Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

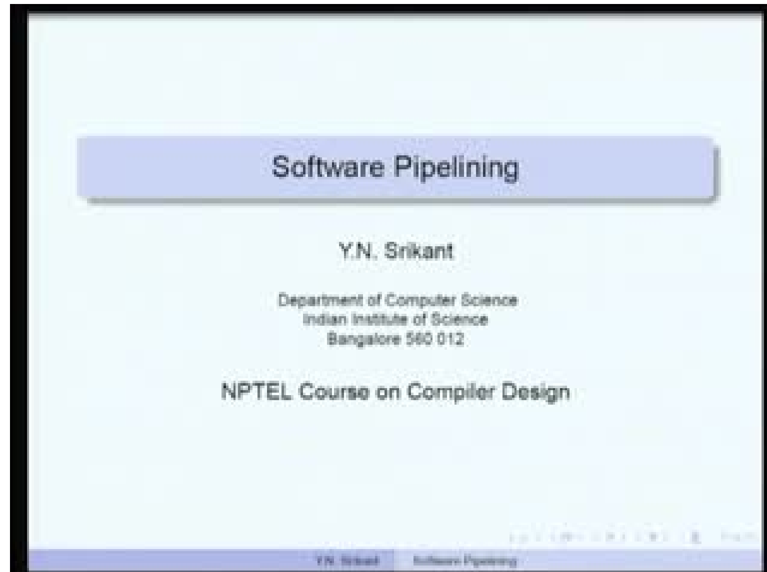
Module No. # 16

Lecture No. # 31

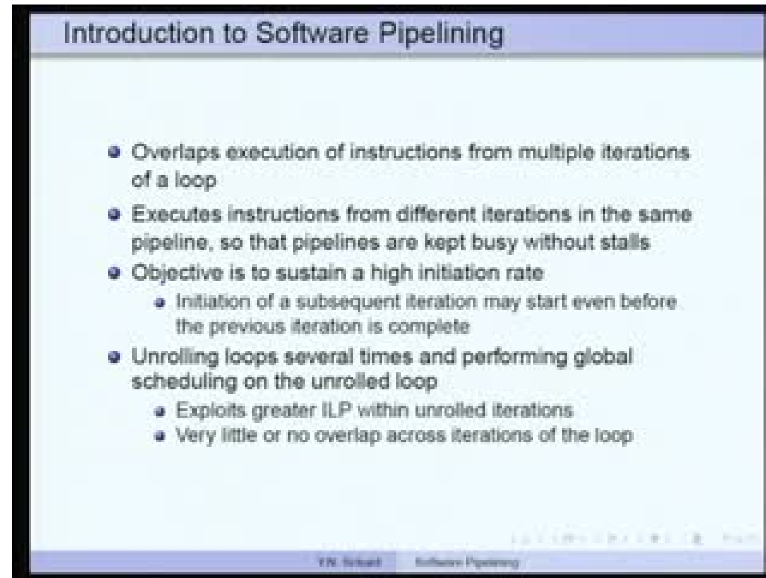
Software Pipelining

Welcome to the lecture on software pipelining. Software pipelining is a new type of optimization at the machine code level. What exactly is pipelining?

(Refer Slide Time: 00:17)



(Refer Slide Time: 00:27)



Everybody is familiar with the hardware pipelining concept. There are different pipeline stages through which, for example, an instruction close, fetch, decode, and so on and so forth.

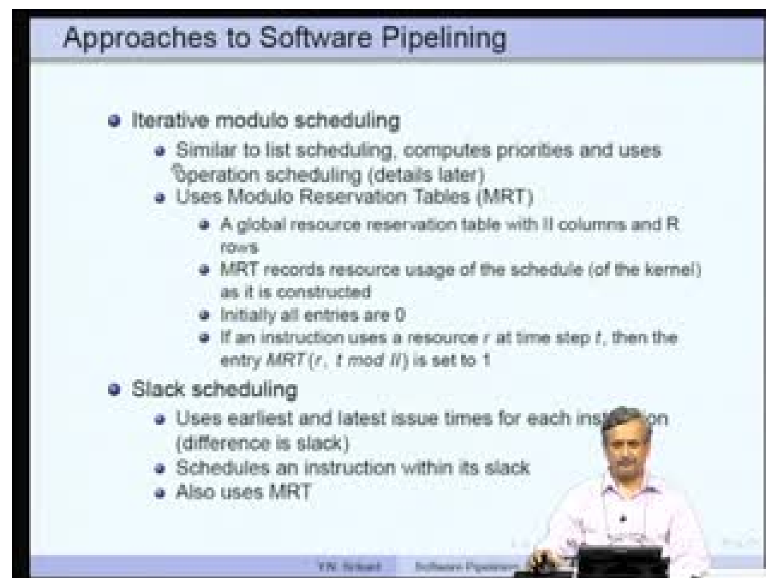
Similarly, software pipelining implies overlapping execution of instructions from multiple iterations of a loop. In hardware, there are different stages of a pipeline and the same instruction flows through all these stages. But in software pipelining, we want to make sure that instructions from different iterations, not necessarily the same iteration, are also permitted to use the functional units of the processor, so the stalls are actually minimal. All the pipelines are kept busy. The hardware pipelines are kept busy without any stalls, so we mix instructions from different iterations of the loop. In other words, software pipelining is meant for a loop. It is not meant for simple straight line code (Refer Slide Time: 01:46).

What is the objective of software pipelining? To sustain a very high initiation rate. In other words, we want to start execution of as many instructions as possible in a single cycle, if possible. That means, initiation of subsequent iterations may start even before the previous iteration is complete. If there are not enough instructions in the same iteration, which can be started in a particular cycle, then it is possible that instructions from the second or third iteration ahead may be started right now. But then, on what basis does one do all this? Obviously, dependencies between instructions. That is, if an

instruction produces a result and it will be used five cycles later, you really cannot say that I will start the instruction, which is using it much earlier than five cycles later; that is not possible. Only when there are no dependencies which hurt, the instructions from other iterations can be started.

Unrolling loop several times and performing global scheduling on the unrolled loop is also a possibility for increasing the parallelism, rather than harnessing the parallelism available in programs. This exploits greater instruction level parallelism within the unrolled iterations. For example, if we unroll a loop once, then two iterations of the loop will be executed as straight line code, one after another, in the body of the loop. The parallelism available in these two iterations will be utilized. If we do it three times, the parallelism in three iterations will be used, and so on and so forth. But there is not much overlap across iterations of the loop. In other words, if we can start the second, third, fourth iteration instructions in the same cycle, this unrolling mechanism will not help. Of course we may say, why not unroll the loop five times and then start the instructions? But then the body of the loop really becomes very large. There will be a code blob attached to loop controlling then.

(Refer Slide Time: 04:13)



There are two approaches to software pipelining -- one is called iterative modulo scheduling and the other is called slack scheduling. In iterative modulo scheduling, we use a mechanism or a methodology which is very similar to instruction scheduling, that

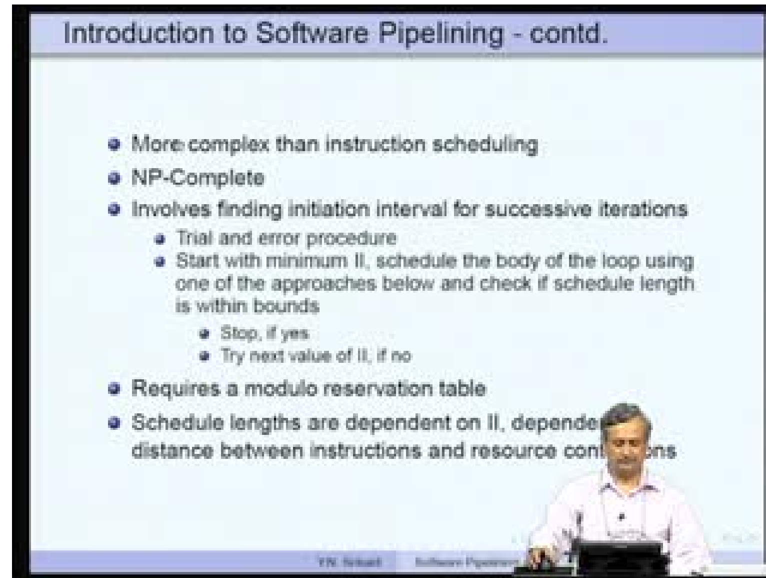
is, list scheduling algorithm. We compute priorities, as in the instruction scheduling algorithms, and use operation scheduling here. In the case of instruction scheduling, we utilize cycle by cycle scheduling. But here, as a rule, we use only operation scheduling, The reason is very simple. It is because we use the modulo reservation table, instead of the global reservation table (Refer Slide Time: 05:05).

What is an MRT (Modulo Reservation Table)? It is a global resource reservation table with its II columns. In other words, II is the Initiation Interval, which will become. That is, every II cycles, I am going to start new iteration. This is what initiation rule is. There are II columns and r rows, where r is the number of resources in the system. We really do not have a table, which is as large as the schedule of the entire program. That is what a global reservation table is. Here, we have just II columns and r rows, instead of t columns, where t is the length of the schedule.

