Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

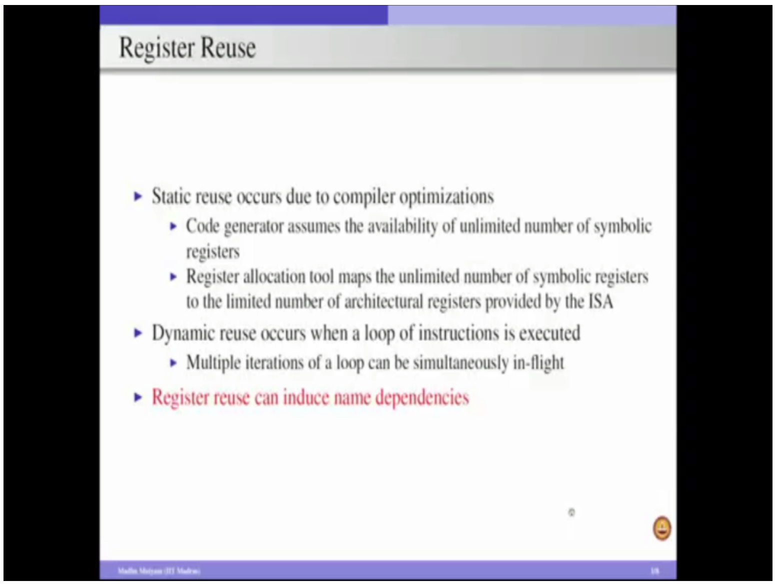
Module – 07
Lecture - 24
Register Renaming

So, as part of this module we are going to discuss register renaming. So, we know that in a load store architecture a typical instruction consists of the opcode, destination operand and the source operands. And for this destination and the source operands we use registers if the instruction is a typical ALU instruction. So, in this instruction for example, if the functional unit is not available, then we are going to get a structural hazard and if the source operands are not available then we are going to get the data hazards, mainly because of true dependence.

If the destination register is not available then we are going to get the false dependence or the name dependencies. And in order to eliminate these name dependencies we are going to use concept called as register renaming. These false dependencies or the name dependencies happen mainly because of reusing of registers. For example, if we consider 2 instructions ADD R1, R2, R3 and there is another instruction MUL R1, R5, R6.

Here these 2 instructions are actually writing to the same register R1. So, in other words like the same register R1 is reused in both the instructions and because of that we are going to have this output dependence and it is part of the name dependencies. So, register use can happen when we compile the code. We know that in the compilation process there are 2 stages one is the code generation phase.

(Refer Slide Time: 02:04)



Register Reuse

- Static reuse occurs due to compiler optimizations
  - Code generator assumes the availability of unlimited number of symbolic registers
  - Register allocation tool maps the unlimited number of symbolic registers to the limited number of architectural registers provided by the ISA
- Dynamic reuse occurs when a loop of instructions is executed
  - Multiple iterations of a loop can be simultaneously in-flight
- Register reuse can induce name dependencies

The other one is register allocation phase. So, here during the code generation we actually translate our instructions into the machine instructions. And during this code generation phase, compiler assumes that there are unlimited number of registers available to use. And with that assumption the code generation phase converts the instructions into machine instructions by using several symbolic registers. But finally, because the architecture is going to have limited number of registers, so, we have to convert these symbolic registers into architectural registers and that is what we are going to do in the register allocation phase.

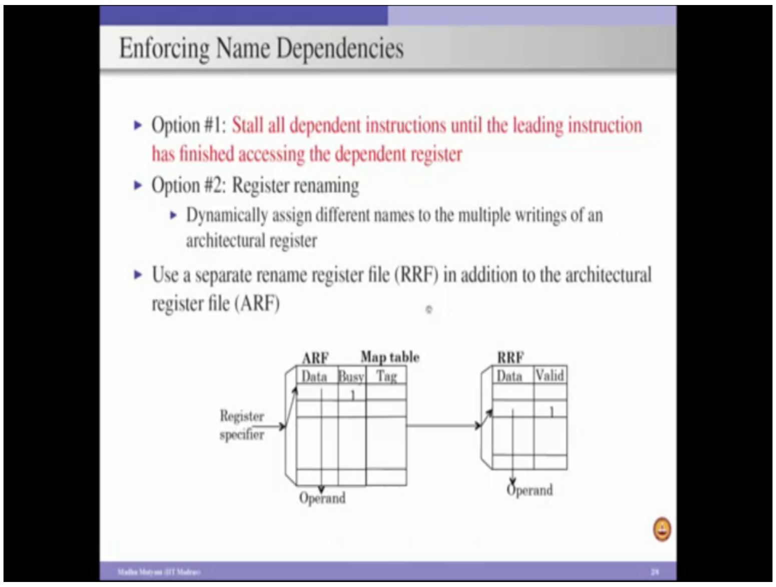
In the register allocation phase we optimize the number of registers that we use for that code, the compiled code. So, in other words the register allocation tool maps the unlimited number of symbolic registers to the limited number of architectural registers provided by the ISA, because each ISA has a specific number of architectural registers to use.

