Pipelining: Basic Concepts and Approaches

RICHA BAIJAL[1]
[1]Student,M.Tech,Computer Science And Engineering
Career Point University,Alaniya,Jhalawar Road,Kota-325003 (Rajasthan)

Abstract-This paper is concerned with the pipelining principles while designing a processor.The basics of instruction pipeline are discussed and an approach to minimize a pipeline stall is explained with the help of example.The main idea is to understand the working of a pipeline in a processor.Various hazards that cause pipeline degradation are explained and solutions to minimize them are discussed.

Index Terms— Data dependency, Hazards in pipeline, Instruction dependency, parallelism, Pipelining, Processor, Stall.

1 INTRODUCTION

TO understand what pipelining is,let us consider the assembly line manufacturing of a car.If you have ever gone to a machine work shop ; you might have seen the different assemblies developed for developing its chasis,adding a part to its body,wheels alignment and painting the parts.All this together bring up your favourite car,every assembly line adding to the perfection and doing their best.Now,if these units wait for resources or we can say that if the second stage is dependent on the first one,then more time will be consumed in building the car.So,we divide the tasks in such a way that their dependency is relaxed.In pipelining a task on computer,we either divide it in such a way that one task is independent of the other so that hardware units can switch the tasks between them using a clock or we add a hardware circuitry to speed up the tasks. The second approach adds up the cost while the first one results in the efficient utilization of available resources,to what we call Pipelining.[1]

2 PARALLELISM AND PIPELINING :

When we implement pipelining in processing a task on computer,we create objects or stages.These stages are independently working and accomplish a certain task. After a certain period of time ,the output from a certain stage is given as input to the next stageThis is done by synchronizing a clock or simply specifying a time period for certain task.Then next stage works on the combined input for a period of time and produces the desired results.Meanwhile,the other stages also work on their inputs. If i have to summarize the concept of pipelining,it is simply achieving maximum throughput from the computer by utilizing resources in parallel manner and avoiding resource deadlocks.

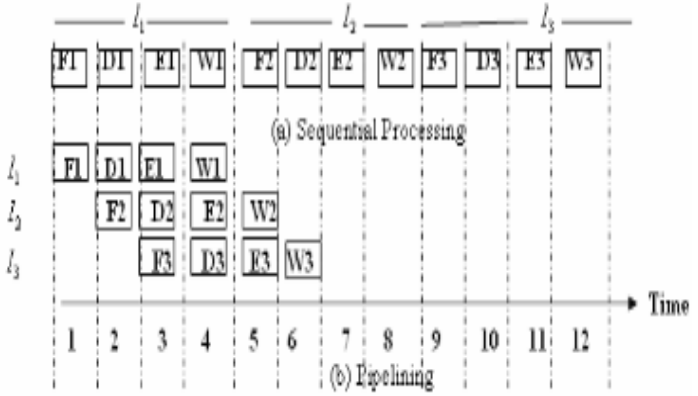
A very good example from our daily lives is :

I want to paint my house. It has 4 rooms.one approach is to employ a painter with his paint bucket and wait until he paints one room. The second approach is to employ another worker with him who in parallel works in the other room and does the paint. Still,2 rooms are idle. These rooms that I want to paint constitute my hardware.The painter and his skills are the objects and the way i am using them refers to the stages.Now,it is quite possible i limit my resources,i.e. I just have two buckets of paint at a time;therefore,i have to wait until these two stages give me an output.Although,these are independent tasks,but what i am limiting is the resources.

I hope having this comcept in mind,now the reader knows what he has to do with his computer to achieve a maximum utilization.[2]

3 DIFFERENCE BETWEEN SEQUENTIAL PROCESSING AND PIPELINING :

Below is an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.[3]



Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in

sequence.

• Each subtask is performed by a given functional unit.

• The units are connected in a serial fashion and all of them operate simultaneously.

• The use of Pipelining improves the performance as compared to the traditional sequential execution of tasks.

In the given fig.,sequential processing is divided into three instructions I_1,I_2 and I_3 and these instruction sets are being processed in serial manner in time slots of 1-4,5-8 and 9-12 respectively.

While in parallel processing,instruction sets I_1,I_2 and I_3 are arranged in parallel manner and processing is done in serial manner. Tasks are divided such that no 2 decode instructions or no 2 write instructions are performed in the same time slot.In parallel processing,the instruction queue has been processed in 6 time slots. As such the time of processing instructions is reduced and efficiency of processor is increased.

