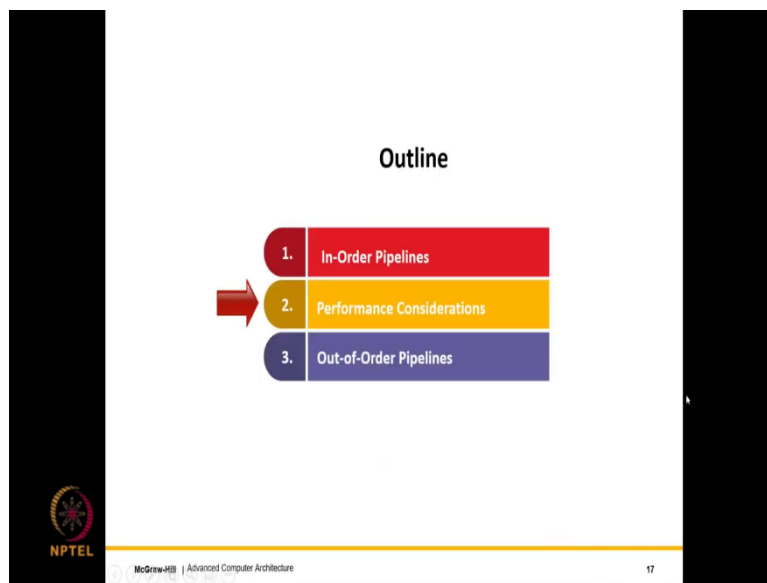


**Advanced Computer Architecture**  
**Prof. Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Lecture - 03**  
**Out – of – Order Pipelines**  
**Part - II**

(Refer Slide Time: 00:52)



Now, let us discuss performance considerations of in-order and of in-order pipelines and motivate the discussion towards Out-of-Order Pipelines.

(Refer Slide Time: 01:06)

**Performance Equation - I**

Is Computer A faster than Computer B

**Wrong Answers:**

- More is the clock speed, faster is the computer
- More is the RAM, faster is the computer

What does it mean for computer A to be faster than computer B

Short Answer: **NOTHING**

Performance is always with respect to a program. You can say that a certain program runs faster on computer A as compared to computer B.

NPTEL

McGraw-Hill | Advanced Computer Architecture

18

So, let us come to the basic question, if I ask you is computer A faster than computer B, what are you going to say? So, many people gave wrong answers. They say higher frequency that is one wrong answer; they say more memory, another wrong answer; any combination thereof, another wrong answer. So, people given whom this question was asked the answers can be different flavors are wrong, but the answers are seldom correct.

So, what does it mean for computer A to be faster than computer B or processor A to be faster than processor B? Well short answer – nothing. So, performance is always with respect to a given program. So, what you do? Is that so, I am kind of you know referring to the processor as a computer because there are other aspects as well like the memory, the hard disk and so on.

But, even if we can isolate the processor well and good or if you want to consider everything in unison, well, that is also fine. So, in any case, let's take the two substrates A and B, where A and B can be individual processors or entire machines with memory and hard disk does not matter.

So, what I would do is that I would run the same program on A and I would run the same program on B and I will simply measure the wall clock time which is the time with a

standard clock; with the wall clock and on whichever system it runs faster I say the that system is faster with respect to that program.

Now, let us say for example, it is a chess program. Then if let us say computer A is faster than computer B, I say that well for chess A is faster than B. So, now, let us say if I want to remove memory and hard disk, well you can always ask me, how will I run a program without involving the memory and disk?

Well, if my program is small enough we will define what it means to be small, but let's say that it does not use memory and disk that much then and if it is predominantly limited to the CPU and its resources we can say that CPU A is faster than CPU B. So, we can see we will see in a later chapters how do we explicitly ensure this, but essentially, whatever the system constitutes we just run the same program at both places sit with a stopwatch wherever it runs faster, we say that look that system is faster.

So, of course, you can say that, but then you can always counter argue with me and you can say that I say a supercomputer is faster than a mobile phone, well, how is that? Well, that is true. If I can say with reasonable certainty that give me any program on computer A it will run faster than computer B. You write any program that you want and I can show you that A will always be faster than B, then I can make a general statement that A is faster than B.

In general, I cannot make that because especially among alternatives where it is not very clear like you know I give you one laptop with a 2 gigahertz processor, another laptop with a 3 gigahertz processor. What is again 2 gigahertzes and 3 gigahertzes? It is a clock speed the clock speed that we have been discussing, it is just the clock speed. So, in that case it is not very clear.

The distinction between a supercomputer and a mobile phone is very clear. One is very large and very expensive and very powerful and the other is small and cheap. So, be the difference between these two is rather stark and it is kind of clear, but if let us say I give you faster laptop and laptop of one kind and laptop of another kind, in that case, it is not that clear. So, then you say performance is with respect to one program it can also be a set of programs.

So, what I can do is that I can take 5 programs. So, let's say run them. so, let us say I can get  $t_1$  A. So, where  $t_1$  A is the time that program one takes on system A. So, what I can actually do is that I can sort of run them at both places and compute these times. How do I compare the times? Well, one is that I can take the arithmetic mean, but that would make little sense because let us say that one programs baseline running time is 1 second and then for the other it is 1 hour, then you can skew the results.

So, what people normally do is they take a geometric mean of the ratios typically whenever you consider a ratio you take a geometric mean in computer architecture. So, what you do is? you do this. You compute a ratio and then you take the geometric mean of the ratios and then you can say that look for this suite of programs suite is a set of programs computer A is 10 % or 15 or 20 % faster than computer B.

(Refer Slide Time: 06:59)

**Performance Equation - II**

$$\begin{aligned} \frac{\# \text{Programs}}{\# \text{Seconds}} &= \frac{\# \text{Programs}}{\# \text{insts}} \cdot \frac{\# \text{insts}}{\# \text{cycles}} \cdot \frac{\# \text{cycles}}{\# \text{seconds}} \\ &= \frac{\text{IPC} \cdot \text{freq}}{\# \text{insts/program}} \quad \left( \frac{5^{100}}{500} \right) \\ &= \frac{\text{IPC} \cdot \text{freq}}{\# \text{insts}} \quad (\text{assume just 1 program}) \end{aligned}$$

*Performance  $\propto \frac{\text{IPC} \cdot \text{freq}}{\# \text{insts}}$*

- IPC is the number of instructions per cycle
- Let us loosely refer to the reciprocal of the time per program as the **performance**

*IPC =  $\frac{1}{\text{CPI}}$*

*IPC  $\propto \frac{1}{\# \text{insts}}$*

NPTEL

McGraw-Hill | Advanced Computer Architecture

19

So, now let us take a look at the performance equation which is arguably one of the most popular and most famous equations in computer architecture. So, we have essentially defined performance to be some quantity as the time that a program takes. So, let us do a little bit of dimensional analysis. So, if I have number of programs in a number of seconds. So, then I can add a few extra variables.

So, I can have of course, programs over here instructions; instructions they will cancel out cycles and cycles, number of cycles number of cycles, hash stands for number of and number of seconds. So, what you will see is that this instruct this essentially comes down if I consider a single program as a number of instructions per program. Roughly, in an on an average number of instructions per program which for a single program is actually the number of instructions.

And, note these are not static instructions these are actually dynamic instructions which means that, so, consider a loop in the body of the loop if there are 5 instructions then there are 5 static instructions, but if the loop runs hundred times there are 500 dynamic instructions. So, a number of instructions here actually points to dynamic instructions.

The number of instructions per cycle is known as the IPC. So, this is an inherent property of the design, the architecture of the processor is very important, we will take a look and the number of cycles per second. well, that we know that is the sequence so the number of clock cycles per second is a frequency.

So, assuming just a single program, which we will assume for the better half of this chapter. The performance I will say is a quantity that is so, it depends on the source. So, many sources would define this quantitative performance we will not do that.

And so, throughout the book what we will do is that we will say that performance, but in some cases for simplicity we will consider this to be a proxy of performance, but in general we will say performance is IPC times frequency divided by number of instructions. This is what we will say in general.