So finally, when we compile the code for a specific architecture, so, our code will use only the available registers specified by that instruction set architecture. So, if the reuse of registers happen due to the compilation, then it is called as the static reuse and because our register allocation phase maps unlimited number of symbolic registers used in the code generation phase to the limited number of architectural registers. So, as a result we may have to reuse several of the architectural registers in our code. And because of that we are going to have the register reuse and because this is happening at the compile time we call it as static reuse, but the reuse can also happen at run time that is a dynamic reuse. So, we can give an example for this, dynamic reuse typically occurs when a loop of instructions is executed. When we consider a deeper pipeline and when we have loop of instructions to be executed, one iteration of the loop body may be at the far end of the deeper pipeline and the other iteration of the loop body may be at the front end of the pipeline.

Once we have this and as a result we may get the name dependencies because the registers that we use may be reused in different iterations of the instructions. And these different iterations are already there in the pipeline and because of that we will get these false dependencies. And we need to eliminate that in order to execute the trailing instructions without having any stalls in the pipeline.

So, in summary so we will have the reuse of instructions happen at compile time or at the runtime also. And so, the compile time reuse happens mainly because of the register allocation phase of the compilation process. And the dynamic reuse happens because of multiple iterations of the loop will be there in different phases, different pipeline stages in our Superscalar processor. So, because of this register reuse, we get name dependencies those are output dependencies or anti dependencies. And we need to take care of these name dependencies in order to improve the overall performance of our super scalar processor.

(Refer Slide Time: 05:48)



Enforcing Name Dependencies

- Option #1: Stall all dependent instructions until the leading instruction has finished accessing the dependent register
- Option #2: Register renaming
  - Dynamically assign different names to the multiple writings of an architectural register
- Use a separate rename register file (RRF) in addition to the architectural register file (ARF)

Madhu Mutyam (IIT Madras) 24

So, one simple option is we stall all the dependent instructions until the leading instruction has finished accessing the depending dependent register. So, for example, if we have an ADD instruction and there is a multiply instruction. And these 2 instructions are actually having the name dependencies. Though these 2 are not dependent using the true dependence, but these two instructions are dependent because of the name dependencies. And now when we stall the trailing instruction, that is multiply instruction until the ADD instruction completes then we unnecessarily create bubble in the pipeline and as a result the performance will be degraded.

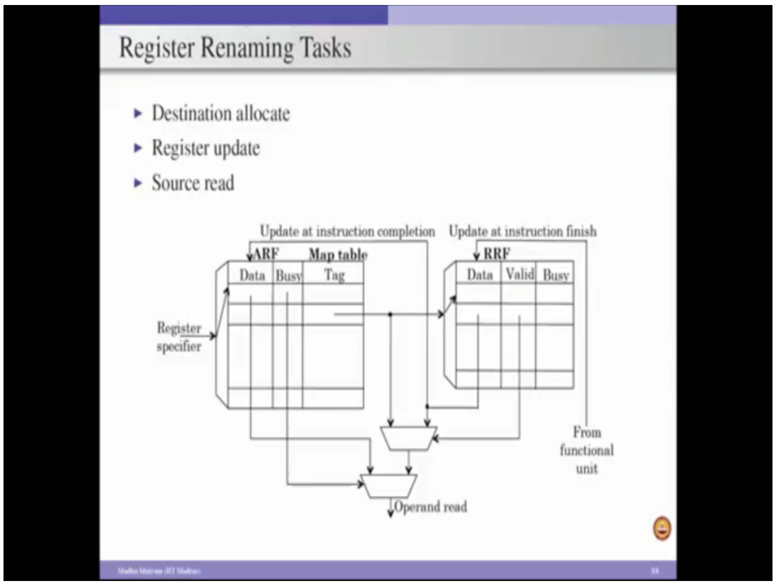
So, we need to go for some other technique. So, the other alternative is we rename the registers because we know that the name dependencies actually happen because of the register reuse. And there is no actual data flow happens between the leading instruction and the trailing instruction in the pair of instruction which are having the name dependencies. So, as a result once we rename the registers, then there would not be any problem from the program correctness point of view.

