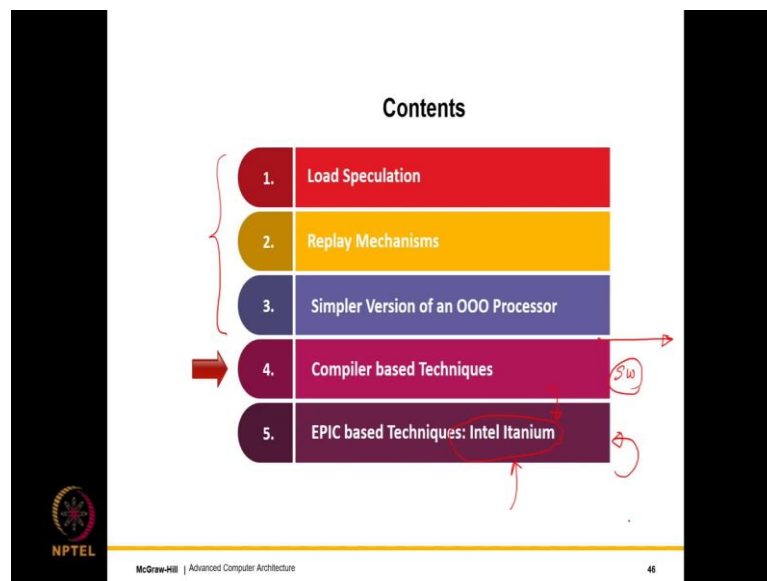


Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Module - 05
Lecture - 14
Alternative Approaches to Issue and Commit Part-III

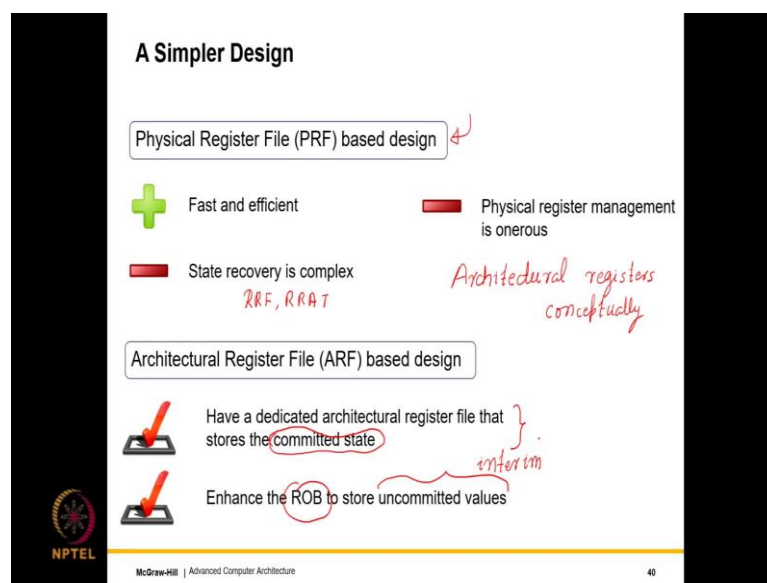
(Refer Slide Time: 00:37)



Let us now discuss the simpler version of an out of order processor. So, we have added a lot of complexity in the past several lectures. So, we started with a basic version, then we added different speculative mechanisms, then we added different replay mechanisms which of course, was required given that we had speculation already so, after that some lightweight state recovery was required which is why we introduced replay.

Let us now look at a different design which is not all that efficient, but it is a different paradigm of executing an out of order processor where we simply perform a little bit of surgery and move around units a little bit.

(Refer Slide Time: 01:23)



So, what we have seen up till now is the physical register file PRF based design, it is fast and efficient. However, there are problems, state recovery is complex, so, we need either a retirement register file or an RRA something of that type. Physical register management is hard, it is onerous, it is difficult. So, these are clearly some of the shortcomings of this scheme and these shortcomings broadly lie in the area of complexity.

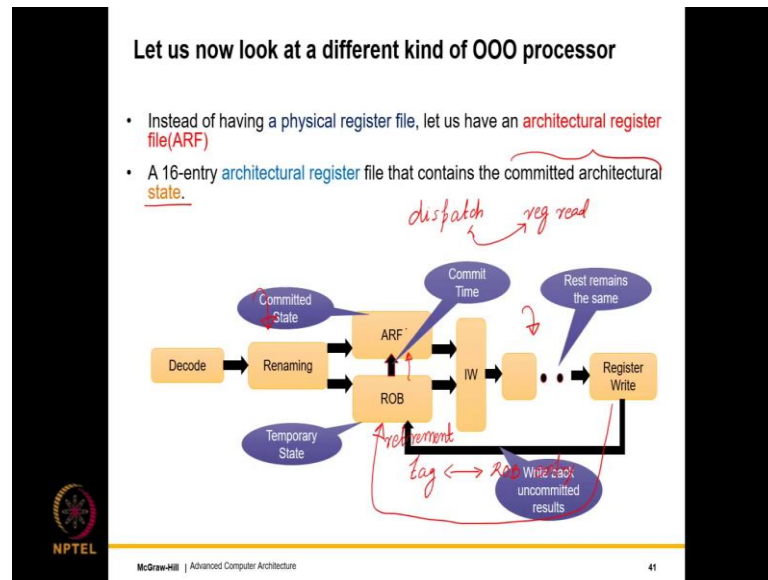
What we can instead do is that instead of having physical register files, we can have architectural registered files ARFs. So, recall that in our previous design, architectural registers only existed conceptually. so, architectural registers, their existence was only a conceptual existence, but instead, let us create a new design where we have an architectural registered file similar to the RRF that we introduced in the previous lecture or the one before that.

So, the architectural register file has it stores the committed state. So, it stores what is called the architectural state or the committed state of the program and thus, recovery is not an issue because we can always revert back to the architectural registered file. So, in this case, we do not have a PRF, but we have an ARF.

So, then, the question that automatically arises over here is that where do we store temporary values? So, temporary values recall that they were being stored in the PRF. So, these were essentially interim values that were produced by instructions in flight. So,

in this case, what we do is that we enhance the reorder buffer with additional value storage capability to store uncommitted values such that all the interim values get stored in the ROB, the ROB is thus wider and fatter and it has more features so to speak.

(Refer Slide Time: 04:03)



So, what was the crux of our discussion in the previous slide that instead of a PRF, we use an architectural registered file or an ARF, this contains the committed architectural state. So, what are the stages that we are looking at right now? Well, decode and renaming remain the same in this design as well.

However, after renaming before dispatch. in this case, essentially the order of dispatch and register read, this order is swapped. So, we read the registers first. So, we read the architectural register file first if our rename table indicates that the value will be found here otherwise, if it is an temporary value, it is an interim value, then we read it from the ROB.

So, the ARF in this case does contain the committed state, but if let us say this contains the most recent version, the most recent value for register, we read it from the ARF otherwise, if there is an instruction in flight that is writing to the same register, we read it from the ROB because the ROB in this case is enhanced.

After we read all the values that are ready, we proceed and also, I have added one more arrow here so, I am showing the animation once again so, values proceed from the ROB

to the ARF at the time of retirement. so, at the time of commit, commit is also called retirement values proceed from the ROB to the ARF at commit time.

Subsequently, they are dispersed, they are returned to the instruction window and then, we have a regular the rest of the mechanism is the same in the sense of the values that have not been produced.

Well, we have the same wakeup, broadcast, select mechanism, execution mechanism, all remains the same, it is just that in this case the tag is the ROB entry basically, instead of the tag being the physical registered ID is ROB entry and so, the values are read once after renaming, they are not read here which was the case in the earlier design.

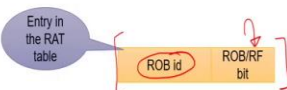
So, this underscores the fact that the rest of the pipeline remains the same. So, simplicity basically is that we have gotten rid of the PRF and all other circuitry including the free list and so on that was associated with the PRF and then finally, we write to the register and of course, the register write phase in this case, the write is actually written to the ROB entry and this value only at commit time goes to the ARF, the architectural register file.

So, as you can see here, I will just that the uncommitted results which the register write stage produces that is written to the ROB entry and these values move to the ARF at commit time.

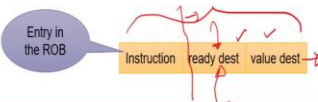
(Refer Slide Time: 08:05)

Changes to renaming

Entry in the **RAT** table



- **ROB/RF** bit → 1 (value in the ROB), 0 (value in the ARF)
- Use the **ROB** if the ROB/RF bit indicates that the value might be there in the ROB
- **Entry in the ROB:** (ready bit indicates if the value is in the ROB (1) or being generated in the pipeline (0))



NPTEL

McGraw-Hill | Advanced Computer Architecture

42

So, let us now look at an entry in the RAT table. So, an entry in the RAT table looks something like this. So, it has an ROB RF bit that is the most important bit over here that is pretty much the operative part of this design. So, this bit basically says will the value be found in the ROB, 1 if the value will be found in ROB, 0 if the value will be found in the architectural register file or ARF.

So, this essentially indicate that whether the value is coming from the committed state or whether the value is being produced by some other instruction in flight, this is what is set is being set. Subsequently, if the value is coming from the ROB, then well, the idea of the ROB entry is stored, and these two pieces of information are what we find in the rename table or the RAT table.

So, we use the ROB if the ROB RF bit indicates that the value might be there in the ROB. Subsequently, what we do is we access the entry in the ROB. The entry in the ROB well, it will have a lot of fields, but I am only showing a subset of the fields that are relevant so, of course, it will have some copy of the instruction packet. So, let us only consider these two fields, this and this.

So, we will have the value of the destination well, if the value has been generated, then yes and then, we will have a bit that indicates if the value is in the ROB or is currently being generated in the pipeline via an in flight instruction. So, the ready bit in this case which is something similar to the available bit that we had in the previous design that was attached to each entry in the rename table, in this case, the ready bit associated with the destination, this essentially says that whether the value is currently there in the ROB entry or not or whether we need to wait for it and get it via the regular work cast, wakeup, bypass mechanism.

So, basically this is what, this ready bit is indicating, and this is very important. So, this is essentially telling us that look either you take the value that is there which is this or you take it from the pipeline using the wakeup, select broadcast, bypass mechanism that we have already discussed.