And, we assume the constant of proportionality is kind of the same for all programs, all processes that we are comparing. So, the IPC so, you need to remember these terms we will be using these terms throughout this chapter throughout this book and that too very liberally. The IPC is the number of instructions per cycle and  $IPC = 1 / CPI$ . Well, CPI is the cycles per instruction.

So, IPC and CPI initially a lot of students confuse. So, it is important for you to know that an IPC I comes first, I stands for instruction. IPC is instructions per cycle, CPI is cycles per

So, the constant of proportionality will cancel, but the most important is the interplay of the IPC, the frequency and a number of instructions that is the most important thing that we need to consider and their relative interplay.

## So, what does performance depend on ...

- #instructions in the program
  - Depends on the compiler
- Frequency
  - Depends on the transistor technology and the architecture
  - If we have more pipeline stages, then the time to traverse each stage reduces roughly proportionally
  - Given that each stage needs to be processed in one clock cycle, smaller the stage, higher the frequency
  - To increase the frequency, we simply need to increase the number of pipeline stages
- IPC
  - Depends on the architecture and the compiler
  - A large part of this book is devoted to this aspect.

Load  
store  
use      [load  
add  
store  
use]

$t(\text{algorithm}) = n \times t_{\text{work}}$

$\left[ \frac{t}{k} + \Delta \right]$

$f_{max} = \frac{1}{f_{min} \times k}$

$t_{clk} = \frac{t}{k} + \Delta$

[allows us to increase the frequency]      for forwarding

McGraw-Hill | Advanced Computer Architecture20

So, the logic requires a lot of instructions or it has a lot of code that does not do anything that is a bad idea. It also depends on the compiler. So, often we will have a lot of code that is not really very optimal and that has a lot of instructions which are known as dead code do not really do much or do not really do anything.

So, most compilers are supposed to removed. So, that is where a smart compiler can definitely reduce the number of instructions. Let us look at frequency and IPC. Wherever you

see this burger sign well, it is good one is that go and eat a burger. So, it will give you some energy to understand the rest; the other is it is not just real food, but it is intellectual food, it is intellectual food for thought. So, that should give you an additional impetus to think.

So, frequency is a rather complicated thing ok. It is kind of crucial to our entire discussion. So, it depends on the transistor technology. well of course, it does because the faster transistors higher we can crank up the frequency. But, it also depends on the architecture. This is the most interesting aspect of the connection between the frequency and the architecture. Architecture in this case is the pipeline.

So, given a fixed transistor technology let's say the total amount of work that we need to do which we can quantify in terms of time. Let this be 't' for a given instruction that the worst case time be 't', which is essentially the total time that it takes to process an instruction. So, do not care about pipelining it is just end to end. So, this is known as the algorithmic work which is the total amount of work that needs to be done by the CPU; so, the maximum of this for processing an instruction.

So, what we do is? if it has a 'k' stage pipeline then essentially we do 't' by 'k' units of work, measure in terms of time of course, per stage. In addition, the stage will have a large delay which we can say  $\Delta$  because every stage has some logic and then it has a pipeline large.

So, we can say this  $\frac{t}{k} + \Delta$ . So, if we assume that the pipeline stages are balanced which means that all the stages have roughly the same time balanced actually means they have exactly the same time, then in this case, each stage takes  $\frac{t}{k} + \Delta$  units of time.

So, the minimum clock period that we can so, what did we say we said that look what happens is that, at every negative edge data comes from this side to this side via a pipeline register. For the entire duration of the cycle it keeps getting processed then when the clock cycle ends at the next negative edge it enters the second latch. So, this is the clock period.

So, the clock period the minimum clock period =  $\frac{t}{k} + \Delta$ , which means that the maximum frequency  $f_{\max} = \frac{1}{\frac{t}{k} + \Delta}$ . So, this is a very very important piece of understanding that as we

increase the number of pipeline stages, what happens? As we increase the number of pipeline stages so, let us say that  $k$  tends to infinity.

In this case, this factor will become 0 and the clock period will be equal to the large delay, which clearly says look beyond a certain point there is no advantage whatsoever in increasing the number of pipeline states. Do not do it because it will be limited by the large delay. So, you cannot increase the frequency further. However, that is a lot of stages when you are talking of 3, 4, 5, 6, 7 stages, there is a definite advantage in increasing the number of pipeline stages. Hence keep doing it.

As you see the maximum frequency at which you can run the processor will keep increasing. And if let's say  $\frac{t}{k}$  is much much greater than  $\Delta$ , then what you essentially say is  $f_{\max} \propto k$ . This is easy to work out right if  $\Delta$  is much lower than this then  $f_{\max}$  is proposed is approximately proportional to the number of stages which means as I increase the number of stages I can keep on cranking up the clock frequency. That is very important.

If I do not do that I have no advantage of pipeline. The only advantage of pipelining as we will see is that. So, what happens if I increase the number of stages each stage becomes smaller. The smaller a stage lower is the clock period smaller is the clock period, hence higher is the frequency that can be supported.

So, to increase the frequency what do we need to do? Well, we simply increase the number of pipeline stages the converse is also true. If we can increase the number of pipeline stages we can also give ourselves a higher frequency. So, note that whether we actually crank up the frequency or not, that depends on other factors, but clearly increasing the number of pipeline stages allows us. So, the key word is allowing us right makes it possible to increase the frequency.

Whether we do it or not we will see, but at least we can do it and given the fact that the performance is proportional to the frequency, what you can clearly see is that look as a pipeline mode I can increase the frequency and my frequency will increase the performance because the performance equation as we just saw is IPC times frequency divided by number of instructions.



Number of instructions beyond a certain point we have no control because the compiler technologies have become reasonably mature, they are not getting better in this regard at least. So, pipelining if we can do more of it we can afford ourselves a higher frequency which is good and the last is IPC. So, IPC is again a complicated beast which is dependent on the architecture and the compiler. Well, why the architecture? Well, one we saw a great example of increasing IPC. How did we see that? Well, we introduced forwarding.

So, forwarding effectively reduced the number of stall cycles which reduced the wasted work, which increased the number of useful instructions we can process per cycle. So, that increase the IPC. So, how can the compiler help increase the IPC? Well, it can rearrange the code in such a way that we lose as few cycles as possible to data hazards and control hazards.

So, in that case, lower the number of stall cycles, lower the CPI and higher the IPC. So, large part of this book is devoted to this aspect of how to increase the IPC and as we have seen the IPC and frequency are also rather connected. So, we will explore these aspects in the next few slides – how they are connected.

But, at least what do we know up till now? We know that look the number of stages in a pipeline is definitely connected with the frequency, how? By this relationship over here that when the number of stages when  $\frac{t}{k}$  is much larger than the large  $\Delta$ , the maximum possible frequency is roughly proportional to the number of stages.

So, if I go from one stage which is a single cycle processor through a 5-stage pipelined processor, I can increase my frequency by 5 times roughly. And, IPC as we said as we have more of these tricks like forwarding as one such trick we can increase IPC similarly consider a load use hazard. Say load use hazard we have a load there we need to have a stall and then we have the use. So, the stall is a wasted cycle which reduces IPC.

So, what I can what a smart compiler can do is? it can have a load, it can have another instruction let us say an add instruction which is of course, not dependent on either of them. Even if it is even if it is not dependent on the load ok. So, this can only be stored for it to get the value.

So, let us assume it is an instruction that is independent of the load. It is an independent instruction I will put an I over here to signal independence. So, then well I can have the use over here, there are no stall cycles and my IPC is remains high. So, a host of tricks give us a high IPC.

(Refer Slide Time: 21:25)

**How to improve performance?**

There are 3 factors: *stalls, compiler, forwarding*  
 • IPC, #instructions, and frequency *transistor tech. # pipeline stages*  
 • #instructions is dependent on the compiler → not on the architecture

Let us look at IPC and frequency

IPC

• What is the IPC of an in-order pipeline?