So, in order to rename the registers what we can do is, at run time we can assign different names to the multiple writings of an architectural register. If we consider an example ADD R1, R2, R3 and MUL R1 R5 R6, here ADD and multiply are using the same destination register that is R1, now what we can do is, we rename R1 of multiply instruction. So, as a result these 2 instructions now appear to be independent and there is no dependence between this. And we can go ahead with the multiply operation without waiting for the ADD operation to be completed. So, we just by renaming the destination registers in the trailing instructions we can eliminate these name dependencies. And we are going to discuss the complete process of how to rename the registers in detail in this module.

So, in order to achieve this register renaming we are going to use “Rename Register File” in addition to the “Architectural Register File”. For each ISA we are going to have a fixed number of architectural registers and all the architectural registers belong to an ARF that is “Architectural Register File”. And we consider other physical registers those are rename registers and we consider the corresponding group as register, we consider the corresponding group as “Rename Register File”.

Once we consider 2 register files, one is for the rename registers the other one is for the architectural registers. Now, we need to come up with a mapping between architectural registers to the rename registers. And that we are going to discuss in the coming foils, but this diagram is going to give a pictorial representation of how the mapping is going to happen between the architectural register file and the rename register file.

(Refer Slide Time: 09:04)



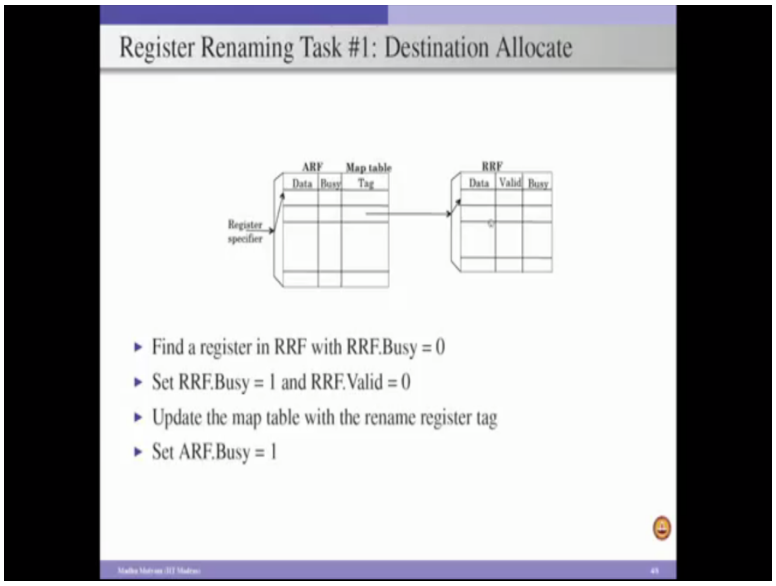
Register Renaming Tasks

- Destination allocate
- Register update
- Source read

So, the register renaming consists of 3 main tasks. One is destination allocation phase, the second one is register update phase, the third one is source operand read phase. Whenever destination operand of an instruction is to be renamed, we come here and we go to the rename register file and find one free register and assign that as the destination operand for the instruction. So, in other words the destination operand of the instruction is renamed to a register in the rename register file. And after that whenever we finish the execution of the instruction then we will come here and we update the rename register first.

And when the instruction is completing then we are going to update the architectural register and whenever we want to read an operand for an instruction we come here and we see whether operand is available in the architectural register file. If it is not available then we will see whether this is mapped to any of the rename register file registers. And if it is so then we go here and read the data and supply the data to the instruction that is requiring this source operand. So, this is the whole process we are going to do and we are now going to discuss each of these steps in detail.

(Refer Slide Time: 10:35)



Register Renaming Task #1: Destination Allocate

ARF
Map table
Data
Busy
Tag
RRF
Data
Valid
Busy
Register specifier

- Find a register in RRF with RRF.Busy = 0
- Set RRF.Busy = 1 and RRF.Valid = 0
- Update the map table with the rename register tag
- Set ARF.Busy = 1

The first phase is destination allocate. So, let us assume that we have an instruction ADD R1 R2 R3. Now we have to rename this register R1 because R1 is the output register or the destination operand for this ADD instruction. So, we have to rename the destination register. Remember in the register renaming concept we are going to rename all the destination registers in all our instructions. So, when we have an ADD instruction ADD R1 R2 R3 we are going to rename R1 with some register in the register renaming file.

Similarly, if some other instruction is there then also the output register we take and then rename. The reason is, because once we rename all our destination registers in our instructions. So, there would not be any name dependencies because the name dependencies happen mainly because of reusing of our output register or the destination register. So, as a result, if we rename our destination registers so, that no instruction, no 2 instructions, in our program uses the same rename register, then we are fine.

