Lecture 26-27: April 23, 2021

Computer Architecture and Organization-I

Biplab K Sikdar

Pipelining

Parallel processing is a solution to achieve high speed computation.

Number of parallel processing mechanisms have been developed in uniprocessor.

Multiplicity of functional units (CDC 6600 had 10 functional units), IBM-360/91 (1968) having 2 execution units (one for fixed-point arithmetic and another for floating point arithmetic); overlapped CPU and I/O operations (I/O processors, DMA etc); use of hierarchical memory system; multiprogramming and time sharing etc.

Most significant attempt to achieve speed-up in computation is *pipelining*.

Pipeline breaks a process into N steps for N-fold increase in processing speed.

Figure 1 represents a simple linear pipeline.

Pipelines is extended to various structures of interconnected processing elements.

Arithmetic pipelining is used in some specialized computers.

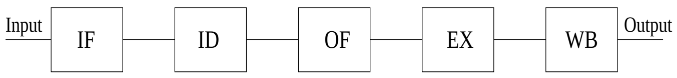


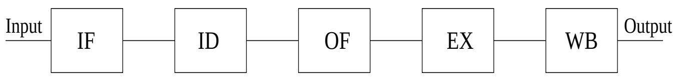
Figure 1: Instruction pipeline

Three categories - instruction pipeline, arithmetic pipeline and processor pipeline.

0.1 Instruction Pipeline

Instruction pipeline: various phases - instruction fetch (IF), decode (ID), operand fetch (OF), arithmetic and logic execution (EX), and store result (WB).

Figure 1 describes such a 5-stage instruction pipeline.



1. Instruction fetch (IF) Processor reads next instruction to be executed.
2. Instruction decode (ID) Processor works out what this instruction is.
3. Operand fetch (OF) Processor reads values required by the operation.
4. Instruction execution (EX) Processor executes instruction on operand values.
5. Result storage (WB) Processor stores result of operation in register/memory.

Execution of 4 instructions in 5-stage pipeline is shown in Figure 2.

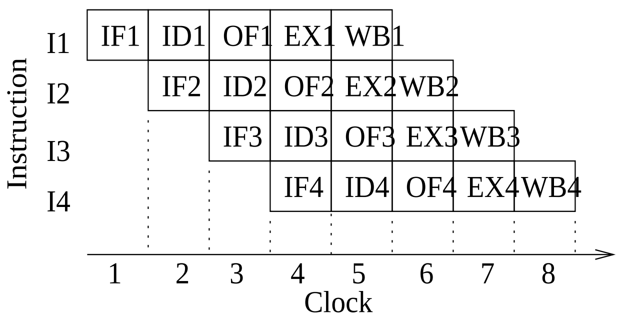


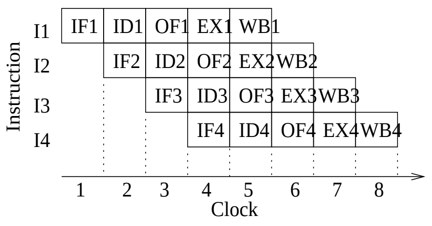
Figure 2: Instruction execution in 5-stage pipeline

0.1.1 Speed up

Execution time of n instructions, each passes through k stages, (without pipeline)

$$CPUtime_{non-pipelined} = \quad n \times k \times CT$$

where CT is clock time taken for computation in each stage.



In Figure 2, first instruction I-1 get executed after 5 clock cycles.

After 6th clock I$_2$, after 7th I$_3$ and so on are completed.

That is, for k-stage pipeline, time taken for execution of n instructions is

$$CPUtime_{pipelined} = \quad \text{k}\times \text{ CT + (n - 1) } \times \text{ CT} \ .$$

So speed-up of the k-stage pipeline execution is

$$\begin{aligned} Speedup_k &= \frac{CPUtime_{non-pipelined}}{CPUtime_{pipelined}} \\ &= \frac{n\times k\times CT}{k\times CT+(n-1)\times CT} \\ &= \frac{n\times k}{k+n-1} \\ &= \frac{k}{\frac{k}{n}+1-\frac{1}{n}} \\ &\simeq k \end{aligned}$$

as for a program $k \ << \ n$, the factors $\frac{k}{n}$ and $\frac{1}{n}$ are close to 0.

In case of 5-stage pipeline of Figure 1 and n = 10,000, the actual speed-up

$$Speedup_5 = \frac{10000\times 5\times CT}{(5+10000-1)\times CT} = \frac{50000}{10004} = 4.998 \simeq 5.$$

Actual speed-up is limited due to other factors - stages need not have same delay i.e, CT in non-pipelined architecture may differ from CT in pipelined architecture.