**1 if there are no stalls, otherwise < 1**

*Observer*  
 $IPC = \frac{2}{4} = 0.5$   
 $< 1$

**Methods to increase IPC**

- Forwarding
- Having more not-taken branches in the code
- Faster instruction and data memories

NPTEL  
 McGraw-Hill | Advanced Computer Architecture  
 21

So, what have we seen? We have seen look there are 3 factors – IPC, instructions and frequency. Instructions are only dependent on the compiler and the programmer. The frequency on the transistor technology, faster transistors, faster frequency let me call it transistor tech, also the number of pipeline stages.

And, what is IPC dependent on? Well, IPC is dependent on of course, the rate of stalls, it is dependent on the rate of stalls, how many stalls? it is dependent on compiler technology of how the code is rearranged to avoid stalls and of course, tricks like forwarding and so on that also increases IPC.

So, what is the IPC of an in-order pipeline that we saw? Well, if it is a perfect pipeline without any stalls. so, consider a 5 stage in-order pipeline that we saw and we can see instructions coming out there is an observer sitting over here.

So, the observer will pretty much see one instruction coming out every cycle, why not? After all there are no stalls. So, this will happen and if this does happen, then in the ideal case the

IPC will be 1. However, we will have stalls. So, the observer might see a valid instruction after that it might see a stall cycle maybe one more stall cycle, then a valid instruction. so, let us look at these four.

So, we have 4 slots, we have 4 cycles. Out of them there are only 2 valid instructions, the rest 2 are stalls. So, the  $IPC = \frac{2}{4} = 0.5$  and why did this happen? Because we had stalls so, the IPC in a pipeline processor is always less than 1.

So, where are we now? The IPC of a pipeline processor we know for sure because of stalls is  $< 1$ . Of course, the IPC in a single cycle processor was higher it was 1, but we will come to that later. And, the methods to increase IPC are well of course, forwarding rate. Having more not taken branches in the code – well, this reduces the number of control hazards grid.

Sometimes, your instruction and data memories can be slow, they can take multiple cycles to return the value. Those multiple cycles will be counted as stall cycles. So, having faster instruction and data memory is also increases the IPC. So, that is the third mechanism.

(Refer Slide Time: 24:36)

### What about frequency?

What is frequency dependent on ...

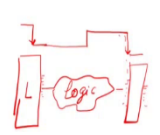
**Frequency = 1 / clock period**


Clock Period:

- 1 pipeline stage is expected to take 1 clock cycle
- Clock period = maximum latency of the pipeline stages

How to reduce the clock period?

- Make each stage of the pipeline **smaller** by increasing the number of pipeline stages
- Use faster transistors





McGraw-Hill | Advanced Computer Architecture

22

So, given that now let us take a slightly deeper look. So, as I said frequency is 1 by clock period and each pipeline stage is expected to take 1 clock cycle, that is the way it is architected. So, every pipeline stages the way it is architected is that you have a pipeline

latch, you have a bunch of you have a large piece of logic and we are added multiplied divided does not matter.

So, at the negative edge of the clock cycle the data is visible to the logic. The logic you do whatever computation you want. And by the end of the current cycle so, maybe like this. Again, the data is here such that at the beginning of the next cycle the data is visible here. Reducing the clock period well use faster transistors or use more pipeline stages.

(Refer Slide Time: 25:39)

The slide is titled "Limits to Increasing Frequency". It contains the text: "Assume that we have the fastest possible transistors" and "Can we increase the frequency to 100 GHz?". A red bracket under "100 GHz" is followed by a handwritten "no". To the right is a green starburst with "NO!". Below the text is a red oval containing the word "Reasons". The slide features the NPTEL logo in the bottom left, the text "McGraw-Hill | Advanced Computer Architecture" in the bottom center, and the number "23" in the bottom right.

Are there limits to increasing frequency? Well, yes there are, even if I have a large pipeline nevertheless there are. Can we increase the frequency to 100 gigahertz? Well, the answer is no. Why is the answer no?

(Refer Slide Time: 25:57)

**Limits to increasing frequency - II**

What does it mean to have a very high frequency?  
Before answering, keep these facts in mind:

- 1 **Thumb Rule**  $P \propto f^3$   
 $P \rightarrow$  power  
 $f \rightarrow$  frequency
- 2 **Thermodynamics**  $\Delta T \propto P$   
 $T \rightarrow$  Temperature
- 3 We need to increase the number of pipeline stages  $\rightarrow$  more hazards, more forwarding paths

NPTEL

McGraw-Hill | Advanced Computer Architecture 24

The reasons are keep these facts in mind. So, we will see this in chapter number 11 on the chapter on power and temperature, the power roughly in modern processors is proportional to the cube of the frequency. So,  $p \propto f^3$ . So, if I double the frequency my power will increase 8 times.

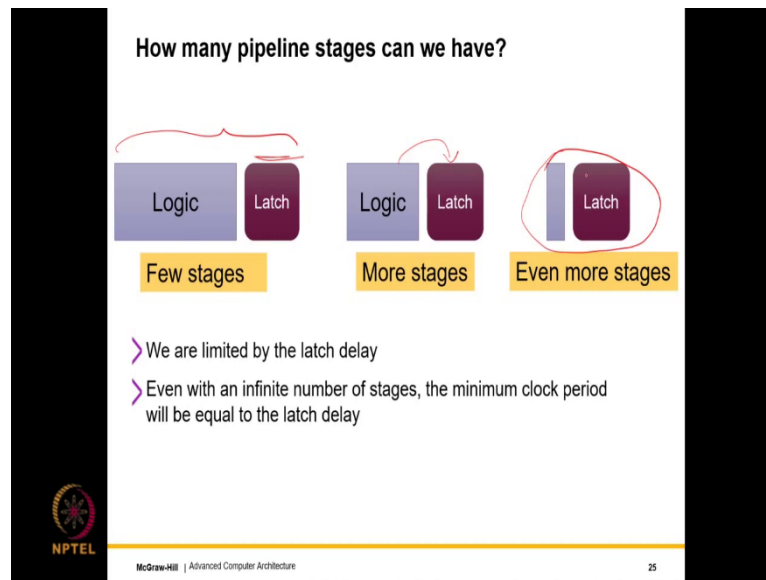
So, as it is our laptops and desktops consume a lot of power a typical laptop might be consuming somewhere between 20 and 50 watts, maybe 100 a few and a desktop would happily consume around 250 watts. So, if I make it 8 times there will be 2 kilowatts that is roughly the amount of power that air conditioner uses that will be too much.

Another result from thermodynamics the change in temperature  $\Delta T$  is proportional to the applied power roughly which means that the increase in temperature as it is chips are very hot. So, while you are playing a game the dye temperature can be as high as the 100 degrees C. So, if this increases further essentially the whole chip will melt.

Additionally, to support a large frequency we need a very large number of pipeline stages that will create a large number of hazards, extremely complicated and expensive forwarding logic with a very large number of forwarding multiplexers and a large amount of logic to figure out which multiplexer to run when and where.

So, because of that there are also in a design reasons of not having many pipeline stages and of course, power is the biggest limiter of frequency. This is probably why in the last 10 or 15 years as of 2020, the frequency has not increased.

(Refer Slide Time: 27:58)




So, how many pipeline stages can we have? Well, there is another argument assume you can still dip your processors and liquid nitrogen and somehow pull them and assume we are the best designers in the world will give you fantastic forwarding circuitry, we are still limited by the latch delay.

So, as you can see in this case, if you have a few stages first stage there is a large amount of logic. So, the latch delay somewhat limited with a larger number of stages they are become roughly equal and if I have a lot of pipeline stages essentially the entire delay of the stage is dominated by the latch delay.


(Refer Slide Time: 28:39)

**Pipeline Stages vs IPC**


$$CPI = \frac{1}{IPC}$$

$$CPI = CPI_{ideal} + \text{stall\_rate} * \text{stall\_penalty}$$

- The **stall rate** will remain more or less constant per instruction with the number of pipeline stages
- The **stall penalty** (in terms of cycles) will however **increase**
- This will lead to a net **increase** in CPI and **loss** in IPC

  
Summary

As we increase the number of stages, the IPC goes down.