And that is the idea we are going to use and we effectively rename all the destination registers in our instructions. Whenever we want to rename the destination register of an instruction we go to the RRF, that is rename register file, take one free register from this and we rename the destination register in our instruction with this Id. And after that we go to the ARF, where we index into the ARF using our destination register specified in our instruction because in our instruction we have the architectural register.

Let us say if we consider ADD R1 R2 R3, R1 is the architectural register which is the destination register for this ADD instruction. We index into ARF using this R1 and we already identified a rename register for this R1 from this RRF. So, we consider a map between this indexed field to the register whatever we selected in the RRF. In other words let us assume that we consider R4 is the rename register in the RRF, which can be used as a rename register for our destination register R1 in our instruction ADD R1 R2 R3.

So, we go to ARF and index into ARF using R1. So, let us say this is the entry and now we consider a map from this to R4 register in RRF. So, once we have this map the next step if any trailing instructions which are going to use R1 output, which is going to be produced by ADD instruction. Now, all these trailing instructions are going to read the data from R4 register that is available in RRF. So, that is the reason why when we rename an architectural destination register in our instruction, we have to go to ARF and consider a map from ARF to RRF in the appropriate location.

So, that all the trailing instructions will be directed to RRF to get the value from the rename register for that particular architectural register. So, the step by step method for doing this destination allocation phase is, find a register in rename register file with RRF busy bit is reset. Remember, whenever we allocate any rename register in RRF, we have to set the busy bit 1. So, that this entry is used as a rename register for some architectural register. So, as a result whenever we want to select an entry in the RRF, we always have to scan through all the busy bits and whichever entry has a busy bit reset we can consider that register as a rename register for our destination register in our instruction.

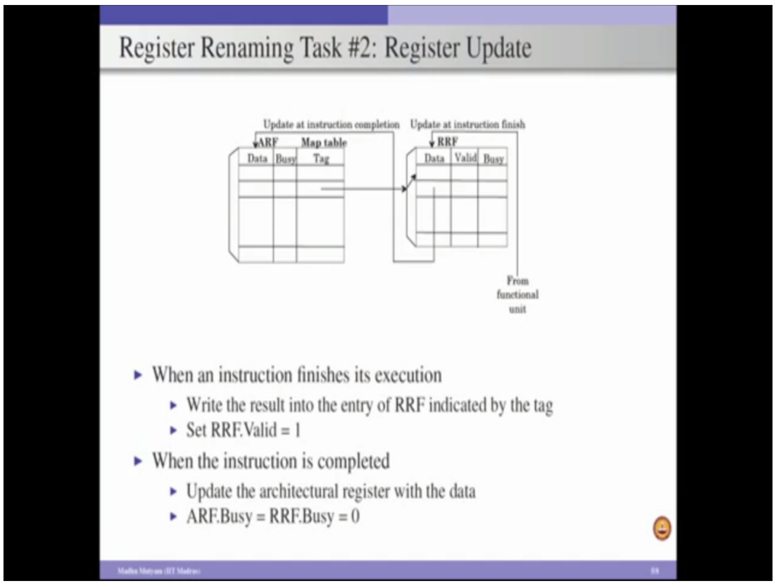
So, first step is to select an entry whose busy bit is 0 in our RRF. So, once we select that we set the busy bit 1 indicating that now this register is going to be used as a rename register for some destination register in our instruction. And also we make RRF valid equal to 0 because we just renamed the register, but we have not computed the value and the computed value is not written to the rename register. So, as a result if any trailing instruction wants to access this rename register to get the operand. So, this valid bit indicates that the data is not available and it should not read the value from this rename register.

And this valid bit will be set only when the instruction is the corresponding instruction is executed or finishes its execution. Only after finishing the execution of an instruction the corresponding value will be sent to this RRF with the corresponding register tag. And we search in the RRF and if there is a match then in that particular entry we write the actual computed value in this data field. And then we set the valid bit to 1 until then the valid bit will be 0. In other words whenever we select an entry in the RRF as a rename register for an architectural register, we have to set the valid bit to be 0. And at the same time we have to keep the busy bit 1. So, that this entry is busy, but this entry is not having any valid data in it. And then, once we do this setting and the resetting of this valid and the busy bits. Now, we will move onto ARF and we map the corresponding entry in the tag, tag field of this map table. So, that this architectural register now is renamed to a rename register in RRF so that any dependent instructions which are going to use this architectural register will be now redirected to this RRF to get the value. So, that is what the third step update the map table with the rename register tag.

