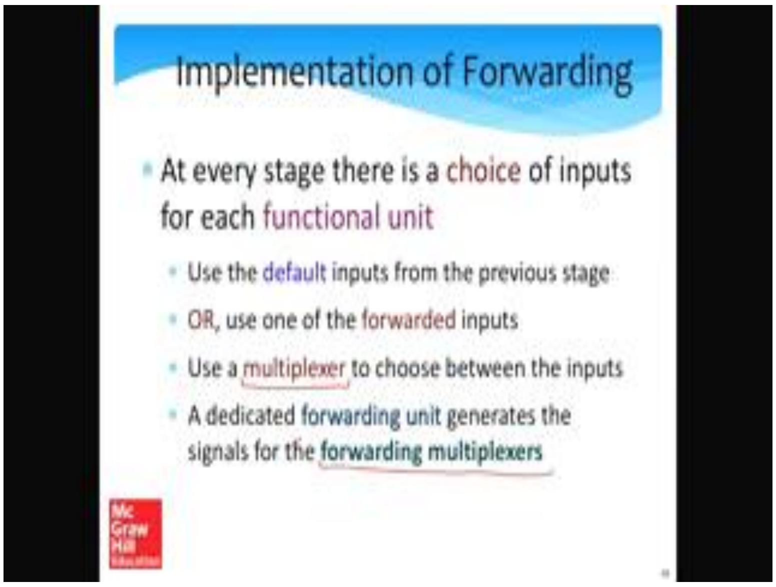
Computer Architecture

Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 28
Principles of Pipelining Part- III

Let us now discuss Implementation of Forwarding.

(Refer Slide Time: 00:25)



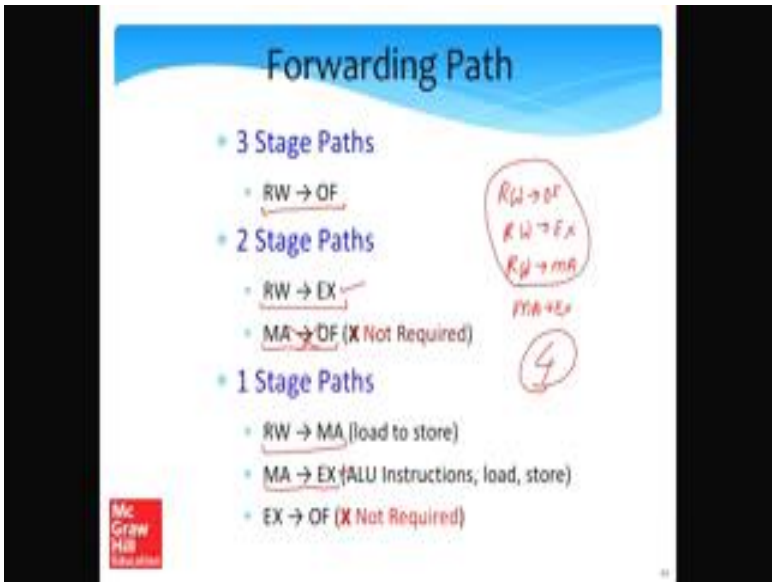
Implementation of Forwarding

- At every stage there is a choice of inputs for each functional unit
  - Use the default inputs from the previous stage
  - OR, use one of the forwarded inputs
  - Use a multiplexer to choose between the inputs
  - A dedicated forwarding unit generates the signals for the forwarding multiplexers

So, at every stage there is a choice of inputs for that particular functional unit. We can either use the default inputs from the previous stage or we can use one of the forwarded inputs. So, when there is a choice we typically use a multiplexer to choose between one of the inputs. And similar to our control path that we had in the case of simple hardwire processor shown in chapter 8; we will have a dedicated forwarding unit that generates the signals for these multiplexers which are called forwarding multiplexers.

So, this will be clear when we discuss the examples. So, before the discussion let us take a look at forwarding paths once again.

(Refer Slide Time: 01:19)



Forwarding Path

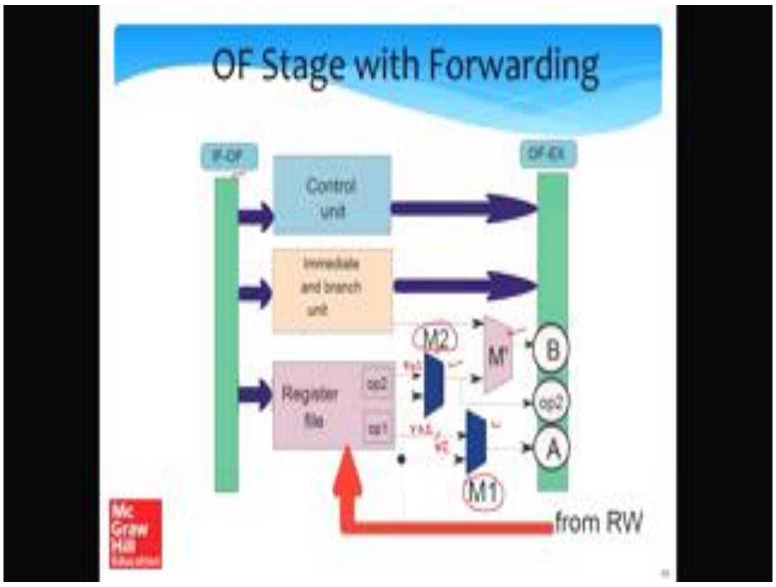
- 3 Stage Paths
  - RW → OF
- 2 Stage Paths
  - RW → EX
  - MA → OF (X Not Required)
- 1 Stage Paths
  - RW → MA (load to store)
  - MA → EX (ALU Instructions, load, store)
  - EX → OF (X Not Required)

RW → OF
RW → EX
RW → MA
MA → EX
4

So, we have four forwarding paths: we have one path from RW to OF which is the long three stage path, we have a two 2 stage path which are RW to EX and MA to OF; sorry MA two OF is not required we do not have it. We have two 1 stage paths which is RW to MA and MA to EX.

Basically OF will have one additional path into it, EX will have two additional forwarding paths: one from RW, one from MA. And the MA stage will have one forwarding input from the RW stage. We have a total of four forwarding paths.

(Refer Slide Time: 02:07)



OF Stage with Forwarding

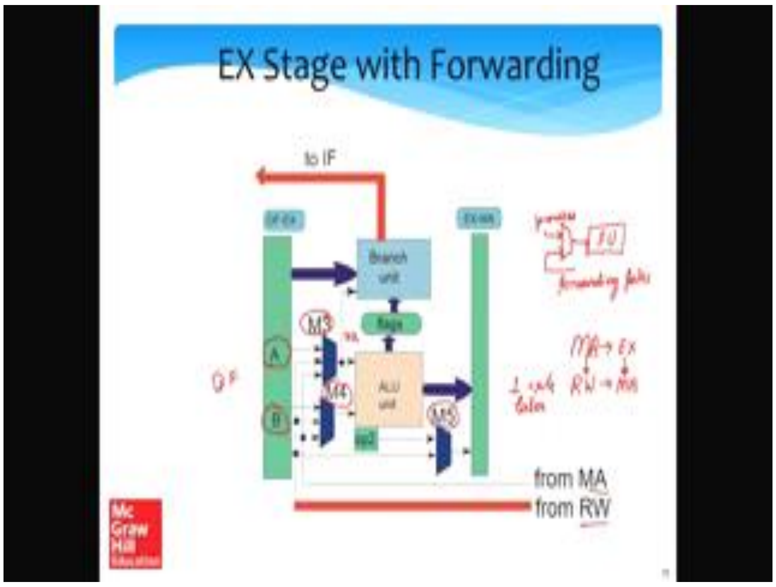
So, let us now take a look at the OF stage with forwarding. So, there is only one forwarding path which is coming into the OF stage and this is coming from the RW stage. So, what we do is that let us first consider the units of the OF stage. The inputs come from the IF-OF register, they are processed by the functional units inside the OF stage and the outputs go to the OF-EX register.

As we had discussed in the previous few slides and in the previous chapter as well there used to be one multiplexer to choose between the output of the register file and the output of the immediate unit. This is for the second operant, whether the second operant is registered or an immediate. So, this was an original multiplexers that we had, so we still maintain it. In addition we add two new multiplexers called forwarding multiplexers. So, the first forwarding multiplexer M 1; multiplexers 1 takes as its input op 1 which is the contents of the first register. So, the contents in the rs 1 field are the first source registers right; its contents 32 bit contents. And also the forwarding input from the RW stage. So, we take both of these values as inputs and we choose one of them. And how we choose one of them we will discuss the logic later.

Similarly at the end of the other register file port op 2 which gives us the value of the second source register rs 2. We also do the same; we have a multiplexers the chooses between the value of the register, and the forwarded input from the RW stage we choose between them. So, whatever we choose we send it to the original multiplexers M dash which chooses between a resister input and an immediate so the rest remains the same. And also we had discussed that in the case of a store instruction it is necessary to essentially store the valve of the second source register that we read. So, that was stored in the op 2 field so that is also there.

So, the only two additional changes that we make in this particular airbrush diagram of the pipeline stage is that the only add to extra multiplexers M 1 and M 2. See in M 1 and M 2 they help us choose between what the register file is giving us which we call the default input, and the forwarded input that is coming from the RW stage; we choose between them. And if forwarding needs to be done we choose the input that is coming from the RW stage, otherwise we choose what the register file is giving us.

(Refer Slide Time: 05:41)



EX Stage with Forwarding

Let us now take a look at the EX stage with forwarding. So, in the EX stage what we do is that we mostly maintained the structure that we had. So in the EX stage what we had is that we had.

So, let us first take see what goes into the EX stage. So, three values going to the EX stage: A op 2 and B. So, A is the contents of the first source register, B is the content of the second source register or the immediate and op 2 is the contents of the second source register. Say A, B and op 2 are what going to the EX stage, so we have been seeing this diagram for quite a time now. Now what do is that we introduce two additional multiplexers M 3 and M 4 which are forwarding multiplexers.

The forwarding multiplexers choose between other values A and B and the forwarded inputs that come from the RW stage and the MA stage. So, one forwarding input comes from the MA stage and one comes from the RW stage, so both of them are sent to these three input multiplexers M 3 and M 4. So, both for M 3 and M 4 what are the inputs again? One input is that comes from the previous stage which is the OF stage. Two more inputs come from other stages the MA and RW stage. So, based up on the forwarding decision we choose one of these inputs as the output and they are sent to the ALU unit.

Also recall that the value of the return address register needs to be sent to the branch unit. So, the return address register is a valid register, its inputs can also be forwarded. So, we do that and so the output of M 3 is also sent to the branch unit as should be the case and the branch units does its computation, and sends the results to the IF stage. We add one more multiplexer M 5 to choose between op 2 which is the register that the store instruction will use and the forwarded input from the RW stage.

So, one trick question that readers can ask is that why are we not using the forwarded input from the MA stage. So, that is very simple to explain the reason we are not doing this is because we will actually require the op 2 value in the MA stage, right. So, instead of doing an MA to EX forwarding and it is much better to follow our previous principle that we forward as late as possible and we forwarded from RW to MA. Because the instruction that is in MA in the current cycle would reach the RW stage in the next cycle. And a current instruction that is there in the EX stage would read reach the MA stage in the next cycle.

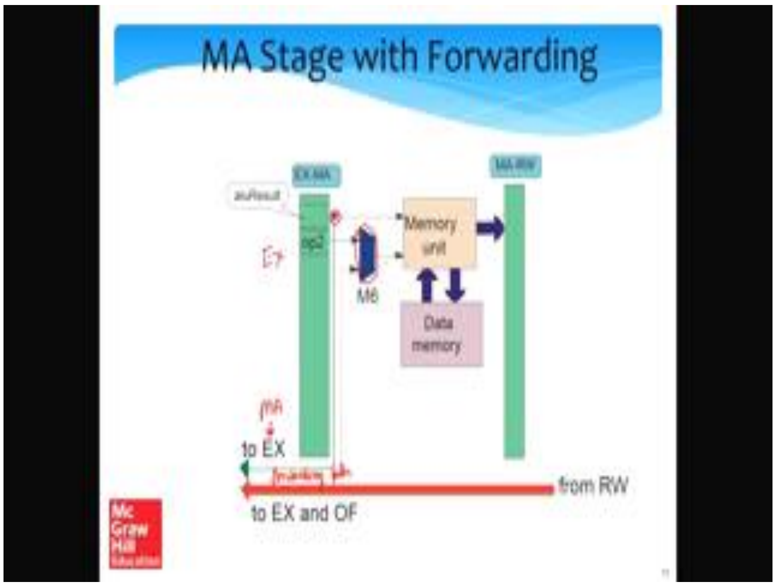
And in the EX MA and RW stages instruction do not get stalled. So, just in case the value for op 2 is being produced by the immediately earlier instruction we can get it from the forwarding path in the next stage. So, we will reduce one input, and also we will go by our self declared principle which is that we forward as late as possible. So, that is the reason we add one more multiplexer over here which is M 5.

So, what the reader would have noticed by now is that adding forwarding to an existing pipeline is actually very very simple; it is super simple. And the idea is very straight forward. Therefore, every input to the functional unit we add a multiplexer before it. So, let us assume that this is a functional unit; we add a multiplexer before it. One of the inputs the multiplexer comes from the previous stage which is the default input, and the rest of the inputs come from the forwarding paths.

So, if there is one forwarding path there will be one additional input, if there are two forwarding paths there will be two additional inputs. Then we choose between one of these inputs and we send it to the functional unit. So, that is the simple idea. Since there are two forwarding paths that come from the MA and RW stages to the EX stage every multiplexer to the input of both the branch unit and the ALU unit will have to choose one input among three. One of them comes from the previous stage and two come from forwarding paths. Op 2 is an exception; the reason its and exception is that it is not used in the EX stage rather it is used in the MA stage.

So, we only need to consider one forwarding path which is the RW stage. Just in case the correct value is there in the RW stage we can forwarded at this point of time, because we cannot forwarded later the instruction will leave the pipeline. And just in case the right value is there is the MA stage instead of doing an MA to EX forwarding we can do an RW to MA forwarding to one cycle later. So, this is according to our principle that we forward as late as possible; right one cycle later.