NPTEL

McGraw-Hill | Advanced Computer Architecture

28

Now, that we have spoken about the frequency, it is now the turn of IPC. So, IPC and CPI will be two terms that we will be using heavily the  $CPI = \frac{1}{IPC}$ . So, typically the CPI is used while doing mathematical calculations with regards to the performance of a pipeline and an IPC is more like a high level quantity that is either measured or is used for doing more high level decisions on deciding which kind of architecture is better for a given program and so on.

So, it does not matter they are still similar quantities one is the reciprocal of the other. So, the CPI can simply be defined as CPI ideal which is the ideal CPI assuming no stalls + the stall rate, the number of stalls per instruction and the stall penalty which is a number of cycles per stall.

So, both of these if we see a number of cycles per stall and a number of stalls per instructions, so, stalls will cancel out and the final unit will be number of cycles per instruction for this quantity as well. So, this can be reasoned out rather simply that for any observers sitting outside if there are no stalls it will see the ideal CPI.

So, ideal CPI if you are not considering other sources of cycle loss then it is 1 and also it will see some stall cycles and a number of stall cycles per instruction is the stall rate times the

stall penalty. So, the stall rate will remain more or less constant it can be imputably reason it has also been seen in experiments.

So, it will remain more or less constant per instruction with the number of pipeline stages; in the sense that well this does tend to increase those slightly as we as the number of pipeline stages becomes high, but we can say that is the reason we have used the word more or less that the stall rate would remain similar. But, the stall penalty will increase which basically means that in terms of cycles the stall penalty will increase.

And, this can be seen very easily over here that if you have a very long pipeline and we will also have these large dependencies, not just between stages, but you know between any two instructions and for we thus have to make an instruction rate for a reasonably long time or a longer time as compared to a simple 5-stage pipeline.

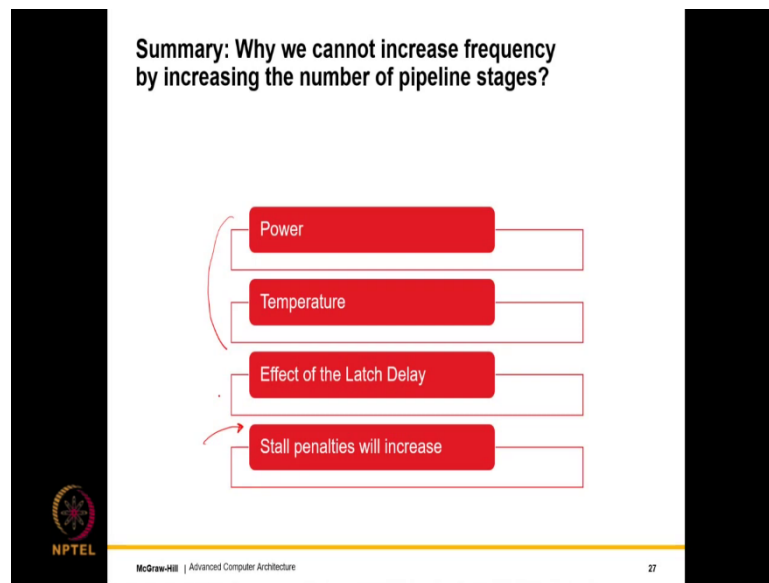
And, it's a stall penalty as we increase the number of stages there is lot of stages it is possible that an instruction here might have to wait for two or three cycles till it can get it is data. So, the stall penalty will increase. So, this will switch means this quantity over here the number of stall cycles or the stall cycles per instruction this is this quantity a number of stall cycles per instruction, this quantity will increase.

And, given the fact that this quantity will increase this will lead to a net increase in CPI and a net increase in CPI is equivalent to a net loss in IPC. So, this is an important learning that we have over here that as a number of pipeline stages increases, the IPC will actually go down right in or in-order pipeline; I mean it is not a generic statement, but in or in-order pipeline as the number of stages increases the IPC will go down because the number of stall cycles is increasing.

So, this is an important point to keep in mind very important point to keep in mind that the IPC will you know start with a good value of 1 and with an increasing number of stages it will just keep going down and down and down.



(Refer Slide Time: 33:17)




So, summary why cannot, why we cannot increase frequency by increase the number of pipeline stages? Well, power and temperature are by far the most important and the most compelling arguments that the power is proportional to a cube of the frequency and the temperature is roughly proportional to the power linearly.

So, the power as this is too much the temperature as this is too much cannot be increased further. There is the effect of the latch delay in the sense that if we have longer pipelines, so, latch delay will dominate the clock period.

Furthermore, the stall penalties as measured as number of stall cycles per instruction will also increase, whatever gains we get by increasing frequency will kind of get negated by reduction in IPC also the complexity; the complexity of designing the forwarding logic the interlocks and verifying the whole thing that will become very complicated.

(Refer Slide Time: 34:25)

Since we cannot increase frequency ...



Increase IPC

NPTEL

McGraw-Hill | Advanced Computer Architecture

28

So, since we cannot increase frequency, what do we do? We increase IPC. So, increasing frequency as we just saw is fraught with problems. So, instead it is a much better idea to actually increase IPC.

(Refer Slide Time: 34:44)

Increase IPC

*issue → send an instruction to an execution unit*

Issue **more** instructions per cycle

2, 4, or 8 instructions

Make it a **superscalar** processor → A processor that can execute multiple instructions per cycle

NPTEL

McGraw-Hill | Advanced Computer Architecture

29

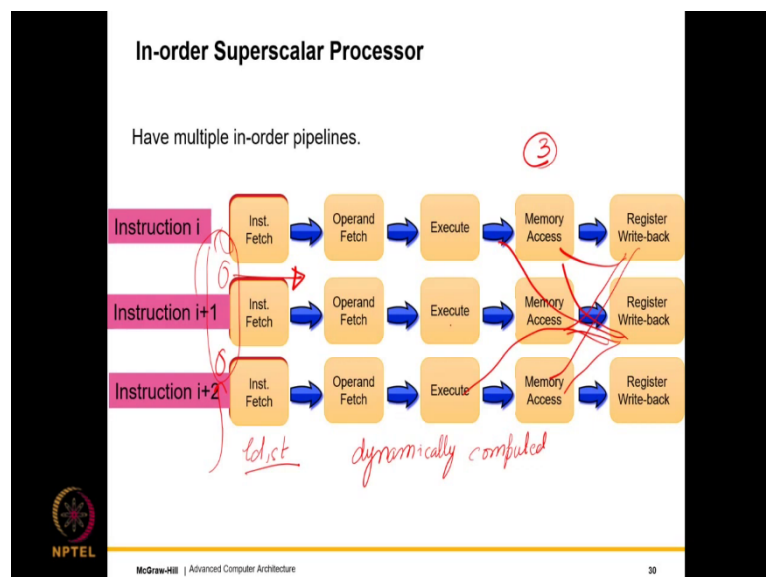
So, what is it that we can do? Well, simple idea is that look do not issue. So, what is issue? So, issue basically means send an instruction to an execution unit. So, at the moment in our

pipeline what is called as the issue width the number of instructions we issue per cycle that is 1.

So, let's increase it. So, let us have a wide pipeline where we fetch multiple instructions together 2, 4 or 8 we decode multiple together and we also issue to execution units' multiple together multiple instructions. So, this is known as a superscalar processor. A superscalar processor pretty much can execute multiple instructions per cycle.

So, this is where there is a built-in instruction level parallelism that it exploits and it executes multiple instructions per cycle.

(Refer Slide Time: 35:52)



So, what we can do is one way of creating an in-order superscalar processor which was similar to the original Intel Pentium idea, which incidentally was a very revolutionary chip published designed in the early 90's. So, that used to have two parallel in-order pipelines, it is called a U pipe and a V pipe.

So, in this case, we can have multiple in-order pipelines. So, we can have multiple in-order pipelines over here and these in-order pipelines as you can see each one of them will have 5 stages and they will just in a fetch instructions run them in parallel if there are no

dependencies, no hazards then well we have an IPC improvement, which the IPC at the most can go up to 3 which is great which is 3 times 1.