(Refer Slide Time: 11:03)

Changes to Dispatch and Wakeup

Each entry in the **IW** now stores the values of the operands

- Reason: We will not be accessing the RF again

What is the tag in this case?


- It is not the id of the physical register.
- It is the id of the **ROB** entry.

What else?

- Along with the **tag**, we need to **broadcast** the value of the operand also, if ~~we will not get the value from the bypass network~~
- This will make the circuit **slower**

(fatter)
- area
- latency
- power

(wakeup)



NPTEL

McGraw-Hill | Advanced Computer Architecture

43

So, each entry in the instruction window gets modified. Previously, it was not storing the values of the operands. In this case, it is storing the values of the operands well, because there is no choice because that we have already accessed the register files, we are not going to access the register file. subsequently, hence each entry of the instruction window also needs to store the values.

This kind of makes each entry fatter, wider, broader whatever may be the context over here whichever objective you would like to use, but pretty much it increases the size and the moment you do that well, negative consequences in terms of area, negative consequences in terms of latency structure becomes slower and clearly power. So, it is kind of an all-round bad idea, it is no doubt simpler, but again simplicity and efficiency are not known to go together.

So, as I said well each entry needs to store the values because we will not access the RF, the register file again. What is the tag in this case? Well, it is not the id of the physical register as it used to be, it is the id of the ROB entry. What else? Along with the tag, we also need to broadcast the value because well where else will it get the value from?

So, because there is no internal storage happening in this part of the pipeline unless the value is broadcasted, we will not get the value from anywhere else because we are also not reading the register file. Hence, the value also needs to be broadcast, this needs to be

done, the value of the operand also because we will well, along with the tag, we need to broadcast the value well of the operand also.

So, see essentially the idea is that even if we do not pick up the value from the broadcast network, you can always argue that look the instruction will get selected and we will pick it up from the bypass network, but it will not happen you know in such a simplistic fashion because it is possible that this cycle the instruction may wake up.

But it actually might end up getting selected 10 cycles later or maybe one operand wakes up and other operands still has not woken up and clearly the value will not be there in the bypass network for such a long time because of that, the instruction window needs to temporarily buffer the value.

So, this part of the sentence should be changed that we essentially need to broadcast the values operand also. Well, I think what is missing here is a comma basically, if we will not get the value from the bypass network.

(Refer Slide Time: 14:37)

Changes to Dispatch and Wakeup

Each entry in the IW now stores the values of the operands

- **Reason:** We will not be accessing the RF again

What is the tag in this case?

- It is not the id of the **physical register**.
- It is the id of the **ROB** entry.

What else?

- Along with the **tag**, we need to **broadcast** the value of the **operand** also if we will not get the value from the **bypass** network
- This will make the circuit **slower**

Handwritten annotations:

- selected later
- 64 → other operand might not have woken up

NPTEL

McGraw-Hill | Advanced Computer Architecture

43

So, let me fix this slide. So, what this is saying is that if we will not get it from the bypass network well. Why? Because we can get selected much later that is one. The second is other operand might not have woken up. So, because of these two reasons, it is possible that we will not get it from the bypass network. So, this is why along with the

tag, you send the value such that the value is there for sure. So, this will clearly make the circuit slower.

So, let us see if we have 128 entries in ROB well, it is a 7-bit tag and recall that even previously with the PRF, we were broadcasting 7-bits, but in this case, a fat 64-bit value needs to be broadcasted and this will increase the inefficiency of the system and make it slower.

(Refer Slide Time: 15:53)

Changes to Wakeup, Bypass, Reg. Write and Commit

- We can follow the same **speculative wakeup** strategy and broadcast a tag (in this case, id of ROB entry) immediately after an instruction is selected. **Tags+values** are broadcast when the instruction is in the write-back stage.
- Instructions directly proceed from the **select** unit to the execution units
- All tags are **ROB** ids. *RF-ROB*
- After execution, we write the **result** to the **ROB entry**
- Commit is simple. We always have the **architectural state** in the **ARF**. *forward --*
- We just need to flush the ROB. *rename table*

NPTEL
McGraw-Hill | Advanced Computer Architecture
44

So, changes to wake up, bypass, the register write and commit well, we can follow the same speculative wakeup strategy and broadcast a tag.

In this case, the tag of course, changes, it is the ROB entry immediately after the select that is and also what is actually broadcast is a tag plus a value, tags plus values are broadcasted and this is the change at this stage, then instructions directly proceed from the select unit to the execution units. So, they there is no register file access stage in between which used to be there, this is not there.

All tags are ROB ids. After execution, we write the result to the ROB entry instead of the physical register. Commit is simple. So, this is important, commit is rather simple. Essentially, what happens is that we save the architectural state in the ARF so, whenever we commit, we write a value to the ARF.

This is again, as I said this is again a power-hungry operation. So, power wise you can see it is inefficient, I am putting a (- -) here. so, the commit part is not power efficient at all, but at least the architectural state is in the ARF. So, this is working in the same manner as the RRF that we had discussed earlier.

And if let us say you want to flush well after a misprediction, all that we do is we flush the ROB and we just recover from the ARF and of course, the rename table has to be modified to incorporate this aspect of execution, but essentially, recovering from a misprediction in this case is more straightforward.

(Refer Slide Time: 17:55)

PRF based design vs ARF based design

+ points in the PRF based design

- A **value** resides in only a single **location** (PRF). Multiple **copies** of values are never maintained. In a 64-bit **machine**, a value is 64 bits wide.
- Each entry in the **IW** is smaller (values are not saved).
- The broadcast also uses 7-bit tags (*id of the phy. register*)
- Restoring state is complicated *← complex.*

+ points in the ARF based design

- Recovery from **misspeculation** is easy *↖*
- We do not need a **free list**
- Values are stored at **multiple places** (ARF, ROB, IW) *↓ ↓ ↓*

NPTEL
McGraw-Hill | Advanced Computer Architecture
45

Let us compare the positive and negative points. Well, the positive points in the PRF based design which are plenty is that a value resides in only a single location. Multiple copies of values are never maintained.

So, this is important to bear in mind that multiple copies of values are never maintained, they are only there in the PRF nowhere else and if let us say we use an RRF which we have already said is inefficient, but if we use an R RAT, then this is definitely true and also, recall that in a 64-bit machine, all values inside the data path are all 64-bits wide so, storing values is inefficient there is no doubt about it.

Each entry in instruction window is smaller again because values are not saved. So, the crux of this design is that do not store big values. The broadcast also uses 7-bit tags

which is essentially the idea of the physical register. Restoring state is complicated, there are complexities with the design so, this is clearly a complex design.

Positive points in the ARF based design. Well, recovery from the mis-speculation is easy, no doubt that is because the ARF functions as a register file that stores committed data, we do not need a free list, we do not have to manage physical registers simpler is just that values are stored well that itself is bad and to make matters worse, values are stored at multiple places namely the ARF, the ROB and the instruction window.

So, as we can see clearly there is a latency power trade off with complexity. So, depending upon what we would like to do and what is the target workload, what is the power budget? how much performance do we want and also, the skills of the engineers can they deal with a more complex design or not, all of these factors have to be taken into account before choosing either a PRF based design or an ARF based design.

So, we have finished the hardware aspect of the processors design. So, this is great in the sense that at least this is an important milestone for us that all the hardware related techniques they are done.

So, we will now discuss a few software techniques. So, software techniques primarily use the compiler to optimise the code and then, hardware can pick up from there in the sense that the compiler can let say optimise 80% and the hardware can use those optimizations and do the remaining 20%, but primarily, it should be understood that these are compiler-based techniques.

And then, we will reach the zenith of compiler-based techniques with Intel Itanium for almost all the work of the processor which is scheduling dispatch rename you name it, all of it is outsourced to software. So, this is essentially a natural extension of this, let us first look at some simple techniques and then, move to Intel Itanium.

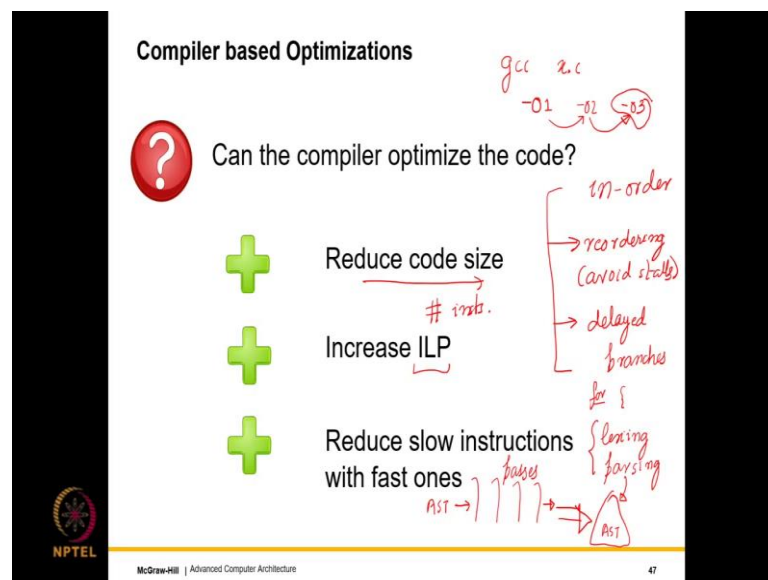
So, Intel Itanium for people who have not heard of it is a flagship Intel server processor for very high-end servers.

So, there is an Intel Itanium 1, Itanium 2, I am not very sure if there is an Itanium 3 or not, but it was a joint effort between Intel and HP to produce a server chip that relies on the compiler heavily and this is an important paradigm of course, Itanium did not see

great days after that so, it is not extremely popular, but nevertheless, it is a very revolutionary design in this space and popularity has many factors depends on the market and different aspects of it.

So, let us have a logical separation between an academic discussion and a market discussion. so, they essentially belong to two different courses.

(Refer Slide Time: 22:43)



So, the key question that we would like to ask is can the compiler optimise the code? Well, yes, before you ask me, it is yes. The reason being that we have already seen this happen. In the case of an in-order pipeline, we have already seen the compiler do a lot. Well, what was it doing? It was reordering the code to avoid stalls.

So, let us say between a load use hazard, it was in inserting some other instruction, then it was leveraging delayed branches to bring instructions before the branch to the two delay slots after it so, this was also reducing stalls and the compiler was also ensuring correctness if the hardware did not have interlocks. So, the compiler did play a role.