(Refer Slide Time: 12:01)


MA Stage with Forwarding

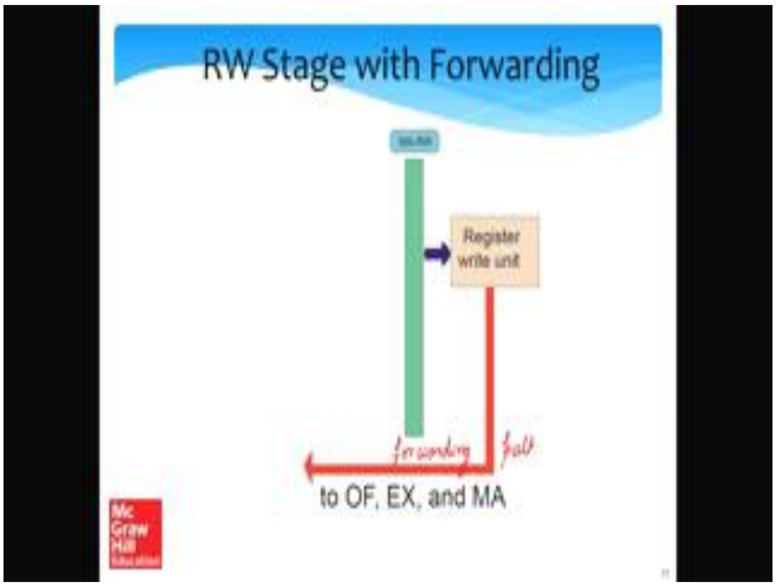
So, let us now move to the next stage which is the MA stage. In the MA stage also we maintain everything; that was there in our previous pipeline. We maintain all the structures that were there in the MA stage; we maintained the memory unit, and the data memory, we maintained everything. Let us first focus on the inputs and then we shall discuss the new structures that we add.

So, the input is the ALU result which is the address. So, for a load this is the only field that is required, and since this is pre computed by the ALU unit it will not be affected with forwarding. So, it can go directly to the memory unit. In addition since it is the result of the ALU this is the beginning of a forwarding path, so when we are forwarding from MA to EX it is this value which needs to be sent to the EX stage because this is the value that the ALU has computed in the immediately presiding cycle. So, we add a wire the green wire which goes to the EX stage, so this is the forwarding path. So, let me maybe write it here.

Now, in the case of a store we need to save the contents of a register into memory. So, this we have been referring to the contents of this register the op 2 field. So, the op 2 field will be affected by forwarding path. So, what we discussed in the last slide is that instead of doing an MA to EX forwarding with op 2 fields we can do a RW to MA forwarding, because we want to forward as late as possible. So, we will use the forwarding path which is coming from the RW stage and going to the EX and OF stages. And what we shall do is that we will add a set of wires between the RW forwarding path and choose the op 2 field which is coming from the previous stage which is the EX stage and the op 2 field which is coming from the next stage which is the RW stage.

So, we will add a single multiplexer which is multiplexer M6, it is our sixth multiplexer; which chooses between these two inputs for the value that needs to b stored. This value is then sent to the memory unit and in the memory unit the load of store happens. And the result of the load is sent to the MA RW register. Recall that the store does not have a register result so will not produce any output other than writing to memory.

(Refer Slide Time: 15:06)

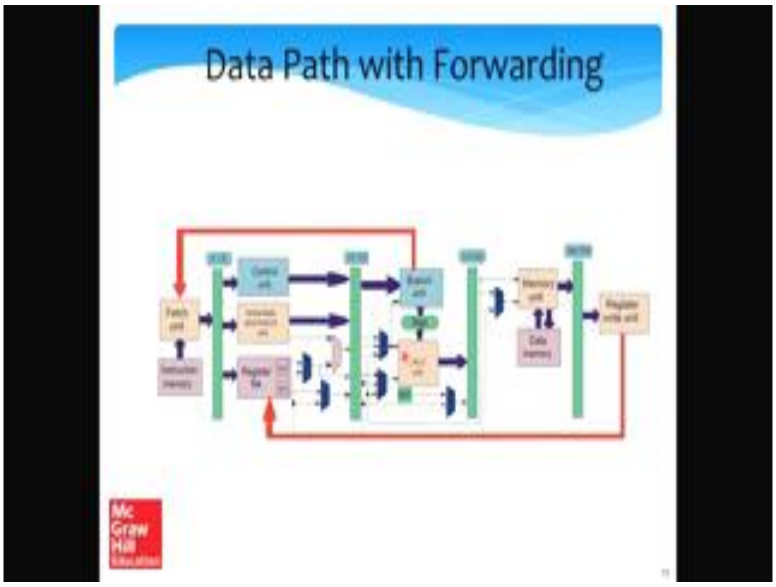


The RW stage is very simple; we will have a register write unit and in the case of RW we actually forward the RW result to three stages. Let me just go back and show the forwarding paths once again. So, forwarding paths RW to OF, RW to EX, RW to MA; so RW is the biggest forwarder in that sense it forwards to three stages and MA to EX. MA to EX we have already seen the forwarding path, you need to take a look at these three.

So, let us go to this path. So the register write unit, the value that it receives from the previous stages the value that needs to be written into the register the same value is forwarded to the OF-EX and MA stages via forwarding path.

So, this is fairly simple some students might find it slightly difficult understand it for the first time; that is the reason let us go back to the airbrush diagram shown over here.

(Refer Slide Time: 16:17)



Data Path with Forwarding

So, what we see is that; so let us first the airbrush diagram remains as it is I have not shown the interlock units, for you know reasons of simplicity but the interlock units will also be there. So, if I have forwarding the only additional structures that I shall have is this blue multiplexers, and if you will count there are six of them; 1, 2, 3, 4, 5 and 6. We added two multiplexers in the OF stage.

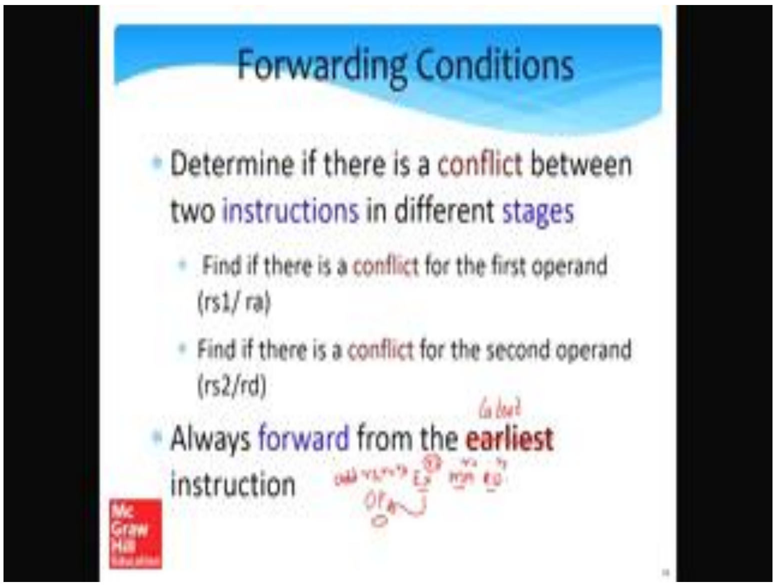
The multiplexers choose between the value that is coming from the RW stage; the last stage. And the inputs generated by the register files, so this multiplexer and this multiplexer. In the EX stage we add three multiplexers, because EX receives forwarded inputs from 2 stages; from the MA stage and the RW stage. So, each of these two multiplexers chooses between the default input coming from the previous stage and inputs coming from the MA and RW stages respectively. We need to add one more multiplexer for the store value because the store value might be coming from the RW stage. So, this chooses between the default value and the value coming from the RW stage.

Then in the MA stage; so MA stage is both the consumer and the producer are forwarded information. So, the value it receives in the previous cycle can be forwarded to other stages namely; the EX stage. So, we add this green line over here which is a forwarding path. And the MA stage at the same time receives a forwarded input from the RW stage. So, the RW stage as the input goes here. So, we choose in the case of a store instruction not required in the case of a load, but in the case of a store we choose between the default input and the input coming from the RW stage with the help of our sixth and last multiplexer. The results go to the memory system and we perform a load of store.

Finally, we have a register write unit and which does the work of the RW stage. We also send the value that we get from the previous stages, so we shall either get a value from the EX stage or a value from a memory unit in the case of a load. So, whatever the value is we send it to the rest of the three stages OF, EX and MA via the red forwarding path.

So, that is let me remove the notation on this slide. So, this is pretty much all the changes that we need to do to implement forwarding. And it is not much if you think about it we have added 6 multiplexers to choose between the default inputs and forwarded inputs; the six multiplexers we can have laser pointer here. So, 6 multiplexers are 1, 2, 3, 4, 5 and 6. Then we have a set of wires that come from other stages, and depending upon the logic we choose between one of the inputs and that input is the input for the functional unit; that is all.

(Refer Slide Time: 20:01)



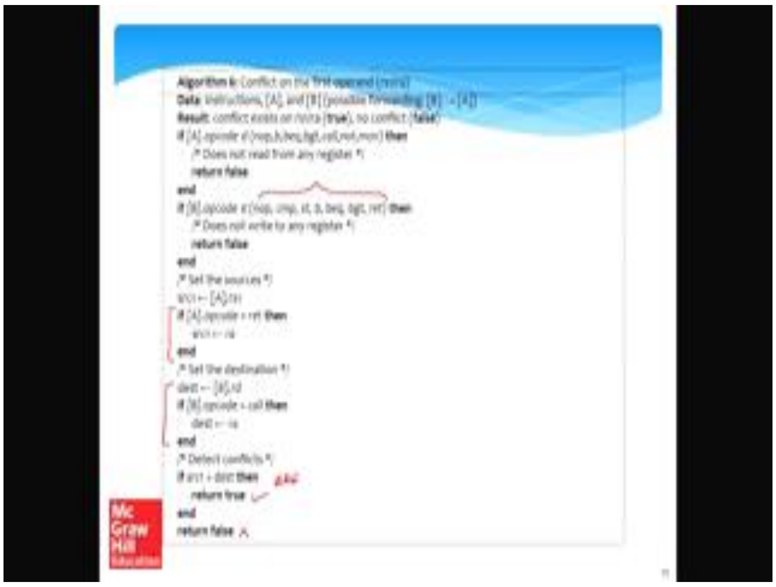
Forwarding Conditions

- Determine if there is a conflict between two instructions in different stages
  - Find if there is a conflict for the first operand (rs1/ ra)
  - Find if there is a conflict for the second operand (rs2/rd)
- Always forward from the earliest instruction

Now let us take a look at forwarding conditions. So, we need to determine if there is a conflict between two instructions in different stages. A conflict means a read after write dependency. So you find if there is conflict for the first operand rs 1 or ra, and we find if there is a conflict for the second operand which is rs 2, rd; rd in the case of a store and for the first case ra in the case of ret instruction. So, we will always let us assume that there is an instruction in the OF stage and it has a read after write kind of dependence conflict with instructions in all the three other stages right, in the EX stage in the MA and RW.

So, basically for the same operant we will always choose the earliest; well, we will always this statement is not actually correct we will always forward from the latest instruction. Which basically means that if the instruction is coming from the EX stage. So, let us assume that all the three instructions over here are. So, this instruction is of the type add r 1 r 2 and r 3, and all the three instruction over here generating a value for r 2. So, we will essentially forward from the EX stage, because that is the latest stage. So, EX stage is essentially overwriting the values of r 2 produced by the two other stages. So, we always forward from the latest instruction that is before the current instruction.

(Refer Slide Time: 21:49)



```
Algorithm 6: Conflict on the first operand (rs1/ra)
Data: instructions, [A], and [B] (possible forwarding [B] → [A])
Result: conflict exists on rs1/ra (true), no conflict (false)
if [A].opcode ∈ (nop,b,beq,bgt,call,not,mov) then
    /* Does not read from any register */
    return false
end
if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
    /* Does not write to any register */
    return false
end
/* Set the sources */
src1 ← [A].rs1
if [A].opcode = ret then
    src1 ← ra
end
/* Set the destination */
dest ← [B].rd
if [B].opcode = call then
    dest ← ra
end
/* Detect conflicts */
if src1 = dest then
    return true
end
return false
```

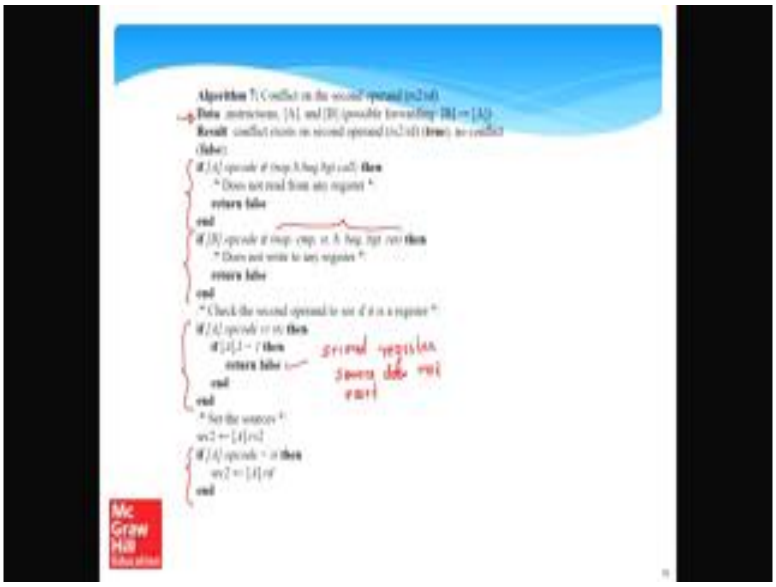
So, let us take a look at the forwarding algorithm which is very similar to the conflict detection algorithm. Let us look at the conflict on the first operand which is rs 1 or ra. So, the input is the instructions A and B and the possible forwarding is that instruction B is coming before instruction A. So, B can forward some data to instruction A. So, if a conflict; sorry if a conflict exists on rs 1 or ra we return true otherwise if there is no conflict we return false. So, let us see; since A in this case is a consumer instruction and it is one of the following nop, p, bdq, bgt, call, not, and mov. It means that it does not use its first source. So, it does not use the first source operand, it does not use the rs 1 field. And since it does not read from any register we cannot have a conflict, so we return false.