0.1.2 Latency/CPI

Pipelined execution of Figure 3 assumes latency 1.

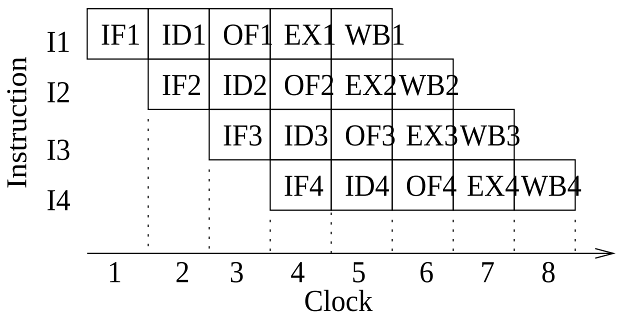


Figure 3:

Delay between issuing of two consecutive pipeline items is one pipeline cycle/clock.

CPI of a pipeline system is proportional to its latency.

CPI of instruction pipeline corresponding to Figure 3 is 1.

Pipelined execution shown in Figure 4 has latency = 2.

Delay between two consecutive pipeline items is two pipeline cycles. Its CPI is 2.

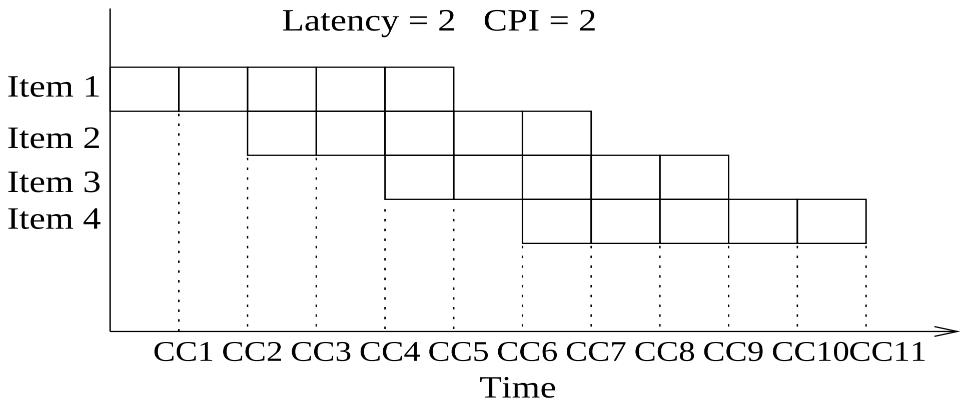


Figure 4: Pipelined execution with latency 2

0.1.3 Design issues

Several difficulties prevent instruction pipelining and hinder speed-up.

Construction of modules that run independently

Timing variations Not all the pipeline stages take the same dealy.

The speed-up will be determined by the slowest stage.

Different instructions can have different operand requirements.

For significant timing differences in stages, following solutions can be considered.

Solution 1 Figure 5.

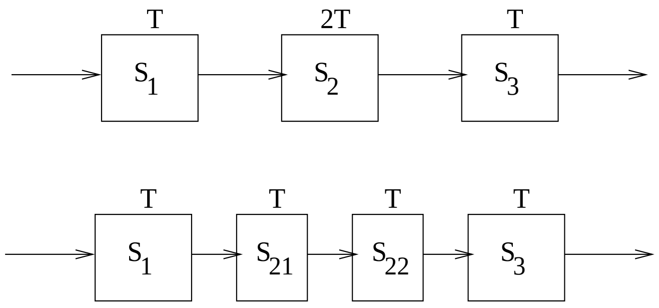


Figure 5: Timing mismatch : solution 1

However, independent moduling (S_{21}/S_{22}) may not be possible.

Another example

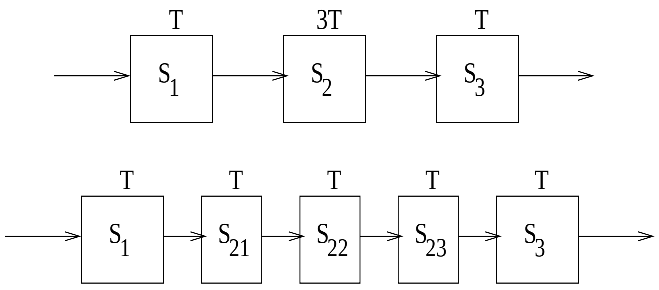


Figure 6: Timing mismatch : solution 1

Solution 2 Figure 7.