And once we do that the next one we are going to do is we set the ARF busy bit. We can read the value from the ARF only when our busy bit is 0. So, if the busy bit in the ARF is set then we should not read the value directly from the ARF. We have to go to the RRF. This is effective in the sense that once the busy bit is 1, it says that you have to go to RRF to get the value and again in the RRF if the valid bit is 0 that indicates that the data is not available right now and the instructions currently executed. And if the valid bit in the RRF is 1 that indicates that value is available and we can supply the value to the dependent instructions.

So, in summary in the destination allocation phase we first have to identify a free entry in the rename register file. And we set the busy bit 1 and also reset the valid bit and we map the ARF with the register tag of this rename register that is selected in RRF. And then we set the busy bit in the ARF field. So, with that the destination allocation process is completed.

(Refer Slide Time: 18:45)



Register Renaming Task #2: Register Update

Update at instruction completion
Update at instruction finish
ARF
Map table
Data
Busy
Tag
RRF
Data
Valid
Busy
From functional unit

- When an instruction finishes its execution
  - Write the result into the entry of RRF indicated by the tag
  - Set RRF.Valid = 1
- When the instruction is completed
  - Update the architectural register with the data
  - ARF.Busy = RRF.Busy = 0

And after that the instruction is issued to the functional unit and functional unit finishes its execution. Now, we have to update the contents, but when an instruction is finish it execution which register we have to update. Whether we have to update the rename register or whether we have to update the architectural register. We already discussed in the previous module that in the Superscalar processor pipeline. So, we have Execute stage and after that there is a Complete stage. And execute stage typically executes instructions in out of order fashion, but the complete stage completes the instructions in the Inorder.

And in order to complete the instruction Inorder we are going to use our ROB, reorder buffer. And from the ROB we always take the first instructions or the instruction which pointed by the head pointer in the ROB, will be processed, will be proceeded to the complete stage and will be completed. And now consider a scenario where the head of the ROB is blocked because of some pending computation, but whereas, the trailing instructions in the ROB finish their execution.

So, as a result even though the trailing instruction finish their computation they cannot proceed with the completion stage because there are some pending leading instructions in the ROB. And In order to respect the in order commit. So, we should not go ahead with the trailing instructions at commit phase. So, as a result whenever any instruction is executed in out of order fashion and some of the leading instructions to this instruction are not yet completed their completion stage then we should not go ahead with trailing instructions for the commit stage.

So, in such scenarios whenever an instruction is finished it execution we have to write values only to the rename registers, but not to the architectural registers. In other words whenever a functional unit completes its execution the computed value will be sent to the RRF rename register file and we update that in the RRF at an appropriate location specified by the register tag because here this RRF will have set of different registers. And by using the register tag of this computer value, we give that as an input to the decoder here in the RRF.

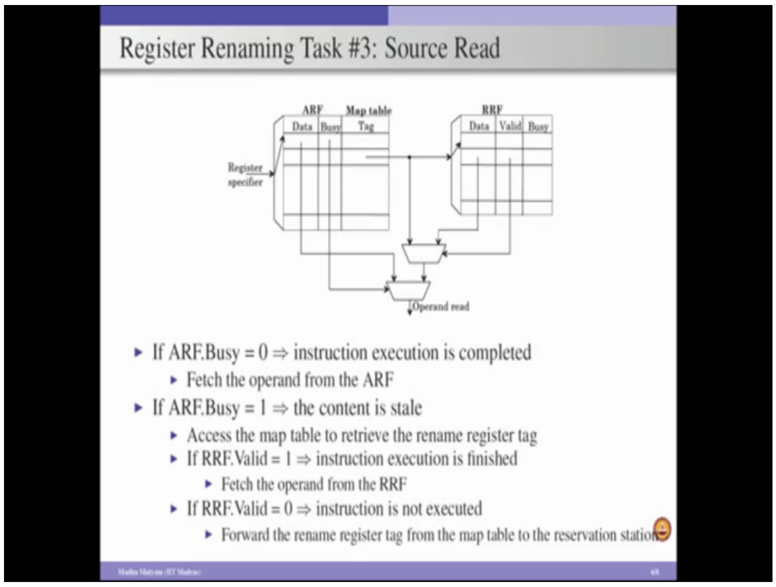
And the decoder is going to select one entry and the selected entry is the entry where we are supposed to write this computer value. And we finally, write the value from this functional unit to the selected entry in that. And once we write that then we have to set the valid bit to 1. So, when an instruction finishes its execution we write the result onto RRF at a location specified by the register tag. And we set the valid bit and as a result any dependent instruction which wants to use this value they can use the value directly from this RRF.