MRT records resource usage of the schedule for the kernel. What is a kernel? That is the set of instructions, which are within what is known as a core of software pipeline. This will become very clear very soon. The usage of resources for that entire program, which is inside the kernel, is actually recorded in the modulo reservation table. Why is it called modulo? All the resource usages are recorded with time modulo II. For example, initially all the entries are 0, but if an instruction uses a resource r at time step t, then the entry MRT r coma t mod II is set to 1. That is how it would be. So everything is modulo the initiation interval.

Slack scheduling is similar. Only difference is, it uses earliest and latest issue times, that is, the deadlines for each instruction and the difference is slack. We have seen this before in the instruction scheduling algorithms. It schedules an instruction, rather tries to schedule an instruction within its slack. Here also, we require the modulo reservation table. It is just that, the priority used is slack, rather than height of an instruction, and so on.

(Refer Slide Time: 07:25)

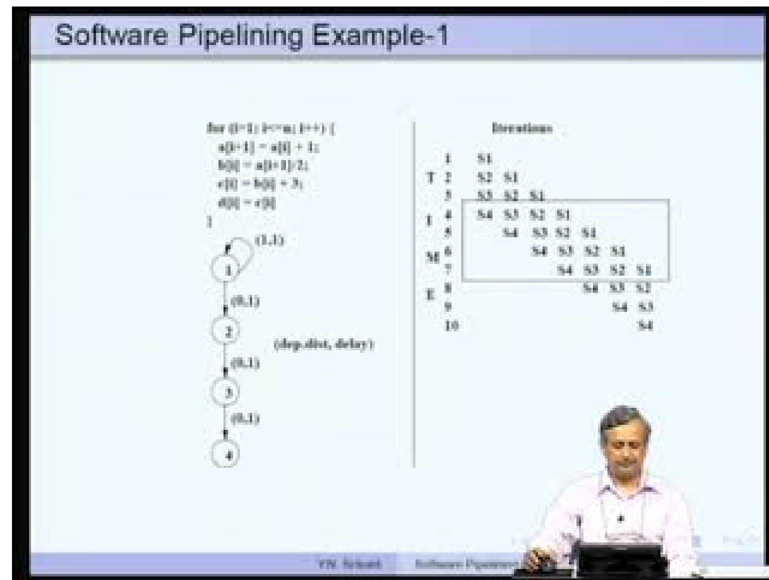


Obviously, the software pipelining algorithms are much more complex than instruction scheduling algorithms. The problem is NP complete. What does it involve? It involves finding initiation interval for successive iterations. What is the initiation interval? I have already mentioned this. It is the number of cycles after which we start the next iteration, so it is not necessarily as many cycles as the body of the loop.

Finding the II (Initiation Interval) is a trial and error procedure. We start with the minimum initiation interval. We will see how to compute this later. Schedule the body of the loop using one of the approaches below and then check if the schedule length is within the bounds, that is, II 0 to II minus 1. We are going to discuss strategies later, but the approaches would be either operation scheduling with some priorities computed or use the slack.

If we have found a schedule for the instructions between 0 and II minus 1, then stop. Otherwise, increment the value of II and make it II plus 1. For a larger II, there is more because the number of time steps available is much more. There is a good chance that we will get a schedule without conflicts. Of course, we use a modulo reservation table. Schedule lengths are dependent on the initiation rule, dependence distance between instructions and resource contentions.

(Refer Slide Time: 09:12)



Let us take a simple example. Here is the body of a small loop. The instructions are a i plus 1 equal to a i plus 1, b i equal to a i plus 1 by 2, c i equal to b i plus 3 and a i equal to c i. Here is the dependence diagram for this particular loop. Instruction 1 has dependence on itself -- a i plus 1 and a i. a i is used from the previous iteration and a new one is written. The first component is the dependence distance and the second component is the delay or the latency. Dependence distance is 1, so we use the result from the previous iteration. The delay for each instruction is assumed to be 1, just for mere simplicity.

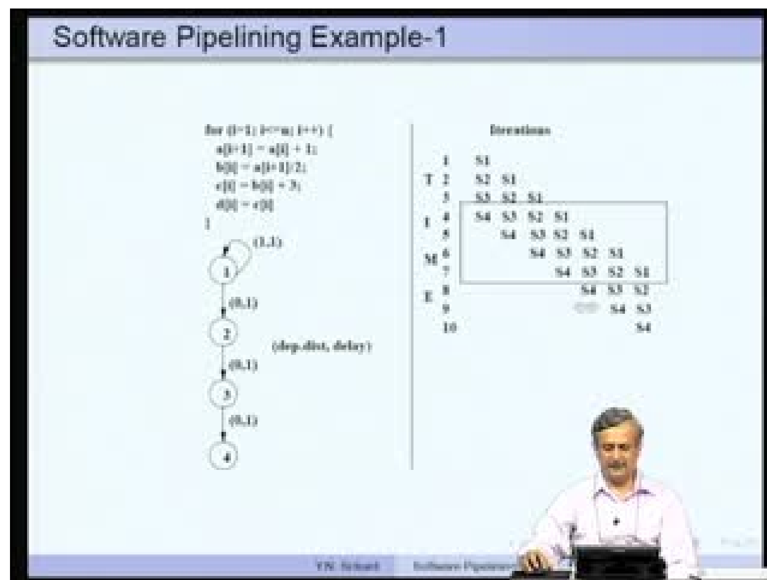
This is a loop and from here onwards, we will have no loops . There is no dependence between iterations anymore after this. All of them are 0 coma 1 and 1 is the delay for or the latency of the instruction.

Let us start executing the instructions one by one. Look at the trace for how it goes. Here is the timeline 1, 2, 3, 4, etc., so we start S1, then S2, then S3 and then S4. Now we go back and start S1, S2, S3, S4, etc., etc. That is the sequential execution. It would be S1, S2, S3, S4, and so on.

In this case, we are executing just one statement in each cycle or in each time step. Just look at it, the statement S1 depends on itself, that is, until the first S1 is completed, we really cannot start the second iteration S1. That is what this dependence is. If we assume that this is satisfied, let us start S1; then when we go to the second cycle, we start S2 of

course, second time step. In the same time step, we also start S1 for the next iteration because there is no dependency between this S1. This S1 really requires results from this S1, so that is already executed. There is no dependence between this S2 and this S1. There is no problem as far as dependencies are concerned. When we start S3 of this first iteration, we would be executing S2 of the second iteration and S1 of the third iteration is begun. In other words, we are trying to execute three statements, all from three different iterations in the same cycle. Then we go to S4, then obviously S3, S2 and S1 are there. There are no more instructions, so the stable state of this particular sequence would be S4, S3, S2, S1, S4, S3 S2 S1 and so on. Finally, after the number of iterations that is required is over, we execute the rest of the instructions S4, S3, S2, S4, S3, and finally S4 and stop.

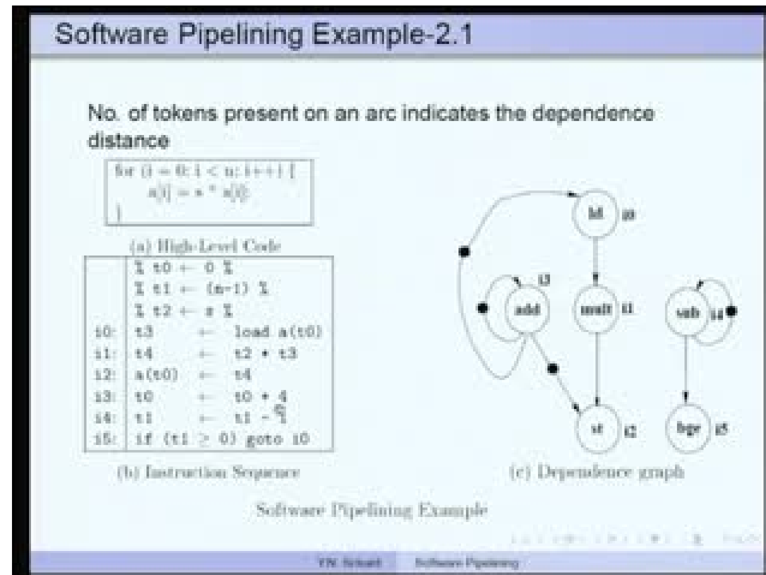
(Refer Slide Time: 12:33)



Here, the steady state, we are going to execute four instructions in each cycle -- S4 of current iteration, S3 of next iteration, then S2 of next plus 1 and S1 of next plus 2. All the four instructions are being executed in the same cycle and we are assuming that there are enough resources integer units, floating point units, branch units, load store units, etc., available and there is no resource conflict between any of these instructions, so that all the instructions can be executed. There is no dependence between these instructions across iterations, except for S1. We can do this. In other words, the loop is actually one instruction, rather one complex instruction, consisting of these four instructions from

four iterations. That is the kernel of our loop. These are the four instructions, which we are going to execute again and again, as many times as necessary.