So, we can see it can do more, it can reduce the code size which is the number of dynamic instructions that can be done. It can increase the ILP. So, it can identify more instructions that can be sort of co-located such that they are fetch together and we can execute more in parallel. It can reduce slow instructions with faster variants which we

shall see this is known as strength reduction, a very important from the point of view of performance and power also and the modern compilers actually do much more.

So, I would like to add one thing that in most undergraduate compiler courses, they only focus on lexing and parsing which is essentially known as the front-end. So, what does lexing do? Well, it takes a C program, if let us say you are building a C-compiler, breaks it into individual tokens like a for loop the word for is a token, then the brace is a token, a variable is a token and so on and these tokens are then sent to a parser like Yacc which makes a parse tree and parse tree is basically a data structure that represents the program, this is further optimise and we generate the code.

So, most modern compiler courses especially at the graduate level, they do not focus on lexing and parsing, they assume that all of this is done, this is easy technology, they just focus on what is known as the abstract syntax tree which is an output of parsing actually.

So, the abstract syntax tree is then processed, processed, processed. so, then like an assembly line, we have different compiler parsers, it just do different operations on the abstract syntax tree and keep on making it more efficient and finally, we generate the code and we produce the binary.

So, these compiler passes, these are something that we will study, and most modern compilers have different optimization levels were depending upon what you specify they will either have more passes or less passes. So, more or the number of passes, more efficient is the code and also, it takes more time to compile. So, if I just do gcc x.c well, this is the default, it will do some optimization not a lot, I can add a flag (-o1) optimization level 1, it will do more (-o2), even more (-o3) maximum.

So, you I invite you to do this that you take a C program and then, you enable different compiler optimizations, run the C program, and see how long it takes. You will see that gradually as you increase the level of optimization, the program becomes faster, faster, and faster and this is a great advantage of modern compilers that look if you want to compile very quickly, then no problem, use a lower optimization level, but then your final code will be slower or you want to spend more time during compilation, in any case not much use (-o3).

(Refer Slide Time: 27:09)

Constant Folding

```
int a = 4 + 6;  
int b = a * 2;  
int c = b * b;
```

Handwritten notes:
 $a = 10$
 $b = 20$
 $c = 400$

→ We can directly **replace** a with 10, b with 20, and c with 400

NPTEL

McGraw-Hill | Advanced Computer Architecture

48

So, let us now come to our first optimization which is known as constant folding. Take a look at the code over here. If you look at it, we are doing arithmetic operations, but if you think about it, there is no reason to actually do them do the addition for example, we can instead analyse this in the compiler, most compilers will actually not do the addition, they will just say $a = 10$, just set $a = 10$.

Once a has been set to 10, most compilers will not do the multiplication, they will just set $b = 20$. Once that has happened, most compilers will just set c to 400 and we are thus avoiding two multiply instructions, two add instructions and also, one additional instruction of putting 4 into a register because in most ISAs, we cannot have two immediates as arguments.

So, given that this piece of code can be optimized somewhat significantly, constant folding is one of the passes, one of the default passes actually in most compilers. So, most compilers even without asking you, they will do it and this will produce reasonably efficient code because most programmers will not or are not expected to know what is efficiency that is why compilers try to help them in the sense, that if programmers are trying to shoot themselves in the foot, compilers give them bulletproof shoes most of the time.

(Refer Slide Time: 28:51)

Strength Reduction

```
int b = a * 8;
int d = c / 4;
int e = b * 12;
```

→

```
int b = a << 3;
int d = c >> 2;
int e = b << 2 + b << 3;
```

fast *slow*

slow *fast*

x4 *x8*

The next optimization is strength reduction. So, often we perform several arithmetic operations of this type. So, these arithmetic operations over here, they one is $a \times 8$, $/ 4$, $\times 12$. So, we very frequently also we compute the half of a value or the quarter of a value and because of this, we typically have multiply and divide operations of this type.

Multiply and divide operations are slow in the sense that they take many cycles, many many cycles and it is much better to replace them with variants that take a single cycle. So, let us say well $\times 8$, this is equivalent to left shifting by 3 positions, So, they say $\times 8$ is tantamount to left shifting by 3 and $/ 4$ is the same as right shifting by 2 positions.

Furthermore, what many smart compilers can do is multiplying let us say something by 12, this can be replaced as the sum of $\times 4$ and $\times 8$. So, this can be replaced as a sum of two shifts, and one add.

So, of course, this depends on the machine in the sense a single operation over here is being replaced with three operations, but sometimes in some machines, this can turn out to be a faster operation mainly because we are parallelizing this so, both of these shifts can be computed in parallel and then of course, we need to perform the addition.

So, all of them are fast operations and let us say that at that point of time, we have three functional units, then this is not that bad an idea. Even though, the conversion of into 12 into this form, this is rare, but nevertheless these two are very popular, these two kinds of

(Refer Slide Time: 31:33)

Common Subexpression Elimination

→

```
int c = (a + b) * 10;  
int d = (a + b) * (a + b);
```

 } *complex*

→

```
int t1 = a + b;  
int c = t1 * 10;  
int d = t1 * t1;
```

] { $\begin{matrix} - \\ - \\ - \\ - \\ - \end{matrix}$ } → { $\begin{matrix} - \\ - \\ - \\ - \\ - \end{matrix}$ }

- Each line in the second example corresponds to one line of assembly code.
- We **do not** compute $(a+b)$ many times.

McGraw-Hill | Advanced Computer Architecture

NPTEL

50

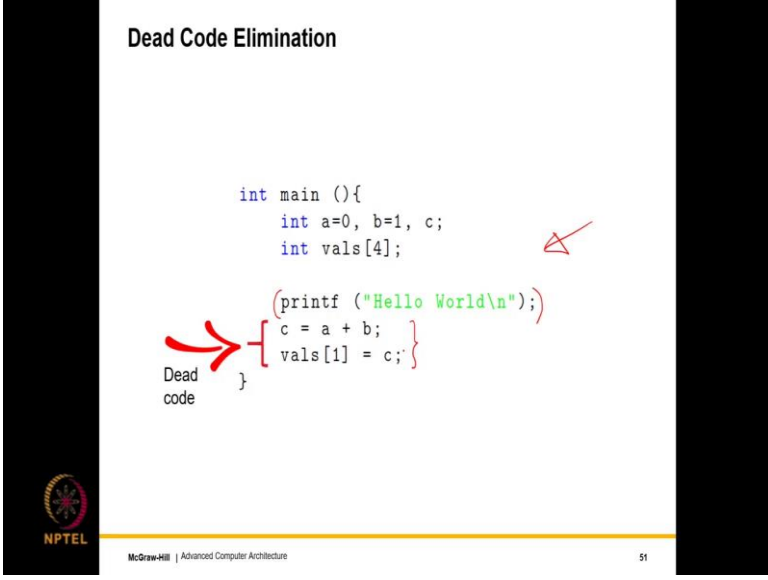
Another common optimization is called common subexpression elimination. So, let us look at these pieces of code. So, in this piece of code, what we see is that the expression $(a + b)$ appears several times. So, if you do not do anything in a naive implementation, we would actually perform this addition three times which is expensive.

However, most compilers today would pre-compute $(a + b)$, store it in a temporary value let us say a register, let us call it t1. So, the way that this code has been written over here and also, the same holds for the previous example that we have. So, each line of C code is very simple. so, it is simplistic in the sense this is complex, but the bottom part is simple and let us say in this example as you can see each line of code corresponds to one line of assembly code.

So, what we do is we compute $(a + b)$, put it in `t1` can be register, then we compute $c = t1 * 10$ and we compute $d = t1 * t1$. So, each line in the second example here, did not hold exactly for the previous in the previous slide, but in this slide, each line in the second example corresponds to one line of assembly code. So, we do not have to compute $(a + b)$ many times, we just compute it once. So, identifying such common subexpressions in arithmetic expressions is something that modern compilers do very well.

So, that is the reason when we have a series of arithmetic expressions, they are able to reduce that to a smaller sequence of computations which uses a combination of constant folding strength reduction as well as common subexpression elimination to reduce the number of instructions as well as to replace them with faster variants.

(Refer Slide Time: 33:57)



The slide is titled "Dead Code Elimination". It displays a C program snippet:

```
int main () {  
    int a=0, b=1, c;  
    int vals[4];  
  
    (printf ("Hello World\n");)  
    {  
        c = a + b;  
        vals[1] = c;  
    }  
}
```

A red arrow points from the text "Dead code" to a bracket that encompasses the block of code containing `c = a + b;` and `vals[1] = c;`. A red checkmark is placed to the right of the code block.

NPTEL

McGraw-Hill | Advanced Computer Architecture

51

Now, let us look at one more kind of optimization known as dead code elimination. So, let us take a look at this program over here. So, any program finally, what matters is what is written to the output. So, the output in this case is Hello World and if you think about it, these two statements have no value.

So, this is dead code in the sense that whatever they compute, this is not a final output of the program. so, it is not leading to a printf statement given the fact that it is not leading to a print statement, this piece of code has no use, it is called dead code.

This I would say is not a part of the default optimization level at least as of today in many compilers, but still, with more advanced optimization levels like o3 and so on it will identify dead code and it will remove it. So, this will reduce the number of instructions that your program executes.

