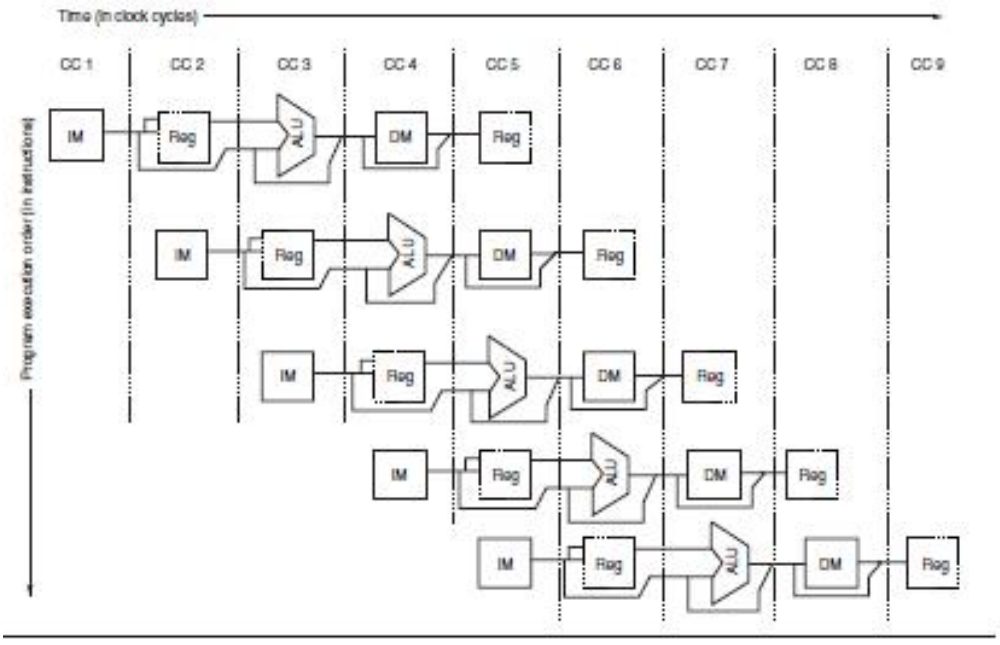
1. Every instruction in the RISC can be implemented in 5 clock cycles. The five clock cycles are IF, ID, EX, MEM and WB.

2. Although each instruction will take 5 clock cycles for its execution the hardware will initiate a new instruction in every clock cycle and will be executing some part of different instructions.
3. The basic RISC pipeline is shown below. On each clock cycle another instruction is fetched and begins its 5-cycle execution.

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

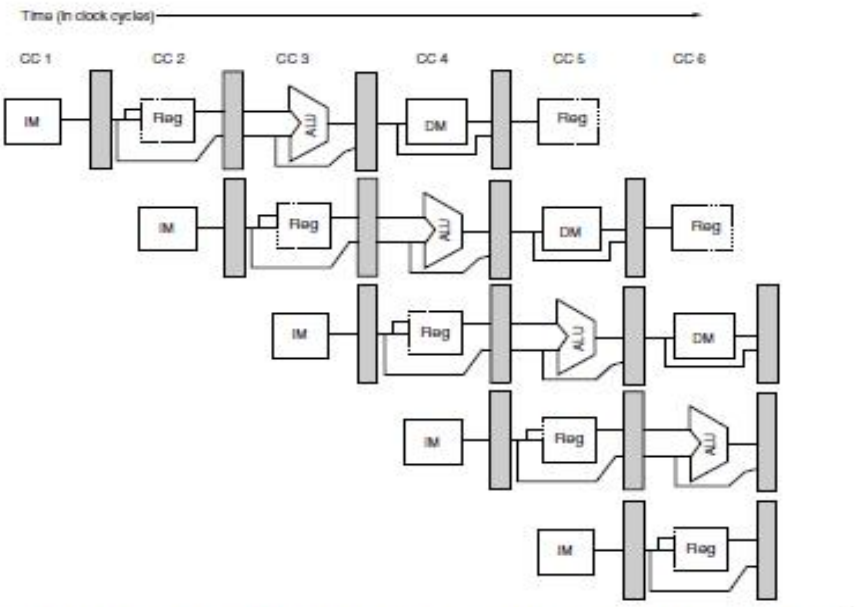
Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = Instruction fetch, ID = Instruction decode, EX = execution, MEM = memory access, and WB = write back.