(Refer Slide Time: 13:38)

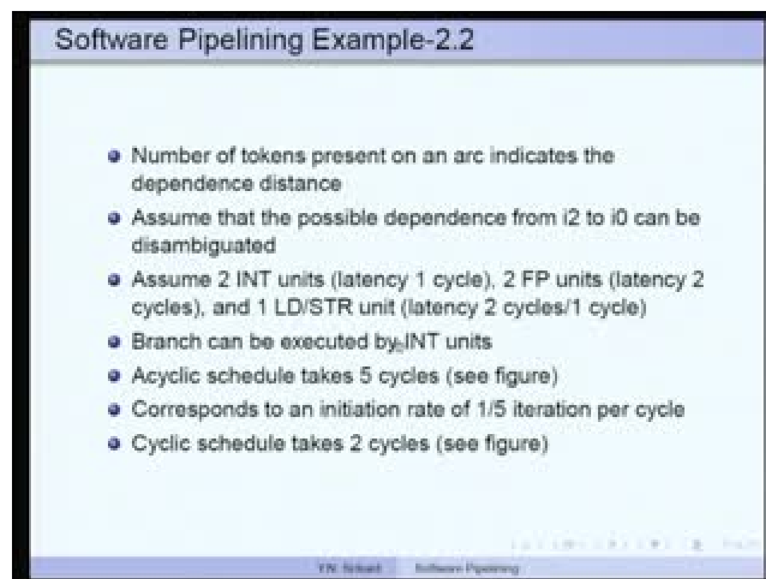


Let us take another example. Here we go down one step and look at machine level instructions, just for the sake of information. The loop is very simple -- a i equal to s star a i. and it executes for n number of times. Initially, Let us assume that variable t0 contains 0, variable t1 contains n minus 1, and t2 contains s and t3 equal to load a coma a of t0. Right hand side is being loaded now. Then t4 is t2 star t3, so that is, s star a I, a t0 equal to t4, that is, this particular item is being stored. Then, we need to go to the loop branching part, so we increment t0 by 4, that is, our i4 bytes. This is the address, t0 is 0. We would like to assume 4 bytes per element of the array, hence t0 plus 4, then the counter is t1, so that is, t1 minus 1. We will check whether t1 greater than or equal to 0, keep iterating.

The dependence diagram for this particular loop is here. Here, i0 is the load instruction, we feeds to the multiply, that is, i1, and multiply feeds to i2. Meanwhile, i3 is add instruction, which is independent of any of these, but that also feeds to i2. That’s the only thing. Then we have this i3 again. There is a self loop. The reason is this - i3 t0 equal to t0 plus 4, and then i2 is store instruction, i4 is a subtract instruction and i5 is the branch instruction.

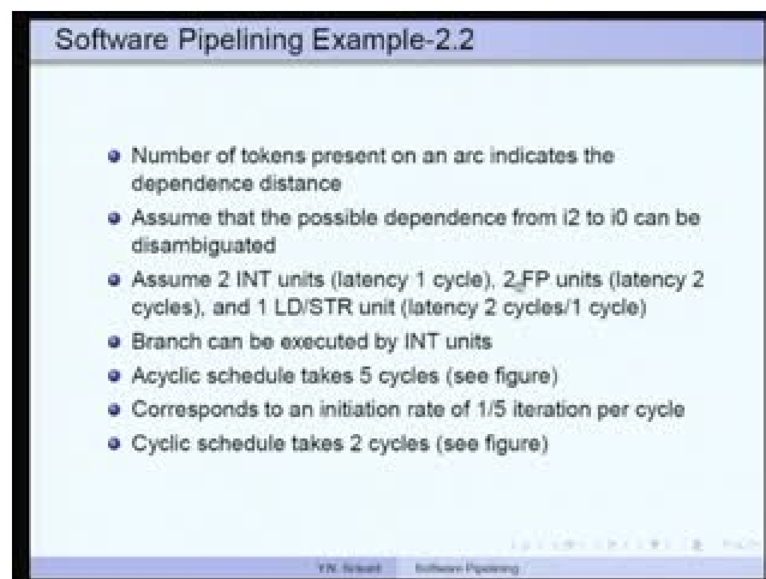
The dots on these indicate that there is a dependence backwards by one iteration. For example, i3 here is t0 is equal to t0 plus 4. This value of t0 is used in the next iteration. That is t0 here. That is why this dependence. Similarly, i3 – this is old value of i3 and this is the new value of i3, so there is a loop. From s3, this value is used here in the next iteration. So, that is the other iteration, the other dependence that is shown here. Similarly i4 has dependence to itself and then it feeds to i5.

(Refer Slide Time: 16:45)



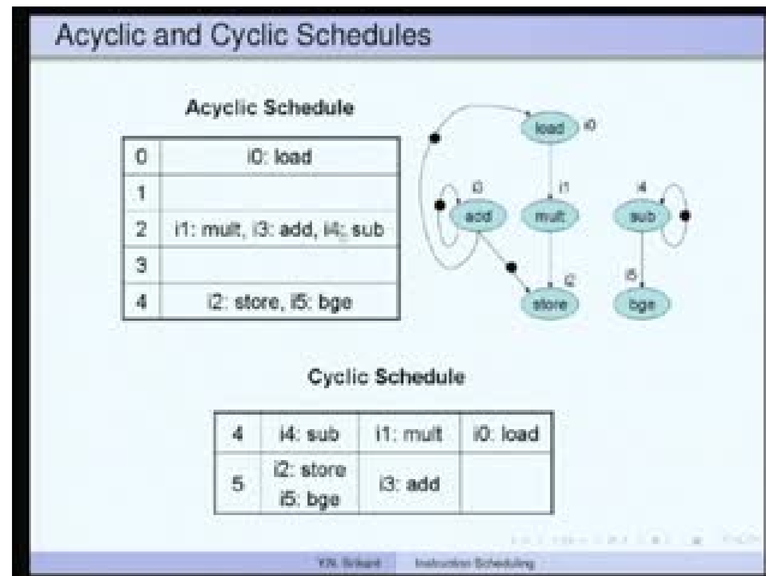
This is the loop structure. Now, the number of tokens on an arc indicates the dependence distance. I explained this already. Assume that the possible distance from i2 to i0 can be disambiguated (Refer Slide Time: 16:57). In other words, if you look at this a t0 and then the store, that is, i2, there is obviously a dependence between these two. This load must happen and then the store. Let us assume that it can be handled by hardware in some way. Software pipeline has to actually respect this particular dependence diagram.

(Refer Slide Time: 17:21)



Assume that there are two integer units -- each latency is one cycle, two floating point units latency two cycles, one load store unit with load requiring two cycles and store requiring one cycle. Let us also assume that the branch can be executed by any int unit. Now, acyclic schedule takes five cycles. I will show you this. It corresponds to an initiation interval of only 1 by 5 iteration per cycle. There are five cycles, so one by is the iteration count, initiation iterate. Cyclic schedule, that is, the software pipeline schedule requires two cycles. How?

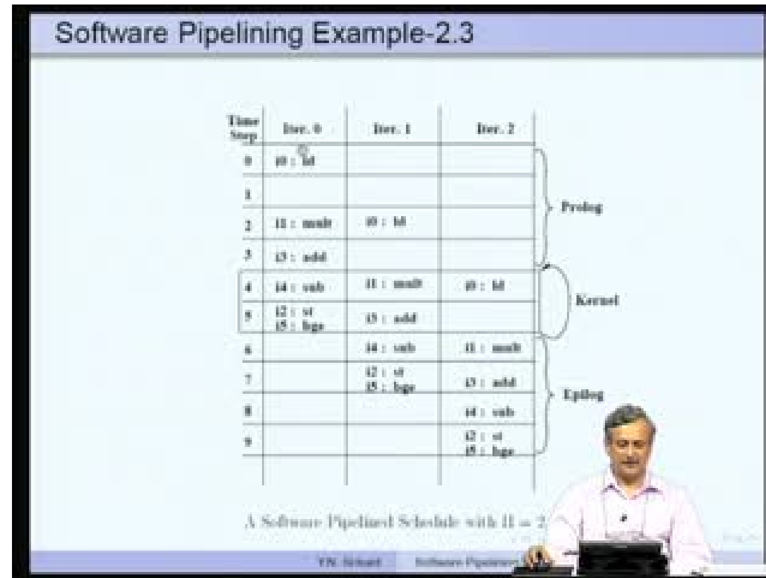
(Refer Slide Time: 18:02)



Let us look at this diagram. You can schedule i0. It requires that result to be used by i1, so we need to have this as a no operation. There is a nothing to do here. Here, we can execute i1, i3, and i4 – all of them in parallel. There is no dependence between these. Then again, this mult requires two cycles, so we hold on i2 scheduled in the fifth cycle, along with i5. That is why we require five cycles for acyclic schedule. This is the regular instruction scheduling.

What about the software pipeline? I will show you the details in the next slide. We are going to have this as our kernel or core of this software pipeline. i4, i1 and i0 execute in the same cycle; i2, i5 and i3 execute in the other cycle. These two are executed together and i3 on another unit. How? Let us look at it.