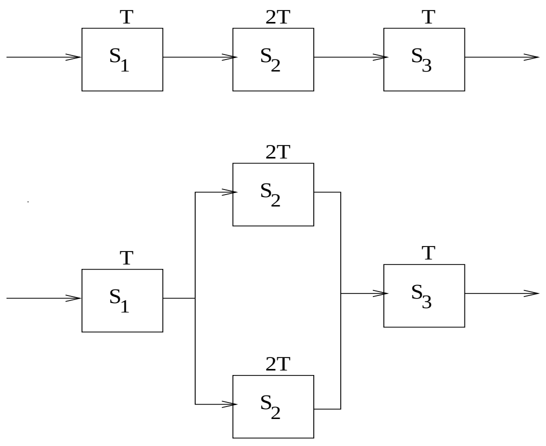


Figure 7: Timing mismatch : solution 2

Needs mechanism to divert pipeline items - one-to-many and many-to-one function.

Require MUX/deMUX and slow down processor. Speedup may not be close to k.

Another example

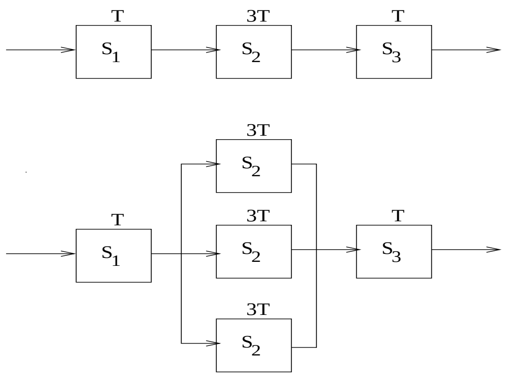


Figure 8: Timing mismatch : solution 2

Pipeline can be synchronous or asynchronous.

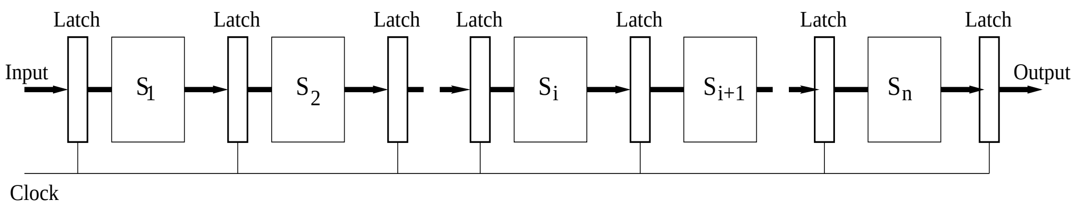


Figure 9: Synchronous pipeline

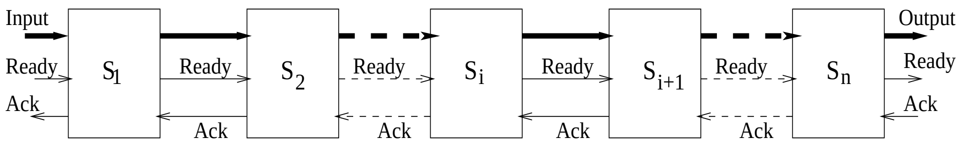


Figure 10: Asynchronous pipeline

Hazards

Structural hazard

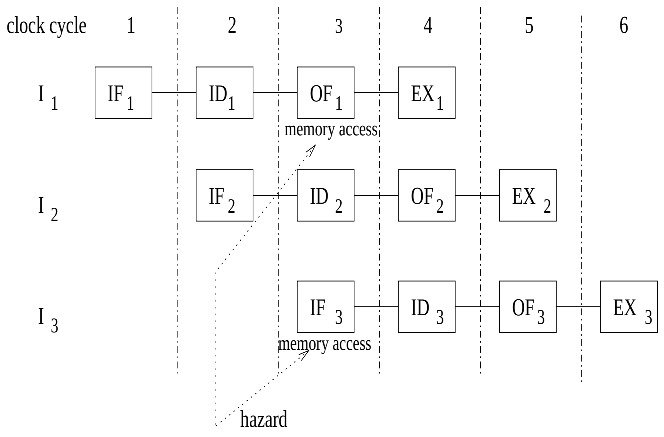


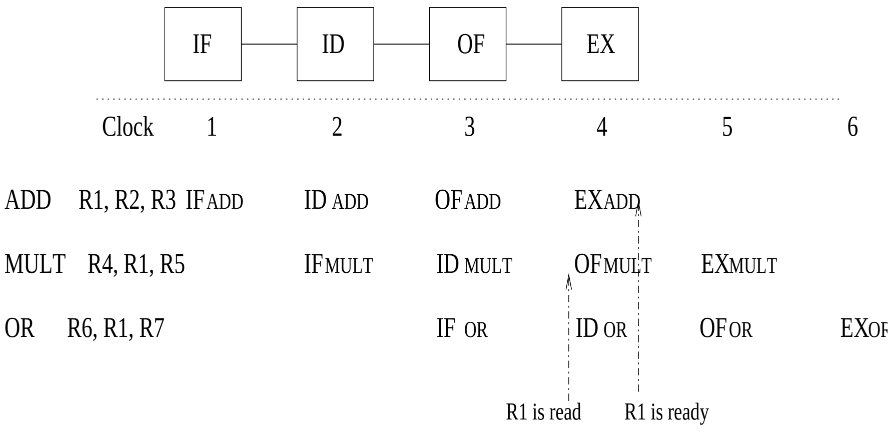
Figure 11: Structural hazard in 4-stage instruction pipeline