Now, suppose that this instruction whatever recently finished its execution comes to the head of the ROB. And now once it comes to the head of the ROB then it has to be going through the complete stage. That is it has to commit to the architectural register. Note that, whenever an instruction is processed through the complete stage the instruction has to write the value to the architectural register specified in that construction. So, as a result when we are moving from the finish stage to the complete stage, we have to take the value from this RRF and we write it back to the architectural register, but how do we do that because we have this map table. And that maps an architectural register to a register in RRF. So, using that mapping we just go back and then update in the ARF architectural register. And after we write the value to the ARF, then this value is a valid value. So, that from now onwards any dependent instruction which wants to use this value stored in the architectural register, it has to get it from the ARF because once we write the data from RRF to the ARF this entry will be no longer valid. And our complete data will be there only in ARF. So, as a result we have to set the busy bit 0. So, that indicates that this mapping is now lost.

Once we have a busy bit 0 so for any dependent instruction or any trailing instruction which want to use this value from the architectural register, it will be supplied directly from the ARF. And there is no connection between ARF to RRF for this particular architectural register. So, that will be indicated by resetting this busy bit. So, in summary the second phase of the register renaming, we are going to update the registers and the register update happens in 2 phases.

When an instruction is finished its execution we update the computed value in the RRF and when the instruction completes its execution. So, remember the previously I mentioned when an instruction finishes its execution and the second time I mentioned when the instruction completes the execution. So, when we say an instruction completes its execution that means it is processed through the commit stage or complete stage. So, during that time we have to write the value, the computed value, to the architectural register and after that we reset the busy bit. So, that the connection between ARF and RRF for this architectural register is disconnected. And the third step in the register renaming is source operand read.

(Refer Slide Time: 25:11)


ARF
Map table
Data
Busy
Tag
RRF
Data
Valid
Busy
Register specifier
Operand read

Register Renaming Task #3: Source Read

- If ARF.Busy = 0 ⇒ instruction execution is completed
  - Fetch the operand from the ARF
- If ARF.Busy = 1 ⇒ the content is stale
  - Access the map table to retrieve the rename register tag
  - If RRF.Valid = 1 ⇒ instruction execution is finished
    - Fetch the operand from the RRF
  - If RRF.Valid = 0 ⇒ instruction is not executed
    - Forward the rename register tag from the map table to the reservation station

When we have an instruction ADD R1 R2 R3. Now, we have to read the values for R2 and R3. R2 and R3 are architectural registers. So, if these values are available in the ARF then we can straight away read the values from the ARF, but if the values are not available and if the busy bit for the corresponding entries set, then we have to go to rename register file to get the data. So, for that we have this sequence of steps. So, if ARF busy bit is equal to 0 that means that the corresponding entry has the valid data.

So, we can supply the data directly from the ARF to the required instruction otherwise we can fetch the operant from the ARF, but if the busy bit is 1 that indicates that this architectural register is renamed to a register in the rename register file. So, that means we have to use this map table and using this mapping we go to the RRF. So, access the map table to retrieve rename register tag and once we have the tag give this tag as an input to the decoder associated with this RRF.

And this decoder is going to select one particular entry in the register file associated with this rename registers. And in that again if the valid bit is, if the valid bit of the selected entry, is one that indicates that the instruction is finished its execution. So, we can supply the data directly from here to the instruction. So, we can fetch the operand from the RRF but.

If the valid bit is 0 that indicates that the parent instruction the producer instruction is not yet finished its execution. So, that the consumer instruction has to be waited for some time or what we can do is in such scenario when the valid bit is 0 we can supply the tag in the data field of this RRF to the consumer instruction. So, when the consumer instruction is not having all the operands ready then we know that we supply the register tag for the source operands. So, that is what we have going to do here.

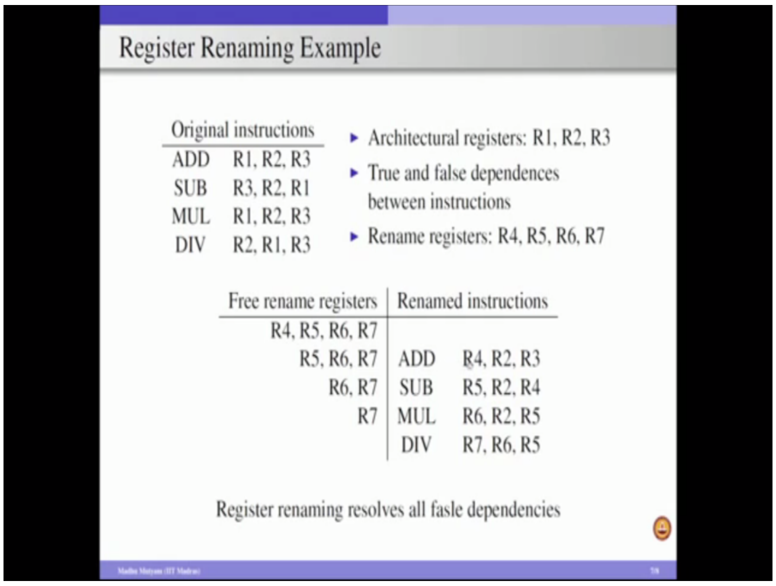
So, in summary in this source read phase. So, when the busy bit of the architectural register is 0 for a specific location, specified by the architectural register the tag which is given as an input to the decoder associated with the ARF. So, we use that we access the busy bit and we access the data field and both will be given. So, one whatever the data we accessed from the data field of the selected entry in the ARF, will be given as an input to this multiplexer. And we give the busy bit whatever we accessed from the selected entry as a select line for this multiplexer.