(Refer Slide Time: 19:04)



Iteration 0, iteration 1 and iteration 2, as usual, have been unrolled. We have load, mult, add, sub store, bge in this iteration, in this sequence. At iteration 1, we start i0 here and then go on. Again i0 here, and then go on, etc., etc. Why are we doing this? Why could not we have started i0 here or here and so on? The point is, we must have sufficient number of resources to execute the instructions of the second or third iteration in the same cycle. We cannot start another load here because load requires two in cycles to complete and there is only one load store unit. We are assuming that more than one load cannot be issued at the same time by the unit. So, that is the reason why i0 has to start here.

(Refer Slide Time: 20:57)

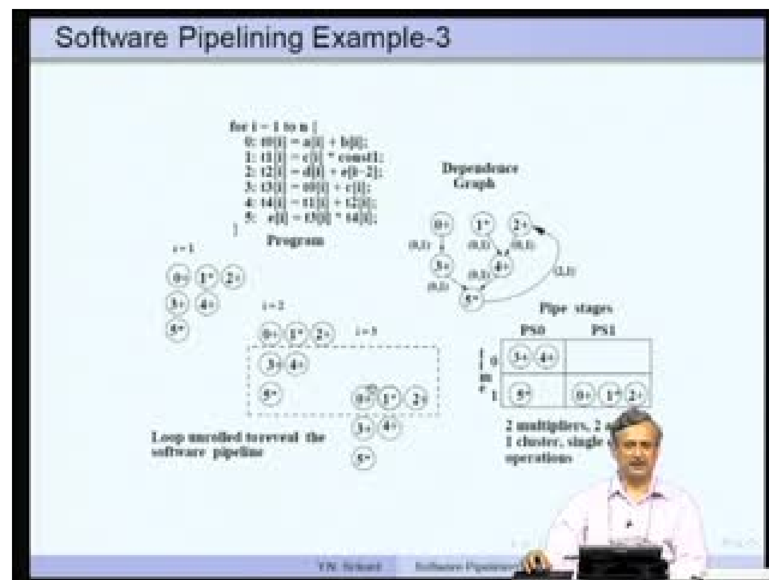
(Refer Slide Time: 21:01)

This, of course, is subject to dependencies, and then this is again is subject to dependencies within the loop, and this also, in the same way. If you observe at this point, we have reached some kind of a steady state and this pattern repeats – i4, i1, i0, i2, i5, i3, i4, i1 and there will be i0. Here if we had peeled iteration 3 as well, i2, i5 and i3, and so on. The same pattern repeats. Because of that, this is the stable state of the software pipeline and all the instructions in the loop are in the kernel of the loop -- this particular part. There is i0, i1, i2, i3 and i4, at various places. The dependencies are also satisfied because, for example, i1 feeds to i2, so i1 is here and i2 is here. This i1 is here and this i2

is here; so in this the loops are also shorter. Many instructions are executed in the same cycle but they are all form different iterations. So, there is no problem regarding which result is used by which one.

The register location would be done appropriately to make sure that results are handled appropriately. What should be remembered here is that many instructions can be executed in the same cycle and these instructions are from different iterations.

(Refer Slide Time: 21:38)



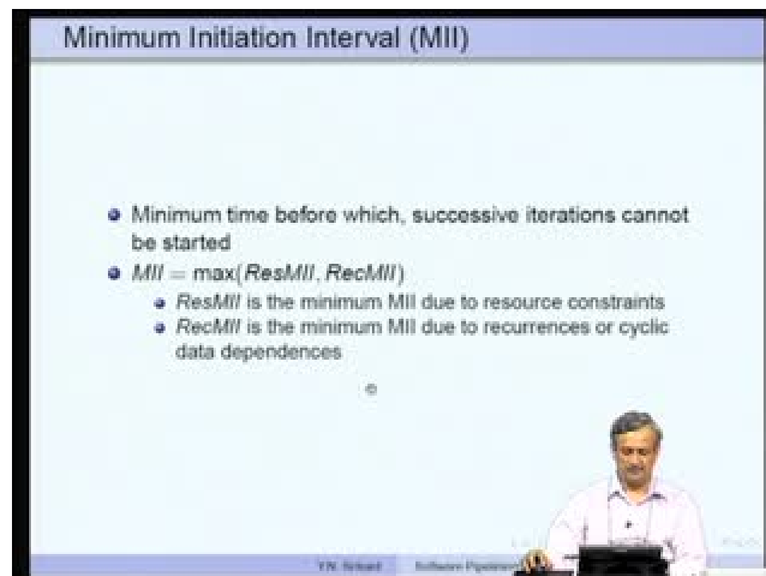
One more example. Here is a small loop consisting of many instructions of this kind – i0; i t0, i equal to a i plus b; i t1, i equal to c i star const 1, etc., etc. Here is the dependence graph for this particular small example. So, 0, 1, 2 -- the operations are written beside the numbers or at the base level. Then, 3, 4 are dependent on them and finally 5 is dependent on 3, 4, and then there is a cycle. This is the cycle that we have. The distance are 0 for all these instructions, except for this – a distance of 2. You can see that the distance is created because of this e i minus 2. This e i is computed in instruction 5, e 2 is actually e i minus 2, so there is a dependence distance of 2 between instruction 2 and instruction 5. Now, let us unroll the loop. Once you have unrolled it, you will see that these are the five instructions – 0, 1, 2 can all be executed in a single cycle; 3, 4 in another cycle; and 5 in another cycle. Let us do so.

We have this group, then this group, for the various iterations of the loop. Once we reach this point, we know that we have a stable state. How to detect this? That will be through

the algorithm. We have 3, 4, 5, 0, 1, 2. This pattern repeats – 3, 4, 5, 0, 1, 2; 3, 4, 5, 0, 1, 2, and again 3, 4, 5, 0, 1, 2. The pattern would repeat. Our software pipeline kernel consists of the same – 3, 4, and then 5, 0, 1, 2. We just say ps0, ps1 because these are stages just to show that they are not from the same cycle. 3, 4 and 5 are from the same iteration, and 0, 1, 2 are from a different iteration. This is time cycle 0, this is time cycle 1, so this is the loop, which would be repeating, as many times as necessary.

This is the concept of software pipelining. We try to execute many instructions from different iterations in the same cycle, in as many cycles as possible.

(Refer Slide Time: 24:00)



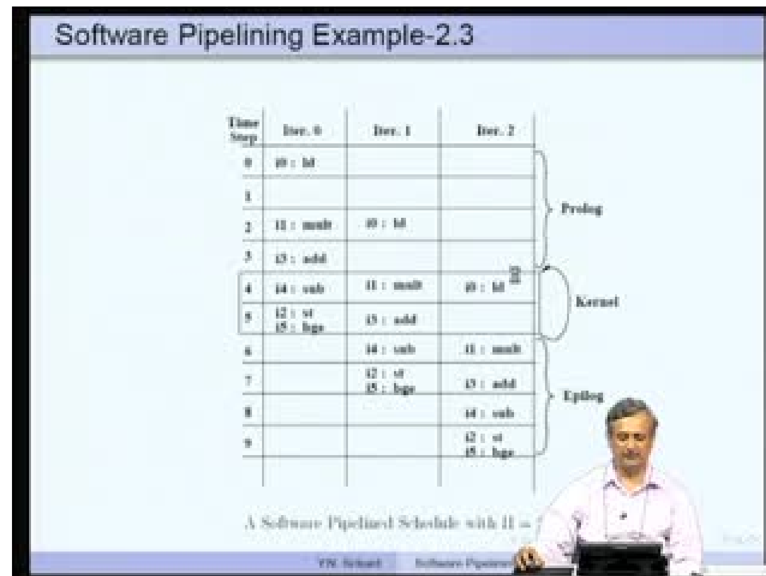
Now, let us start looking at the algorithm for software pipelining. We need to compute what is known as minimum initiation interval? How many cycles need to pass before the next iteration can be begun? That is minimum time before which successive iterations cannot be started. MII is the maximum of ResMII and RecMII. What is ResMII? This is the minimum imitation interval, due to resource constraints. RecMII is the minimum II due to recurrences or cyclic data dependencies. Let us elaborate.

(Refer Slide Time: 24:48)

Resource Minimum Initiation Interval (ResMII) is very difficult and very expensive to compute exactly. It is almost like solving the problem before we compute it. For function -- pipelined function on it, it can be approximated as maximum of N_r by F_r, over all the resource units r. What is N_r? N_r is the number of instructions that execute on a functional unit of type r and F_r is the number of functional units. Look at all the instructions in the loop body, which execute on a particular type of resource, then divide it by the number of function units of type r. That gives you ResMII for that particular resource. You do this for all the resources and then take the maximum.

(Refer Slide Time: 25:50)

(Refer Slide Time: 26:50)



Let us look at an example. The previous example, here is the graph. We know that there are two integer units, 2 f p units, and 1 load store unit. If you count the number of instructions of each type, int, f p and load store, we have add, sub, then this bge, which is again a branch instruction that are executed in the int a l u. There is only one mult instruction, which is the floating point variety, so 1. Then, there are two instructions load and store, so that is 2. The number of int units is 2, f p units is 2, and load store units is 1. So, the maximum of all this is 1.5 5.5 and 2, that is, 2. Resource MII is 2. That means, you cannot have a software pipeline kernel of length less than 2. What we got here was really a software pipeline kernel of length 2. That is the optimal part.