(Refer Slide Time: 35:11)

Silent Stores

```
int arr[5], a, b, c;

arr[1] = 3;
a = 29;
b = a * arr[0];
arr[1] = 3; /* Not required */
printf ("%d \n", (arr[1] + b));
```

Silent store

- Silent stores write the same value that is already present

NPTEL

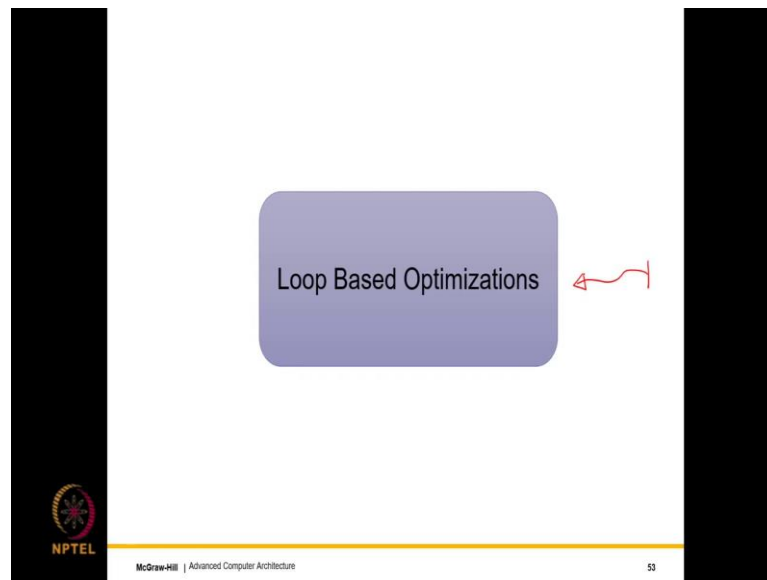
McGraw-Hill | Advanced Computer Architecture

52

A silent store is somewhat more sophisticated version of dead code, often it is hard to identify at compile time, but whenever it is identified, it should be removed. So, what we see over here is that we set the first element of the `arr[1] = 3` and then, do a little bit more of computation. The key operative point over here. So, of course, there are many lines, I am not showing them, but I am only focusing on this part. So, the key operative point over here is that we set the first element of the `arr[1] = 3` and we are setting it to 3 once again.

So, this process is problematic. The reason it is problematic is that if 3 is already written in the memory location, we are writing 3 once again which is not, I mean establish which is not achieving any purpose. so, this is a silent stored in the sense that even if this store was not there, nothing would have happened. So, the right the same value that is already present so, this store can be and should be removed.

(Refer Slide Time: 36:31)



So, now, we have discussed a couple of regular simple optimizations starting with constant folding, strength reduction, dead code removal, silent stores, most of the common subexpression elimination we have discussed five. So, these five are pretty much present in most compilers as of today and compilers furthermore optimized loops because a program spends most of its time, some say 90% of its time in loops and thus, optimizing loops is of paramount importance.

(Refer Slide Time: 37:13)

Loop Invariant based Code Motion

Original

```
for (i=0; i<N; i++){  
    val = 5;  
    A[i] = val;  
}
```

Loop Invariants Moved

```
val = 5;  
for (i=0; i<N; i++){  
    A[i] = val;  
}
```

- There is no point setting (val = 5) repeatedly.

The diagram illustrates the transformation of a loop by moving a loop-invariant statement out of the loop body. In the "Original" code, the statement `val = 5;` is inside the `for` loop. A red arrow points from this statement to the "Loop Invariants Moved" section, where it is placed before the `for` loop. Another red arrow points from the `val = 5;` statement in the "Loop Invariants Moved" section back to the `val = 5;` statement in the "Original" code, indicating the movement. The "Loop Invariants Moved" section shows the `val = 5;` statement followed by the `for` loop. The "Original" section shows the `for` loop with `val = 5;` inside it. The "Loop Invariants Moved" section shows the `val = 5;` statement followed by the `for` loop. The "Original" section shows the `for` loop with `val = 5;` inside it.

So, let us take a look at this piece of code. So, this piece of code over here assuming N is a very large number is setting the variable $val = 5$ in every single iteration. So, in every single iteration, what we are doing is that we are setting $val = 5$ which needless to say is not something that we would like to do over and over again because it is increasing the number of instructions within the loop and that too significantly N is a very large number if here number of times and this is being done will be N times and if N is a million, we will do this million times and each one of it is a silent store.

So, what we do is that we move this instruction over here, we move it over here and this can be done automatically, most compilers have sophisticated analysis process to identify such patterns, they will identify this pattern and they will move $val = 5$ before the loop. So, this is also called a loop invariant in the sense that this is not varying in the loop. so, this remains constant for all the iterations of the loop.

So, this can be moved to a point which is before the loop because there is no point in setting $val = 5$ repeatedly and this will make a loop function very efficiently in the sense out of two C statements, we are removing one and that is substantial.

(Refer Slide Time: 38:59)

Induction Variable based Optimization

Original

```
for (i=0; i<N; i++){  
    j = 6*i;  
    A[i] = B[j] + C[j];  
}
```

Induction variable

Optimized

```
j = 0;  
for (i=0; i<N; i++){  
    j = j + 6;  
    A[i] = B[j] + C[j];  
}
```

Replace a multiply with an add

- An add operation is **faster** than a multiply operation. Hence, it makes sense to **replace** multiplies with adds.

NPTEL

McGraw-Hill | Advanced Computer Architecture

55

The other is induction variable-based optimization. So, in this case, the variable j which is derived from the loop variable i is known as an induction variable. So, as I mentioned the induction variable is derived from the loop variable. In this case, it is $6 * i$ and then,

we have some expressions that it involves both i and j . So, the particular instruction that we are concerned with the statement is $j = 6 * i$.

So, we do not seem to like that. Why? Because we do not like the multiplication instruction, our claim is that the multiplication instruction is slow that is our claim. So, that is why we do not like it. What we would instead like to do is that we would try to replace a multiply with an add which is much much faster if possible and most compilers can recognise this pattern.

So, they will set $j = 0$ outside the loop and simply replace $j = j + 6$ which is the same as this. If we just take a look at it, i is being incremented by 1 every cycle. This means that j is being incremented by 6 every cycle and incrementing by 6 every cycle $6 * i$ is basically $j = j + 6$ and this is by far a more efficient method of dealing with multiplies which is convert them to an addition in this way.

And since this is happening within a loop, the potential advantages are much more because as I said programs spend most of their time roughly 90% of the time within a loop and this is why we get to see the advantage over here.

(Refer Slide Time: 41:09)

Loop Fusion

Original

```
for (i=0; i<N; i++) /* Loop 1 */  
{  
    A[i] = 0;  
    for (i=0; i<N; i++) /* Loop 2 */  
    {  
        B[i] = 0;  
    }  
}
```

loop overhead

Fuse the loops

Optimized

```
for (i=0; i<N; i++){ /* Loop 1 */  
    A[i] = 0;  
    B[i] = 0;  
}
```

programmer

- Loop fusion **reduces** the instruction **count** and the number of branches significantly

NPTEL

McGraw-Hill | Advanced Computer Architecture

56

Another idea loop fusion well, let us take a look at these two loops. So, we have two loops, and we iterate from 0 to $(N - 1)$ and in the two loops all that we do is we set elements of one array to 0 and the elements of another array to 0 $A[i]$ and $B[i]$. So, what

we can do is we can fuse the loops in the sense, we can create an optimized loop that will set A i and B i both to 0 at the same time.

So, the additional overhead of the branches and the checks and the loop overhead right of the second loop will go away. So, this will reduce the number of runtime instructions and bring in efficiency. The idea here is that we completely got rid of these instructions, we just have a single loop now and we fuse this operation and this operation.

So, loop fusion is also done by compilers, but sometimes you have a loops are complicated so, the programmers are supposed to write efficient code such that they fuse the loops without bringing in, without passing on the effort to the compiler, the programmers sometimes should do it particularly, if in the feel that it is complicated something that the compiler might not catch.

So, loop fusion will reduce the instruction count drastically and also the number of branches quite significantly. So, as I said it will remove the overhead of one of these for loops completely and that will bring in the desired amount of efficiency that we needed which basically means that instead of two loops, we will just have one.

(Refer Slide Time: 43:01)

Loop Unrolling - I

```
sum = 0;
for (i=0; i<10; i++){
    sum = sum + i;
}
```

Original loop

Assembly code

```
mov r0, 0      /* sum = 0 */
mov r1, 0      /* i = 0 */

loop:
  cmp r1, 10   /* if (i == 10) exit */
  beq .exit
  add r0, r0, r1 /* sum = sum + i */
  add r1, r1, 1 /* i = i + 1 */
  b .loop      /* next iteration */

exit:
```

NPTEL

McGraw-Hill | Advanced Computer Architecture

57

Now, let us come to loop unrolling which by far is maybe the most popular optimization in this space. So, let us take a look at this. So, loop unrolling incidentally used to be much more popular in the older days when we had in order pipelines, but even now also,

loop unrolling is extremely popular and it has many other applications other than this simple optimization as I will show because it forms the basis for far more sophisticated optimizations and also while programming hardware, when we have loops in Verilog or VHDL, they are also unrolled and a lot of work on the loops is done.

So, let us consider a simple 10 iteration loop, all that it does is just does $\text{sum} = \text{sum} + i$ and of course, sum is initialized to 0. So, let us take a look at the assembly code r0 is mapped to sum, r1 is mapped a loop iterator i, every iteration we compare r1 with 10 if it is equal, we just exit. otherwise, we just add the add; add to the sum, we add to i and we jump to the next iteration.

So, this assembly code is simple. We have seen something similar many many times in the previous lectures, I am not going into this in detail.

(Refer Slide Time: 44:35)

Loop Unrolling - II

C code

```
for (i=0; i<10; i+=2){
    sum = sum + i + (i+1);
}
```

(5 iterations)
(equivalence)

$\times 2$
 $\text{sum} += (i \ll 1) + 1$

5

Assembly code

```
mov r0, 0      /* sum = 0 */
mov r1, 0      /* i = 0 */

.loop:
cmp r1, 10     /* if (i == 10) exit */
beq .exit
add r0, r0, r1 /* sum = sum + i */
add r1, r1, 1  /* i = i + 1 */
add r0, r0, r1 /* sum = sum + i */
add r1, r1, 1  /* i = i + 1 */
b .loop
.exit:
```

loop code

Advantage: fewer total instructions and specifically fewer branch instructions

{ fewer mis predictions }

↓

So, what we can do is that these 10 iterations can be fused into groups of two iterations. So, instead of incrementing i by 1, we can increment i by 2 and we can replace this sum operation with this which as you can see with the previous slide, this is equivalent. So, there is a degree of equivalence, it is the same basically what we are doing, it is just that we are doing more work within the loop and instead of 10 iterations, we have reduced the number of iterations to 5.

Given that we have reduced it to 5, there is an advantage. The advantage over here is that we have reduced the loop overhead which is essentially the branches and the checks and all of these. So, let me come to the assembly code as you can see, it will be visible very clearly. So, we set sum to 0, we set i to 0 well fine, then this is the main body of the loop. So, we do the same comparison, compare r1 with 10 and if it is equal, we exit. So, this is basically the loop code.

