UNIT-5

Instruction Set Architecture:

Instruction set architecture (ISA) to refer to the actual programmer visible instruction set in this book. The ISA serves as the boundary between the software and hardware

Seven dimensions of an ISA :

1. Class of ISA -all ISAs are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating point data, while MIPS has 32 general-purpose and 32 floating-point registers The two popular versions of this class are register-memory ISAs, such as the 80x86, which can access memory as part of many instructions, and load-store ISAs, such as ARM and MIPS, which can access memory only with load or store instructions. All recent ISAs are load-store.

2. Memory addressing—virtually all desktop and server computers, including the 80x86, ARM, and MIPS, use byte addressing to access memory operands. Some architectures, like ARM and MIPS, require that objects must be aligned. The 80x86 does not require alignment, but accesses are generally faster if operands are aligned.

3. Addressing modes—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), two registers (based indexed with displacement), and two registers where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement

ARM has the three MIPS addressing modes plus PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes, autoincrement and autodecrement addressing,

4. Types and sizes of operands—Like most ISAs, 80x86, ARM, and MIPS support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).

5. Operations—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point.

6. Control flow instructions—All ISAs support conditional branches, unconditional jumps, procedure calls, and returns. All three use PC-relative addressing, where the branch address is specified by an

address field that is added to the PC. MIPS conditional branches (BE, BNE, etc.) test the contents of registers.

80x86 and ARM branches test condition code bits set as side effects of arithmetic/logic operations. The ARM and MIPS procedure call places the return address in a register, while the 80x86 call (CALLF) places the return address on a stack in memory.

7. Encoding an ISA—There are two basic choices on encoding: fixed length and variable length. All ARM and MIPS instructions are 32 bits long, which simplifies instruction decoding. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable length instructions can take less space than fixed-length instructions.

Pipelining:

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.

The time required between moving an instruction one step down the pipeline is a processor cycle. The length of a processor cycle is determined by the time required for the slowest pipe stage. In a computer, this processor cycle is usually 1 clock cycle. The pipeline designer's goal is to balance the length of each pipeline stage. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

Time per instruction on unpipelined machine / Number of pipe stages

Pipelining yields a reduction in the average execution time per instruction.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream

Key properties of RISC architectures are :

- i All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
- ii The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively. Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- iii The instruction formats are few in number, with all instructions typically being one size.

Most RISC architectures, like MIPS, have three classes of instructions:

1. ALU instructions— These instructions take either two registers or a register and a sign-extended immediate operate on them, and store the result into a third register. Typical operations include add (DADD), subtract (DSUB), and logical operations (such as AND or OR) In MIPS, there are both signed and unsigned forms of the arithmetic instructions; the unsigned forms, which do not generate overflow exceptions—and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g., DADDU, DSUBU, DADDIU).

2. Load and store instructions—These instructions take a register source, called the base register, and an immediate field (16-bit in MIPS), called the offset, as operands. The sum—called the effective address is used as a memory address. In the case of a load instruction, a second register operand acts as the destination . In the case of a store, the second register operand is the source of the data that is stored into memory. The instructions load word (LD) and store word (SD) load or store the entire 64-bit register contents.

3. Branches and jumps—Branches are conditional transfers of control. There are usually two ways of specifying the branch condition in RISC architectures: with a set of condition bits (sometimes called a condition code) or by a limited set of comparisons between a pair of registers or between a register and zero.

Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows:

1. **Instruction fetch cycle (IF):** Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential

2. **Instruction decode/register fetch cycle (ID**): Decode the instruction and read the registers. Decoding is done in parallel with reading registers

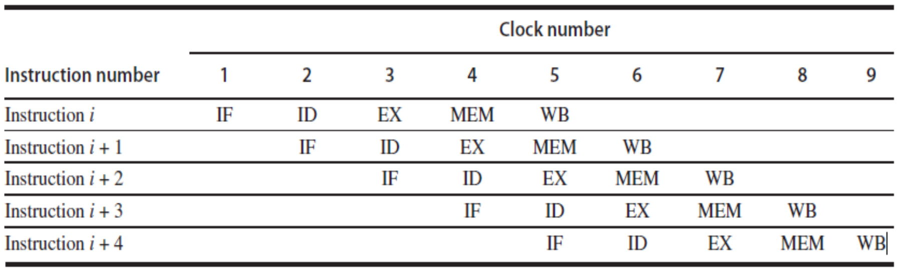
3.**Execution/effective address cycle (EX):** The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.
■ Memory reference—The ALU adds the base register and the offset to form the effective address.
■ Register-Register ALU instruction—The ALU performs the operation on the values in the register file.
■ Register-Immediate ALU instruction—The ALU performs the operation on the first value read from the register file and the sign-extended immediate.

4. **Memory access (MEM):** If the instruction is a load, the memory does a read using the effective address. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. **Write-back cycle (WB):**

Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

The Classic Five-Stage Pipeline for a RISC Processor:



	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

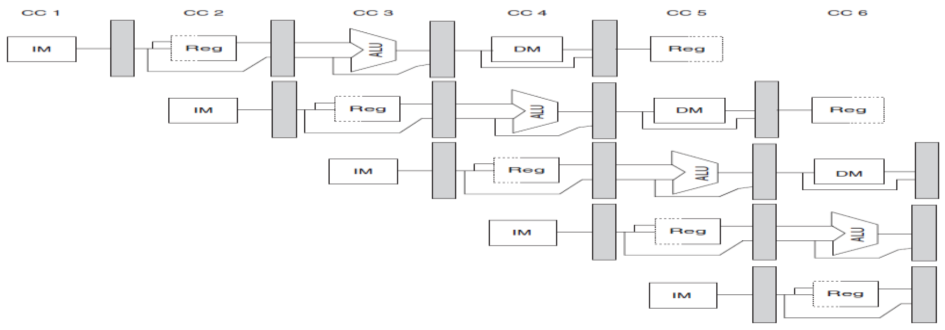
The Classic Five-Stage Pipeline for a RISC Processor Is implemented simply starting a new instruction on each clock cycle. Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline .

During each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions
The processor must make sure that two different operations are not performed on the same data path resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict.

Separate instruction and data memories with separate instruction and data caches are used. The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. The register file is used in the two stages: one for reading in ID and one for writing in WB.

To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target during ID

In pipeline, we must ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle.



The Major Hurdle of Pipelining—Pipeline Hazards

There arethree classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Structural Hazards

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.

The most common instances of structural hazards arise when some functional unit is not fully pipelined, then a sequence of instructions using that un-pipelined unit cannot proceed at the rate of one per clock cycle.

Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. 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. This will generate a structural hazard.

Structural hazard can be overcome by using Stall. The pipeline is stalled for 1 clock cycle when the data memory access occurs. A stall is commonly called a *pipeline bubble* or just *bubble*, since it floats through the pipeline taking space but carrying no useful work.

Data Hazards

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 ofthese 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. 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 data hazard can be solved using Forwarding or Stalling

Forwarding (also called bypassing and sometimes short-circuiting) is a technique in which the result of the previous instruction is moved directly to the next instruction required

Branch Hazards or Control hazards

Methods to overcome:

1. The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known.

2. higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed.

3. Predicted-not-taken or predicted untaken scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction

4. An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target.

5. Another scheme is to use delayed branch