(Refer Slide Time: 26:57)

Resource Minimum Initiation Interval (*ResMII*)

- Very expensive to determine exactly
- For pipelined function units

$$ResMII = \max_{\forall r} \left(\left\lceil \frac{N_r}{F_r} \right\rceil \right) \quad (1)$$

where N_r represents the number of instructions that execute on a functional unit of type r, and F_r is the number of functional units of type r

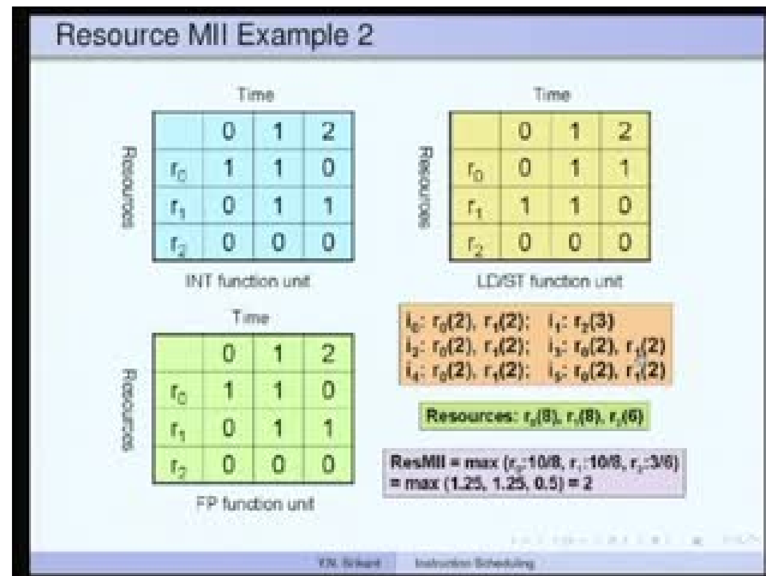
- For non-pipelined FUs or FUs with complex structural hazards

$$ResMII = \max_{\forall r} \left\lceil \frac{\sum_a N_{a,r}}{F_r} \right\rceil \quad (2)$$

where $N_{a,r}$ represents the maximum number of t... steps for which instruction a uses any of the stages of ... functional unit of type r. For example, for a ... FU, $N_{a,r}$ equals to the latency of the functi...

For non-pipeline function units or function units with complex structural hazards, ResMII is defined as max over all the resource function units. All the resources of sigma N_a coma r over all a. Hence, a is the instruction, and N_a coma r represents the maximum number of time steps for which instruction a uses any of the stages of a functional unit of type r. If I am using any particular resource, it does not matter which stage of that resource, I account it for the function for the instruction a. I sum up all the requirements of the instruction a, that is, sigma overall and sigma overall a., then divide it by the number of function units of type r and take the max for all such resources. You get the ResMII.

(Refer Slide Time: 27:58)



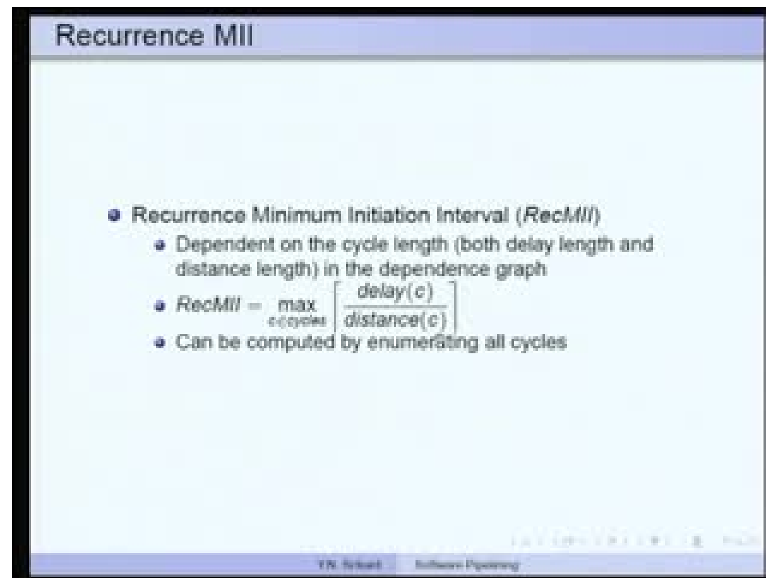
Let me show you an example. Let us say, these are the reservation tables for int unit, l d store unit and function unit. These are purely fictitious. This is not possible in practice. Now, for instruction $i_{0,}$, which is a load, load would require two cycles of r_0, two cycles of r_1 and none of $r_{2.}$. So, r_0 is 2 and r_1 is 2. Here, i_1 is a floating point instruction, so it requires all in this FP unit. So, we would require, this is slight mistake, this should have been 1 1 1 here. So r_2 is required three times, the rest are all 0. Then, i_2 would require r_0 two times and r_1 two times, i_3 would require r_0 two times and r_1 two times, and so on, and so forth. Now, if you sum up all the r_0 requirements, you get 8; if you will sum up all the r_1 requirements, you get 8; r_2 requirements, you get 6. This is the number of resources -- r_0 is available in 8 units, r_1 is available in 8 units, and r_2 is available in 6 units. Why?

We have two integer units, so r_0 is required in three phases. Rather we would say, we have 4 into 2, 8 of these 0 0, this is 1, 2, and r_0 is required here also -- 1 and 2. As I said, this should be zeros, only these should be ones. Now r_1 is required, we would have eight resources again because two of this and two of this -- 4 into 2. So again, eight resources and r_2 would be available in six resources. If this is 1, 1, 1 each, we would have three of them, then into 2, would be 6.

Then, you divide. This is the just an assumption that so many units of r_0, so many units of $r_{1,}$ so many units of r_2, etc., are available. Now, you divide 10 by 8, 10 by 8 and 3 by

6, you get max of 2. With a complex pipeline, this is what you really do. You count the usages of the various units, divide it by the number of units of that kind, and you get ResMII.

(Refer Slide Time: 30:23)



What is recurrence MII? Recurrence MII initiation interval is actually the minimum initiation interval due to cycles. This is dependent on the cycle length, both delay length and distance length, in the dependence graph. It is defined as max over all the cycles. What we really do is you enumerate all the cycles, take one cycle at a time, compute delay in the cycle, divide it by the distance in the cycle, take the max over all cycles, and that will be your RecMII. Let us see how to compute it.

(Refer Slide Time: 31:05)

Recurrence MII Example

$$RecMII = \max(RecMII_{\text{cycle on } i3}, RecMII_{\text{cycle on } i4}) \quad (5)$$

$$RecMII = \max\left(\frac{1}{1}, \frac{1}{1}\right) = 1 \quad (6)$$

(Refer Slide Time: 31:55)

Minimum Initiation Interval (MII)

- Minimum time before which, successive iterations cannot be started
- *MII* = max(*ResMII*, *RecMII*)
  - *ResMII* is the minimum MII due to resource constraints
  - *RecMII* is the minimum MII due to recurrences or cyclic data dependences

For this example again, we have two cycles here -- One is here and the other is here. This is delay of 1 and distance of 1. This is also a delay of 1 and distance of 1. So, we have 1 by 1 and 1 by 1. The max is 1. For RecMII, this would be 1. Initiation interval ResMII was actually 2. For this example RecMII is 1, so the minimum initiation interval would be max of this, that is, rather the minimum of this, that would be just the 1 or 2. For example in this case, we really say MII is the max of ResMII and RecMII not the min, even though, it is a minimum initiation interval. The reason is the restriction on the

resources or the restriction on the cycles. This will stop you from initiating the next iteration. So, you really cannot take minimum here. It has to be the maximum.

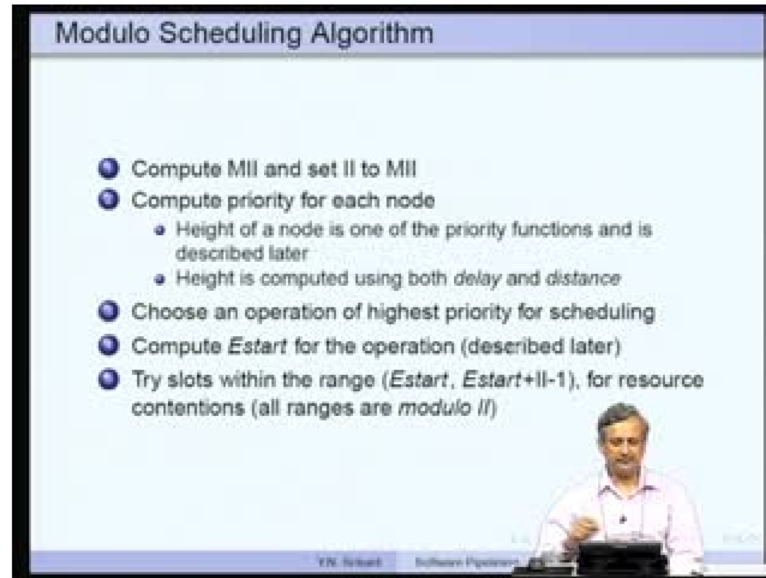
(Refer Slide Time: 32:18)

Now, let us take another example. Let us compute ResMII and RecMII for our cluster example. Here is the code and this is the dependence graph. If you look at it, you have ResMII. You really have only two types of instructions. Add adder, so two multipliers, two adders and one clusters. You have just 1 and 2, two multiply instructions and four add instructions. Hence, four adders and two multipliers. Add instructions and we have two of each, so 3 by 2 and 2 by 2. That would be 2 coma 1. The maximum is 2. ResMII is 2. So, we really have minimum possibility of 2 because of ResMII for the kernel length.