Data hazard

I_{i+k} must not be permitted to fetch an operand that is yet to be stored into by instruction I_i.

ADD R1, R2, R3
MULT R4, R1, R5
OR R6, R1, R7

Instructions MULT and OR, after ADD, use the result generated by ADD.



IF
ID
OF
EX
Clock 1 2 3 4 5 6
ADD R1, R2, R3 IFADD IDADD OFADD EXADD
MULT R4, R1, R5 IFMULT IDMULT OFMULT EXMULT
OR R6, R1, R7 IFOR IDOR OFOR EXOR
R1 is read
R1 is ready

Figure 12: Data hazard in 4-stage instruction pipe

Control hazard

In order to fetch ’next’ instruction, pipeline system must know which instruction is required next.

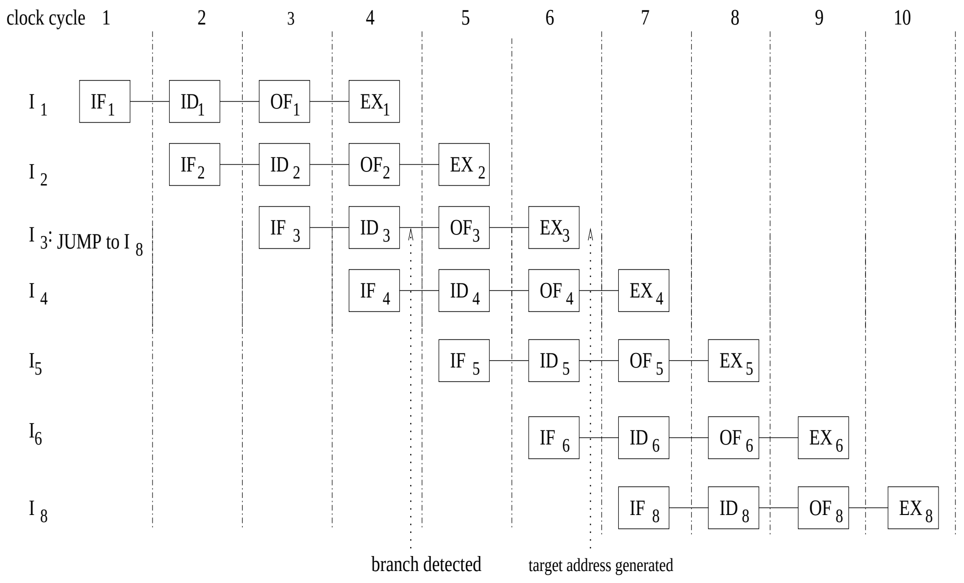


Figure 13: Control hazard in 4-stage instruction pipeline

If present instruction is a conditional branch, next instruction may not be known until current one is processed.

Pipeline may be slowed down by a branch instruction because we do not know which branch to follow.

Interrupts

Interrupts (Figure 14) insert 'extra' instructions into execution of instruction stream.

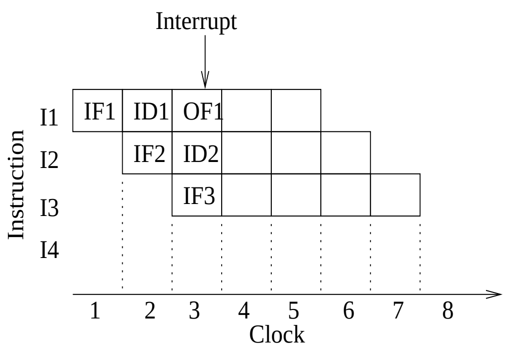


Figure 14: Interrupt in pipelining

Interrupt effect is taken when one instruction has completed.

But in pipelining, next instruction enters before current one has completed.

Solution 1: Simplest solution is to wait until all instructions in pipeline complete -that is, flush pipeline from starting point, before acknowledging interrupt.

For frequent interrupts, it makes pipeline inefficient; interrupt handling is delayed.

Solution 2: Select a point in the pipeline (say, EX stage). Instructions which have passed this point are only allowed to complete and then interrupt is acknowledged.