Similarly, B b is a part of this set which is nop, cmp, store, bdq, bgt and ret; it means that we are not writing to any register. So, we cannot have a read after write kind of dependency if any registers are involved so we also return false. Otherwise we set the sources. So, we say that src 1 is a s ra 1 field. However, we take this special case into account that if A is the return instruction then we set src 1 as ra. So, refer to chapter 8 we have discussed the case; so refer to chapter 8 for this.

Similarly we set the destination. So, in the case of A we take consider the destination variables so mainly the entire algorithm needs to be implemented in hardware. So, we are only showing the pseudo code over here. So, the destination register should ideally be the rd field in the instruction B. However, if instruction B is a call instruction then the call instruction writes to the return address register, so dest will be equal to ra.

Once we have set the source and the destination we need to detect conflicts. So, if src 1 is equal to dest which means that the first source of A is equal to the destination of B, then a read after write kind of a conflict on registers exists so we return true otherwise we return false.

(Refer Slide Time: 24:33)



```
Algorithm 7: Conflict on the second operand (rs2/rd)
Data: instructions, [A] and [B] (possible forwarding: [B] → [A])
Result: conflict exists on second operand (rs2/rd) (true), no conflict
(false)
if [A].opcode ∈ (nop, b, beq, bgt, call) then
    /* Does not read from any register */
    return false
end
if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
    /* Does not write to any register */
    return false
end
/* Check the second operand to see if it is a register */
if [A].opcode ≠ st then
    if [A].I = 1 then
        return false
    end
end
/* Set the sources */
src2 ← [A].rs2
if [A].opcode = st then
    src2 ← [A].rd
end
```

So, let us look at a similar algorithm for the second source operand which is rs 2 and rd in the case of a store. So, the inputs are instructions A and B, and there is a possible forwarding path from instruction B to instruction A. So, a conflict will exist on the second operand rs 2 and rd; if a conflict exists we return true otherwise we return false similar to the previous algorithm. So, let us do one more thing, let us first look at instruction A. So, if this is a part of this set nop, dbq, bgt, so it is a branch instruction or it is a call instruction this means that it does not read from any register in specific it does not use a second source operand of the instruction.

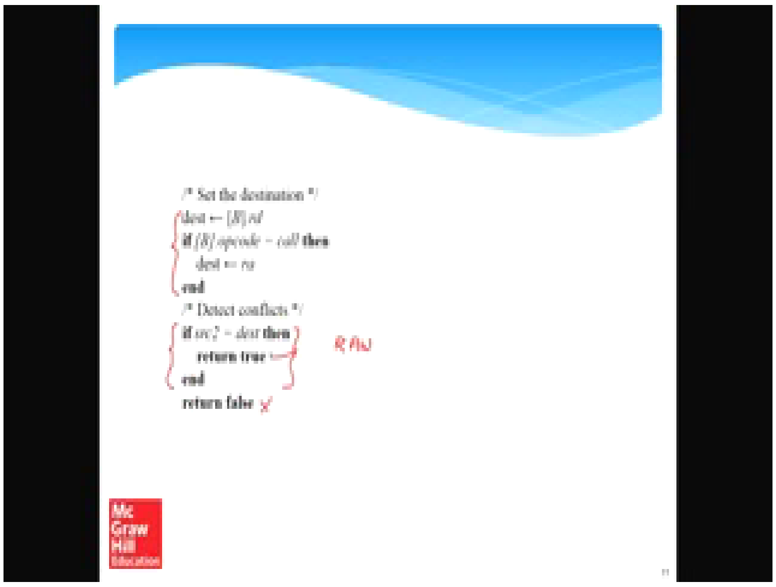
Since it does not read from any register let us straight away return false. Now let us consider instruction B. So, again if instruction B is the part of this step this set no op compare store branch different flavors of branches and return. It means that it does not write to any register so we can also return false, because if you are not writing to any register we cannot have a ra w dependence on a register. So, we will also return false.

Now let us see if the second operand is a register or an immediate. So, let us do this little check and we had done the same check for interlocks, so let us do it right now. If A is not a store instruction, so why do we again say that A is not a store instruction, because a store instruction by default has a second source. So, store instruction has two sources pretty much; it does not have a register destination and the store instruction has immediate as well. Again you can see that this is a consequence of the special case that was done for store instructions. So, moment we have special cases we require a lot of logic in the pipeline to handle these special cases. So, that is the reason special cases should be avoided as far as possible, but when we did make an exception for the store instruction we did not have a choice.

So, coming back if A is immediate field is set to 1 this means that it does not have a second register source. So, we can happily return false. So, this means that the second register source you know does not exist.

Now that we have at this point asserting that the second register source does exists we set src 2 to a dot s rs 2. And if it is a store instruction we set src 2 to a dot rt.

(Refer Slide Time: 28:01)

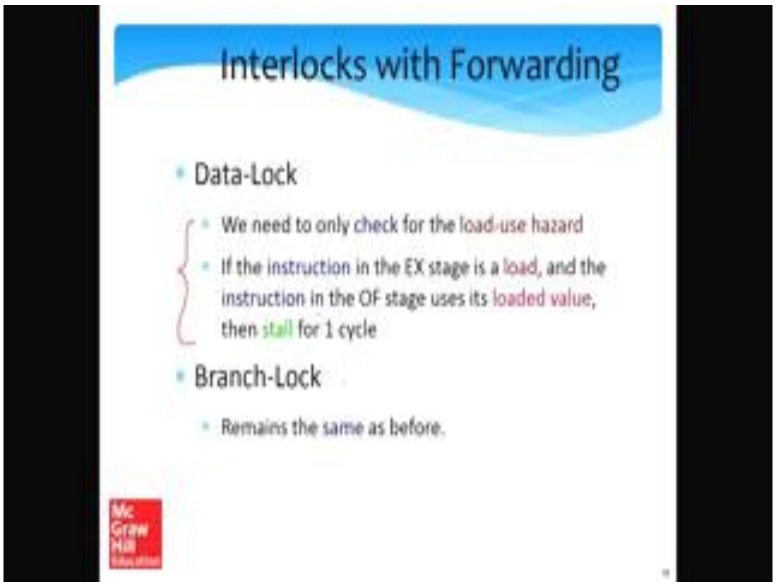


```
/* Set the destination */
dest ← [B].rd
if [B].opcode = call then
    dest ← ra
end
/* Detect conflicts */
if src2 = dest then
    return true
end
return false
```

RAW

Then we set the destination to b dot rd, and what we have done in the previous slide which is that in the previous algorithm right; algorithm 6 is that if B is off code is a call instruction then we set the destination to the return address register. This is because the call instruction writes to the return address register. Hence forth detecting conflicts is very very easy, we say src 2 is equal to dest then we return true means the conflict exists. There is a read after write dependency with registers involved, otherwise we return false. So, this was the algorithm for the second source register.

(Refer Slide Time: 28:49)



Interlocks with Forwarding

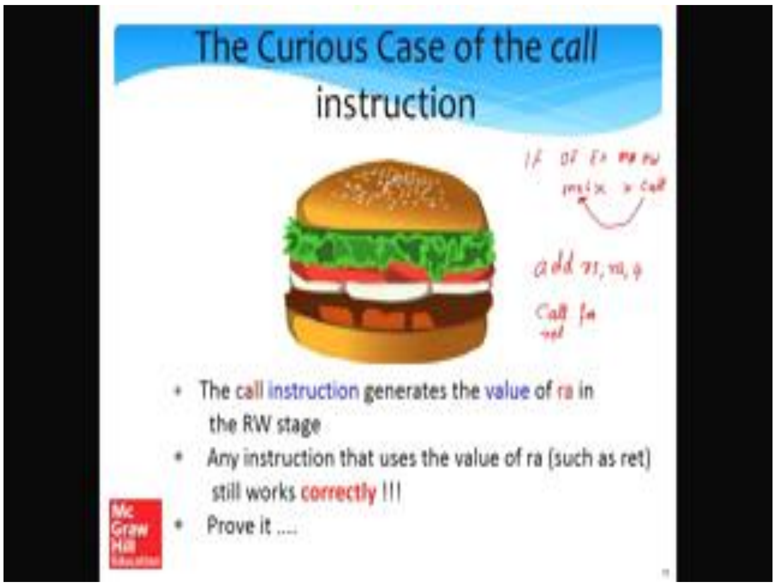
- Data-Lock
  - We need to only check for the load-use hazard
  - If the instruction in the EX stage is a load, and the instruction in the OF stage uses its loaded value, then stall for 1 cycle
- Branch-Lock
  - Remains the same as before.

Let us now look at the relationship between interlocks and forwarding, because we will still require interlocks so we will still require a data lock. So, we will require it to check for the load use hazard. So, the load use hazard exists which means that in consecutive stages we have a load first and then an instruction that uses the result of the load an ALU instruction at EX stage which uses the results at the load. We will have a load use hazard. So, it is all needs to be introduced.

The way we shall do this is that if the instruction in the EX stage is the load, and instruction in the OF stage uses its loaded value and it will use it in also in the EX stage, then we need to stall for 1 cycle. So, this is the load use hazard. So, this is very easy to detect and we can use a same conflict detection circuitry to find the existence of a load use hazard and then use the same circuitry to stall for one cycle.

The branch lock remains the same as before where we use option number three. And if the branch is taken we cancel nullify convert to bubbles the instructions in the IF and OF stages.

(Refer Slide Time: 30:11)


The Curious Case of the *call* instruction

- The call instruction generates the value of ra in the RW stage
- Any instruction that uses the value of ra (such as ret) still works correctly !!!
- Prove it ....

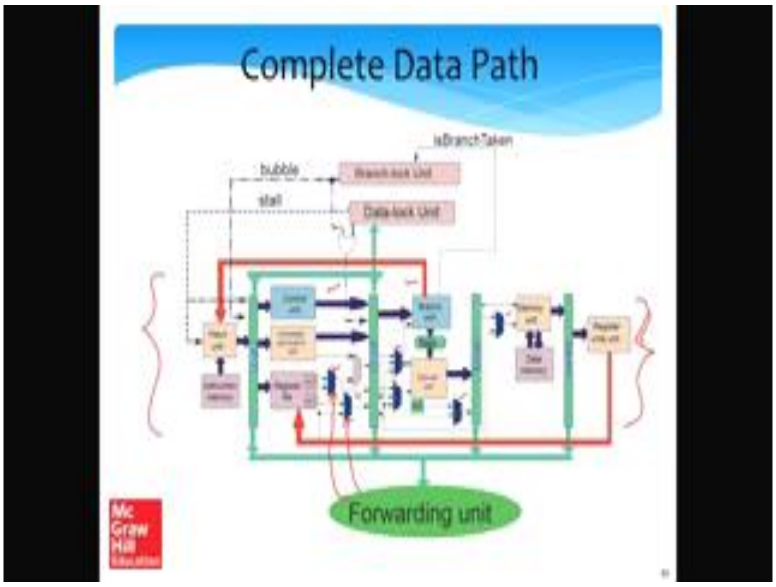
IF OF EX MA RW
add r1, ra, 4
Call fn
ret
Mc Graw Hill Education

Let us now look at the curious case of the call instruction I have a burger over here, because there is some food for thought. So, the call instruction generates the value of ra the return address in the RW stage; that is when we add pc equals sorry ra is pc plus 4. Say any instruction that uses the value of ra, you can always you know nothing stops us from writing an instruction of this type add r 1 ra 4, right. So, any instruction that uses the value of ra such as ret or you know instruction of this type you can always define course snippet of this type called dot fu and ret.

So, any such instructions still work correctly why right you need to prove it. I think I will give an outline of the proof. So, the call is taken branch. If call is a taken branch and I just draw a pipeline diagram, what will happen is that in a given cycle with the call instruction is over here there will be no instructions there will be bubbles in the previous 2 stages right, bubbles in EX and MA. The reason being that it is a taken branch so basically the two instructions immediately after it will get converted to bubbles.

So the earliest point; sorry the latest point at which instructions can be there is at this point. Here you can have an instruction which will use the valve that the call instruction will produce. And this can use the RW to OF forwarding path, no problem in that. And since you know we are only adding pc plus 4 that is a very simple additions a very fast addition. So, in quickly to the addition and we can forward it from RW to OF. So, any instruction which will be there right the closest it can come to call instruction is it will come into the OF stage. So, it can get the value from the RW to OF forwarding path, as a result it will get the correct valve.

(Refer Slide Time: 32:43)



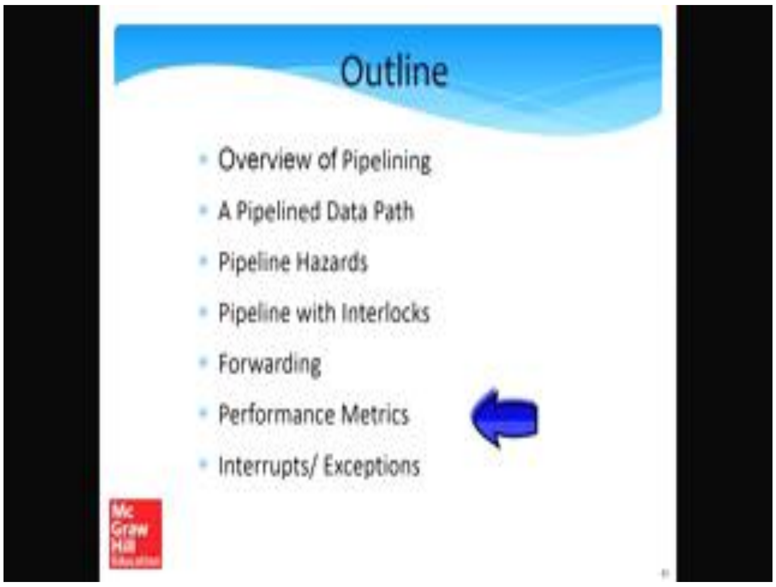
So, let us take a look at the complete data path again with forwarding and you know our pipeline in full glow and glory. So, this is the basic pipeline that we had. So, we have gradually been building on it we added our pipeline registers, subsequently we added a forwarding multiplexers; 6 of them. So, the forwarding multiplexers have to be controlled, so the way we will do it is that all the pipeline registers will send information about their instructions to the forwarding unit, the forwarding unit will compute the control signals of all these multiplexers. Have not shown them you know in the interest of edibility, but pretty much there are lines from these two each of the multiplexers right to control it.