What about RecMII? Actually you have to take the cycle. There is only one cycle, so no question of taking a max there. Add the delays and distances. If you add the delays – 1 plus 1 plus 1, that is 3. You add the distances 0 plus 0 plus 2, that is 2. So, 3 by 2 is 1.5 and its ceiling is 2.

RecMII also says 2, ResMII also says 2. We have demonstrated that it is possible to get a software pipeline of length 2. In this case, we have achieved it. But in general, you may not be able to get a software pipeline of length MII. You may actually get something more than that because of difficulties in scheduling.

(Refer Slide Time: 34:07)



What is the algorithm? The algorithm is – compute the minimum initiation interval and set the II value (initiation interval value) as MII. We are going to try and get a software pipeline of length MII the first time. How to do that?

We are assuming that some distance height of height etcetera is used not the slack. Let us compute priority for each node. What are the priorities? Height of a node is 1 of the priority functions and we are going to say how to compute it a little later. Height is computed using both the delay and distance components of the dependence graph or on the arcs, we have this dependence – the delay and the distance. We are going to use it to compute the height. Let us assume that priority, that is, height, is computed for all the nodes.

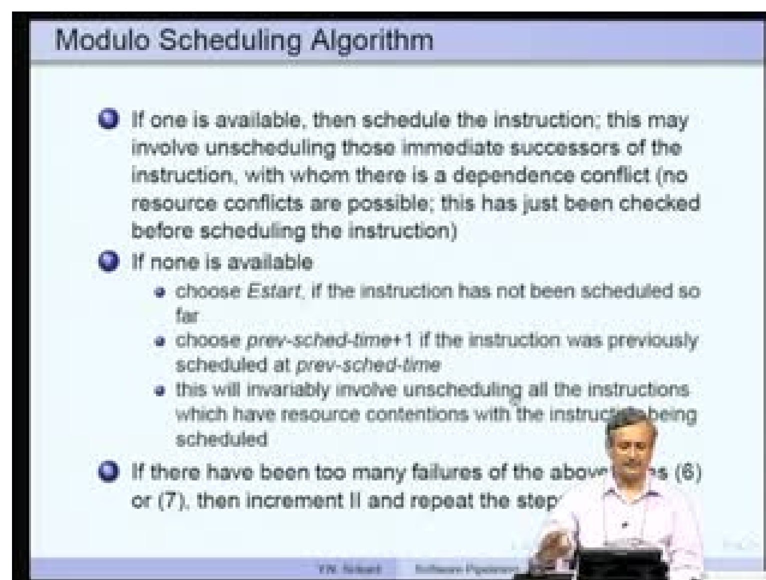
Choose an operation of highest priority for scheduling. Pick up. This is operation scheduling, so there is no ready queue. We are just going to pick up the highest priority node and start operation and start scheduling.

How to do that? You compute earliest start time for the operation. Again, we are going to show details of this a little later. Now, you start and try to schedule the operation between in the interval e start and e start plus II minus 1. Why? Why should we try scheduling the instruction in this range? The reason is that the software pipeline is supposed to be of length II cycles and every II cycles we are going to begin a new iteration. Now, we are looking at the instructions of the loop in one iteration, that is, the

body. We must be able to fit all the instructions of the loop within a window of II cycles. That is why e start is the earliest start time for an instruction. So, we look at e start plus II minus 1. All these are modulo II. If we go beyond II slots, we actually rap around and come from the top. We try all the instructions – e start, e start plus 1, e start plus 2, etc., etc. Modulo of course II and see if we can fit these instructions into these slots.

Then, checking whether the instruction fits into a slot implies checking for resource contentions. If there are enough resources, then we can schedule it. Otherwise, no. How do you check resource contention? That is where the modulo reservation table that I have mentioned earlier comes into picture. MRT records resource usages in every cycle. So, you just check whether the resource that you need for this particular instruction is available. If so, schedule it.

(Refer Slide Time: 37:28)



If a slot is available, then schedule the instruction. No problem as such. This may involve actually unscheduling those immediate successors of the instruction, with whom there is a dependence conflict. Why? This is operation scheduling, so we have just picked up the instruction, computed its e start and put it there. It may so happen that some other instruction, which is following it in the dependence graph may be scheduled in an inappropriate slot. That may happen because of the step number 7, which we are going to see later.

There is some other instruction, which is now colliding with it because of the precedence problems, so it has happened. Now, we must remove the successor, which is colliding with our instruction, which has just now been placed and perhaps the removed instruction will find a different slot in the next scheduling cycle.

It is not possible to have any resource conflicts because we have already checked whether resources are available. Whatever precedence conflicts happened because of the successors, the successors were all removed, so they will be rescheduled again.

Suppose we are not able to find a slot, that means, the slot has already been occupied by somebody. There is a resource contention in every slot in between e start and e start plus II minus 1.

Now, just choose e start. If the instruction has not been scheduled so far, this is the first time that we are scheduling the instruction. In that case, you will choose the e start slot and forcefully fit that instruction into the start e start slot.

If we forcefully fit an instruction there, there may be some instructions, which are already fitting there, so we have to remove them. That means, unscheduling those instructions and remove them and finally schedule them in some other slot later on in the second cycle, third cycle, fourth cycle, and so on and so forth.

Choose prev minus sched time plus 1, if the instruction was previously scheduled at prev sched time. If this is the first time we are using, then e start. If it is not the first time, then it would have been scheduled previously in slot, let us say prev sched time, so we use prev sched time plus 1. If the instruction was previously scheduled at prev sched time. We had scheduled it previously, but then you know it was kicked out, and so on, so that is why we are rescheduling it now. That is what we mean. In the previous time, we had scheduled it at prev sched time. Now add 1 to it and try the next slot.

This will invariably involve unscheduling all instructions, which have resource conflicts with the instruction being scheduled. There may be instructions in the current slot or next slot, and so on and so forth, which now collide because of resource contentions. All these are now unscheduled and we are going to try scheduling all over again for these removed instructions.

This looks like a very violent case with too many instructions removed and so on. No, not necessarily. The fear may be that there may be failures and we may be going in a cycle – each instructions removing something else, just like the thrashing problem of operating systems.

Now, we are going to keep a counter. If there have been too many failures of the type 6 or 7, then increment the initiation interval and repeat the steps. Once they pass the counter value for this thrashing, rather we cross the counter value for this thrashing, we say this initiation interval is not good enough for us, and may be we need a larger initiation interval with extra space, extra cycles. So, we increment the II and repeat the scheduling all over again. All the partial schedules are thrown away and scheduling is started afresh.

(Refer Slide Time: 42:23)

Operation Scheduling

- Ready list has no use here because unscheduling of previously scheduled instructions is possible
- MRT with *II* columns and *R* rows is used to record commitments of scheduled instructions
- Conflict at time T means conflict at $T + k * II$ and $T - k * II$

$$Estart(P) = \max_{Q \in Pred(P)} \begin{cases} 0, \text{ if } Q \text{ is unscheduled} \\ max(0, SchedTime(Q) + Delay(Q, P) \\ \quad - II * Distance(Q, P)), \text{ otherwise} \end{cases}$$

$$Height(P) = \begin{cases} 0, \text{ if } P \text{ is the STOP pseudo} - op \\ \max_{Q \in Succ(P)} (Height(Q) + Delay(P, Q) - \\ \quad II * Distance(P, Q)), \text{ otherwise} \end{cases}$$

- Note that only scheduled predecessors will be considered in the computation of *Estart*

How do we compute e start and height? We know that we are going to use operation scheduling. What does operation scheduling imply? That there is no ready list because unscheduling of previous instructions is possible here in any operation schedule. So, ready list has no value.

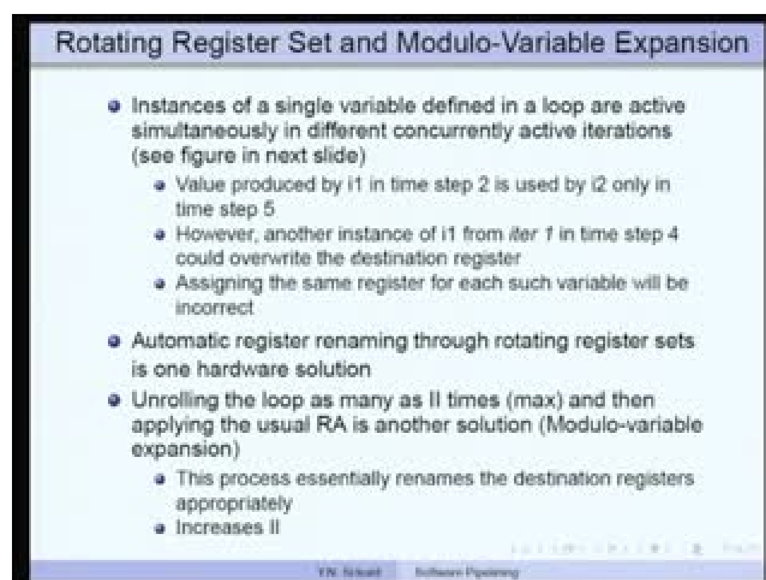
We use an MRT with II columns and R rows to record the commitments of scheduled instructions. I have already explained this before. Conflict at time T implies conflict at T plus k star II and T minus k star II. This has to be checked, rather this is certain. So, k equal to 0, that means T, that is a conflict. k equal to 1 implies T plus II and T minus II.