4. But the overlapped execution of multiple instructions inside the pipeline should not introduce new type of conflicts.
5. Hence to avoid conflicts three methods are used in a pipeline.
6. First we separate the instruction and data memory which we would typically implement with separate instruction and data caches. The use of separate caches eliminates a conflict for a single memory that would arise between IF(Instruction Fetch) stage and memory access(MEM) stage.
7. Second the register file is used in two stages one for reading in ID and one for writing in WB. Thus to handle reading and writing to same register perform the register write in first half of the clock cycle and register read in the second half of the clock cycle. It is indicated by solid and dashed lines in the figure given below.
8. Figure shows the pipeline and abbreviation IM is used for instruction memory and DM for data memory and CC for clock cycle.



9. The PC should be updated to start the execution of the new instruction in every clock cycle. PC points to the address of the next instruction to be fetched for execution. But for branches the PC is not changed until ID stage and it causes problem.

10. Also pipeline registers are present between the successive stages of the pipeline so that all the results from a given stage are stored into register that is used as the input to the next stage on the next clock cycle.
11. For example: The result of the ALU operation is computed during the EX stage, but not actually stored until WB, it arrives there by passing through two pipeline registers. The pipeline registers are named as IF/ID, ID/EX, EX/MEM and MEM/WB.



A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

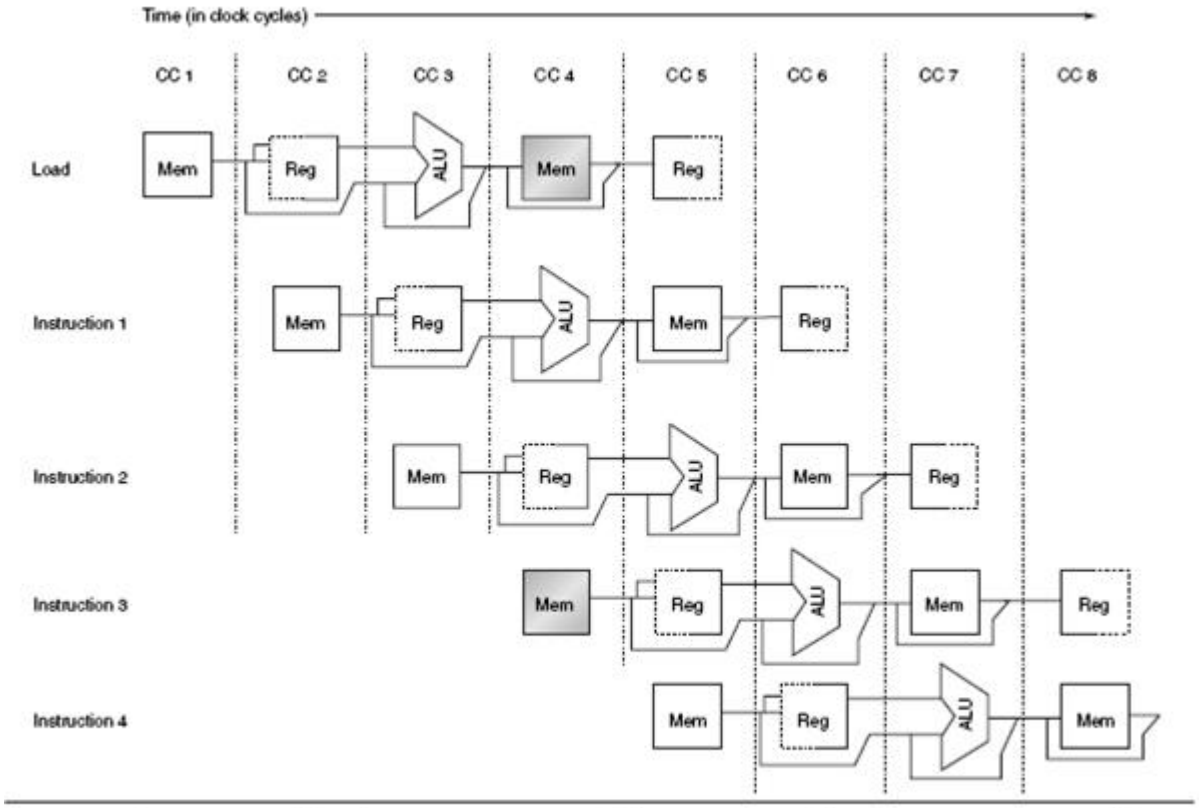
Q.6 **Pipelining**:
It is the implementation technique where multiple instructions are overlapped in execution. Today pipelining is the key implementation technique used in to make fast CPUs.