(Refer Slide Time: 36:51)

**In-order Superscalar Processor - II**

- There can be dependencies between instructions
- Have  $O(n^2)$  forwarding paths for an  $n$ -issue processor
- Complicated logic for detecting dependencies, hazards, and forwarding
- Still might not be enough ...
- To get the peak IPC ( $= n$ ) in an  $n$ -issue pipeline, we need to ensure that there are no stalls
- There will be no stalls if there are no taken branches, and no data dependencies between instructions.
- Programs typically do not have such long sequences of instructions without dependencies

NPTEL  
McGraw-Hill | Advanced Computer Architecture 31

What is the problem? There are lot of problems. So, the problems will come when there are dependencies between instructions. Dependencies between instructions will always be there. So, there is no way that we can actually get rid of it and if you have an  $n$  issue processor we can have dependencies between all pairs of instructions. So, we will essentially have  $O(n^2)$  dependence detection as well as forwarding paths this will become a massive bottleneck.

So, we will also have complicated logic for detecting dependencies for detecting hazards forwarding and you get the peak IPC =  $n$  in an  $n$  issue pipeline we need to ensure there are no stalls; stalls will be there because of hazards. There will also be stalls because of taken branches we will fetch a lot of instructions in the wrong path all of them need to be nullified cancelled.

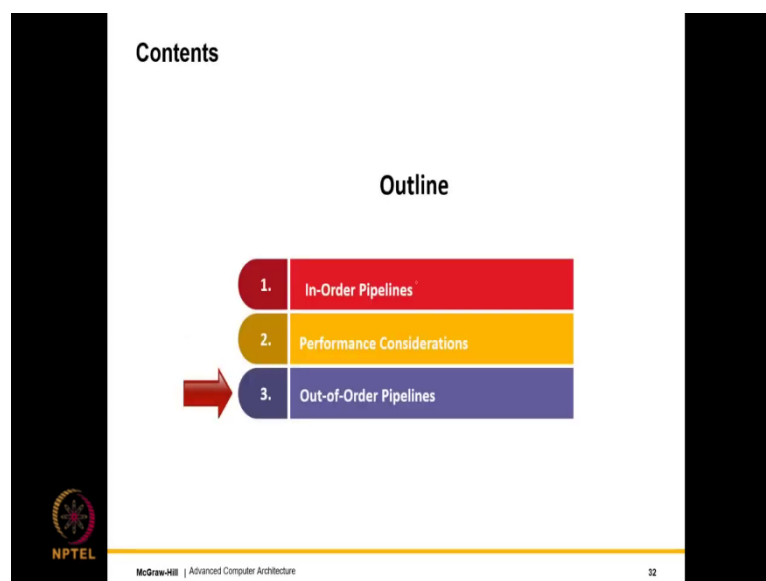
Furthermore, there will be other things that when we are creating. So, if you just look at the diagram over here essentially we will be creating a bundle of three instructions 1, 2, 3 and sending the entire bundle in parallel. It is good if they are independent, otherwise we need to have internal forwarding we will have lot of internal forwarding paths right from here to here to here to here in a lots and lots of paths. And also we need to have forwarding between instructions of the bundle.

What can further make our life different is? if we difficulties if we have load and store instructions because see finding registered dependencies is easy. Registers are named storage locations. So, the decode and operand fetch units can easily figure out and detect register dependencies. However, they will find it very hard to detect dependencies across memory addresses because memory addresses are never specified in instruction they need to be computed.

The memory addresses are dynamically computed in the execute stage and there it is possible that different new dependencies will kind of come up from nowhere there can also be dependencies between instructions in the same bundle the three instructions that proceed together that will make life hard.

So, then also you know finding long sequences of instructions without dependencies or with very few dependencies is in a sense impractical. So, that is the reason what Intel Pentium did is that they had two parallel pipes. More than that would have been impractical and having 3 or 4 is clearly hard. See even with a multi issue in-order pipeline the best that we can do is get an IPC of two most likely it will be much lower than this. So, the game as we see, are not there.

(Refer Slide Time: 39:59)



So, what do we do? Well we will have to change our thinking. So, instead of processing in-order we should process out-of-order which means that we will not observe or respect program order. So, this is out-of-order processing it is called an out-of-order pipeline.

(Refer Slide Time: 40:29)

What to do ...

Don't follow program order

single-cycle

$f_{max} \propto k$

```
mov r1, 1
add r3, r1, r2
add r4, r3, r2
mov r5, 1
add r6, r5, 1
add r8, r7, r6
```

Too many dependence

NPTEL

McGraw-Hill | Advanced Computer Architecture

33

So, what does it mean to not follow program order? Well, what it means? So, let me just take a little walk back into in order see in-order pipeline what we said is that look we have a sequence of stages and we have a sequence of dynamic instructions. A dynamic instruction has been defined before. So, it is the instructions that a processor encounters so, they keep coming in you keep executing them.

Whenever you detect something has gone wrong while you stall and as compared to well what was before this? Before this we had a single cycle processor which did not have any pipelining. So, entire thing happened in one cycle. It was one long cycle one ultra-long cycle; in this case these are short cycles. So, well, where was the advantage coming from pipelining? Well, it did not come from IPC because the IPC reduces with pipelining.

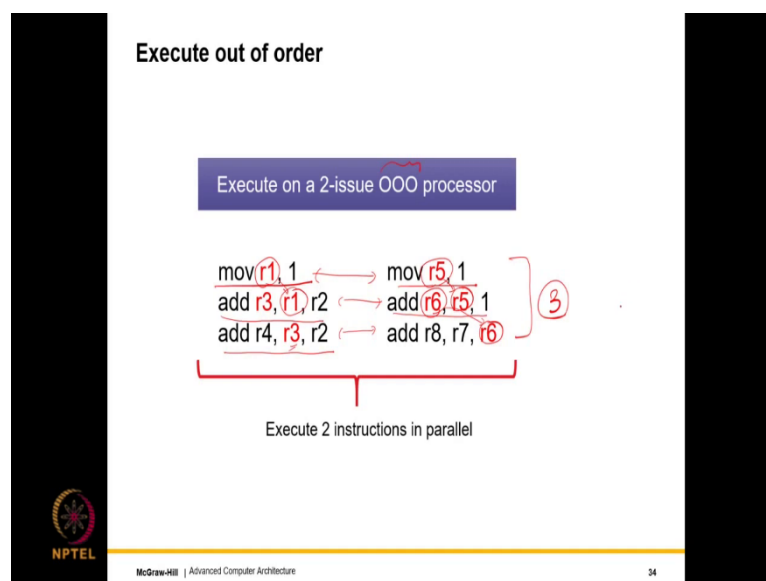
The advantage came because we could crank up the frequency and  $f_{max} \propto k$ . So, that is where the advantage came from, but then we found there are limits and the main limits were that look finding independent instructions was hard. Particularly, when we considered parallel

in-order pipelines that is where finding independent instructions became difficult and the main reason was that we are kind of being constrained by program orders.

So, that is the main problem that we cannot even if there are independent instructions way down in the instruction stream we cannot pull them out and execute them. We will have to proceed in program order which would basically mean in this diagram first execute this instruction, then this, then this, then this and so on which in a certain sense is not really fair.

So, let us look at the structure of dependencies over here. So, you see there are too many dependencies, but there is a certain pattern. So, the pattern is that the first three instructions have a dependency of this kind as you see or read after write. Again here another pattern, but between them between this block and that block well they are independent. Does this give us an opportunity? Well, yes, it does let us move to the next slide.