Similarly we will have a data lock and branch lock circuitry exactly the way we had before, no difference in that. It is just at the data lock logic changes it becomes slightly more simpler. That is the reason I have not taken inputs from the other 2 stages into the data lock unit, its only the IF-OF stage in the OF-EX stage because this instruction this instruction, these two matter for computing if there is a load use hazard or not. So, that is the reason inputs have been taken only from these two registers, then the stall and the bubbles circuitry remains the same.

Just to summarize at this particular point: we have a fairly complicated processor on our screen, so it is a five stage pipeline, it is called an five stage in order pipeline because instructions are not overtaking each other they move in program order. And we have pipelining, we have interlocks, we have forwarding, almost everything that is required to get a high performance in order implementation right, in order processor implementation. So, this is great it is just that either it is fairly complicated. So, you will see in the website of the book that we at least are distributing the logisim and VHTL models of the base processor. So, it does not have pipelining and interlocks on forwarding. But normally as assignments we ask students to create a pipeline with interlocks pipeline stages and forwarding, and they are able to do it; it is not a lot of work in a month or two it is possible to do it and it is a lot of fun.

So, I would advice all the readers and listeners, readers of the book and listens of this video that you know the right way and the most effective way of learning how to design a pipeline processor is to actually build one. So, you can take a base processor that we have built in VHTL that does not have pipelining from the books website and then add pipelining to it. It is a lot of fun.

(Refer Slide Time: 35:52)

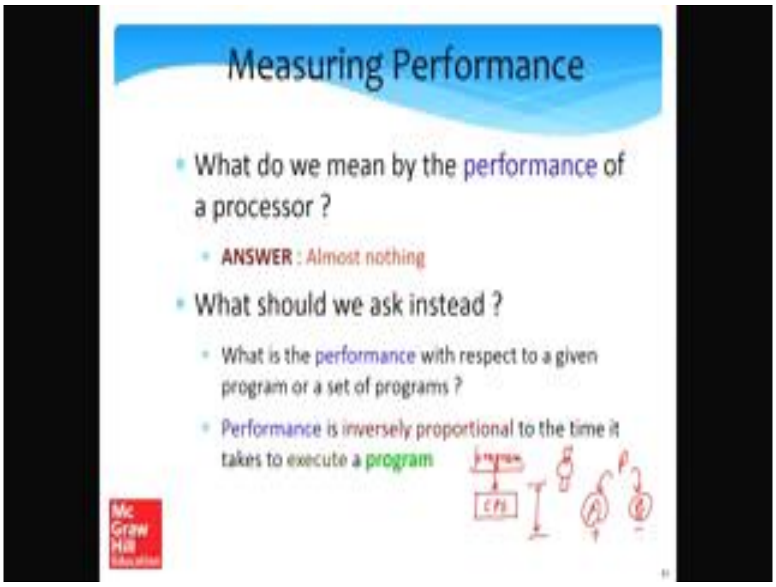


Outline

- Overview of Pipelining
- A Pipelined Data Path
- Pipeline Hazards
- Pipeline with Interlocks
- Forwarding
- Performance Metrics
- Interrupts/ Exceptions

So, now that we have seen forwarding we can move on to the performance matrix. So, how to find out if a pipeline is fast or slow.

(Refer Slide Time: 36:01)



Measuring Performance

- What do we mean by the performance of a processor ?
  - ANSWER : Almost nothing
- What should we ask instead ?
  - What is the performance with respect to a given program or a set of programs ?
  - Performance is inversely proportional to the time it takes to execute a program

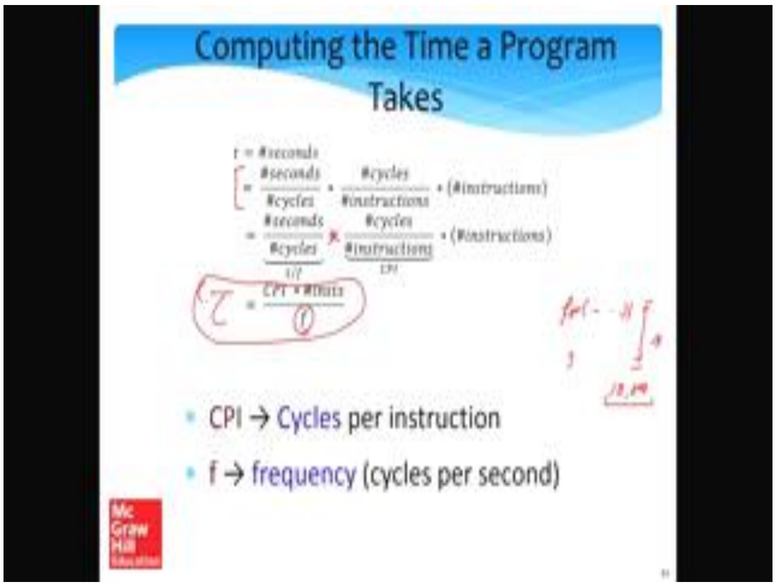
Program
CPU

So, what we typically mean by the performance of a processor? The people will say one processor is faster than the other. So, what do we mean by that? Well, the answer is almost nothing. So, we should be asking a different question because of our problem in our previous question. We should be asking what is the performance of the processor with respect to a given program or a set of programs, because it is possible that processor A is fast for a certain program and slow for another program and vice versa. Processer B is faster or some set of programs and slower for another set of program. So, we should ask these questions in the context of a set of programs.

Second, how do we define performance? So, let us define performance very simply. As a quantity that is inversely proportional the time it takes to execute a program. So, what we do is that we take a CPU, take a processor and we give it a program to execute and simultaneously we start a stopwatch. I am a bad drawer so hopefully this is showing a watch. So, we start a stopwatch and so the program starts and program ends, and the total amount of time it takes that is indicative or the performance of the processor. So, the faster is the processor the slower; sorry the faster is the processor the less is the time it takes to execute a program. And we essentially refer to a performance as the quantity which is inversely proportional. So, the time it executes to takes to execute a given program.

So, even typically here many other things clock frequency and number of course and so on. So, performance is none of that performance is simply a quantity that is inversely proportional till the time it takes to execute a program. If you want to compare a processor A and processor B then you take processor A and B give both of them the same program to execute; the one that is faster has higher performance, one that is slower has lower performance.

(Refer Slide Time: 38:36)



Computing the Time a Program Takes

$$
\begin{aligned}
\tau &= \#seconds \\
&= \frac{\#seconds}{\#cycles} * \frac{\#cycles}{\#instructions} * (\#instructions) \\
&= \underbrace{\frac{\#seconds}{\#cycles}}_{1/f} * \underbrace{\frac{\#cycles}{\#instructions}}_{CPI} * (\#instructions)
\end{aligned}
$$

$$\tau = \frac{CPI * \#insts}{f}$$

- CPI → Cycles per instruction
- f → frequency (cycles per second)

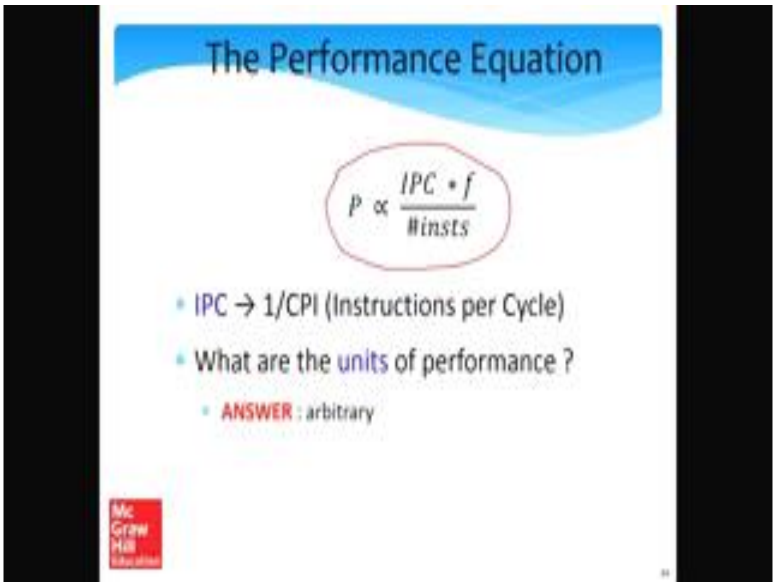
So, let us do a little bit of math and let us derive what is called a performance equation which is very very important. It is the crux of the entire pipeline math. So, let us define the time that a program takes as in number of seconds'. So, then let us do a little bit of very very simple algebra. Let us say number of seconds divided by the number of the clock cycles multiplied by the number of clock cycles divided by the number of instructions. These are number of instructions the program you know actually executes on the pipeline.

Let us assume this as the number of dynamic instructions. So, let me just quickly differentiate means static and dynamic instructions. Assume that there is for loop. So, it is possible that the in the body of the for loop inside the binary of the program there are only 10 instructions. There is the for loop is running for a 1000 times the number of instructions a processor is seeing is 10000 right; 10 times 1000. So, when we talk about number of instructions we refer to this quantity. It is the number of instructions is the processor sees, there are called dynamic instructions. And then we multiply by the number of instructions.

So, what we see is that it is a simple algebraic thing that we have done here. So, the number of cycles will cancel out and number of instructions will cancel out; sorry this should be a multiplication over here. So, number of seconds divided by number of cycles is actually 1 by the clock frequency, because the clock frequency is the number of cycles per second so this is 1 by the clock frequency 1 by f. So, f comes in the denominator here.

The number of cycles divided by the number of instructions is called the CPI; cycles per instructions this is very important. And the number of instructions will refer to as (Refer Time: 40:34) So, pretty much the time that a program takes is CPI the number of clock cycles per instruction multiplied by the total number of dynamic instructions in the program that the processor sees divided by the frequency. As you can see this is the time a program takes it is not just the frequency there are other variables. So, let us define performance which is a quantity which is proportional to 1 by the time.

(Refer Slide Time: 41:05)



The Performance Equation

$$P \propto \frac{IPC \cdot f}{\#insts}$$

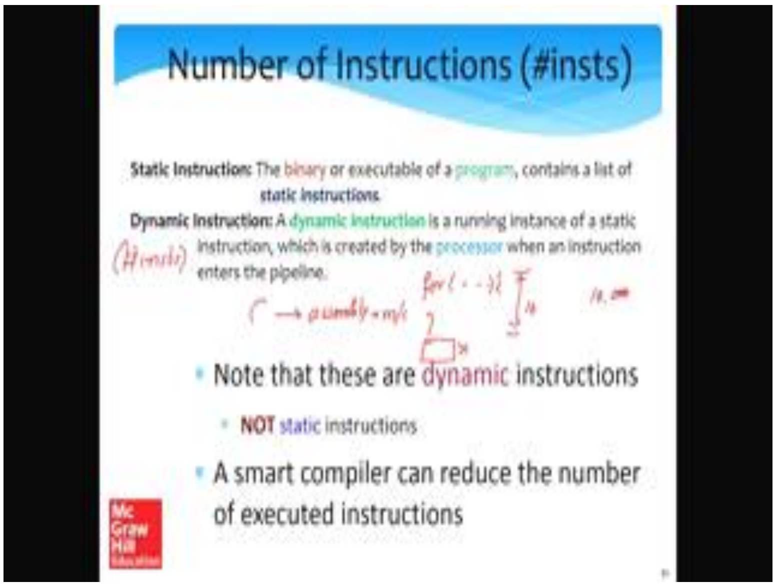
- IPC → 1/CPI (Instructions per Cycle)
- What are the units of performance ?
  - ANSWER : arbitrary

So, performance is 1 by this, so we will see this will basically be f divided by CPI times number of instructions. Let us refer to 1 by CPI is IPC; IPC is instructions per cycles. So, the performance will be proportional to IPC instructions per cycle times the frequency divided by the number of instructions.

So, this is a very very important equation, it is very simple it is very important and most of the students get it wrong that makes a even more important. And still we will appreciate this equation in many of its forms; well let us at least try to memorize it for the time being. So, this says that the performance of a processor with respect to a given program is the IPC number of instructions on an average that we execute per cycle multiplied by the frequency divided by the number of dynamic instructions in the program. So, it is three quantities determining the performance not one, but three. That is important to remember. And what are the units of performance, well they are arbitrary so you can think of it as 1 by seconds but it is not important it is arbitrary. As long as the definition is being used in the same manner it is being used consistently.

So, let us again remember the performance equation, which is performance is proportional to IPC; instructions per cycle multiplied by the frequency divided by the number of instructions.

(Refer Slide Time: 42:50)



Number of Instructions (#insts)

Static Instruction: The binary or executable of a program, contains a list of **static instructions**.

Dynamic Instruction: A dynamic instruction is a running instance of a static instruction, which is created by the processor when an instruction enters the pipeline.

- Note that these are dynamic instructions
  - **NOT** static instructions
- A smart compiler can reduce the number of executed instructions

So, the number of instructions numinsts; let us see what determines it. So, let us once again for the benefit of the listener differentiate between static instructions and dynamic instructions. So, a static instruction is something like this; the binary or the executable of a program will always contain a list of static instructions. As we had shown in the for loop if we have 10 instructions inside the for loop, then it is essentially 10. But now if the for loop runs for 1000 hydrations the total number of instructions that the processor sees is 10000.