Various hazards in Pipeline are as follows
1) Structural Hazard
2) Data Hazard
3) Branch Hazard

Structural Hazard

- Structural Hazards arise due to resource conflicts. For example, a processor may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle.
- Structural hazards also arise when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle.

- Thus when a processor is pipelined, the overlapped execution of instruction requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- When the structural hazard is encountered, the pipeline will stall one of the instructions until the required unit is available. Such stalls will
- Figure shows a structural hazard between load instruction and instruction 3. Both load instruction and instruction 3 want to access one memory port at the same time and hence there is a resource conflict.



A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

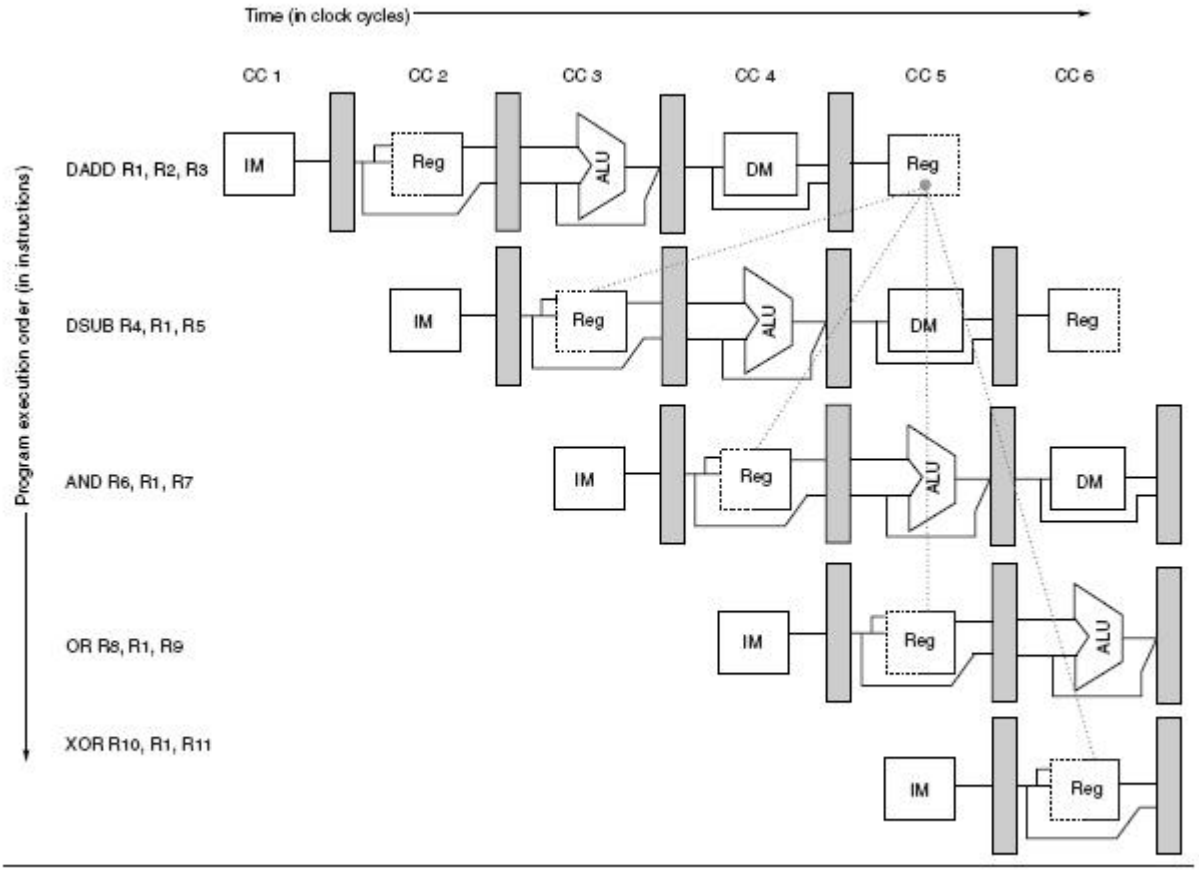
Data Hazard

- Data hazard occurs when one instruction is dependent on the result produced by the other instruction.
- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Consider the pipelined execution of these instructions:

  DADD R1,R2,R3
  DSUB R4,R1,R5
  AND R6,R1,R7
  OR R8,R1,R9

XOR R10,R1,R11

- All the instructions after the DADD use the result of the DADD instruction i.e. R1. As shown in Figure the DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage. This problem is called a *data hazard*.