(Refer Slide Time: 43:06)



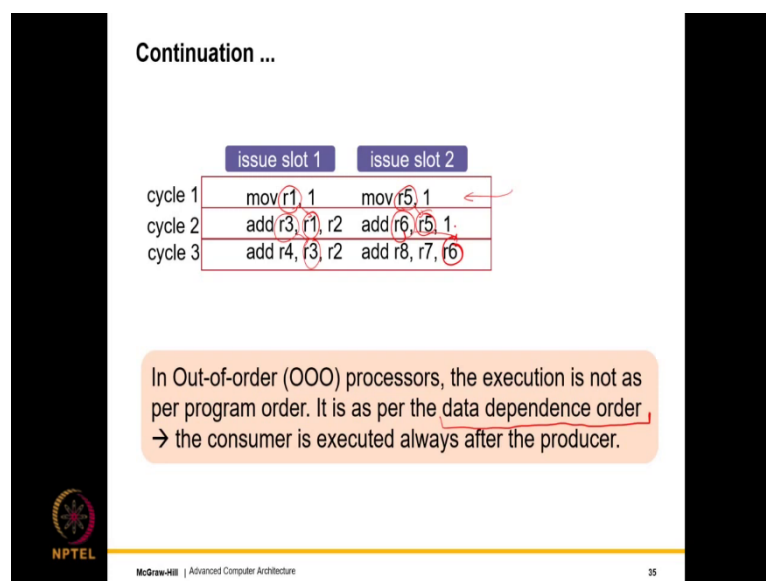
If I were to execute it on a two issue out-of-order processor out-of-order will be given a triple O designation. If I were to do that what I would do? Is that in the first cycle I will fetch this instruction and this instruction. Just go back; it is essentially this instruction and this. Fine, I can do that in the next cycle I can execute this. I do not have to I do not need in the stall cycles because I can forward the value of r1.

Similarly, I can fetch and execute this. Again, stalling is not required forwarding is good enough. Finally, I can get the last instruction. So, since I have an issue width of 2, I will also have a fetch width of 2 which means I can fetch 2 instructions, 2 independent instructions per cycle decode them and send them down to the execution units.

So, as you can see over here we are executing 2 instructions in parallel when we are doing that it is true that they have dependencies, but still since we are executing out-of-order we can without waiting for any without you know waiting for instructions before an instruction in program order to complete.

We are breaking the order extracting more parallelism as you can see right here and instead of taking 6 cycles, which we would have done in a previous example with forwarding enabled of course, we are doing it in 3 cycles which of course, is a fantastic speed up of 2 times.

(Refer Slide Time: 45:05)

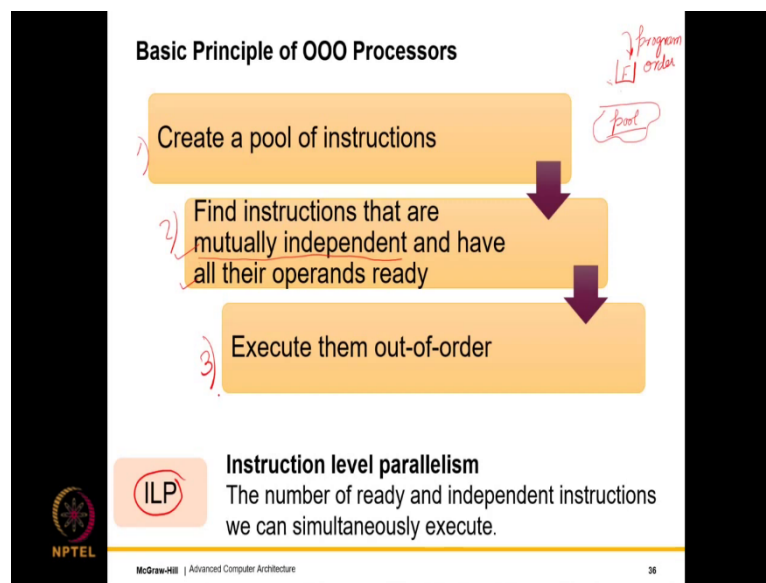


So, this shows what we were discussing that cycle 1, 2 and 3 and this is how we execute the instructions. So, just take a look at this diagram, take a look at this. So, the advantage of an out-of-order processor is rather clear that the execution is not as per the program order instead, it is as per the data dependence order in the sense that the producer instruction always executes before the consumer that is the only criterion that you have the or in other words the consumer is executed always after the producer.



So, that is the criterion that we have that is the criterion that we will use that the consumer will be executed after the producing instructions. So, in this case as we see this is always happening is a producer, this is the consumer, the producer, consumer. So, this is the data dependence order that we are following and because of that we could extract more parallelism out of the program by executing our instructions out of program order not in program order.

(Refer Slide Time: 46:18)



So, what is the basic principle? so, the basic principle is rather important what we do is we create a pool of instructions. So, basically always fetch in program order that is important. So, we always fetch in program order and then naturally if we have an execute bandwidth of let us say 4 or fetch bandwidth also has to be 4 to match it. So, we will see it needs to be similar. but, now let us assume that if the execute width is 4, issue width this 4.

We are also fetching 4, then we fetch instructions put them in a large pool of instructions in a large instruction pool. So, the hardware term is the instruction window. So, we will use the term instruction window and pool kind of interchangeably. The next step is to find instructions that are mutually independent and whose operands are ready.

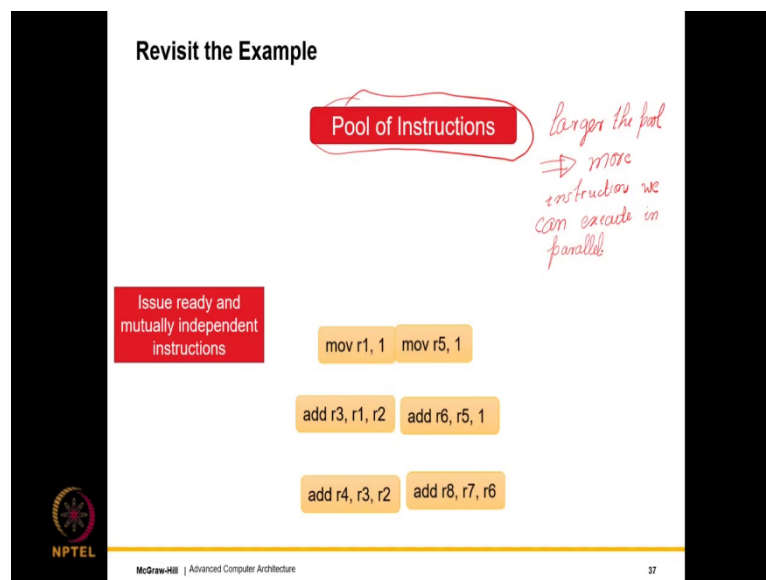
So, these are the two important criteria. The first is that I find a set of instructions to execute together. There should be no dependencies between them that is criteria one that is very

important because there is a dependency between them we cannot forward one has to wait for the other and all their operands have to be ready such that they are ready to execute. These two are obvious and then we execute them out of order.

So, this will give us the parallelism that we saw and that was huge I mean as compared to a small improvement this was giving us 2 to 3 times additional performance. So, the key term that we are using here we use it in the last slide as well is ILP Instruction Level Parallelism which is essentially the number of ready and independent so ready meaning that the operands are ready.

So, inputs are ready and independent instructions we can simultaneously execute in a cycle, this is ILP. So, clearly some programs will have high ILP sometimes the programs may have lower ILP with the compilers can compile them in such a way that more ILP can be exposed. So, it does not matter in whichever way. So, we essentially extract the ILP and on the basis of that we execute instructions in parallel.

(Refer Slide Time: 48:55)



So, here example of a pool of instructions we revisiting what we had these are all our instructions. So, you want to as we said we issued ready and mutually independent instructions. So, here we go an animation follows. We first bring in two ready and independent instructions. Once r1 and r5 are computed, we again bring in two more, again

bring in two more and in three cycles we are all done. Here is the greatness of out-of-order execution.

So, what is the most important factor? Well, it is to quickly build a large pool because larger is the pool the more instructions we will find to execute in parallel. So, larger the pool alright larger the pool more instructions we can execute in parallel. So, more is the ILP that we can exploit.