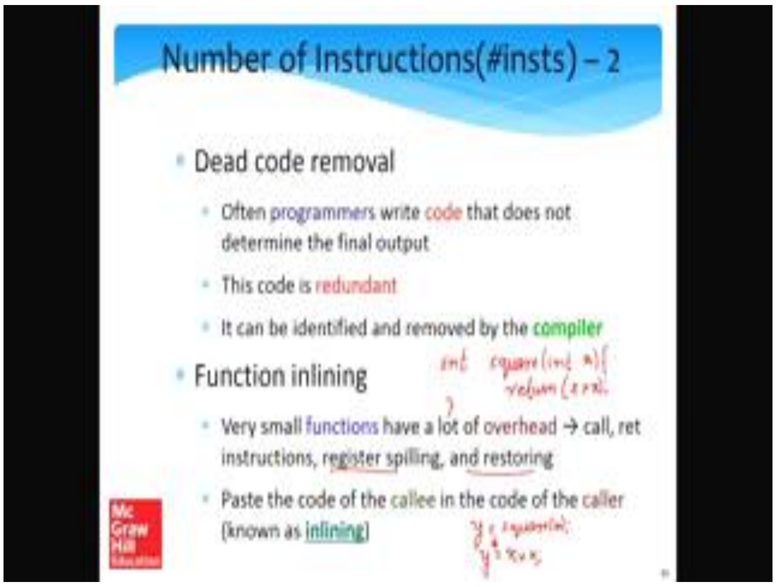
And each 10000 of them are a dynamic instruction which is pretty much running instance of a static instruction and this is created by the processor when the instruction enters the pipeline. So, even if there are 10 instructions in the code of the for loop of a for loop executes a 1000 times the 10000 10 times 1000; 10000 instructions are pretty much being created by the processor. So, dynamic instruction which is exactly numinsts is a number of instructions that the processor actually sees.

So, basically how do we you know have exercise some degree of control over dynamic instructions. Well, the dynamic instructions are pretty much a function. So, dynamic instructions you know with different kinds of pipelines and so on is still remain the same. it is a function of the core. So, what do we do we first write C code which is converted to assembly code and assembly code has a one to one correspondence with machine code I will just write mc to mean machine code.

So, smart compiler can essentially reduce the number of dynamic instructions; the number of executed instructions by essentially generating code which is more compact. Instead of generating codes which requires a lot of instructions a smart compiler can do a lot of things to ensure that the generated code is fairly compact and we also do not have ret code. For example, lot of time you know programmers would write some pieces of code which are actually not used. So, the compiler can remove all of that; that is one thing that the compiler can do.

Other things that the compiler can do are if let us are any points some computations has been repeated it can avoid that. In the similar manner when we are generating assembly code where the compiler needs to keep in mind that reducing the number of instructions will actually really help us.

(Refer Slide Time: 45:58)



Number of Instructions(#insts) – 2

- Dead code removal
  - Often programmers write code that does not determine the final output
  - This code is redundant
  - It can be identified and removed by the compiler
- Function inlining
  - Very small functions have a lot of overhead → call, ret instructions, register spilling, and restoring
  - Paste the code of the callee in the code of the caller (known as inlining)

```
int square(int x){
    return (x*x);
}
```

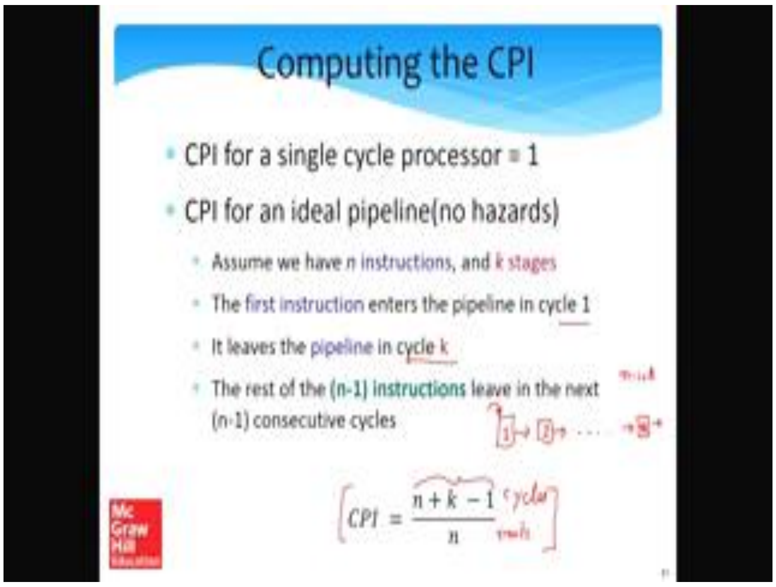
```
y = square(x);
y = x*x;
```

So, let us look at some of these schemes once again. So, dead code removal means that program is often write a lot of code which does not determine the final output, so it can run but it has no effect. So, this code is redundant so this can be identified and removed by the compiler. Often what programmers also do is they will create very very small functions, and small functions have a lot of overhead.

We need to call the function, we need to return from the function, we need to spill registers and restore registers. For example, I can define a function of this type; instead of that it is a much better idea we can you know write a piece of code which is y is equal to square of x. It is a much much better idea to replace this with code snippet which says y equals x time x. So, we will be saving call instruction ret instruction many of these registers spill and restore instructions. And we will be able to say fair amount of code if we sort of remove this function and replace it with the logic of the function.

So this is called function in lining, which means that we paste the code of the colane in the code of the collar. So, compilers will do function in lining by default, and function in lining seriously helps us reduce the lines of code especially lines of codes which in the sense are not required. And programmers never the less do declare very very small functions and this has a performance over head. And inlining will reduce to some to a certain extent, this is an automatically by compilers.

(Refer Slide Time: 48:09)



Computing the CPI

- CPI for a single cycle processor = 1
- CPI for an ideal pipeline(no hazards)
  - Assume we have *n* instructions, and *k* stages
  - The first instruction enters the pipeline in cycle 1
  - It leaves the pipeline in cycle k
  - The rest of the (n-1) instructions leave in the next (n-1) consecutive cycles

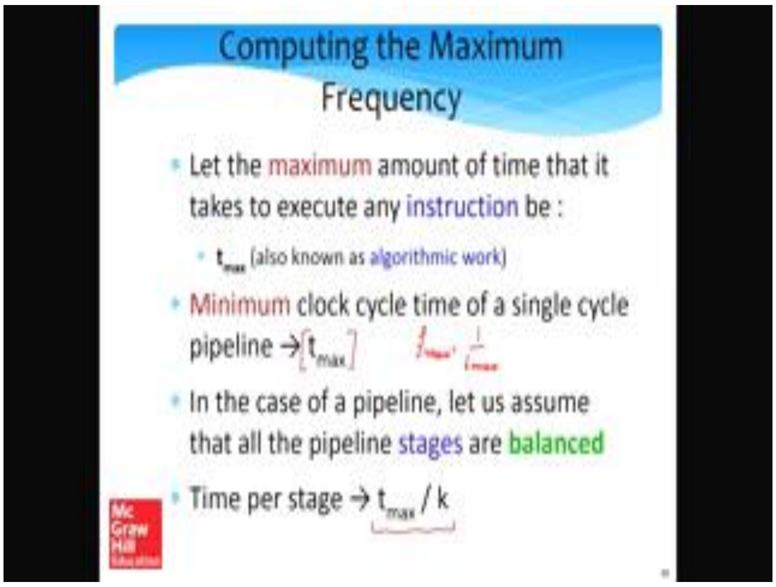
$$\left[CPI = \frac{\overbrace{n+k-1}}{n} \right]$$

Let us now take a look at the CPI. So, the CPI for the single cycle processor is 1, because all instructions take one cycle to execute. Now let us consider the CPI for an ideal pipeline which does not have any hazards. Let us assume we have n instructions then we have k stages. So, the first instruction enters the pipeline in cycle number 1 and will leave the pipeline cycle k because there are no hazards and thus no stalls. The rest of the n minus 1 instruction will leave in the next n minus 1 consecutive cycles.

So, the total number of cycles that we spend is n minus 1 plus k, because you know we spend the first k minus 1 cycles not doing anything. So, for n instruction to pass through a k stage pipeline it will take n plus k minus 1 cycles. And since there are n instructions the CPI will be n plus k minus 1 divided by n. So, this is also very important equation and should be appreciated and understood to its full extent.

The CPI in this case if a pipeline has k stages; let me maybe draw it in one more sense and show. In the first cycle if the first instruction enters the first stage then at the end of the kth cycle it will leave the pipe line. Subsequently, in every single cycle one instruction will leave the pipeline. So, then the total amount of time required will be n minus 1 plus k same as n plus k minus 1. So, CPI will be n plus k minus 1 divided by n.

(Refer Slide Time: 50:11)



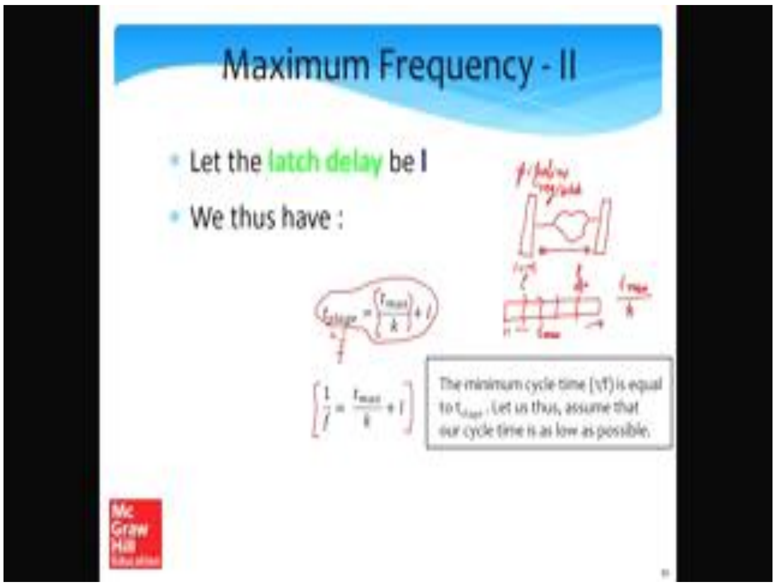
Computing the Maximum Frequency

- Let the maximum amount of time that it takes to execute any instruction be :
  - t_{max} (also known as algorithmic work)
- Minimum clock cycle time of a single cycle pipeline $\rightarrow \boxed{t_{max}}$ $f_{max} = \frac{1}{t_{max}}$
- In the case of a pipeline, let us assume that all the pipeline stages are balanced
- Time per stage $\rightarrow \underbrace{t_{max} / k}$

Let us now compute. We have looked at the CPI, so what is the third component? It is the frequency. So, let the maximum amount of time that it takes to execute any instruction within the processor the t max and let us call this refer to this as the amount of algorithmic work. Let us again assume that we will not in the case of a single cycle pipeline, in the case of single cycle processor which has only no pretty much 1 stage we will not divide the execution of an instruction into two clock cycles right because we do not have a pipeline.

So, in this case the minimum clock cycle time that we can afford is t max, because t max is pretty much the maximum time it takes to process a single instruction. So, what will be the maximum possible frequency f max will be 1 divided by t max. In the case of a pipeline let us assume that all the pipeline stages are balanced which means that they do the same amount of work, it takes the same amount of time to flow through them. So, the time per stage in the total amount of work you know that you do the total time it takes for an instruction to finish its execution is t max of there are k pipeline stages the time per stage will be t max divided by k.

(Refer Slide Time: 51:51)



Maximum Frequency - II

- Let the latch delay be l
- We thus have :

$$t_{stage} = \left(\frac{t_{max}}{k}\right) + l$$

$$\left[\frac{1}{f} = \frac{t_{max}}{k} + l\right]$$

The minimum cycle time (1/f) is equal to t_{stage}. Let us thus, assume that our cycle time is as low as possible.

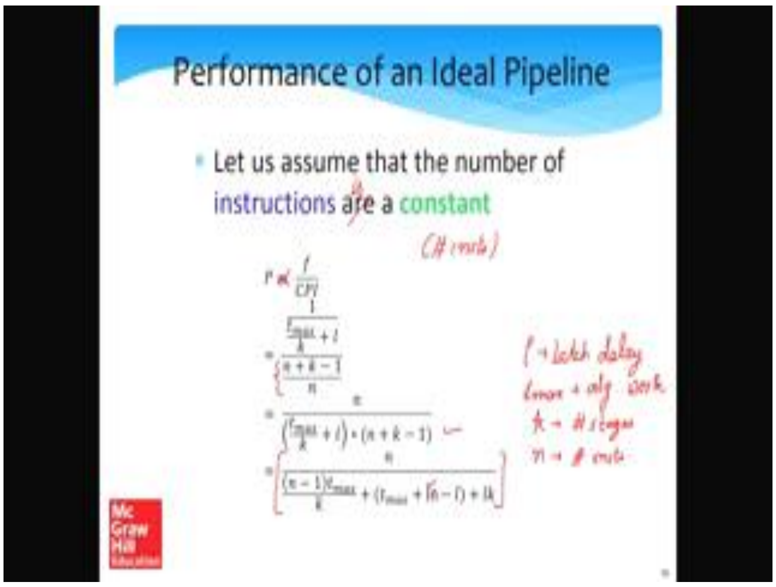
Furthermore, so in pipelines we have pipelines. So the typical structure is like this; that we have a pipeline register slash latch. Then we have some logic and then again we have one more pipeline latch. So, a pipeline register and a pipeline latch is the same thing. So, pretty much they delay is that the time it takes to pass through this logic, right the logic can be add or multiplier MA stage OF stage does not matter and the delay of this you know passing through the pipeline register.

So, let us the latch delay be l. As I said in the case of pipeline a pipeline register and a pipeline latch as the same thing otherwise they are not the same, but we will interchangeably use the term pipeline register and pipeline latch or just a latch; so let the latch delay be l. So, the time per stage is t max divided by k which is the amount of work that you need to do in the logic part right t max by k plus l. So, the minimum cycle time that you can afford is t stage. As a result if you want to set the largest possible frequency right run the processor as fast as possible we shall thus have 1 by f; t stage is equal to 1 by f. Where f is the largest possible frequency at which you want to run the processor.

So, 1 by f will be equal to t max by k plus l. Let me explain this formula once again. So, let us assume that you know single cycle processor which is not pipeline the maximum time it takes for an instruction to go from fetch part to the right part is t max. Now, if you divide this in to k parts the time per stage will be t max by k; also when we are dividing it into k part so the division is not always perfect. Basically the signal also passes through a pipeline latch.