4 PERFORMANCE OF A PIPELINE :

In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space time chart (called the Gantt's Chart) is used. The chart shows the succession of the sub-tasks in the pipe with respect to time.[4]

5 PERFORMANCE ANALYSIS OF A PIPELINE :

In the following analysis, we provide three performance measures for the goodness of a pipeline. These are the Speed-up S(n), Throughput U(n), and Efficiency E(n). It should be noted that in this analysis we assume that the unit time T = t units In the following analysis, we provide three performance measures for the goodness of a pipeline.

1.Speed-up S(n): Consider the execution of m tasks (instructions) using n-stages (units) pipeline. As can be seen, n + m-1 time units are required to complete *m* tasks.

Speed-up S(n)= $\frac{Time\ using\ Sequential\ Processing}{Time\ using\ Prallel\ Processing}$

$= \frac{(m \times n \times t)}{(n+m-1)\times t} = \frac{m \times n}{m+n-1}$

$\lim_{m\to\infty} S(n)$= n (i.e., n-fold increase in speed is theoretically possible)

For the given example : m=10,n=4,t=13

S(n)= $\frac{10\times4}{10+4-1} = \frac{40}{13}$ = 3.07 and maximum speed up that can be achieved is 4 times because n=4.

2. Throughput U(n)=no.of tasks executed per unit time $=\frac{m}{(n+m-1)t}$

$\lim_{m\to\infty} U(n)$=1 assuming that t=1 unit time.

t=1 i.e. U(n)= $\frac{m}{n+m-1} = \frac{10}{4+10-1} = \frac{10}{13}$ =0.8

It means that in our example,the processor utilization using pipelined tasks is 80 %.

3. Efficiency E(n)=Ratio of the actual speed-up to the maximum speed-up

i.e. $= \frac{Speed-Up}{n} = \frac{m}{n+m-1}$

$\lim_{m\to\infty} E(n)$=1

Which is same as throughput.

Note : Practically, m and n are very large.

6 INSTRUCTION PIPELINE :

We discussed the performance of a pipeline in the above section. But,practically things are different.There is a possibility that a instruction might delay in its execution in order to resolve a pipeline hazard. This situation is referred to as a stall or a bubble in pipeline execution. Let us first draw a pipeline with a bubble or stall :[5]

Let us understand this pipeline now :

Here,Instruction I_2 is incurring a *cache miss* which requires 3 time units during instruction fetch.

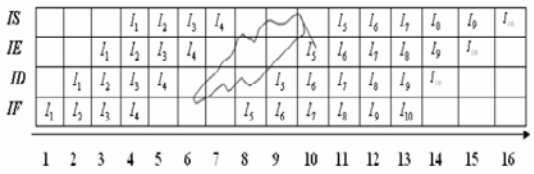
		1	2	3	4	5	6	7
ADD	**R1**,R2,R3	IF	ID	EX	MEM	**WB**		
SUB	R4,R5,**R1**		IF	ID_{SUB}	EX	MEM	WB	
AND	R6,**R1**,R7			IF	ID_{AND}	EX	MEM	WB

A *cache miss* stalls all the instructions on the peipeline both before and after the instruction causing the miss. This delay of three units time has worsened the pipeline performance.

6.1 Understanding pipeline "stall" :

1 Due to instruction dependency : An instruction is being executed in a pipeline. The result of its execution is an input to the next instruction. Therefore ,the latter instruction cannot start its execution until it gets the input from the previous instruction. Such kind of dependency is called instruction dependency.[1,5]

Pipeline "Stall" due to Instruction Dependency:



2 Due to data dependency : Data dependency in pipeline occurs when a source operand of instruction I_i depends on the result of executing a preceding instruction I_j ,i>j .

The hazards discussed here involve registers.

Types Of Data Dependency with Example :[5]

		1	2	3	4	5	6	7
ADD	**R1**, R2, R3	IF	ID	**EX_{add}**	**MEM_{add}**	WB		
SUB	R4, R5, **R1**		IF	ID	**EX_{sub}**	MEM	WB	
AND	R6, **R1**, R7			IF	ID	**EX_{and}**	MEM	WB

(i)RAW (Read After Write): j tries to read a source before i writes it,so j incorrectly gets the old value.
This is the most common type of hazard and can be overcome using *forwarding*.[7]
Example:
The key insight in forwarding is that the result is not really needed by SUB until after the ADD actually produces it.The only problem is to make it available for SUB when it needs it. If the result can be moved from where the ADD produces it (EX/MEM register), to where the SUB needs it (ALU input latch), then the need for a stall can be avoided.
Using this observation , forwarding works as follows:

-The ALU result from the EX/MEM register is always fed back to the ALU input latches.
-If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Forwarding of results to the ALU requires the additional of three extra inputs on each ALU multiplexer and the addtion of three paths to the new inputs.