Now, what we do is we do something interesting. We compute $\text{sum} = (\text{sum} + i)$. We increment i with 1 and add it to the sum once again or we could have multiplied $i * 2$, added 1 to that and then added that to the sum whatever is more efficient. So, what we could have done is we could have done $\text{sum} += (i << 1)$ which is same as $X \ 2$ and then, we could have $+ 1$. So, this would have done, this would have had the same effect.

So, regardless how we do it? We compute the updated value of sum and we also increment if you see we increment i twice, incremented once's here, incremented once's there, but the advantage is that we reduce the number of; the number of instructions that implement the loop code, these two instructions implement the loop code and this instruction implements the code of the loop.

So, we are basically reducing branches that is what it comes down to, we are reducing branches and other code that is important in maintaining the number of iterations of the loop. So, well, reducing branches has two advantages. So, the minor advantage is that we are also reducing the total number of instructions which needless to say is beneficial. So, that we have already seen from the performance equation. The specifically, we have fewer branch instructions, and this is the crux.

So, fewer branch instructions mean fewer mispredictions, a fewer times you have to deal with the branch predictor. So, even accessing the branch predictor requires time and what happens is that in a modern program or the loop itself will have a lot of instructions, these instructions can destructively areas with the branch that can increase miss-predictions.

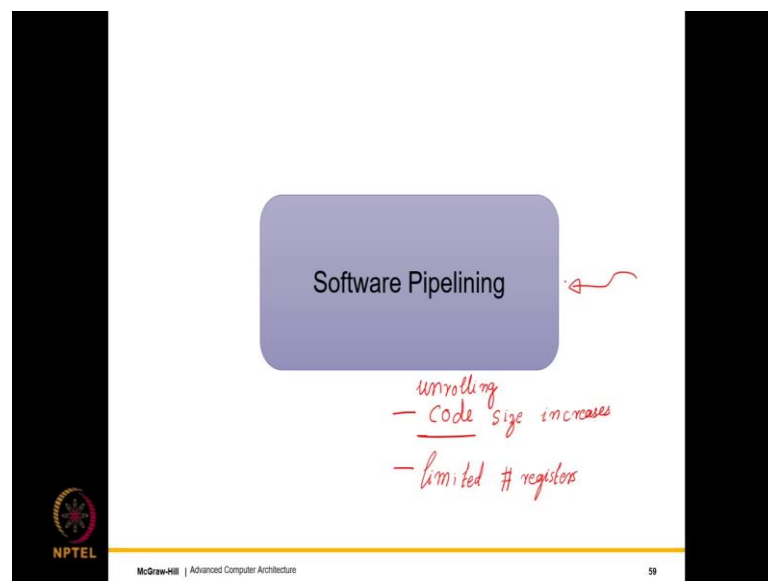
So, in general, if we can reduce the number of branches well, that is great right ok that is absolutely fantastic. If we can reduce the number of branches because less branches are both less instructions as well as less less predictions. So, essentially it is reducing the

sum total of this branch heading and so, that is one advantage of unrolling. Of course, unrolling has other advantages which is also as I said reducing number of instructions.

Also, another advantage of unrolling which we shall see it is more important for in ordered pipelines is that stalls associated with branches are not there and another good thing in out of order pipelines is that we get a larger window of instructions that are branch free.

So, let us say instead unrolling it by a factor of 2 so, this 2 is called unrolling factor. Again, let us say unroll it by a factor of 5, in that case, I will have a large number of instructions without a branch and these instructions can then be scheduled very efficiently using our out of order pipeline scheduling logic. So, that is unrolling, it has multifaceted advantages. So, these are those.

(Refer Slide Time: 49:33)



So, before we move to the next topic which is software pipelining which essentially builds on what we discussed, I want to highlight some of the negative aspects of unrolling and the negative aspects of unrolling will become amply clear as we discuss software pipelining as well.

The first negative aspect is so, let me write it unrolling what is bad with it, a (-) means negative, (+) means positive. So, the first negative aspect is that the code size actually increases. So, there is an increase in the code size which means there is additional

pressure on the instruction cache, this is not something that you would ideally want particularly, the instruction cache sizes low.

The other is that in many cases, we have to add more registers to get good performance and unrolling. So, we have a limited number of registers, but I do not want to talk a lot about this because the entire section on software pipelining is in a sense devoted to that. So, I would not say more, but clearly what we have seen up till now is there is an increase in the code size and the code size will put pressure on the instruction memory, which is conceptual, it is actualized in terms of the I-cache or the instruction cache.

So, software pipelining is trying to do what hardware pipelining does. In hardware, we are trying to do that in software somewhat in an abstract and conceptual sense and so, I will discuss the rest of the lecture will be on software pipelining and we will end over there.

(Refer Slide Time: 51:33)

```
C code
int A[300], B[300];
...
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