All these are modulo II. That is why, if we are going to begin another iteration II cycles later, conflict at T implies conflict in the next iteration, which is started II cycles later. And there would have been one iteration running II cycles earlier also, so conflict at T implies conflict in the previous iteration and the successive iteration, which was started minus II cycles before and II cycles later. That is what this really says.

What is e start? Earliest start time is 0, if Q is unscheduled. What is Q? Q is a predecessor of P. We are going to consider only scheduled predecessors in the computation of e start. Unscheduled predecessors are not used. Otherwise, if it is already scheduled, then, max of 0 coma sched time of the predecessor Q plus delay of the arc from Q to P minus II star distance Q coma P. This is the modulating moderating factor. This modulation, which is introduced by the modulating factor make sure that we take all the factors available to us for the computation of e start. So, height is similarly computed as 0, if P is the stop pseudo op – the last one – because we compute from the bottom upwards. It is max of height Q plus delay P coma Q minus II star distance P coma k. Very similarly again, this distance P coma Q, this is the factor that modifies the height. Otherwise, it would have been the height computed as in the case of instruction scheduling. Now, there is a modulating factor here, that is, the distance star II.

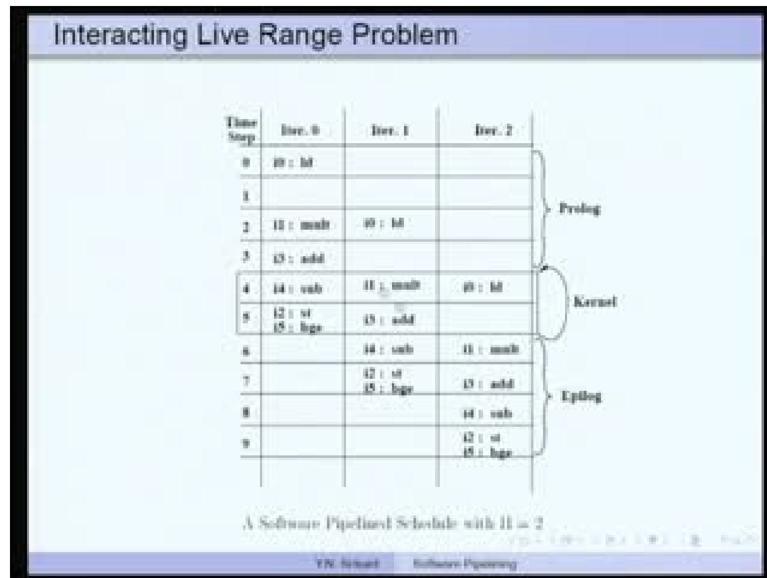
So, e start P and height P are computed in this fashion. They are used for the scheduling as I described before.

(Refer Slide Time: 45:39)



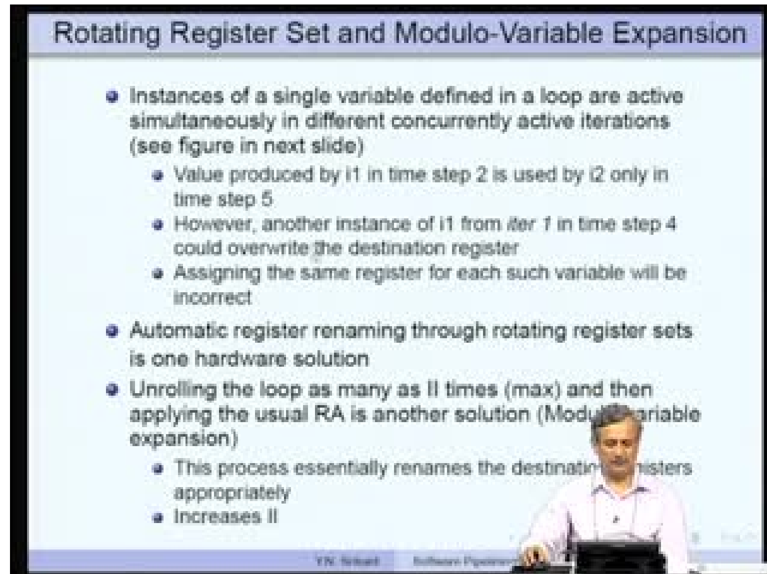
So far, we did not worry about any register requirements. What happens, there are not enough registers, and things of that kind. So, let us start looking at that particular problem. Instances of a single variable defined in the loop are actually active simultaneously in different concurrently active iterations. Let us look at the figure.

(Refer Slide Time: 46:20)

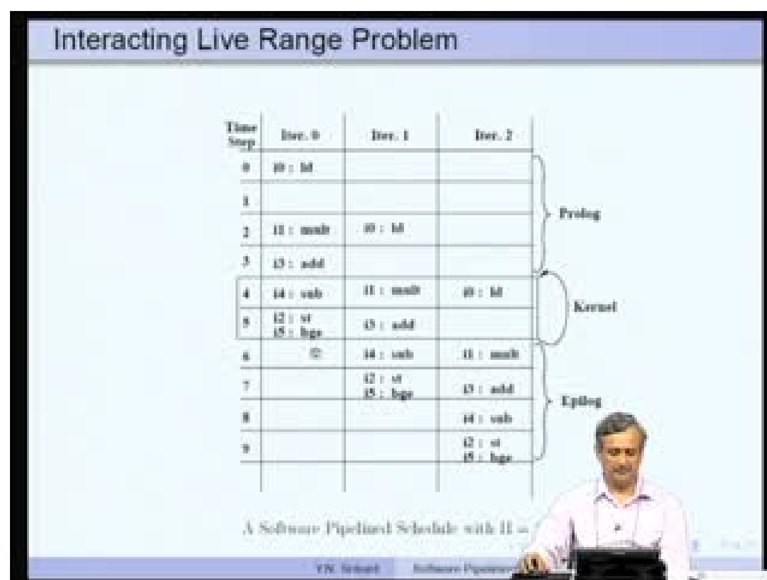


In the figure, value produced by i1 in time step 2 is used by i2 only in time step 5. So here is the figure. So i1 produces a value in time step 2 and i2 actually uses it only in time step 5. That is what is here. Now, what happened, because of the software pipeline here, another i1 has already begun. This i1 had actually produced result, this i2 had not yet consumed the result and the next i1 has already started, so it should not happen that whatever i2 is supposed to store is already over written by i1, even before i2 uses it. That is what the problem here is.

(Refer Slide Time: 47:04)

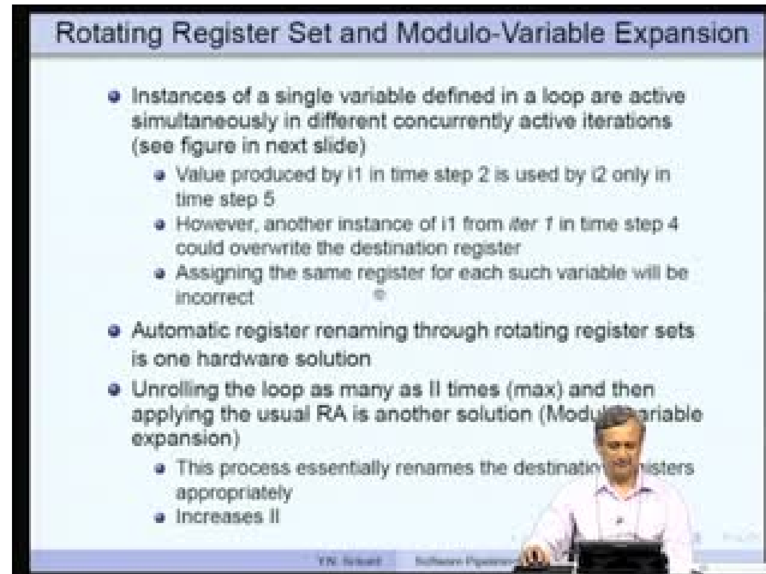


(Refer Slide Time: 47:20)