The paths correspond to a forwarding of:
(a) the ALU output at the end of EX,
(b) the ALU output at the end of MEM, and
(c) the memory output at the end of MEM.

Without forwarding our example will execute correctly with stalls:

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**stall**	**stall**	**ID_{sub}**	EX	MEM	WB	
AND	R6, **R1**, R7			**stall**	**stall**	IF	**ID_{and}**	EX	MEM	WB

As our example shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started three cycles earlier. Forwarding can be arranged from MEM/WB latch to ALU input also. Using those forwarding paths the code sequence can be executed without stalls:

The first forwarding is for value of **R1** from **EX_{add}** to **EX_{sub}** . The second forwarding is also for value of **R1** from **MEM_{add}** to **EX_{and}**.
This code now can be executed without stalls.

Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. [5],[6]

(ii) 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 is 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. The DLX integer pipeline writes a register only in WB and avoids this class of hazards.

WAW hazards would be possible if we made the following two changes to the DLX pipeline:

▪move write back for an ALU operation into the MEM stage, since the data value is available by then.
▪suppose that the data memory access took two pipe stages.

Here is a sequence of two instructions showing the execution in this revised pipeline, highlighting the pipe stage that writes the result:

LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	**WB**
ADD R1, R2, R3		IF	ID	EX	**WB**	

Unless this hazard is avoided, execution of this sequence on this revised pipeline will leave the result of the first write (the LW) in R1, rather than the result of the ADD.

Allowing writes in different pipe stages introduces other problems, since two instructions can try to write during the same clock cycle. The DLX FP pipeline , which has both writes in different stages and different pipeline lengths, will deal with both write conflicts and WAW hazards in detail.

(iii) WAR (write after read) - j tries to write a destination before it is read by **i** , so **i** incorrectly gets the new value.

This can not happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.

Because of the natural structure of a pipeline, which typically reads values before it writes results, such hazards are rare. Pipelines for complex instruction sets that support autoincrement addressing and require operands to be read late in the pipeline could create a WAR hazards.

If we modified the DLX pipeline as in the above example and also read some operands late, such as the source value for a store instruction, a WAR hazard could occur. Here is the pipeline timing for such a potential hazard, highlighting the stage where the conflict occurs:

SW R1, 0(R2)	IF	ID	EX	MEM1	**MEM2**	WB
ADD R2, R3, R4		IF	ID	EX	**WB**	

If the SW reads R2 during the second half of its MEM2 stage and the Add writes R2 during the first half of its WB stage, the SW will incorrectly read and store the value produced by the ADD.

(iv)RAR (read after read) - this case is not considered as a hazard .

6.2 When are the Stalls Required *?*

Unfortunately,not all potential hazards can be handled by *forwarding*.

Consider the following sequence of instructions: [5,6]		1	2	3	4	5	6	7	8
LW	**R1**, 0(R1)	IF	ID	EX	**MEM**	**WB**			
SUB	R4, **R1**, R5		IF	ID	$\mathbf{EX_{sub}}$	MEM	WB		
AND	R6, **R1** R7			IF	ID	$\mathbf{EX_{and}}$	MEM	WB	
OR	R8, **R1**, R9				IF	**ID**	EX	MEM	WB

The **LW** instruction **does not** have the data until the end of clock cycle 4 (**MEM**) , while the **SUB** instruction needs to have the data by the beginning of that clock cycle ($\mathbf{EX_{sub}}$).

For **AND** instruction we can forward the result immediately to the ALU ($\mathbf{EX_{and}}$) from the MEM/WB register(**MEM**).

OR instruction has no problem, since it receives the value through the register file (**ID**). In clock cycle no. 5, the WB of the LW instruction occurs "early" in first half of the cycle and the register read of the OR instruction occurs "late" in the second half of the cycle.

For **SUB** instruction, the forwarded result would arrive too late - at the end of a clock cycle, when needed at the beginning.

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called ***a pipeline interlock***, to preserve the correct execution pattern. In general, a pipeline interlock ***detects*** a hazard and ***stalls*** the pipeline until the hazard is cleared.

The pipeline with a stall and the legal forwarding is:

		1	2	3	4	5	6	7	8	9
LW	**R1**, 0(R1)	IF	ID	EX	**MEM**	WB				
SUB	R4, **R1**, R5		IF	ID	***stall***	$\mathbf{EX_{sub}}$	MEM	WB		
AND	R6, **R1** R7			IF	***stall***	ID	EX	MEM	WB	
OR	R8, **R1**, R9				***stall***	IF	ID	EX	MEM	WB

The only necessary forwarding is done for R1 from **MEM** to $\mathbf{EX_{sub}}$.
Notice that there is no need to forward R1 for **AND** instruction because now it is getting the value through the register file in ID (as **OR** above).