```
Assembly code
1 /* Assume the base address of A is in r0
2 /* Assume B is in r1
3 mov r2, 0 /* i = 0 */
4 mov r0, 0 /* offset = 0 */
5 .loop:
6 cmp r2, 300 /* termination check */
7 beq .exit
8 add r3, r1, r0 /* r3 = addr(B) + offset */
9 ld r5, 0[r3] /* r5 = B[i] */
10 add r4, r0, r0 /* r4 = addr(A) + offset */
11 st r5, 0[r4] /* A[i] = r5 (= B[i]) */
12 add r2, r2, 1 /* i = i + 1 */
13 lsl r0, r0, 2 /* offset = i * 4 */
14 b .loop
15 .exit:
16
17
18
19
20
21
```

NPTEL

McGraw-Hill | Advanced Computer Architecture

60

Let us now discuss software pipelining. So, we will use this piece of code as a running example throughout the lecture. So, in this piece of code, what we do is we consider two 300 element arrays A[i] and B[i] and we just copy the contents of array B into array A that is all that we do.

So, the assembly code looks like this, we assume that the base address of A is in register r0, the base address of B is in register r1 and then, the counter i is mapped to r2 and we

also define an offset variable which is mapped to r10 why r10? we will see later, but basically we need to create a lot of temporaries and that is why having r10 where it is a good idea it sort of keeps the code clean.

So, what do we do? Well, first we check for termination. If you have reached termination, we exit. So, this part of the code we have seen so, there is; there is nothing new with this part. So, we first have a load block. So, in the load block, we for array B, we add the offset to r1 that gives us the address of B[i]. The address of B[i] is put into register r3, then the contents of the address which is essentially B[i] is transferred to the registered r5. So, r5 at this stage contains B[i].

Then, let us consider the next two instructions. In this instruction, we do the same, but instead of array B, we do it with array A. So, what we do is that let r0 contain the base address of array A so, we add r10 to it, r10 is the offset so, then A[i], the address of A[i] rather gets transferred to r4 and again, the contents of A[i] are anyway given by the contents of B[i] which is in register r5 so, this is stored over here. So, this basically achieves this $A[i] = r5$ which was set in line 11 to B[i]. So, this is basically doing this for us.

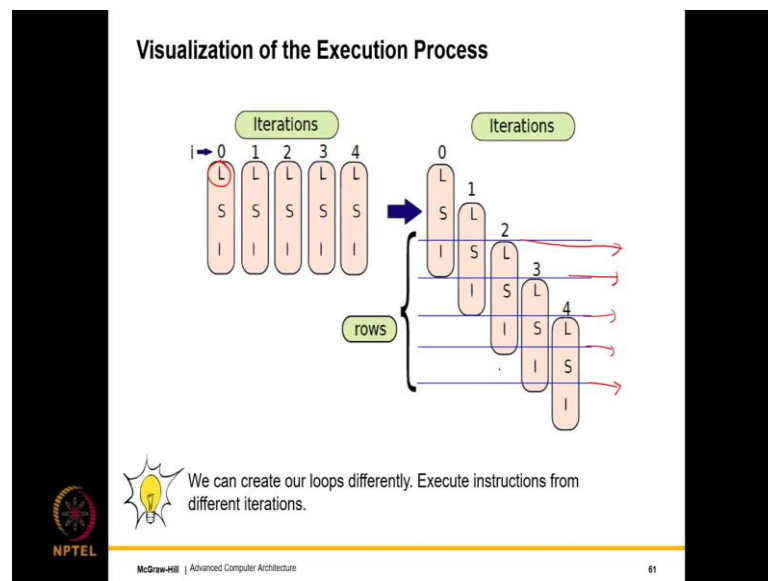
Then, what do we do? Well, then we do the customary increment so, we increment r2 and at the same time, we multiply ($r2 * 4$). So, instead of using a complex multiplication instruction, we instead use the left shift instruction. So, the left shift instruction over here left shifts r2 by 2 positions or in effect, multiplies it by 4 and this is the offset which is stored in r10 and again b loop.

So, what we see over here? If I were to get rid of all the ink on the slide, what we see over here is there are three blocks of statements, those three blocks of statements work something like this L, S and I. So, the way that they actually work is that so, L is a load block and in the load block, what we do is that we load the value of B[i] into register r5, then we have a store block where we store the contents of register r5 into A[i] and then, we have an increment block where each block contains two instructions.

So, what we see is that this can run in an in-order pipeline, there will not be a load use hazard because there is one instruction that separates the load and the use. However, if we have longer memory latencies like longer than one cycle, then of course, there will be a problem.

Furthermore, if there is a need to parallelise some of these blocks so, we will see how? It does not appear to be parallelizable right now, but we can modify the code in such a way that this structure can be parallelized then of course, it should be possible to do so even if the latency to load a value from the data cache is more than one cycle like it is of the order of two, three, four cycles, it should still be possible. So, this is where we would like to look at software pipelining.

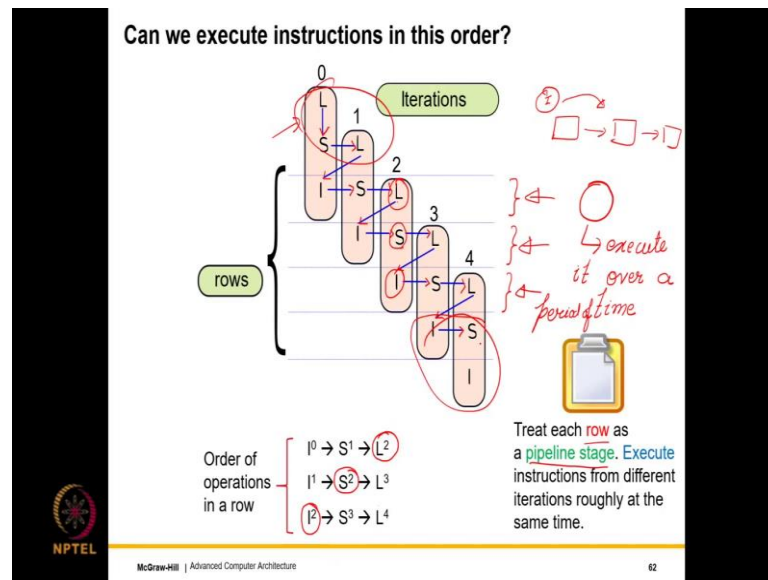
(Refer Slide Time: 56:27)



So, let us look at these three blocks across the space of iterations. So, we have a L block as you can see over here L, S and I. So, we have L, S and I for all the; for all 300 iterations, I am only showing for the first 5. So, I can write these in a different way, I can write L, S, I here, then I can write it the next iteration, I can just write it next, after that, then after that so, nothing stops me from writing it in this order, then I can create these rows as you can see.

So, I have created this row over here, I have created this row over here, this row over here so, these rows have been created. So, well why have I created these rows? Well, this would be clear because I wish to execute instructions in a certain order and that order will become clear, but let us understand that every iteration has three of these blocks and these blocks can alternatively be written in this fashion where we can look at a different order of execution.

(Refer Slide Time: 57:45)



What is the order of execution? Well, it is this order where basically I will execute it like this. So, I am just adding my arrows over here so, just give me a second or so to add these arrows. So, what I can do is that I can execute them in this order, first the L block, then S and L, then I, S and L, I, S and L and so on.

So, if I look at the steady state, if I let us ignore this part and I ignore this part, but if I look at the steady state and let us say the iteration number is the superscript over here. so, order of operations in a single row would basically be for iteration 0, block I, iteration 1 block S, iteration 2 block L, I 0, S 1, L 2, then I 1, S 2, L 3, I 2, S 3, L 4 so on and so forth.

So, it is important to first appreciate the fact that all the blocks that we would have executed in the normal execution with this special order of execution which is also known as a software pipelined way of execution, we are executing exactly the same blocks, let us not worry about dependencies and correctness right now, we will fix that, but why is it called pipelining let us understand.

So, how does a conventional pipeline work? Well, we have stages, and we pass an instruction from stage to stage. At a given stage and instruction, we can say is partially executed or semi executed. So, if I were to have one instruction over here, it will go from this stage to this stage in a certain sense, more of the instruction is getting executed.

For example, in the first stage if you have fetch the value, in the second stage we decode the value, in the third stage we execute so, it is kind of executing it more, more, more and more. So, if we treat the rows as pipeline stages so, this is let us say one stage, another stage, another stage what we add let us say we treat an entire iteration over here as kind of one instruction, then we see that the iteration pretty much proceeds across the pipeline stages.

So, the iteration across the stages is proceeding and this is why we can say that this is one way of software pipelining where the way the compiler will generate the code is that it will execute the entire computation row wise. So, let us not as I said, let us ignore this part for the time being. So, it will generate code let us say for this row where it will execute I 0 S 1 L 2, then it will execute S 1 and I 1.

So, if I were to see just take a look at L 2, it first executes this, then it executes this instruction the store and then I 2. So, it is as if what is happening is that is execution is happening across stages. So, first in let us say this row, we execute the L instruction of iteration 2 the L block, then in this row, we execute the S block and then in this row, we execute the I block.

So, this is in a sense conceptually similar to what we would do in a pipeline where instruction would keep on getting executed partially, but more and more as it passes through the stages, the same is roughly happening for each iteration that iteration in this case is three blocks and as it passes through the rows. so, the rows in a sense, in this case are conceptual for the time being. As it passes to the rows, we can treat each row as a pipeline stage.

And also, what we are doing is that in each row, we are executing instructions from different iterations roughly at the same time so, that also is something that we are doing and here, maybe you can see its slightly differs from a normal pipeline, but well, as I said this is just about a high-level similarity nothing more than that, but the main aim is that we are splitting the execution of a single iteration across stages or over time.

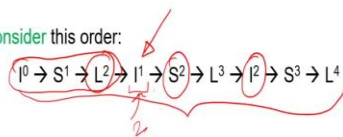
So, what is pipelining? The pipelining is that you take a bunch of computation whether it is a single instruction or an iteration, it does not matter and then, you kind of execute it over time, it is not over time in over time sense, but you just executed over a period of time so, maybe I can. In a certain sense, we are doing that over here.


So, what we are doing over here is something very similar where we are executing in one row the L instruction, then we go to the next stage, the next stage is the next row the S block and then the I block. So, way it kind of executing it in this fashion.

So, if you look at it, we are still executing the same set of instructions exactly the same set, do not worry about correctness for the time being, but the order of execution is different that is important, the order is different that needs to be kept in mind. What are we achieving that is important.

(Refer Slide Time: 63:55)

Advantages of Software Pipelining

- Consider this order:

- The gap between the L, S, and I blocks is one block
- This means that we can absorb delays
- We can accommodate multi-cycle loads without stalls
- The blocks I, S, and L can possibly be executed concurrently
- There is a problem
 - How do we execute three blocks (belonging to different iterations) possibly concurrently?
- Solution: Use different loop iterators



McGraw-Hill | Advanced Computer Architecture

63

So, the advantages of software pipelining, which is essentially this order over here, we must be achieving something. So, let us look at again the second iteration, let us look at this. So, what we are basically achieving over here is that between L 2 and S 2 instead of having no instructions which was the case over here just take a look. So, L and S basically, L ends and S begins that is not happening over here, we have a new block of instructions.

So, given the fact that we have added this extra block, the advantage is that we can absorb more delays in the sense that if we have a slow load instruction that does not take one cycle, but it takes more than one cycle let us say two, three, four cycles, you can still absorb it because we have a block in the middle and of course, four cycles we cannot absorb, but let us say three cycles you can absorb, if we are assuming that this block takes two cycles. So, one cycle it would have taken anyway plus two more.


So, multi-cycle loads can easily be accommodated which is clearly a major advantage. Furthermore, it is possible to execute the I, S and L blocks of the three different iterations concurrently. So, what are we doing? We have essentially created these blocks.

(Refer Slide Time: 65:51)

Advantages of Software Pipelining

- Consider this order:

$$I^0 \rightarrow S^1 \rightarrow L^2 \rightarrow I^1 \rightarrow S^2 \rightarrow L^3 \rightarrow I^2 \rightarrow S^3 \rightarrow L^4$$
- The gap between the L, S, and I blocks is one block
- This means that we can absorb delays
- We can accommodate multi-cycle loads without stalls
- The blocks I, S, and L can possibly be executed concurrently ILP
- There is a problem
 - How do we execute three blocks (belonging to different iterations) possibly concurrently?
- Solution: Use different loop iterators 0, 1, 2



McGraw-Hill | Advanced Computer Architecture

63

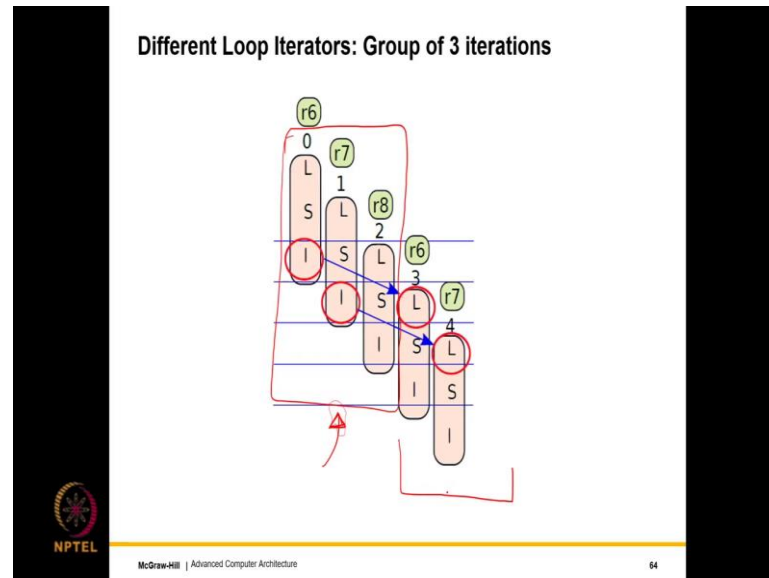
And these blocks so, if I were to just show this in a different way in a different manner what I can do is I can divide these into groups of three, if you would see that in each group we have blocks from different iterations and given that we have blocks from different iterations, what would essentially happen is that there is a possibility that we can execute them in parallel as long as there are no dependencies between them, this will further increase our ILP.

So, this is an ILP enhancing technique, if it is possible to execute them in parallel because we are using different loop iterations. So, that possibility is there, but we will look at it slightly later.

But if we were to execute them not in parallel, but what is called concurrently, it is a concurrently in computer science is not parallel, it does not indicate simultaneously, but it just means that without any specific ordering specified between them well, then we should move in that direction so, one can be, we can use different loop iterators for the three separate iterations which is 0, 1 and 2. So, let me explain this with an example it will become much clearer.

So, where are we moving towards? Well, we are moving towards an implementation of software pipelining that is correct, does not have dependencies and where we can possibly execute these three instructions, these three blocks in parallel. The solution that I will show will not take us 100% there, but the solution after that will take us 100% there.

(Refer Slide Time: 67:33)



So, let us try to understand what is happening. So, what we are suggesting is that to break the chain of dependencies. so, I will tell you in a second why it is important.

(Refer Slide Time: 67:55)

C code

```
int A[300], B[300];
...
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

Assembly code

```
1 /* Assume the base address of A is in r0
2    and B is in r1 */
3 mov r2, 0 /* i = 0 */
4 mov r10, 0 /* offset = 0 */
5
6 .loop:
7     cmp r2, 300 /* termination check */
8     beq .exit
9
10    add r3, r1, r10 /* r3 = addr(B) + offset */
11    ld r5, 0[r3] /* r5 = B[i] */
12
13    add r4, r0, r10 /* r4 = addr(A) + offset */
14    st r5, 0[r4] /* A[i] = r5 (= B[i]) */
15
16    add r2, r2, 1 /* i = i + 1 */
17    lsl r10, r2, 2 /* offset = i * 4 */
18
19    b .loop
20
21 .exit:
```

L
S
I

So, if you can see, let us look at this I, S, L order. so, let me erase the ink on this slide, I will just show you. See if you look at it, what I is doing is that this is for one iteration, this is for one more, this is for one more. So, just take a look at it, it is in the reverse order I, S, L.