And if the busy bit is 0 that indicates that this first operand which is given as an input to the multiplexer will be the output for the multiplexer. And that will be given as an operand for our instruction, but if the busy bit is 1 whatever the data we read from here will not be used will not be selected then we have to select only the second input for this multiplexer. And the second input for this multiplexer is coming from another multiplexer and whose select line is the valid bit of a particular entry.

And this entry is selected by the tag of rename register, but the tag of rename register is available from this map table of the corresponding entry. And this is going to give a tag of a rename register and we use that as an input to this rename register decoder as well as we give that as one input to the second multiplexer. And once we select one particular entry in the RRF using this rename register file decoder, we access the data in the data field and that will give us a second in the second multiplexer.

This valid bit whatever we access from the selected entry will be given as a select line. So, if the valid bit is 0 this value is not readily available in RRF and we have to supply only the tag to the consumer instruction. And if the valid bit is 1 then this value is the valid value and then we just supply this data. So, this is the overall process that happens in register renaming logic. So, we have 3 components. One is the destination allocation, the second one is register update, the third one is the source read. So, now we are going to consider an example to discuss this entire process.

(Refer Slide Time: 30:31)



Register Renaming Example

Original instructions	
ADD	R1, R2, R3
SUB	R3, R2, R1
MUL	R1, R2, R3
DIV	R2, R1, R3

- Architectural registers: R1, R2, R3
- True and false dependences between instructions
- Rename registers: R4, R5, R6, R7

Free rename registers	Renamed instructions	
R4, R5, R6, R7		
R5, R6, R7	ADD	R4, R2, R3
R6, R7	SUB	R5, R2, R4
R7	MUL	R6, R2, R5
	DIV	R7, R6, R5

Register renaming resolves all fasle dependencies

Consider a simple piece of code consists of 4 instructions ADD R1 R2 R3, SUB R3 R2 R1, MUL R1 R2 R3 and DIV R2 R1 R3. And now we can clearly see here there are many dependencies. For example, this SUB instruction is using R1 which is produced by ADD. So, effectively there is a true dependency between ADD and SUB, for this R1. At the same time SUB and ADD also using R3 where R3 is used as a destination register for SUB. And R3 is used as a source for ADD instruction and also these 2 instructions uses R2, but both are using as a source operand. So, effectively here read after read, but that is not going to create any hazard, but this one is. So, write after read. So this is WAR write after read hazard and which is going to be because of the name dependence and there is read after write.

So, read after write which is because of true dependence. So, effectively among these 2 instructions we have one true dependence and the one name dependence. And now you can consider the other ones. So, here you can see multiply and ADD again both instructions are using same R1 as a destination. So, effectively output dependency is there which is also part of name dependence between these 2 instructions. And the other 2 operands are same in both the instruction. So, as a result there would not be any hazard. And between multiply and SUB R1 is used in multiply as a destination and R1 is used as a source in SUB. So, effectively here write after read, there is another name dependence between these 2 instructions. And there is a true dependence between the SUB and multiply because of R3.

Finally, with respect to DIV. So, DIV uses R2 as destination register but whereas all other R3 instructions are using R2 as a source. So, effectively here there is write after read hazard is there because this is writing to R2, whereas all other instructions are reading from that. So, we have to read this and then there is a write operation. So, write after read. So, there is name dependence between DIV and all other instructions. And of course, there is a true dependence between DIV and multiply because of this R1. And also there is a true dependence between DIV and SUB because of R3.

So, effectively this code has the true dependence and the false dependence. In other words this code has data dependence, output dependence, anti dependence. So, all these are there but we know that the using register renaming we can eliminate all the name dependencies. So, now we will see how we can do that. So, we assume that our ISA has 3 architectural resisters R1 R2 and R3 and that is the reason why our code also consists of only those architectural resistors R1 R2 R3.