There are techniques to reduce number of stalls even in this case, which we consider next.

6.3 Pipeline Scheduling :

Generate DLX code that avoids pipeline stalls for the following sequence of statements:[5]

```
a = b + c ;
d = a - f ;
e = g - h ;
```

Assume that all variables are 32-bit integers.Wherever necessary,explicitly explain the actions that are needed to avoid pipeline stalls in your scheduled code.

Solution:
1.The DLX assembly code for the given sequence of statements is:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LW Rb, b	IF	ID	EX	**M**	WB													
LW Rc, c		IF	ID	EX	**M**	WB												
Add Ra,Rb, Rc			IF	ID	stall	**EX**	M	WB										
SW Ra, a				IF	stall	ID	EX	**M**	WB									
LW Rf, f					stall	IF	ID	EX	**M**	WB								
Sub Rd, Ra, Rf							IF	ID	stall	**EX**	M	WB						
SW Rd, d								IF	stall	ID	EX	**M**	WB					
LW Rg, g									stall	IF	ID	EX	**M**	WB				
LW Rh, h											IF	ID	EX	**M**	WB			
Sub Re, Rg, Rh												IF	ID	stall	**EX**	M	WB	
SW Re, e													IF	stall	ID	EX	**M**	WB

Running this code segment will need some forwarding. But instructions LW and ALU(Add or Sub), when put in sequence, are generating hazards for the pipeline that can not be resolved by forwarding. So the pipeline will stall. Observe that in time steps 4, 5, and 6, there are two forwards from the Data memory unit to the ALU in the EX stage of the Add instruction. So also the case in time steps 13, 14, and 15. The hardware to implement this forwarding will need two Load Memory Data registers to store the output of data memory. Note that for the SW instructions, the register value is needed at the input of Data memory. The better solution with compiler assist is given below.

Rather then just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid these stalls by rearranging the code sequence to eliminate the hazards.[7]

2.Suggested version is (the problem has actually more than one solution) :

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**Explanation**
LW Rb, b	IF	ID	EX	M	**WB**											
LW Rc, c		IF	ID	EX	**M**	WB										
LW Rf, f			IF	ID	EX	M	**WB**									
Add Ra, Rb, Rc				IF	**ID**	**EX**	**M**	WB								**Rb** read in second half of ID; **Rc** forwarded
SW Ra, a					IF	ID	EX	**M**	WB							**Ra** forwarded
Sub Rd, Ra, Rf						IF	**ID**	**EX**	M	**WB**						**Rf** read in second half of ID;
LW Rg, g							IF	ID	EX	M	**WB**					**Ra** forwarded
LW Rh, h								IF	ID	EX	**M**	WB				
SW Rd, d									IF	**ID**	EX	M	WB			**Rd** read in second half of ID;
Sub Re, Rg, Rh										IF	**ID**	**EX**	M	WB		**Rg** read in second half of ID; **Rh** forwarded
SW Re, e											IF	ID	EX	**M**	WB	**Re** forwarded

The same color is used to outline the source and destination of forwarding.
The blue color is used to indicate the technique to perform the register file reads in the second half of a cycle, and the writes in the first half.

Note: Notice that the use of different registers for the first, second and third statements was critical for this schedule to be legal! In general, pipeline scheduling can increase the register count required.

7 CONCLUSION :

In this paper, the basic principles involved in designing pipeline architectures were considered.Our coverage started with a discussion on a number of metrics that can be used to assess the goodness of a pipeline.We then moved to present a general discussion on the main problems that need to be considered in designing a pipelined architecture.

– In particular two main problems are considered :- Instruction and data dependency.

References :

[1] Book : Computer Organization by Hamacher.
[2] Parallelism and pipelining by David G. Messerschmitt,University Of California.
[3] Pipelining Design Techniques by Mostafa Abd-El-Barr & Hesham El-Rewini
[4] Project Management Graphics by Edward Tufte: B.S. and M.S. in statistics, Stanford University, 1964. Ph.D. in political science, Yale University, 1968
[5] Website : http://www.cs.iastate.edu/
[6] Website : https://www.cs.uaf.edu/2011/fall/cs441/lecture/09_20_pipelining.html ;lecture by Dr. Lawlor
[7] Website : https://courses.engr.illinois.edu/cs232/sp2010/lectures/L13.pdf