(Refer Slide Time: 50:45)

**Pool of Instructions: Instruction Window**

- Needs to be **large enough** such that the requisite number of mutually independent instructions can be found.
- Typical instruction window sizes: 64 to 128
- How do we create a large pool of instructions in a program with branches? We need to be sure that **all the instructions** are on the **correct path**

```
for (i = 1; i < m; i++) {  
    for (j = 1; j < i; j++) {  
        if (j % 2 == 0) continue;  
        ....  
    }  
}
```

Example

The slide includes a diagram of an Instruction Window (IW) as a queue and a control flow graph showing a loop with a branch. The NPTEL logo is in the bottom left, and the footer reads 'McGraw-Hill | Advanced Computer Architecture' and '38'.

So, how large should the pool be? Well the pool is we will tell pool more as a kind of a conceptual entity and the hardware structure that stores this pool of instructions we will call it the instruction window which is nothing, but a simple queue of instructions.

So, the queue is filled in program order, but then of course, you extract instructions from the instruction window in data dependent order and then they are sent to the execution units. You have a set of functional unit's adders, subtractors, dividers, multipliers etcetera. So, instructions from here go there.

So, the typical instruction windows sizes range from 64 to 128 instructions, they need to be large. So, because larger is the instruction window the more independent and ready instructions we can find. The more instructions we can execute in parallel, higher will be the

IPC. So, the important question that we need to answer is how do we create such a large pool of instructions because we will have branches in the pool.

So, we need to ensure that the entire pool, the entire window all the instructions are on the correct path they are not on the wrong branch path because typically we will have a new statement which will have two directions again one more if statement. See, if you were to create an instruction pool and let us see if this is the correct path.

So, we need to somehow guess the branch directions and fill instructions from the correct path because any branch if your kind of get it wrong, then all the instructions after it will be in the wrong path. So, this the entire window is gone. So, the entire window basically has to be populated with instructions from the correct path that is very very important.

So, given this let us take a look at a simple loop. So, if you see a loop well as you can see there are lot of branches in here we need to test if  $I < m$ , if  $j < I$ , if  $j / 2$ . So, with all of these things for such a loop like this if we kind of unwind it we will have a path like this and for each of the branches we need to somehow predict it correctly because we would not have executed that many branches.

And, so, to essentially fill a large window without executing the branches we will have to predict them and fill the window with instructions from the correct path with a very high probability.

(Refer Slide Time: 53:39)

**Problems with creating an Instruction Pool**

100  
20  
Typically 1 in 5 instructions is a branch  
branch  
direction  
target  
Predict the directions of the branches, and their targets  
Solutions

NPTEL  
McGraw-Hill | Advanced Computer Architecture  
39

So, how do we create such a large instruction pool? Well, the option of evaluating, so, let us say that 1 in 5 instructions is a branch and we have a 100 entry sized instruction pool then it will have 20 branches.

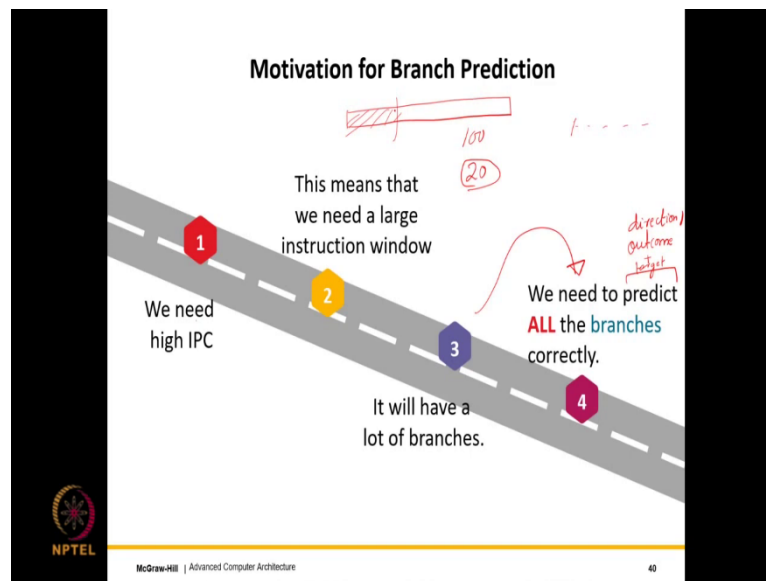
So, we clearly do not have the option of picking out those 20 branches and evaluating them because every branch would be dependent on a few instructions before it will be dependent on their result. So, they also need to be executed similarly another branch they also it also needs to be executed essentially all the instructions that determine the outcome of the branch all of them need to be executed. So, for us to actually pre-execute these 20 with high priority is not an option.

So, just in case it crossed your mind let's say pre execute branches you know like you know give them ultra high priority, this is not an option because there will be many other instructions that determine the outcome of the branch. So, this is not feasible strategy of filling up the window because we will end up executing the program.

Our aim was to somehow fill up the window then find independent instructions from there quickly and execute as many as we can. So, given that this is not an option the only option we are left with is to somehow predict the directions of the branches and the targets.

So, given a branch, so, let us see beq. So, it has two attributes one is the direction of the branch is it taken not taken the other is a target of the branch. So, the target of the branch is where the branch points too? Even that needs to be predicted because that has to be computed and we do not have enough time. So, we want to predict as much as we can in terms of the direction and target and fetch instructions.

(Refer Slide Time: 55:57)



Let us now retrace our journey. So, we said that we need high IPC because well reducing the number of instructions and also increasing the frequency seem to be typical goals. So, some amount of saturation we had already reached. For high IPC we need to exploit a large amount of instruction level parallelism.

Hence we need a large instruction window such that we can find a lot of ready and independent instructions in one cycle, but because to sustain a large instruction window let's say an instruction window has 100 entries it will have roughly 20 branches. Those 20 branches we cannot execute we do not have the time for executing them because I mean the branches do are never executed independently.

Branches are the outcome of a branch is determined by a large number of instructions before it they have to be executed. So, clearly we cannot fill up a window that quickly by executing branches that is not practical. We will have to predict branches both their outcome as well as

their target and all of these branches we need to predict all of the branches. As I said prediction involves two things outcome and target outcome we can say outcome or direction.

So, it is taken not taken and the target right direction. So, you can say direction slash outcome plus target. So, note that if there are 20 branches, all 20 of these branches all of these branches have to be predicted correctly to fill the entire window with 100 correct instructions. Even if one of these branches is predicted incorrectly the entire pool of instructions is pretty much useless or at least still that branch and all the instructions fetched after it they are useless.

So, as far as we are concerned it is a problem even if there is a single miss prediction because a large part of it will become large part of the pool even if we treat it as a queue will pretty much become useless and we will have to remove them and again re-fetch that will cause a major issue. So, we want our ideally speaking we want our entire pool to be correct.

(Refer Slide Time: 58:38)

The Maths of Branch Prediction	
Number of instructions	$n$
Number of branches	$n/5$
Probability of predicting any given branch <b>incorrectly</b>	$p$
Probability of predicting <b>ALL the branches</b> correctly	$(1-p)^{n/5}$
Probability of making at least a single mistake (branch misprediction) in a pool of $n$ instructions.	$P_n = 1 - (1-p)^{n/5}$

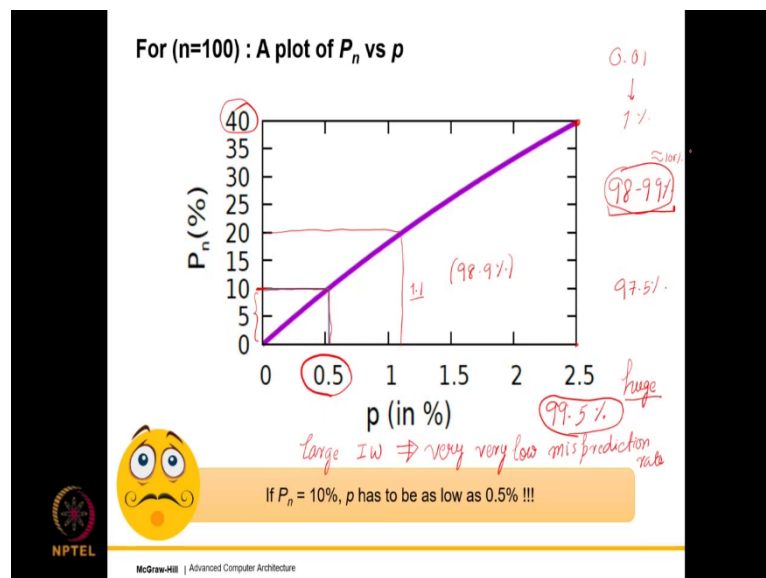
Consider the maths of branch prediction. So, let us say that we have 'n' instructions. So, on average n instructions will have n by 5 branches. So, let us assume that the number of branches is n over 5, (n / 5). So, let the probability of predicting any given branch incorrectly wrongly be 't' and let us assume is the same for all branches the probability of making a mistake for a given branch.