The pipeline latch has a certain delay associated with it, so let us refer to it as l. So, the total delay is t max by k which is the amount of time it takes for the instruction to pass through the logic part of the pipeline, and the latch which is t max by k plus l. So, if we assume this is the minimum time you know it will take for one instruction to complete 1 stage. So, we assume that the frequency is the maximum possible value as large as possible we will have 1 by f is equal to t max by k plus l. So, this will give us the highest possible frequency.

(Refer Slide Time: 54:55)



Performance of an Ideal Pipeline

- Let us assume that the number of instructions ~~are~~ is a constant (# insts)

$$P \propto \frac{f}{CPI} = \frac{\frac{1}{\frac{t_{max}}{k} + l}}{\frac{n+k-1}{n}} = \frac{n}{\left(\frac{t_{max}}{k} + l\right) * (n+k-1)} = \frac{n}{\frac{(n-1)t_{max}}{k} + (t_{max} + l(n-1)) + lk}$$

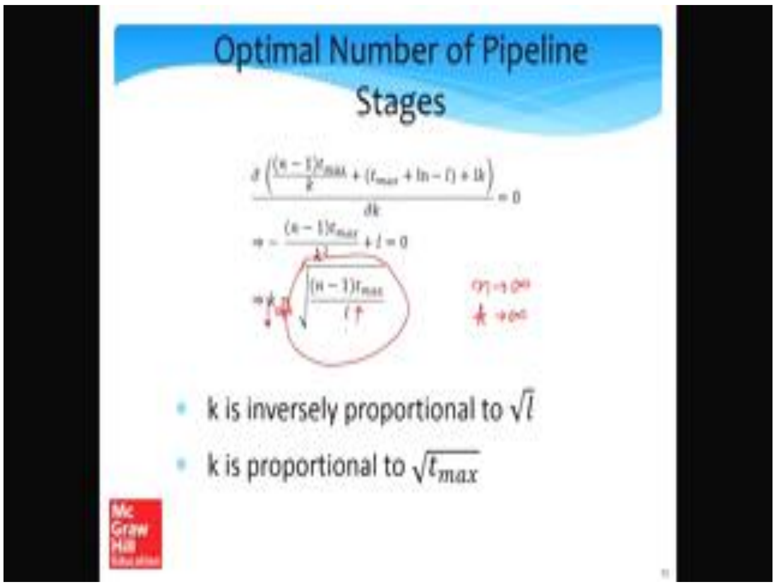
l → latch delay
t_{max} → alg work
k → # stages
n → # insts

So, let us now work and find the performance of an ideal pipeline. So, let us assume that the number of instructions is a constant, so grammatical mistake over here. So, let us assume that this number is a constant and let us also try to in a sense remove the proportionality sign, so the interest of readability. The proportion of (Refer Time: 55:25) is still there, but since they are removing number of (Refer Time: 55:30) from the equation we are assuming that it is a constant for all kinds of pipelines. Because this is a function of the program it has nothing to do with the pipeline.

We will have the performance, well equal to this is I am sort of taking little bit liberty here should be proportional does not matter. So, it is f divided by CPI or f times IPC. So, the f that we have computed is 1 by t max by k plus l and the CPI that we had computed is n plus k minus 1 divided by n. A little bit of algebraic manipulation gives us this expression and a little bit of algebraic manipulation then gives us this expression.

Let us keep going forward. In this case what do we see, we see that latch delay is here n is the; so let us look at what these terms mean. L is the latch delay, t max is the algorithmic work or the total time it takes for the instruction to go from one end of the pipeline to the other let us call it the algorithmic work, k is the number of stages; that is all and n is the number of instructions.

(Refer Slide Time: 57:07)



Optimal Number of Pipeline Stages

$$\frac{\partial\left(\frac{(n-1)t_{max}}{k} + (t_{max} + l(n-1)) + lk\right)}{\partial k} = 0$$

$$\Rightarrow -\frac{(n-1)t_{max}}{k^2} + l = 0$$

$$\Rightarrow k = \sqrt{\frac{(n-1)t_{max}}{l}}$$

$n \to \infty$

$k \to \infty$

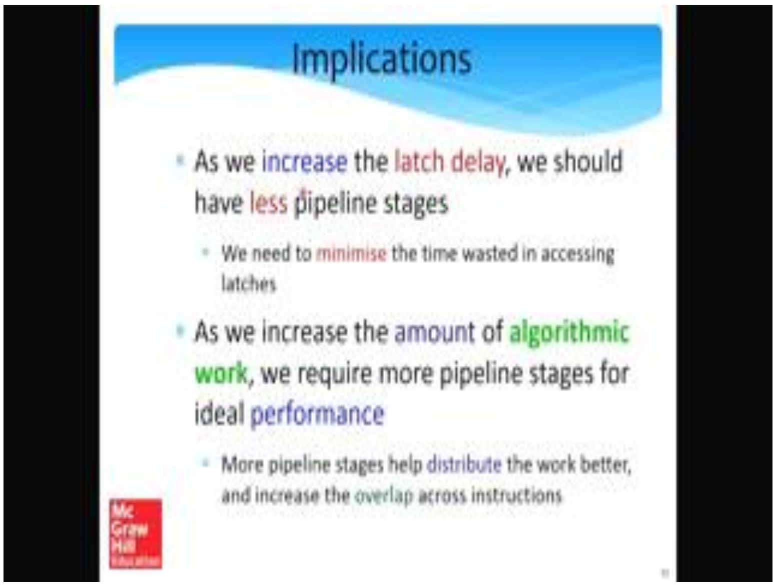
- k is inversely proportional to $\sqrt{l}$
- k is proportional to $\sqrt{t_{max}}$

So let us do one thing, let us try to differentiate this equation with respect to k to find the optimal number of pipeline stages. So, when we take a derivative with respect to k this is the equation that we get and this is the final solution that we get, this is the optimal number of pipeline stages.

As you see a pipeline with no hazards the number of optimal number of pipeline stages is dependent on the number of instructions. In fact, as the numbers of instructions tend to infinity the number of optimal pipeline stages will also tend to infinity, which is not really very practical thing but since we have assumed an absolutely ideal pipeline we are getting very ideal yet unrealistic results. Also the number of optimal pipelines stages in this case k you know the k opt. K optimal is proportional to the square root of t max square root of total amount of algorithmic work, and it is inversely proportional to the latch delay. So, the higher is the latch delay, as the latch delay increases the optimal pipeline stages actually decrease. And lower is the latch delay we can have more pipeline stages.

So, given the fact that we have appreciated this let us move to the case of non-ideal pipeline which actually has hazards and the interlocks and stalls and bubbles and everything that will give us a much more realistic formula to work with. And we will use exactly the same methods of analysis.

(Refer Slide Time: 58:51)

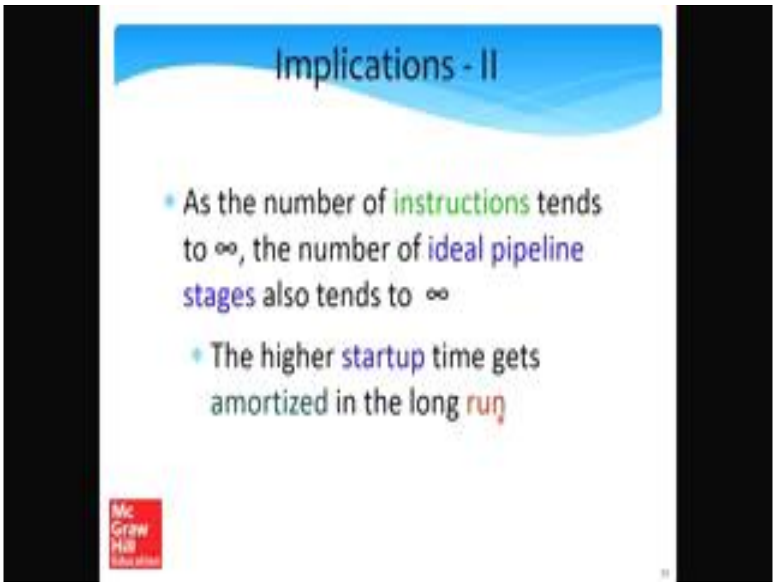


Implications

- As we increase the latch delay, we should have less pipeline stages
  - We need to minimise the time wasted in accessing latches
- As we increase the amount of algorithmic work, we require more pipeline stages for ideal performance
  - More pipeline stages help distribute the work better, and increase the overlap across instructions

Before moving to a non-ideal pipeline we will just spend quick half a minute discussing the implications of all the math that we did. So, the first thing is that as we increase the latch delay we should have you know lesser pipeline stages right a lower number of pipeline stages. This is because we need to minimize the time wasted in passing through latches, accessing latches. As we increase the amount of algorithmic work we require more pipeline stages for ideal performance, this is because more pipeline stages will help distribute the work better and also consequently increase the overlap across instructions between instructions.

(Refer Slide Time: 59:38)

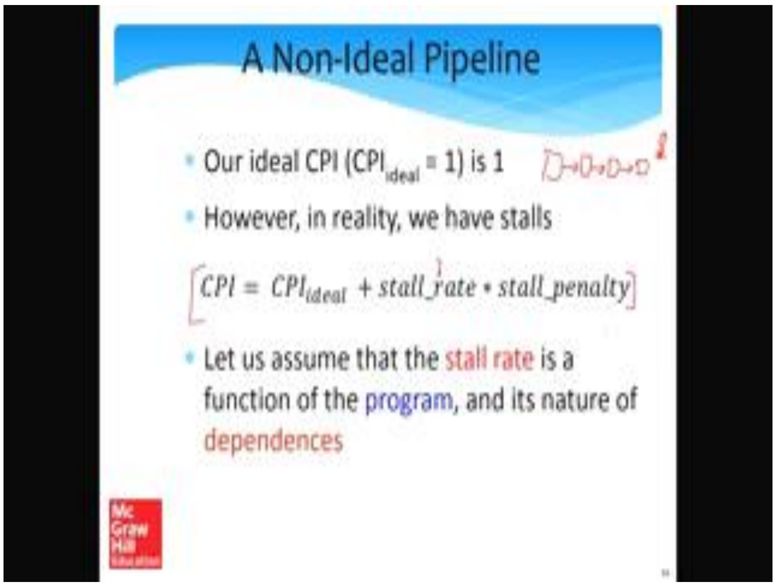


Implications - II

- As the number of instructions tends to ∞, the number of ideal pipeline stages also tends to ∞
  - The higher startup time gets amortized in the long run

The second is that as the number of instructions tends to infinity the number of ideal pipeline stages also tends to infinity, and the higher start up time will get you know sense amortized in the long run. Well, again this is for ideal pipeline so let us quickly move to a non-ideal scenario; real realistic scenario.

(Refer Slide Time: 59:58)



A Non-Ideal Pipeline

- Our ideal CPI ($CPI_{ideal} = 1$) is 1
- However, in reality, we have stalls

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

- Let us assume that the stall rate is a function of the program, and its nature of dependences

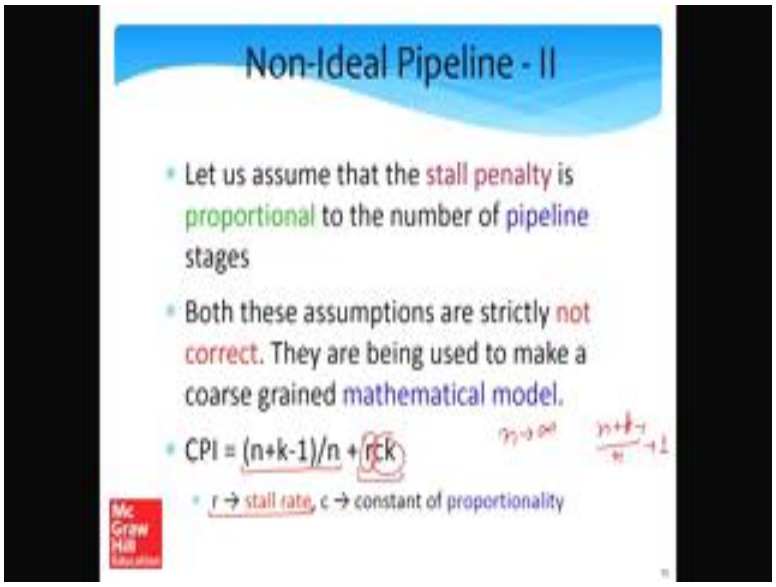
So ideal CPI; CPI ideal is one when we have no stalls, in reality we do have stalls. So, ideal CPI is one you know when number of instructions also tends to infinity, but in reality we do have stalls. So the CPI, the clock cycles per instruction will be the clock

CPI ideal which in this case is 1 plus the stall rate; the stall rate is the number of stalls per instructions times the stall penalty it will simply be a vetted mean. So this is also a very important formula which should be appreciated and in a sense remembered for later use.

So, this is trying to say that in any non-ideal processor the CPI or the clock cycles per instruction is equal to CPI ideal. CPI ideal is basically the CPI on a pipeline with no stalls. So, there every cycle one instruction will come out of the pipeline. Since every cycle one instruction is coming out this clock cycles per instruction; so assume that we have a k stage pipeline right, at the end of the pipeline we have a person standing here. In an ideal pipeline you would see that every cycle one instruction is coming out. So, the person would compute the CPI ideal to be equal to 1.

Now assume that there are some stalls; there are some stalls the person here can also find that given an instruction what is the probability that it will suffer from a stall; that is point number 1. And point number 2 is that fth shuffles from a stall. So, how many cycles does it stall right; that is a stall penalty. So, we need to make some assumptions here for a case stage pipeline. So, let us make some standard assumptions which are being made in other texts as well. So, let us assume that the stall rate is the function of the program and its nature of dependency.