So, let us say I were to execute this first, then this will essentially change the value of the offset correct and then, that will cause problems over here and so, then it will actually not be correct. So, what I should do is that for these three, I should use different loop iterators and different offsets. So, that will ensure it is correct which is exactly what I am showing here. For iteration 0, 1 and 2, I use r6, r7 and r8, then for 3, 4 and 5, I again use r6, r7 and r8 and I claim it is correct, I will show in a second why, but I claim it is correct.

And if you can see the dependencies well, there will be a dependency between the r6 S so, I will have a dependency with L here, again I will have a dependency with L here, but I will see how to take care of it.

So, what in a sense I am proposing is that we consider these group of nine blocks as one iteration and then, we execute all of that together again, we consider another group of nine blocks and execute it together. So, we will see that there will be no problem in terms of correctness. So, let me show you the code it will become amply clear.

(Refer Slide Time: 69:33)

Code with Different Loop Iterators

```

/* First Row */
add r6, r6, 3 /* I0 */
lsl r10, r6, 2

[
  add r4, r0, r11 /* S1 */
  st r5, 0[r4]
  add r3, r1, r12 /* L2 */
  ld r5, 0[r3]
]

/* Second Row */
add r7, r7, 3 /* I1 */
lsl r11, r7, 2

[
  add r4, r0, r12 /* S2 */
  st r5, 0[r4]
  add r3, r1, r10 /* L3 */
  ld r5, 0[r3]
]


/* Third Row */
add r8, r8, 3 /* I2 */
lsl r12, r8, 2

[
  add r4, r0, r10 /* S3 */
  st r5, 0[r4]
  add r3, r1, r11 /* L4 */
  ld r5, 0[r3]
]

```

Handwritten annotations:

- Red circles around `I0`, `I1`, `I2`, `S1`, `S2`, `S3`, `L1`, `L2`, `L3`, `L4`.
- Red arrows indicating dependencies between instructions.
- Red text: `0,3,6` and `7,10` under `I0`; `1,4,7` and `7,11` under `I1`; `2,5,8` and `7,12` under `I2`.
- Blue box: "Unroll the loop 3 times"



McGraw-Hill | Advanced Computer Architecture

65

So, here is the code. So, as I said we will have three blocks so, the three blocks will be these blocks I0, S1, L2, I1, S2, L3 and I2, S3, L4. So, these are the blocks so, what we do is we unroll the loop. so, recall we had discussed loop unrolling a while ago. So, we unroll the loop 3 times so, we will have nine of these blocks and then, we order them in this fashion I0, S1, L2, I1, S2, L3 and I2, S3, L4.

So, what we do is that for the 0th iteration, for this iteration, what we do is that we considered r6 to be the loop iterator for the 0th iteration and then, we consider r6 so, it's a pair of r6 and r10. So, what we will do is that we will assign r6 and r10 to the iterations which has (0, 3, 6, 9) and so on, again we will assign r7 and r11 to the iterations of the form 1, 4, 7 and so on and we will assign r8 and r12. So, r 8 is the iterator, r12 is the offset, 4, 2, 5, 8 and so on.

So, if you see this is exactly what we are doing so, for I0, we execute this block which is in r6, r10 player and again, we have an L 3 and S 3, so, just take a look at L 3 and S 3. So, what we do is that we compute the address by adding the offset which is r10 which we presumed to have happened correctly. So, why do we presume it has happened correctly? Well, the reason that we do that do it is because of this block over here.

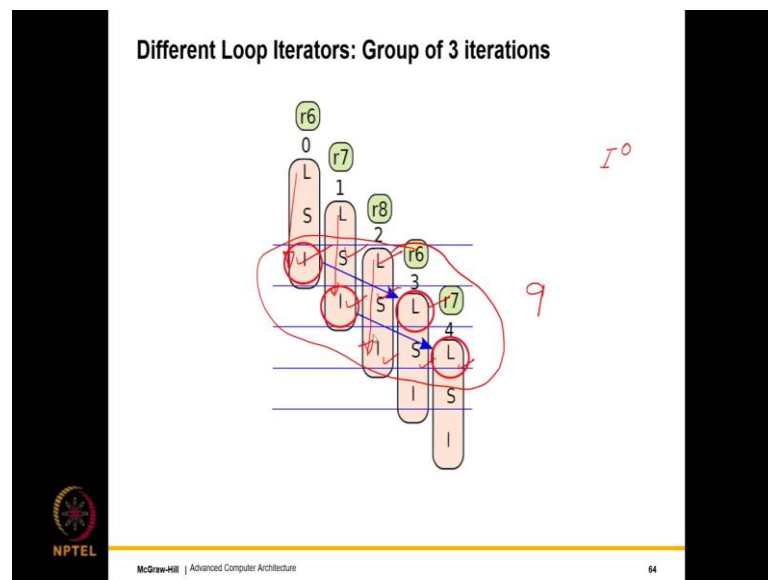
So, instead of adding 1 to the iterator given the fact that it has been unrolled 3 times, instead of adding 1, we add 3 so, this is the crux. The fact is that since we are considering 0, 3, 6, 9 and so on instead of 1, we add 3 so, that is the crux of this argument, we add 3 to r6 and then, we compute the offset accordingly which is just r6 times 4. So, r10 is computed correctly and no other iteration uses r10, you can verify.

The next iteration that actually uses r10 is again the third iteration L3. So, this uses r10 correctly, then you compute r3 and from r3 you compute r5, r5 is not used in the I2 block in the middle and r5 is again used, again we use r10 over here and r5 is again used to store the value into r4.

So, we can see a very similar pattern here for others iteration. So, let me show that I am still keeping this part there for you. So, it is that can be a running reference. So, what I will show now is that I will show it show for the one series. So, just take a look at S1 alright, I1 and L4. So, in S1 again, we assume that r11 is computed correctly which is the offset and we store the value in r5.

We take the value in r5 and we store it in the address. Henceforth, r5 is not required. See L2 can overwrite r5, we do not care, then we have I1 again, the I iterator is incremented by 3 not 1 and we compute and the corresponding offset, which is r11 and then, r11 is used in L4 to again compute the address and both the value in r5.

(Refer Slide Time: 74:17)



So, what we see over here is that we have actually taken three iterations, unroll the loop by a factor of 3 and the three iterations which would look somewhat like this, I will show you the iterations that we have taken. So, we have essentially taken I0, S1, L2. So, just take an I0, S1, L2. So, it is this we have taken, this, this, this, this, this, this and then I2, S3 and L4 so, we have essentially taken these iterations. So, these are 9 blocks.

So, these 9 blocks of course, if I would have executed them in a traditional manner, then they we would have proceeded in this fashion, first these, then these, then these starting from I0, but we in this case we actually did not do that, what we did was more interesting where we just considered these three rows and the instructions in these three rows, these 9 blocks were considered together and then by unrolling the loop 3 times and then of course, we rearrange them.

To eliminate dependencies, we had to use more registers. So, for different iterations, we used different loop iterators and corresponding offsets r6, r10 one pair, r7, r11 one pair, r8, r12 one pair and it showed that look, we are not having any dependencies otherwise, there would have been a problem, had we not use them.

Had we not use them consider S1 and L2 both would have required an offset and this offset would have come from I0 and this clearly would have been wrong. So, that is the reason we have used separate ones and that has to a large extent broken the dependency and this execution order is correct. so, there is no problem with this.

And what is the key advantage? Well, the key advantage is we have been able to insert a block the I block between an L and S block. So, this allows us to absorb multi-cycle loads and even from the perspective of an out of order pipeline, it gives us more instructions to work with an enhanced ILP.

(Refer Slide Time: 76:35)

Code with Different Loop Iterators

```

/* First Row */
add r6, r6, 3      /* I0 */
lsl r10, r6, 2

add r4, r0, r11    /* S1 */
st r3, 0[r4]

add r3, r1, r12    /* L2 */
ld r5, 0[r3]

/* Second Row */
add r7, r7, 3      /* I1 */
lsl r11, r7, 2

add r4, r0, r12    /* S2 */
st r5, 0[r4]

add r3, r1, r10    /* L3 */
ld r5, 0[r3]

/* Third Row */
add r8, r8, 3      /* I2 */
lsl r12, r8, 2

add r4, r0, r10    /* S3 */
st r5, 0[r4]

add r3, r1, r11    /* L4 */
ld r5, 0[r3]

```

We cannot execute S1 and L2 in parallel because of the dependence on r5

Unroll the loop 3 times

So, now, let me point out the problem. So, the problem is that we also wanted to execute these three blocks concurrently, this is not possible. So, this is what I was hinting at in the previous slide that we cannot execute S1 and L2 in parallel because they actually depend on r5.

So, what we want is that we want to read r5 first and we want to write to it later, but well, if they are executed in parallel, we have a hazard, it is write after read hazard and this will cause an issue. It will not cause an issue in an in order in a multi-issue in order pipeline, this will cause an issue. In an out of order pipeline well, no, this will not cause an issue because renaming will as you can see will take care of it that this will be in the sense given to a separate register and that register will again be read over here and its life will end.

So, out of order pipeline, this issue is not there, but in order pipelines, this issue is there and software pipelines were made predominantly in the days of the in order pipeline so, it is much better to get rid of this little dependency over here so, then you will see that we genuinely do not have dependencies because it is all r6, r10 over here that is what we are writing to this uses other registers and this uses other registers.

So, we are simply creating a different set of registers for each iteration if we do that so, the dependency between S1 and L2 will go.

(Refer Slide Time: 78:15)

The slide displays assembly code for three rows of instructions, with handwritten annotations in red and a callout box in a purple bubble.

```

/* First Row */
add r6, r6, 3 /* I0 */
lsl r10, r6, 2

add r4, r0, r11 /* S1 */
st r21, 0[r4]

add r3, r1, r12 /* L2 */
ld r22, 0[r3]

/* Second Row */
add r7, r7, 3 /* I1 */
lsl r11, r7, 2

add r4, r0, r12 /* S2 */
st r22, 0[r4]

add r3, r1, r10 /* L3 */
ld r20, 0[r3]

/* Third Row */
add r8, r8, 3 /* I2 */
lsl r12, r8, 2

add r4, r0, r10 /* S3 */
st r20, 0[r4]

add r3, r1, r11 /* L4 */
ld r21, 0[r3]
  
```

Handwritten annotations include red circles around register names (r21, r22, r20) and red arrows indicating data flow between instructions across rows. A purple callout box on the right contains the text: "If we had 32 registers, we could do this very easily and elegantly".