So, our system consists of some physical resistors or also called as rename resistors. And these rename resistors are R4, R5, R6, R7 and we are going to use these registers for renaming some of these architectural resistors in our code. So, we have this list of free rename registers because we have not renamed any of the architectural resistors so far. So, as a result all our rename registers are available. So, every time whenever we want to rename the destination register of an instruction, we pick one free rename register.

So, we start with the first instruction ADD R1 R2 R3. So, we pick the first free rename register which is R4. And we rename this R1 with R4. So, after renaming our ADD instruction which was previously ADD R1 R2 R3 is now becoming like ADD R4 R2 R3. So that any dependent instruction which is going to use the value from R1, now is going to use the value from R4 only. So, next we have SUB instruction.

And we have to rename our second destination register which is R3. So, in order to rename, we select the second the next free rename register that is R5. And now you can see here. So, SUB R3 is renamed with R5, R2 is used as it is, but now SUB actually uses R1 earlier, but now R1 already renamed to R4. So, as a result in our SUB instruction R1 is replaced with R4. Now, we have to go for third instruction which is MUL R1 R2 R3.

So, here we have to rename R1 ok. So, for that we select R6 as the free rename register and we replace R1 with R6. So, MUL R6 R2. And multiply is actually using R3 as the one of the source operands, but R3 is renamed to R5. So, as a result our multiply instruction becomes like MUL R6 R2 and R5. Now, see here in the DIV instruction we are using R1 as one of the source operands, but R1 is here renamed to R4, And here it is renamed to R6. So, now for this DIV instruction whether we are going to use R6 or R4 for this R1.

So, remember when we apply the renaming. So, we always have to consider the latest rename register for the subsequent instructions. So, as a result for this DIV instruction we are going to consider R6 for R1. So, we consider the other free rename register that is R7 and rename R2 with R7. So, as a result DIV R2 becomes DIV R7 and after that the source operands R1 is actually taking the latest rename register that is R6. And R3 is already renamed in the SUB instruction to R5. So, we consider R5.

So, now this entire piece of code is translated to this after applying our renaming logic. So, here we renamed all the destination registers in our code. So, R1 is replaced with R4 first time, R3 is replaced with R5, second R1 is replaced with R6 and R2 is replaced with R7. Now, we will see after applying this renaming, register renaming, whether any name dependency is there.

You can see here ADD and SUB. So, here ADD is supplying the value or producing the value in R4 and SUB is using that as one of the source operands. So, effectively there is a true dependence between ADD and SUB, but there is no other dependence between ADD and SUB because ADD is using R2 R3 as source operands, but whereas SUB is using R5 which is a different register compared to these 2 as the destination the register.

Of course, there is dependence between ADD and SUB through this R2, but that is a read after read. So, that is not going to create any hazard. Now, consider the third instruction MUL. So, MUL is using R6 as the destination, but none of the previous instructions using this R6, either as a source or the destination registers. And R2 is not going to create any problem with respect to the previous one, but R5 is used as a source operant in multiply, but whereas R5 is used as a destination in SUB.

So, there is a true dependence between the SUB and MUL. Of course, in the original code also there is a true dependence between SUB and MUL through this R3. So, as a result we have not changed the true dependence between MUL and SUB after renaming but we eliminated the output dependence because previously there was an output dependence between MUL and ADD through this R1, but now one R1 is replaced with R4 the other R1 is replaced with R6.

So, there is no output dependence between ADD and MUL. And the last one DIV, DIV uses R7 as the destination the register and which is not used anywhere in the remaining instruction. And also here there is a true dependence between MUL and DIV through R6. Previously also there was a true dependence between MUL and DIV using this R1. So, that is maintained as it is, MUL and DIV in the original code has a name dependence because of R2, but now these 2 are not having any name dependence because here we are using R2, but whereas here we are using R7.

So, these 2 are different. So, that also we eliminated. And DIV and SUB in the original code has a true dependence because of R3, still that is maintained here DIV and SUB has true dependence between R5 and R4. So, as a result our register renaming concept eliminates the complete name dependencies among the instructions. And at the same time it is not going to change the true dependence thing. So, it retains the true dependence as it is. So, that means like we are not going to violate the program correctness.

At the same time because the name dependencies are eliminated we can freely execute the subsequent instructions because the instructions those which were having the name dependencies earlier were not able to execute, but now after applying the register renaming. So, these instruction become independent instructions and we can execute them independently. So, as a result that improves the overall performance.

So, with that I am concluding this register renaming concept. And in the next module we are going to discuss the dynamic scheduling by using Tomasulo algorithm. So, in other words we are going to discuss Tomasulo algorithm in detail in the next module.

Thank you.