(Refer Slide Time: 62:10)



Non-Ideal Pipeline - II

- Let us assume that the stall penalty is proportional to the number of pipeline stages
- Both these assumptions are strictly not correct. They are being used to make a coarse grained mathematical model.
- CPI = $\underline{(n+k-1)/n} + \boxed{rck}$
  - $r \rightarrow$ stall rate, $c \rightarrow$ constant of proportionality

$n \to \infty$
$\frac{n+k-1}{n} \to 1$

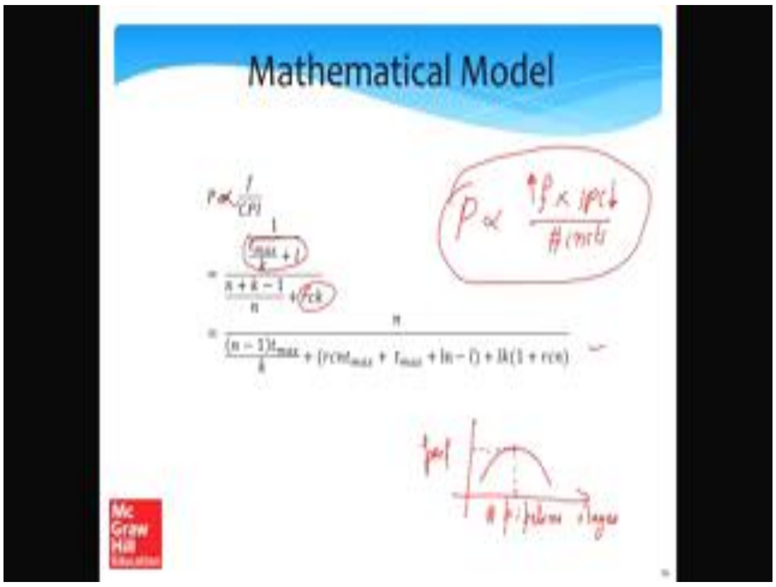
Let us further assume that the stall penalty is proportional to the number of pipeline stages; it is a reasonable assumption to make. The reason that we say that is because you know if we have a 20 stage pipeline for a load use hazard will not stall for 1 cycle, it will stall for some number of cycles which is proportional to 20. We stall from one cycle in a high stage pipeline, but we divide each of these single stages into foremost stages, hopefully we expect that instant of stalling from one cycle in a 20 stage pipeline you might stall for 3-4 cycles in a ideal scale.

So, both these assumptions are exactly not correct hence they cannot be mathematically proven, but they are sort of empirically observed and people think that it is more or less correlates with what they have seen in real experiments and it also correlates with their common sense. We will use this to make a mathematical model. We will see the CPI is the CPI if that we have observed before which is n plus k minus 1 by n. So, this is pretty much the ideal CPI where we are assuming that n is less than infinity for a small value of n, because this number as n tends to infinity this number in as n tends to infinity; n plus k minus 1 by n will tend to 1.

So, let us assume that for even for small values of n this does capture your ideal CPI. Furthermore let r be the stall rate, so we are saying that the stall rate is dependent on the instruction it is not dependent on anything else. And since it is dependent on instruction it will remain independent of the pipeline so this is one constant. And we see that the stall penalty is c times k; where k is the number of pipeline stages and c is the constant of proportionality.

As a result the stall rate which is again independent of the number of pipeline stages and the stall penalty which is proportional to the number of pipeline stages ck. So, stall rate times stall penalties rck.

(Refer Slide Time: 64:38)



Mathematical Model

$$P \propto \frac{1}{CPI}$$

$$= \frac{\frac{1}{\frac{t_{max}}{k} + l}}{\frac{n+k-1}{n} + rck}$$

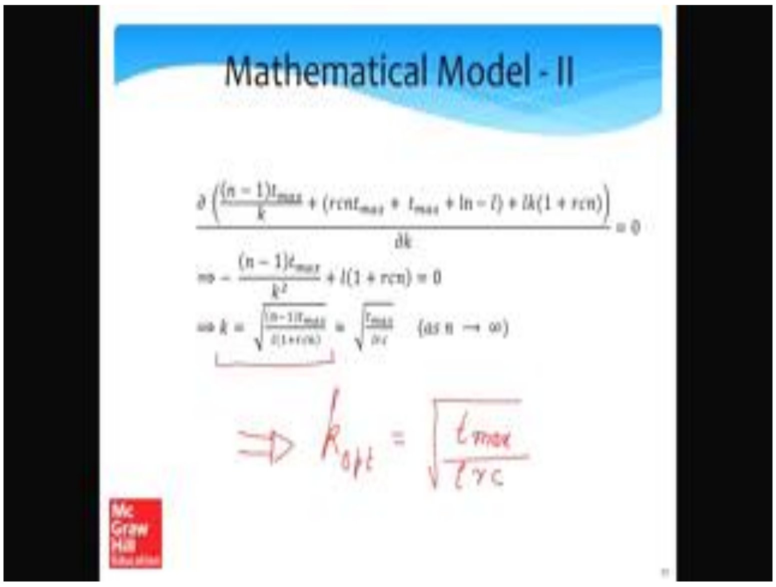
$$= \frac{n}{\frac{(n-1)t_{max}}{k} + (rcnt_{max} + t_{max} + ln - l) + lk(1 + rcn)}$$

$$P \propto \frac{\uparrow f \times ipc \downarrow}{\# insts}$$

perf
# pipeline stages

So, given this equation let us modify our previous equation. So, we are assuming p is equal to or is proportional to take a little bit of liberty here. And since number of instructions remains constant in the design space we have removed it from our equation. So, f is still 1 by t max by k plus l, but the CPI equation has changed. So, this is the additional factor that has come. So, let us further do some algebra and simplify this expression; so we come here.

(Refer Slide Time: 65:12)



Mathematical Model - II

$$\frac{\partial\left(\frac{(n-1)t_{max}}{k} + (rcnt_{max} + t_{max} + ln - l) + lk(1 + rcn)\right)}{\partial k} = 0$$

$$\Rightarrow -\frac{(n-1)t_{max}}{k^2} + l(1 + rcn) = 0$$

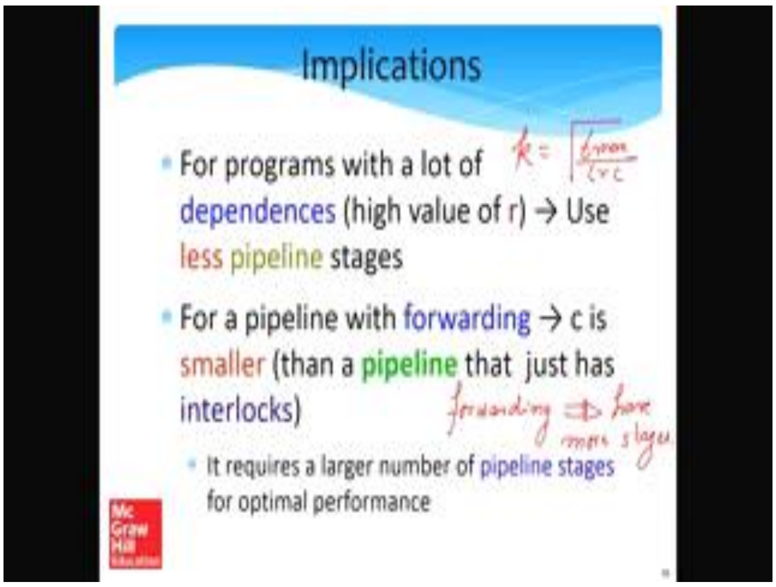
$$\Rightarrow \underbrace{k = \sqrt{\frac{(n-1)t_{max}}{l(1+rcn)}}} \approx \sqrt{\frac{t_{max}}{lrc}} \quad (as\ n \to \infty)$$

$$\Rightarrow k_{opt} = \sqrt{\frac{t_{max}}{lrc}}$$

Now, let us again differentiate this with respect to k. If we differentiate this expressional respect to k we shall get the optimal number of pipeline stages. So, then after differentiation we come to this expression; as n tends to infinity we get the optimal number of pipeline stages let us call it k opt is equal to the square root of t max, t max is the algorithmic work divided by the latch delay multiplied by r which is the stall rate, and c which is the constant of proportionality for the stall penalty right; square root of t max by lrc.

So this is the result that we get after all our algebra, and let us quickly try to appreciate what this equation is trying to tell us.

(Refer Slide Time: 66:05)



Implications

- For programs with a lot of dependences (high value of r) → Use less pipeline stages
- For a pipeline with forwarding → c is smaller (than a pipeline that just has interlocks)
  - It requires a larger number of pipeline stages for optimal performance

$k = \sqrt{\frac{t_{max}}{lrc}}$

forwarding ⇒ have more stages

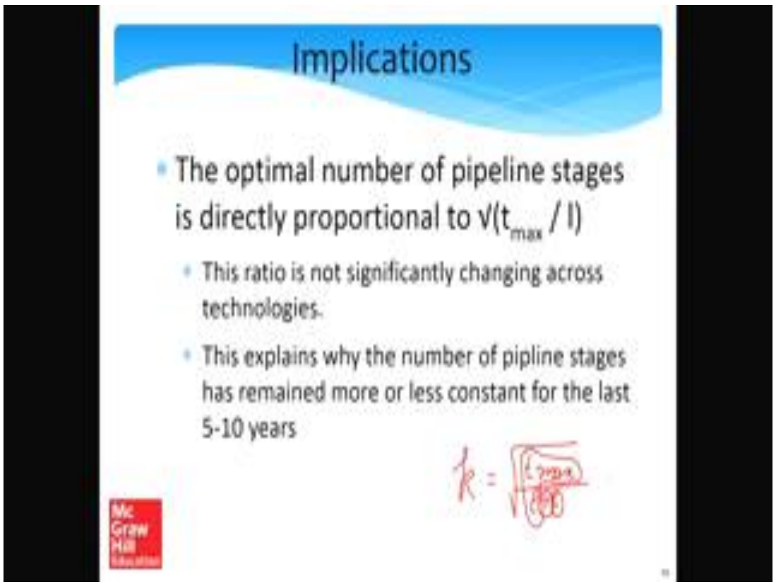
For programs let us look at the implications once again. Let me just write it once again k optimal is square root of t max divided by lrc; l is the latch delay, r is the stall rate and c is the constant of proportionality for the stall penalty. So, for programs with the lot of dependency, massive number of dependency usually have high value of r right high stall rate we will need to use lesser use less pipeline stages right. Why is this; this case because we do not you know; so let us consider the pathological case.

Where every instruction is has a load use hazard with instruction just before it. That is possible. In this case we actually do not get gain much by heavily pipeline in the processor. So, pipelining only works when the number of dependencies between instructions are limited such that we even take actual benefit of pipelining will (Refer Time: 67:27) and the number of bubbles that will introduce will be low.

So, that is the reason as we increase r, increase the stall rate it calls for reduction in the number of pipelines stages. Now for a pipeline forwarding the stall penalty will be lower much much lower, so c is smaller than a pipeline that uses interlocks. There is a inter locks we are stalling for more cycles for three cycles for forwarding most of the time we do not stall any and even if we do stall in the case of a load use hazard we stall for only one cycle; so c is smaller.

In this case for a pipeline that is using forwarding since c is smaller we can the optimal number of pipeline stages will actually be more. So, for a pipelining with forwarding it makes sense to make it deeper, deeper means add more stages. It will require a larger number of pipelines stages to get optimal performance. Let me may be summarize this forwarding implies have more stages.

(Refer Slide Time: 68:44)



Implications

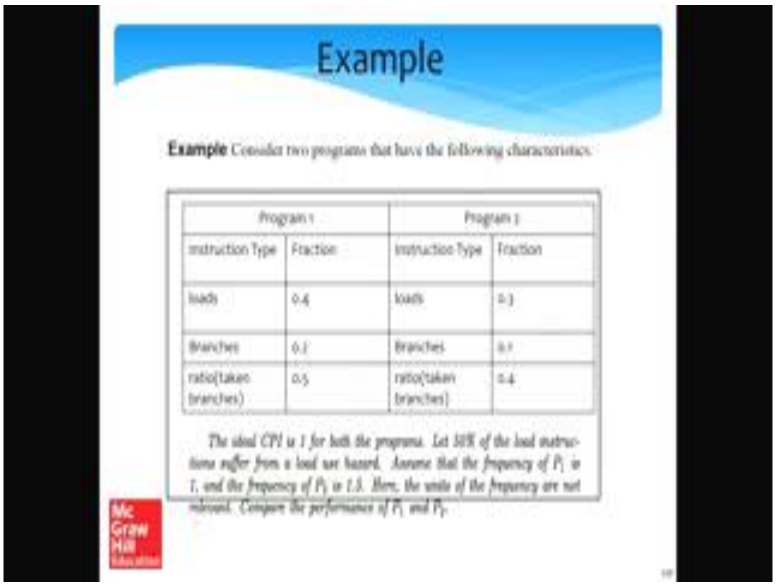
- The optimal number of pipeline stages is directly proportional to √(t_{max} / l)
  - This ratio is not significantly changing across technologies.
  - This explains why the number of pipline stages has remained more or less constant for the last 5-10 years

$k = \sqrt{\frac{t_{max}}{[illegible]}}$

Let me once again write this expression. So, the optimal number of pipelines stages is directly proportional to square root of t max by l. The ratio of square root of t max by l which is the total amount of works that needs to be done to process a single instruction divided by the latch delay is not really changing across technology. See even as transistors gets smaller, just look as transistors gets smaller t max will reduce because smaller transistors are faster. So, for an instruction to go from the beginning of the pipeline to the end of the pipeline it will take a lesser amount of time.

Simultaneously, latches are also getting faster. Roughly by a similar factor so t max by l will remain the same, as a result the number of pipeline stages will not change very significantly. So, this explains why the number of pipeline stages has remained more or less constant for the last 5 to 10 years, because t max by l remains more or less constant, r is anyway dependent on the nature of the programs and c is the constant which also remains the same. So, since over the years transistors get smaller its only t max by l that we expected to see some change in, but that is also not changing. So, it does not really make sense to increase the number of pipeline stages.

(Refer Slide Time: 70:24)



Example

Example Consider two programs that have the following characteristics.