NPTEL
McGraw-Hill | Advanced Computer Architecture
66

To do that, what we do is let us hypothetically assume that we have 32 registers, then we can do this very easily and very elegantly. So, what we can do is that we can in a sense create a new register r21 which stores output of S1. so, which S1 uses and then, r22 which is the load value of L2 and again, r 22 is used over here and again r20 is used by L3 which is again used by S3.

So, we have effectively partitioned the set of registers between the iterations. So, there will be no conflict between them at least in these three so, I and in these three and these three.

So, I would advise the reader to take a look at these sequences of instructions and convince himself or herself that genuinely there is no problem. For example, r7 and r11 are the ones registers that are used here, but they are not used here. Similarly, r4, r0, r12

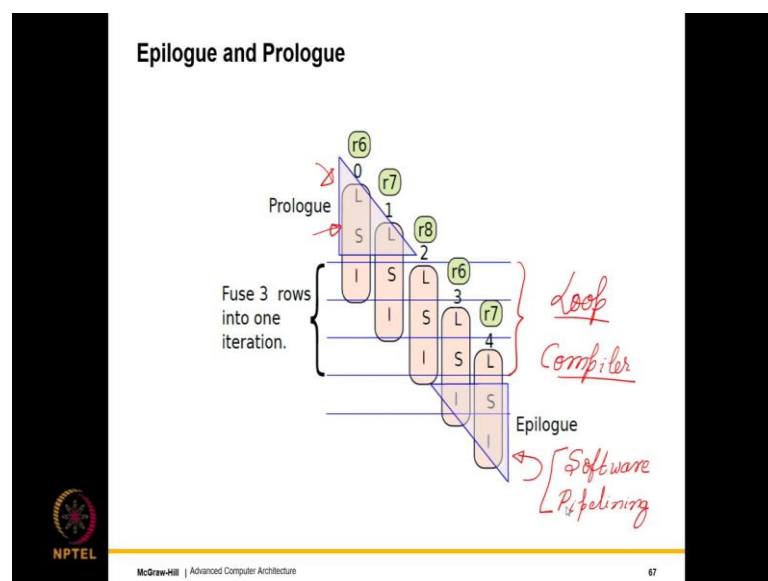
and r22 are used, but they are not used here and so on and so forth, the set of registers that are used here are disjoint from the set of registers that are used here.

Hence, there is no problem and we can execute these three blocks in parallel and this is an advantage in order setups and when we will discuss the Itanium processor that predominantly relies on the software to identify parallelism and to do the scheduling, you will find that such approaches are extremely helpful where we can pretty much run these three blocks simultaneously in parallel. So, these approaches are extremely beneficial.

So, one point that we need to appreciate over here is that we clearly need more registers to do this because well, I could have used r13, 14 and 15, but they are reserved for other purposes like the stack pointer and so on so, I did not want to touch them. Hence, I am proposing a new solution that uses more registers.

Further expounding on this fact that we need more registers for doing software pipelining, this will become even more clearer when you will see that Itanium processor actually uses a lot of registers and it also has special support including what you have seen that different set of registers for different iterations so, a lot of this support is actually built into the hardware.

(Refer Slide Time: 81:21)



So, we have discussed kind of the steady state where we have looked at these three rows, but what about this part which is known as the prologue and this part which is known as the epilogue. So, this part also deserves give attention and this part is not being pipelined, but of course, this diagram has not been drawn to scale.

So, if you have 300 iterations, most of the iterations can be software pipelined, but you will always have a few of these iterations, instructions over here part of the epilogue and prologue that cannot be broken down or rearrange the way that we have actually done. In that case, well we can add some additional code at the beginning and at the end. So, this is the compiler can do automatically at the beginning and at the end to basically ensure that the entire loop executes correctly.

So, the summary over here is that a loop is actually a pretty complex thing from the point of view of a compiler because it is a full encyclopaedia of optimizations that can be applied. Particularly, software pipelining is very important because it exposes ILP. More so in the days of this free out of order pipelines and also in special pipelines that we shall see in Itanium and so on, software pipelining used to be more important.

Nowadays, with renaming and so on its shine is not that bright, but nevertheless, it is a very very important mechanism for simpler processors that are predominantly in order and need to deal with large multi-cycle delays. So, there is a very very rich literature in compilers of how to automatically transform loops into their software pipeline versions.

(Refer Slide Time: 83:23)

Solution without Unrolling

Original C code

```
int A[300], B[300];
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

Simpler C code

```
int A[300], B[300];
int i = 0;
loop: if (i<300){
    t = B[i]; /* L */
    A[i] = t; /* S */
    i++; /* I */
    goto .loop;
}
```

Diagram illustrating the transformation of the original C code into a simpler C code structure, showing the loop body and the goto statement. The diagram also includes a handwritten note: "i = -1; t = B[0];" and a handwritten note: "A[299] = t;".

Handwritten notes in the diagram:

- $i = -1; t = B[0];$
- $A[299] = t;$
- Handwritten numbers: 297, 298, 299

NPTEL

McGraw-Hill | Advanced Computer Architecture

68

So, now, the we showed one way, one method of software pipelining, but the it turns out that there are other methods as well. So, the alternative solution that does not use renaming we will look something like this. So, let us first go from here to here. So, I have slightly expanded this C code such that one line roughly corresponds to one line of assembly or one or two lines, not more than that.

So, what we can see is that we can see that we are creating a loop over here. So, the loop over here is similar to what we would do in assembly that we will check for termination and if it has not terminated, well we will again have an unconditional branch statement and furthermore, what we will have over here? We will have the L, S and I blocks. So, here, each block actually corresponds to two lines of assembly, the same way that we have been showing is just another way of writing.

So, what I will do over here is that I will slightly reorder them such that we do not have to unroll, but still the code will be correct. So, what I do is I initialise $i = (-1)$ important, I initialize the temporary variable $t = B[0]$, the first element and then, I go till 297. so, i essentially goes from minus 1 till 297. So, the first thing that we do is we increment i so, then it goes from 0 to 298.

Then, what we do is we set $A[0] = t$. For the first iteration it will be nothing, but $B[0]$. In the second iteration, we will see, then what we do is we set the temporary $t = B[i] + 1$ and so, basically the value that the next iteration will set we store that in a temporary over here and then, we go to the next iteration. In the next iteration, they again do the same.

In this case, it is easy to convince oneself that $A[i] = t$ will actually be set correctly. So, this will actually be set correctly because what is this t ? This t is essentially coming from the previous iteration where we did it for $B(i + 1)$, but now, i has changed so, the $(i + 1)$ of the previous iteration is these iterations i and so, basically it is the correct t . so that previous iterations $B(i + 1)$ is being funnelled into this iteration A , we think of it $A(i + 1)$ and it is happening via the t and the value of t is flowing across iterations.

So, anytime, there is a dependence between iterations, we call it a loop carry dependence which we are seeing over here. So, we keep doing this, we will not be able to do it for the last iteration. So, for the last iteration, it will be $A[299]$.

So, A 299 which is the last so, basically B 299 so, after $i = 298$, B 299 will be putting t and then, we will come here, we will exit the loop and this value of t will get transferred over here which means that the entire iteration has been done correctly. So, the entire process of transferring the values from B to A has been done correctly.

So, I am erasing the ink, it is important for the readers to stare a little bit at this code and convince themselves that this is indeed correct ok keep staring. So, hopefully you have appreciated the broad idea that we do we are doing software pipelining still I will beat in a different sense without unrolling.

So, pretty much we have taken the original iteration and sliced it across iterations of this transformed code, again it is a form conceptual form of pipelining, and it does allow us to have more instructions between a load and a store. So, in this sense, a load is happening over here, this is the load and then, what we do is we again have a loop, we come here, we have some checks, we increment i and then, we do the store, it is a, multi-cycle loads latency can be absorbed.

(Refer Slide Time: 88:35)

Unrolling and Mixing

C code


```
int A[300], B[300];
for(i=0; i<300; i++){
    t1 = B[i];
    t2 = t1 * 5;
    A[i] = t2;
}
```

SW pipelined version

```
int A[300], B[300];
for(i=0; i<300; i+= 3){
    t1 = B[i];
    t2 = B[i+1];
    t3 = B[i+2];

    t11 = t1 * 5;
    t12 = t2 * 5;
    t13 = t3 * 5;

    A[i] = t11;
    A[i+1] = t12;
    A[i+2] = t13;
}
```



McGraw-Hill | Advanced Computer Architecture

69

Let us now look at another way of unrolling. So, let us take a look at a C code. So, here basically, we load the value of B $[i]$, we multiply it with 5 and then, the data is stored in A $[i]$. So, in this case, what we do is we unroll it by a factor of 3, if you mix the different iterations in a different way so, what we do is we just take a look at the outer for loop

and outer for loop, we have $i += 3$ over here which means I consider 3 iterations at one row.

So, what we use is instead of one temporary, we use three temporaries' t_1 , t_2 and t_3 . So, we load in the value of $B[i]$, $(B[i+1])$, $(B[i+2])$, then what we do is that we multiply again we generate three more temporaries and again we write it. So, the advantage is that let's say the difference, the time difference between reading the value of $B[i]$ and actually using it, there are two instructions in between.

So, even if let us say the load has a large latency that is still fine, the reason is that the pipeline is not idle, the pipeline is busy in issuing other load instructions or doing other work and here, what we do is that we simply compute the result and we do a store and the stores in this case are not on the critical path, the loads are, but the nice thing is that we have essentially increase the distance between the load which is over here and its used over here such that we can sustain or we can tolerate a much longer load use delay.

So, this is basically what we have done over here. This is of course, again a different version or a different kind of software pipelining in a different flavour, but again this is also a very key and important example in this space.

Well, so, this is always good news to say that we have completed 4 out of 5 sections of this chapter. What is left is to kind of take all our compiler techniques, put them together, put all of them together and create what is called an epic processor, the example of which being Intel Itanium.