The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

- The AND instruction is also affected by this hazard. As we can see from Figure, the write of R1 does not complete until the end of clock cycle 5. Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results.
- The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write. The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

Q.7 A MIPS instruction implementation in five clock cycle

1. *Instruction fetch cycle* (IF):

IR ← Mem[PC]; NPC ← PC + 4;
Operation: Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

The IR is used to hold the instruction that will be needed on subsequent clock cycles; likewise the register NPC is used to hold the next sequential PC.

2. *Instruction decode/register fetch cycle* (ID):
A ← Regs[rs];
B ← Regs[rt];
Imm ← sign-extended immediate field of IR;
Operation: The values from two source registers are read into two temporary registers (A and B) for use in later clock cycles. If instruction contains 16 bit immediate field it is sign extended and stored into the temporary register Imm, for use in the next cycle. Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the MIPS instruction format

3. *Execution/effective address cycle* (EX): The ALU operates on the operands prepared in the prior cycle, performing all given functions below depending on the MIPS instruction type.

- Memory reference:
  ALUOutput ← A + Imm;
  Operation: The ALU adds the operands to form the effective address and places the result into the temporary register ALUOutput.
- Register-Register ALU instruction:
  ALUOutput ← A *func* B; *Operation:* The ALU performs the operation specified by the function code on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.
- Register-Immediate ALU instruction:
  ALUOutput ← A *op* Imm; *Operation:* The ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. The result is placed in the temporary register ALUOutput.

***4. Memory access/branch completion cycle* (MEM):** The PC is updated for all instructions: PC ← NPC;
Memory reference:
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
Operation: Access memory if needed. If instruction is a load, data returns from memory and is placed in the LMD (load memory data) register; if it is a store, then the data from the B register is written into memory. In either case the address used is the one computed during the prior cycle and stored in the register ALUOutput.
Branch: If branch instruction is taken branch PC is replaced by branch target address.

***5. Write-back cycle* (WB):**
Register-Register ALU instruction: Regs[rd] ← ALUOutput;
Register-Immediate ALU instruction: Regs[rt] ← ALUOutput;
Load instruction Regs[rt] ← LMD;
Operation: Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput); the register destination field is also in one of two positions (rd or rt) depending on the effective opcode.

Q.7 B. Exception in MIPS

1. Synchronous versus asynchronous—If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is *synchronous*. With the exception of hardware malfunctions, *asynchronous* events are caused by devices external to the CPU and memory.

Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.

2. ***User requested versus coerced—***If the user task directly asks for it, it is a *user-requested* event. In some sense, user-requested exceptions are not really exceptions, since they are predictable. They are treated as exceptions, however, because the same mechanisms that are used to save and restore the state are used for these user-requested events. Because the only function of an instruction that triggers this exception is to cause the exception, user requested exceptions can always be handled after the instruction has completed. *Coerced* exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable.

3. ***User maskable versus user nonmaskable***—If an event can be masked or disabled by a user task, it is *user maskable*. This mask simply controls whether the hardware responds to the exception or not.

4. ***Within versus between instructions***—This classification depends on whether the event prevents instruction completion by occurring in the middle of execution—no matter how short—or whether it is recognized *between* instructions. Exceptions that occur *within* instructions are usually synchronous, since the instruction triggers the exception. It's harder to implement exceptions that occur within instructions than those between instructions, since the instruction must be stopped and restarted. Asynchronous exceptions that occur within instructions arise from catastrophic situations (e.g., hardware malfunction) and always cause program termination.

5. ***Resume versus terminate***—If the program's execution always stops after the interrupt, it is a *terminating* event. If the program's execution continues after the interrupt, it is a *resuming* event. It is easier to implement exceptions that terminate execution, since the CPU need not be able to restart execution of the same program after handling the exception.

Q.8. ILP: The ability to execute multiple instructions parallel inside a pipeline is called as Instruction Level Parallelism (ILP)

Various types of dependencies are as follows

1. Data Dependencies
2. Name Dependencies
3. Control Dependencies.

Data Dependence

An instruction *j* is *data dependent* on instruction *I* if either of the following holds: instruction *i* produces a result that may be used by instruction *j*, or instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

For example: Consider following instruction

```
ADD.D    F3, F1,F2
MUL.D    F4,F3,F5
```

Here the instruction MUL.D depends on the instruction ADD.D for the register F3.

Various types of Data Hazard

A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order,* that is, the order that the instructions would execute in if executed sequentially one at a time

Consider two instructions *i* and *j,* with *i* preceding *j* in program order. The possible data hazards are

RAW *(read after write)*: *j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value. This hazard is the most common type and corresponds to true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*.

WAW *(write after write)—j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

WAR *(write after read)—j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value. This hazard arises from an antidependence.