Program 1		Program 2	
Instruction Type	Fraction	Instruction Type	Fraction
loads	0.4	loads	0.3
Branches	0.2	Branches	0.1
ratio(taken branches)	0.5	ratio(taken branches)	0.4

The ideal CPI is 1 for both the programs. Let 50% of the load instructions suffer from a load use hazard. Assume that the frequency of P_1 is 1, and the frequency of P_2 is 1.5. Here, the units of the frequency are not relevant. Compare the performance of P_1 and P_2.

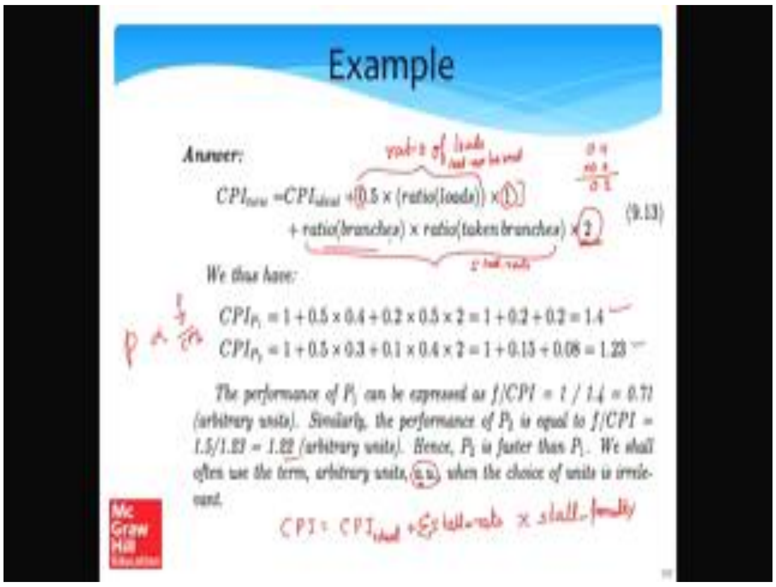
So, let us now do a little bit of math. So, hopefully we have understood this. I need to make another very very important point of this point. So, what is real advantage of pipelining? The real advantage of pipelining let us actually go back to this equation over here. So, there are three factors in performance: one is the frequency; one is the IPC performance equation can be thought of like this as the performance is proportional to the frequency multiplied with the IPC divided by the number of instructions. So, what pipelining allows us to do? Is it allows us to crank up the frequency? It is t max by k plus l. So, as we increase k in a certain; so we decreasing the time per stage and so then we can crank up the frequency.

With the frequency increases hopefully the performance will increase, simultaneously as the frequency increases the number of stages increase so this k over here is getting off set by k in the denominator. So, the IPC that will be an effect on IPC the IPC will most likely come down, but at one point you know you will have a maximum so for a lot of paper which have plotted their performance in the y axis versus the number of pipeline stages what they have actually found is that at till one point we can increase the frequency and we will get higher performance, after that the performance kind of decreases. So, they choose the optimal point and the optimal point gives us the optimal performance as well as the optimal number of pipeline stages.

See any way students can plot this equation and see you know for representative values and how things change. So, let us consider a typical math's. So, let us consider two programs where the fraction of loads is 40 percent loads 20 percent branches and 50 percent of the branches are taken branches. Another programs which has 30 percent loads 10 percent branches and 40 percent of the branches are taken branches. The ideal CPI is one for both the programs and let 50 percent of the load instruction suffer from a load use hazard. Assume that the frequency of P 1 is 1 and the frequency of P 2 is 1.5 units do not matter.

So, compare the performance of P 1 and P 2. So see we will see many many examples of this type where we are given an instruction means, you are given the fraction of load use hazards. The frequencies of you know processor 1 and processor 2 and let us compare the performance.

(Refer Slide Time: 73:19)



Example

Answer:

$$CPI_{new} = CPI_{ideal} + \boxed{0.5} \times (ratio(loads)) \times \boxed{1}$$
$$+ ratio(branches) \times ratio(taken\,branches) \times \boxed{2} \quad (9.13)$$

We thus have:

$$CPI_{P_1} = 1 + 0.5 \times 0.4 + 0.2 \times 0.5 \times 2 = 1 + 0.2 + 0.2 = 1.4$$
$$CPI_{P_2} = 1 + 0.5 \times 0.3 + 0.1 \times 0.4 \times 2 = 1 + 0.15 + 0.08 = 1.23$$

The performance of P_1 can be expressed as $f/CPI = 1/1.4 = 0.71$ (arbitrary units). Similarly, the performance of P_2 is equal to $f/CPI = 1.5/1.23 = 1.22$ (arbitrary units). Hence, P_2 is faster than P_1. We shall often use the term, arbitrary units, (a.u.), when the choice of units is irrelevant.

$$CPI = CPI_{ideal} + \Sigma\, stall\_rate \times stall\_penalty$$

So, we know that the CPI mu, right the CPI is CPI ideal plus the stall rate tends the stall penalty. So, we will take the ratio of loads multiply them with 0.5 because half of the load suffer from a load use hazard.

So, this is the ratio of loads that have a load use hazard. So, this is pretty much the stall rate multiplied by the stall penalty for a load use hazard is 1 cycle plus the ratio of branches, multiplied by the ratio of taken branches within that. So, this is the stall rate for branches, multiplied with the stall penalty and the stall penalty for taken branches is 2 cycles. So, we compare compute the CPI by taking values from the previous table, you find the CPI of P 1 to be 1.4 and the CPI of P 2 to be 1.23.

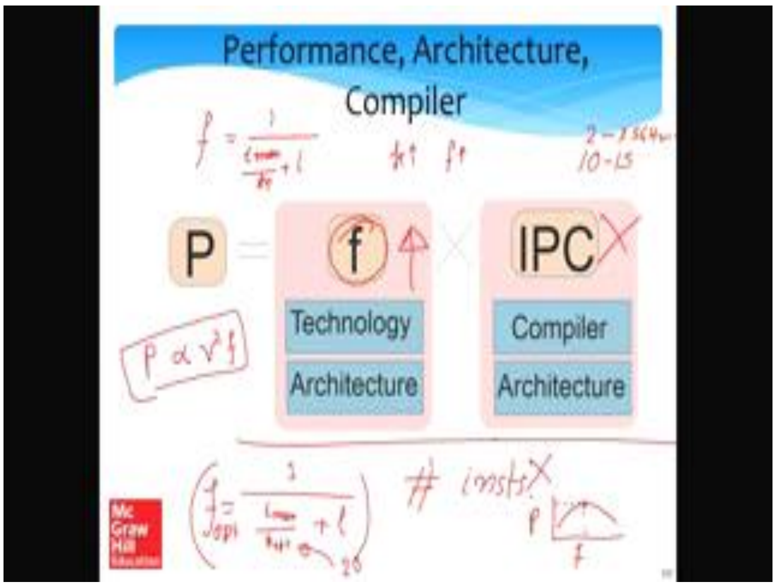
So, the performance can be expressed as the quantity which is f divided by CPI which is 0.71 for P 1 units are arbitrary. Similarly the performance of P 2 is equal to f divided by CPI which is 1.5 by 1.23 which is 1.22. Since the performance of P 2 is higher we can say that P 2 is faster than P 1. So, we shall often use the term and for arbitrary units we will use the term a u, when the units do not matter. But what is the most important take away point from this slide is the way that we compute the CPI.

So, CPI is CPI ideal plus stall rates times stall penalty. The stall rate is at per instruction what is the probability that you will have the stall. And there can be multiple reasons for a stall. You can have stalls because of loads; you can have stalls because of branches. They are disjoint events so we can put a summation here. So, for the case of a load we observe that given an instruction let us say ratio load tells us that for every instruction is a probability that probability of 0.4 that it is a load. And for every load the probability of 0.5 that it will lead to a load use hazard.

So, the total probability that given an instruction that will lead to a load use hazard is 0.4 times 0.5 which is 0.2. And the stall penalty for a load use hazard is 1 cycle, so we get stall rate times stall penalty for loads as this expression. Similarly for branches we need to find what percentage the instructions are branches, and out of the branches what percentage are taken branches. That will give us the stall rate because in for every taken branch we stall. And we will get the stall penalty; stall penalty is two cycles per taken branch. So, when we multiply both of them we will get the number of stall cycles for taken branches.

So, when we add all of these components we will get the CPI mu, and subsequently getting performance is very simple because performance is proportional to f divided by CPI when the numbers of dynamic instructions are the same.

(Refer Slide Time: 77:19)



Performance, Architecture, Compiler

So, let us take a look at these factors once again. We have in a sense performance is equal to; well we have been slightly taking some liberty should technically be proportional but, we have been taking some liberty let us keep on taking it. So, performance is pretty much f times IPC divided by number of instructions, but number of instructions will is completely dependent on the compiler. So, I have sort of taken it out of this particular expression; sorry. So, the frequency is dependent on several things. So, what is the frequency? The frequency we said is 1 divided by t max by k plus l. So, it is dependent on algorithmic work which is the time and instruction takes to go from the beginning of the pipeline till the end divided by the number of pipeline stages plus the latch delay. So, t max and l are completely dependent on the way that the circuit is made and speed of transistors. So, this is pretty much dependent on the transistor technology and the circuit, transistor plus; sorry transistor plus circuit technology.

The other factor in determining the frequency is k which is the number of pipeline stages and this is determined by the architecture, right the way we design the chip the computer architecture of the chip that determines the factor k. As we see that the frequency is determined by the technology the transistor technology in terms of t max and l and k is determined by the architecture; so t max and l from the technology and k from the architecture.

Similarly when we talk about IPC which is 1 by CPI it is of course determined by the compiler because you can order the code in such a way that there are no load use hazards it can be done. So, compiler has a role to play. The architecture definitely has a role to play because look schemes like forwarding are architectural schemes; interlocks and forwarding are architectural schemes there are many many more architectural tricks that can be done to ensure that our stall rate and the stall penalty go down. So, these determine the IPC of the processor. And so the IPC is definitely determined by the architecture. And the strongest you know the most potent IPC increasing mechanism that we saw in this lecture is forwarding.

So, what is forwarding do? Forwarding essentially reduces the number of stalls, it reduces the CPI and it increases the IPC. We have the other factor which is the number of instructions; and the number of instructions is dependent purely on the compiler. So basically we have 2-3 things: we have the technology, we have the architecture and we have the compiler and it the interplay of these three that determine the performance.

And here again I would like to state the most important thing which almost all students miss: that where does the benefit of pipelining come from. So, it clearly does not come from the number of instruction because this is a function of the program, so the benefit cannot come from here. The benefit cannot come from IPC I will tell you why, because let us consider. So, what is the IPC of an ideal pipeline? The IPC of an ideal pipeline the number of instructions per cycle of an ideal pipeline for a person who is standing over here you know I seen that this is the end of the 5 stage pipeline. So, an ideal pipeline one instruction will be coming out every cycle. So, this person will see the ideal IPC to be 1. So, I can say that the IPC ideal is 1.

And what is the IPC of a single cycle processor without any pipelining? Well, every instruction is completed in a single cycle so the CPI is 1 and consequently the IPC is 1. But in any real pipeline which will have hazards and stalls and bubbles the IPC is expected to be less than 1, because we will actually have stalls so we need to stop we cannot process every instruction in one cycle; we have dependency. So, the number of instructions remains the same it cannot be a factor IPC decreases. So, definitely cannot be a factor. The only factor that is responsible for increasing performance is frequency.

So, let me delete the ink on the slide, because I want to make a very important point. The only factor which one second; the only factor which allows us basically we know that in this equation you know in this particular equation there is nothing that its really helping the case of pipeline in the number of instruction are in IPC is not. The only advantage we are getting from pipelining is that we can reduce t max by k by increasing the number of pipeline stages, so if we increase the number of pipeline stages we can increase the frequency. And it is only this factor which is giving us the benefit that if we increase the number of pipeline stages we will reduce the time per stage, as a result we can increase the frequency. And by increasing the frequency we will be increasing one term in the performance equation.

The IPC will maybe slightly decrease as we have been seen, but that is as long as the product increases we are just fine. And so that is the advantage. The advantage of a pipeline you know listen to this very carefully; the advantage of a pipeline is that it allows us to increase the frequency; when we increase the frequency the numerator of the performance equation one term of the numerator of the performance equation increases. And as we increase the frequency we need to increase the number of pipeline stages also. Ultimately that will have negative effects in terms of IPC.

As a result you know we cannot increase the frequency of arbitrarily high, but frequency will definitely from our math the optimal frequency will be 1 divided by t max divided by k optimal which is the one square root of t max by lrc plus l. We will have an optimal frequency at which the performance speaks. Let us say I plot you know the number of pipeline stages or the frequency versus performance I will see you know some graph like this where there is an optimal frequency at which the performance is highest and this is exactly what pipelining allows us to do. We can keep on increasing the frequency till some point and we will get higher and higher and better and better performance. And that is the advantage.

Now let us say that we are not allowed to increase the frequency because somebody will say that you know this is not something that a transistor can support, or the participation will become very high so then the natural limits are placed on increasing the frequency. But still if we are pipelining a processor and we do not increase the frequency we are not going to get any performance benefit. The performance benefit only comes the increase in the frequency. And that is the crux for the pipelining idea; that if we do not increase the frequency after pipelining a processor we get nothing in terms of performance, only when we increase the frequency we get something.

There are of course, limits to how much we can increase the frequency too. Even if let us say from this expression it comes out that the optimal number of pipelining stages is 20 there might be other constraints on why we cannot have that (Refer Time: 86:22) pipeline. And also the power that a processor dissipates is actually proportional to v square which is v is the supply voltage times frequency. So, as we increase the frequency the power dissipation increases. So, this is also a problem, we cannot have processor the dissipates a lot of power.

So, this places limits on the frequency, and placing limits on the frequency places limits on the number of pipeline stages that we can support as well. That is one reason why at least over the last 10 to 15 years. The numbers of pipeline stages have remained roughly constant between 10 and 15 and the frequencies have also remained roughly constant between 2 and let us say 0.5 gigahertz.

So, let us finish this you know very very interesting discussion on pipelines, so this current lecture ends here, the last part the fourth part of this lecture series will look at advanced topics on how to handle interrupts and exceptions. So, that will be the last part of this lecture series.