So, the probability of predicting all the branches correctly, so, what is the probability of predicting a single branch correctly? Well, it is  $(1 - p)$ . The probability of predicting all of them correctly is  $(1 - p)$  given the fact that they are independent events we are assuming that they are independent events. So, the probability of predicting all of them correctly is  $(1 - p) \times (1 - p) \times \dots \times (1 - p)$ ,  $\frac{n}{5}$  times, So, this makes it  $(1 - p)^{\frac{n}{5}}$ .

So, probability of making at least a single mistake a single branch miss prediction in an in this full pool of  $n$  instructions of many single mistake 1, 2, 3, 4, 5 does not matter, but at least a single mistake is 1 minus this number which is  $1 - (1 - p)^{\frac{n}{5}}$ . and let this value be  $p_n$  which is of course, has to be a function of  $n$ .

So, what is  $p$ ?  $p$  is a probability of miss predicting a single branch and  $p_n$  is probability of making a single mistake a single branch miss prediction in the entire pool which means that with this probability the entire pool needs to be jumped.

(Refer Slide Time: 60:40)



So, if I were to just plot a probability of  $p_n$  versus  $p$  and of course, you know in this case  $p$  is in percentage both are in percentage. So, essentially probability of 0.01 would become 1%. So, as you can see if 90% of the time I want my window to be all correct which means my  $p_n$



= 10% that is the number of mistakes that I am allowed to do, there is a number of times I am allowed to have a window with at least 1 miss predicted branch.

If I were to bound  $p_n = 10\%$  and I look at the corresponding point on one of the curve. So, this is roughly 0.5% slightly more maybe 0.55% and some, approximate it to 0.5 might not be a very accurate one but 0.55 let's say or does not matter, but we can see that roughly the probability of miss predicting a single branch has to be in the ballpark of this figure which means that a successful prediction probability of any given branch has to be 99.5 % which is huge.

Predicting anything with such a massive accuracy is clearly going to be a very hard task and this is something that we have seen you know the math from here if I were to just plot it over here. So, what am I doing I am essentially plotting  $p_n$  with small p. If I were to do that and the x-axis represents the probability of making a mistake with any given branch and  $p_n$  is the probability of the entire window getting junked.

If let us say and when will entire window get jumped if there is at least a single mistake because then we know that not everything is correct. And, so, if we were to do that then we see even if let us say I say that look 80 % of the time I want my window to be all correct and I draw here still if I were to see well this probably 1.1 percent.

So, even here also 98.9 % of the time your per branch prediction accuracy has to be 98.9 %, which is very high which means that the branch prediction error rate miss prediction rate has to be extremely low. This means that if you want if we want to sustain a large instruction window this automatically implies that we need a very very low. I am deliberately adding two varies to emphasize the fact that the branch prediction has to be extremely accurate a very very low miss prediction rate.

If it is not the case, then we can quickly see that we will have disaster at our hands. So, let us say instead of  $0.5 = (2)^{1/2}$ . So, well even  $(2)^{1/2}$  error is not bad, it is 97.5% of the time, correct. So, here also as you can see we would end up with roughly 40 % error rate for the


window which means that 60 % of the time the entire window will be correct, 40 % of the time then at least be one miss prediction and the window.

So, pretty much we cannot trust the contents of the window. So, a 60 – 40 ratio is still quite bad because as we will see when we design the pipeline having a wrong branch in the window is a very very expensive thing in terms of time. So, we typically do not jump the entire window we try to kind of save all the branches before the all the instructions before the miss predicted branch before it in program order.

But, nevertheless it does have a very substantial performance penalty and as a result getting a correct window is very very important. And, we are talking of extremely low miss prediction rates extremely high prediction accuracies and extremely low is of the range of 0.5, 1, 1 and half, 2 % which means that we need to be 98 to ideal branch predicted for a single branch its accuracy should be in the range of 98 to 99 % to sustain a window of 100.

This automatically places a limit on the instruction window size it cannot be more, if it is let us say thousand then this number will even will get very very close to 100 % we can never design such an accurate such an accurate predictor in practice. So, it is far better that we stick with what we have and a practical window size is thus limited to maybe 150 instructions maximum not more than that, primarily because the branch predictors accuracy is limited.

(Refer Slide Time: 66:12)



If we need a large instruction window, we need a very accurate branch predictor. The accuracy of the branch predictor limits the size of the instruction window.

NPTEL

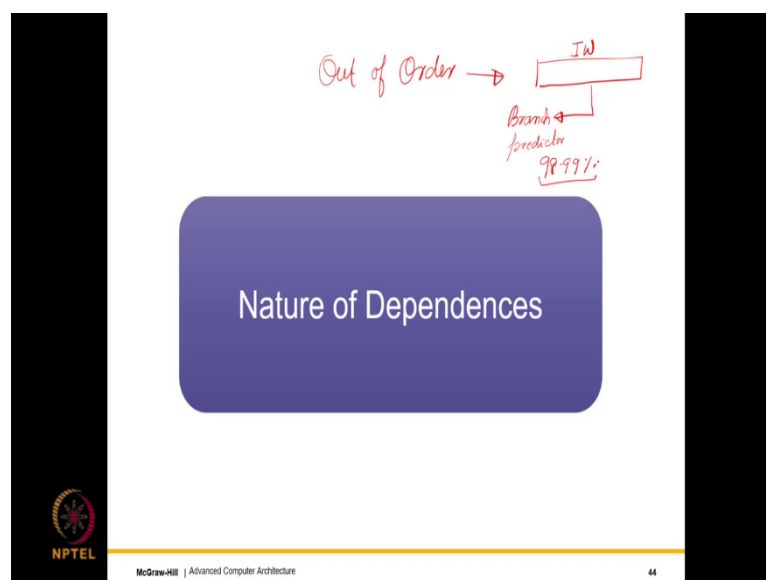
McGraw-Hill | Advanced Computer Architecture

43

So, what did we just learn that look if we need a large instruction window we need a very accurate branch predictor. And, there is something that you can study there in information theory there is a result called the Fano's inequality which says that look there are theoretical limits to predicting a given sequence. So, in this case, a sequence of branches and this has to do with the information content of the stream.

So, given the fact that we run into theoretical limits our branch predictor also has hard limit on this prediction accuracy. So, this limits the size of the instruction window and we cannot go more than that. It is not practical to go beyond.

(Refer Slide Time: 67:01)



So, now we will take a turn towards analyzing. So, now, where are we? We are at the point where we have realized that out-of-order execution is required that we have kind of convinced ourselves. Then for out-of-order execution, we need a large instruction window which is a large conceptual pool of instructions.

From this we pull out ready and independent instructions. To create a large instruction window where all the instructions are preferably on the correct path, we need a very accurate branch predictor, where the branch predictor is predicting both the target as well as the outcome and we are talking of extremely high accuracies over here as we just saw from a mathematical analysis 98 to 99%.

And, with that we can fill our window, we can execute. Have we solved all the problems? We have not because in an out-of-order pipeline we will be looking at many more kinds of dependencies other than the simple raw dependencies raw read after write dependencies that we have seen in the in-order pipeline. So, we will handle the nature of dependencies in the next lecture of this lecture series.