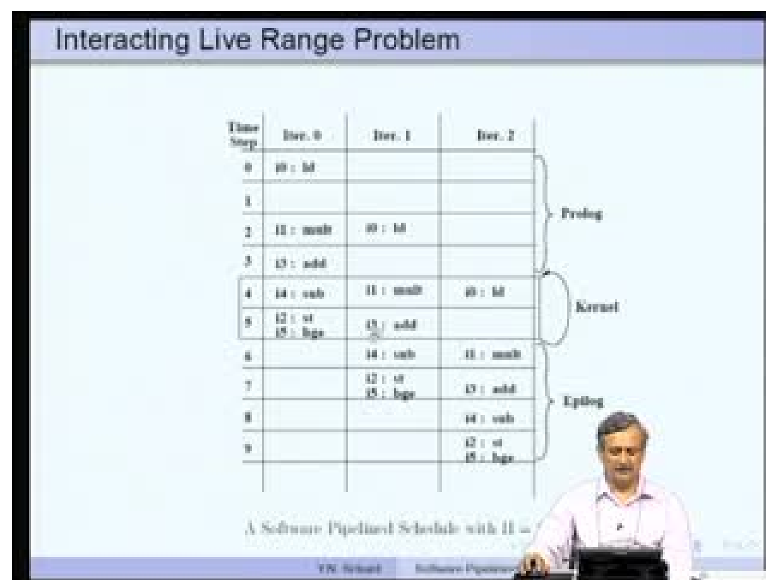


Another instance of i1 from iteration one in time step 4 could overwrite the destination register. Assigning the same register for each such variable would be incorrect. In other words, this variable which is computed by i1 and written into by i2 in the iteration 0 must get a register, which is different from the register assigned to i1. and written stored by i2 later on. This register and this register – they are in two different iterations but they are probably going to execute together. This register and this register must be separate. That is the basic idea.

(Refer Slide Time: 47:13)

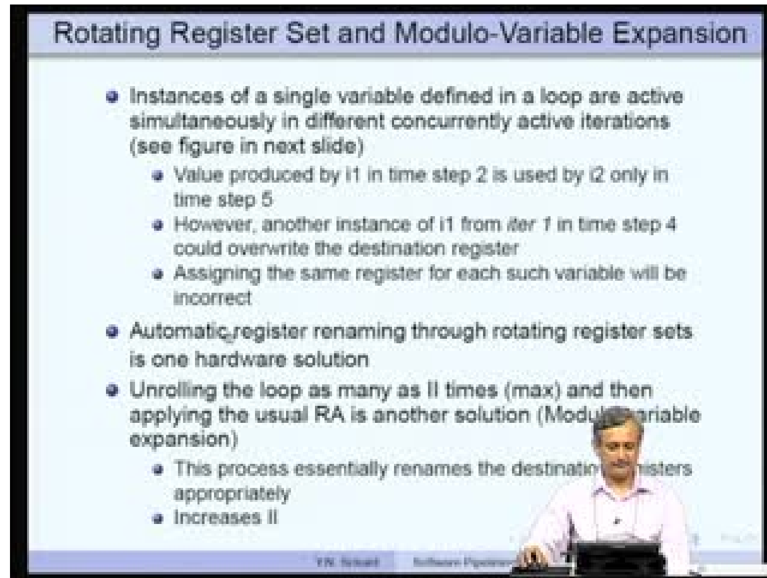


(Refer Slide Time: 48:15)



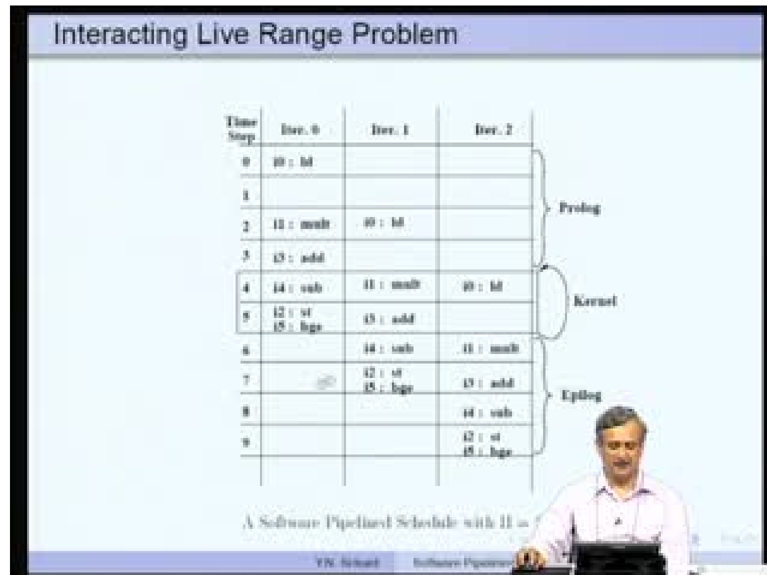
Automatic register renaming through rotating register sets is one hardware solution. Automatically, for each iteration, the register numbers are changed and a set of registers is available again for the instructions to use. So, there may be, let us say, a register r_0 available to i1 in iteration 0, this becomes r_0 prime in the second iteration, $r_{0,}$ double prime in the third iteration, and so on and so forth. That is what we mean by rotating register set. There are set of registers – a vector of registers – one for each iteration. For $r_{0,}$ itself will have different instances 1 for each iteration.

(Refer Slide Time: 48:43)



That is one hardware solution. Unrolling the loop as many times as II, that is the maximum, and then applying the usual register location is another solution. This is called modulo variable expansion. This process essentially renames the destination registers appropriately, but increases the II as well.

(Refer Slide Time: 49:06)



(Refer Slide Time: 49:59)

Rotating Register Set and Modulo-Variable Expansion

- Instances of a single variable defined in a loop are active simultaneously in different concurrently active iterations (see figure in next slide)
  - Value produced by i1 in time step 2 is used by i2 only in time step 5
  - However, another instance of i1 from *iter 1* in time step 4 could overwrite the destination register
  - Assigning the same register for each such variable will be incorrect
- Automatic register renaming through rotating register sets is one hardware solution
- Unrolling the loop as many as II times (max) and then applying the usual RA is another solution (Modulo-variable expansion)
  - This process essentially renames the destination registers appropriately
  - Increases II

What happens here is – i1 here is i2 in iteration 0. Here is another i1 and i2 in the next iteration. Suppose we unroll the loop, we have II of 2 here, so these instructions will all be here. We have unrolled the loop, then we assign two different registers for this i1 and this i1. Automatically, this i2 and this i2 will use two different registers by the usual register location, and then we find a software pipeline. There should be no difficulty in this case because there can be only two instances of i1 active in any kernel. Since we have taken care of renaming the registers, there should be no difficulty. But then we have unrolled the loop; that means, the number of instructions in the body has increased, so we may have to increase the initiation interval and hence efficiency may become lesser.

(Refer Slide Time: 50:12)

Register Spilling in Software Pipelining

- Register requirement is higher than the available no. of registers
  - Spill a few variables to memory
  - Register spills need additional loads and stores
  - If the memory unit is saturated in the kernel, and additional LD/STR cannot be scheduled
    - II value needs to be increased and loop must be rescheduled
  - Reschedule loop with a larger II but without inserting spills
    - Increased II in general reduces register requirement of the schedule
  - Generally, increasing II produces worse schedules than adding spill code

What to register spilling? Register requirement is higher than the available number of registers. So, we spill a few variables, obviously. Register spills need additional loads and stores, so if the memory is already saturated, then we may have to actually increase the II value and retry, or we could reschedule the loop with a larger II but without inserting the spills. Increased II, in general, reduces register requirement. This may be alright. Generally, it is better to spill and then increase the II, if necessary, rather than straight away increase the II, and schedule the code.

(Refer Slide Time: 50:54)

Handling Loops With Multiple Basic Blocks

- Hierarchical reduction
  - Two branches of a conditional are first scheduled independently
  - Entire conditional is them treated as a single node
    - Resource requirements is union of the resource requirements of the two branches
    - Length of schedule (latency) equal to the max of the lengths of the branches
  - After the entire loop is scheduled, conditionals are reinserted
- IF-Conversion and then scheduling the predicated code (resource usage here is the sum of the usages of the two branches)

What happens if there are if, then, and else branches inside the body of a loop? We can perform what is known as hierarchical reduction. So, two branches of a conditional are first scheduled independently -- then and else parts are scheduled independently. Now, the entire conditional is then treated as a single node. How? Resource requirements of this entire conditional is the union of resource requirements of two branches. Why? The reason is each one or either one of the branches is going to be executed, not both. So, we just take the union of these. The length of the schedule latency is equal to the maximum of the lengths of the schedules of branches. Again, the same reason; only one of them will be active at any point in time. Now, we have reduced the entire conditional to a single pseudo op with resource requirements and the delay. Now, we schedule the loop. After the entire loop is scheduled, conditionals are reinserted. So, the loop is back to its original shape and then it is used.

The second possibility is doing if conversion. If you convert if statements with predicates, then the body of the loop blows up. If there are four statements in the then part and five statements in the else part, the if converted body will contain nine statements. So, scheduling the predicated body involves more resource usage here. It is actually the sum of usages of the two branches. It is not actually the union as in the case of hierarchical reduction. You really require more resources. There will be more resource contentions as well. In general, after if conversion, you may actually end up with a higher II than in the case of hierarchical reduction. Further, you will also need predicated instructions and the hardware support for all this.

This gives you an overview of software pipelining algorithm and issues of register location, etc. This is the end of the lecture.

Thank you