Report 5 - Superscalar CPU With Dynamic Scheduling

Group 2
E/16/203 - Lakshan S.A.I
E/16/332 - Samaraweera A.A.R.V

Limitations of a Pipelined Processor

When we consider the pipeline processor, it has complex designs to increase the throughput of the processor. But processors that have complex instructions where every instruction behaves differently from the other are hard to pipeline. We can have a longer pipeline to increase the throughput of our processor, but in that case, the depth of the pipeline increases the hazards related to it. Mostly it has worse problems with branch instruction hazards.Processors have reasonable implements with 3 or 5 stages of the pipeline because of this hazard limitations.

What is a Superscalar CPU?

A superscalar processor is designed to achieve an execution rate of more than one instruction per clock cycle for a single sequential program.Superscalar design techniques typically include parallel instruction decoding, parallel register renaming, speculative execution, and out-of-order execution. These techniques are typically employed along with complementing design techniques such as pipelining, caching, branch prediction, and multi-core in modern microprocessor designs.In a superscalar computer, the central processing unit manages multiple instruction pipelines to execute several instructions concurrently during a clock cycle. This is achieved by feeding the different pipelines through a number of execution units within the processor. To successfully implement a superscalar architecture, the CPU's instruction fetching mechanism must intelligently retrieve and delegate instructions. Otherwise, pipeline stalls may occur, resulting in execution units that are often idle.

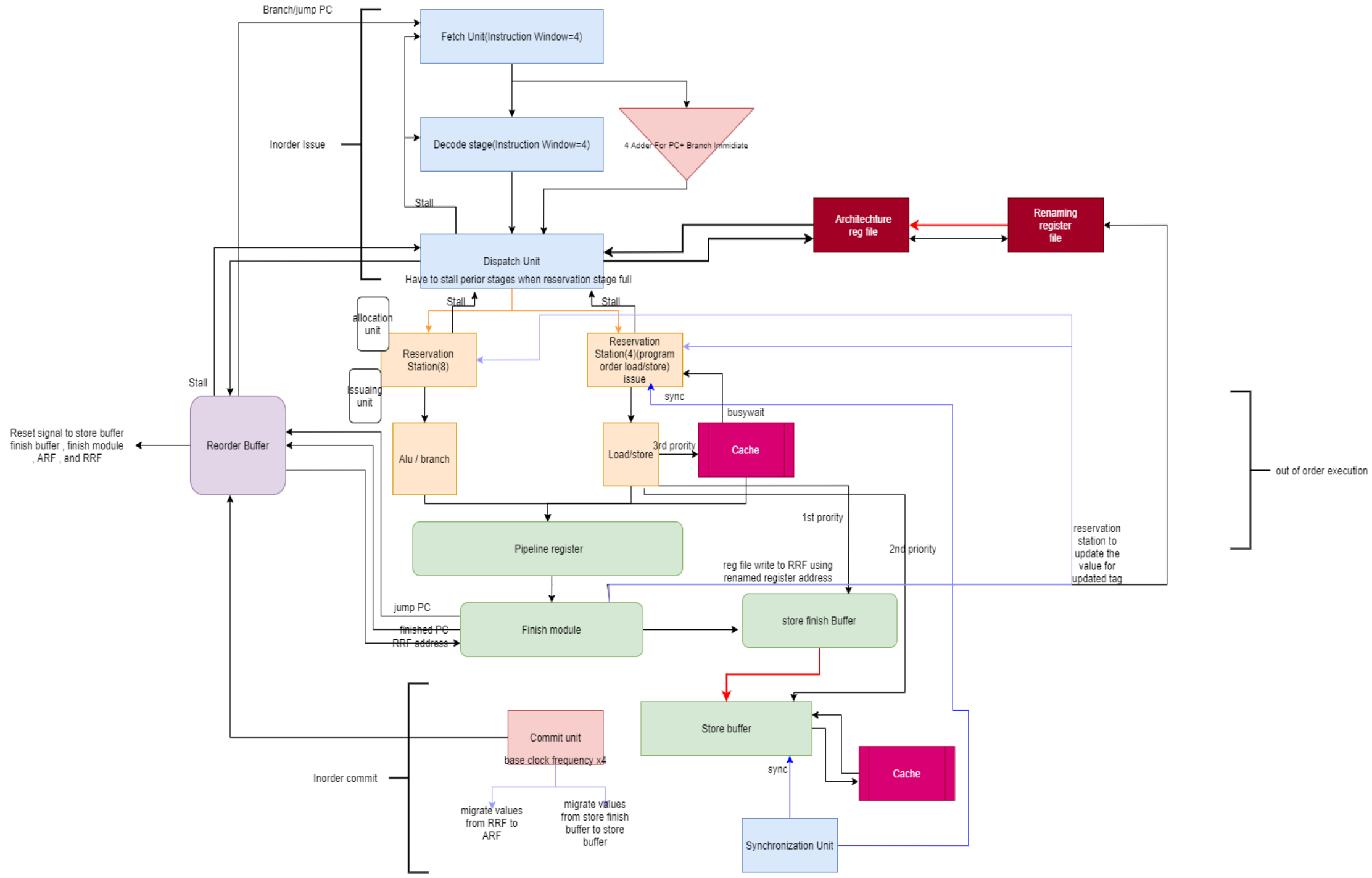
What is Dynamic Scheduling?

Dynamic Scheduling is a technique in which the hardware rearranges the instruction execution to reduce the stalls, while maintaining data flow and exception behavior.This allows code compiled for one pipeline to run efficiently on a different pipeline. Also here, we are using hardware speculation, a technique with significant performance advantages, built on dynamic scheduling. Also this is the process used for handling cases when dependencies are unknown at compile time. In a dynamically scheduled pipeline, all instructions pass through the issue stage in order; however they can be stalled or bypass each other in the second stage and thus enter execution out of order. Instructions will also finish out-of-order

Main details of Our Design

- 4 instruction per clock cycle
- Have 2 clocks
    - base clock
    - Secondary clock 4X in frequency to the base clock
- In order issue
- Out of order execution
- In order commit

Our Design Diagram



1. Instruction fetch stage

From previous design

We have designed our previous fetch unit to fetch 1 instruction per clock cycle along with its address (PC) and the address of the next instruction (next PC) and a signal (inst hit) to identify whether the instruction is a valid one or not.

New design

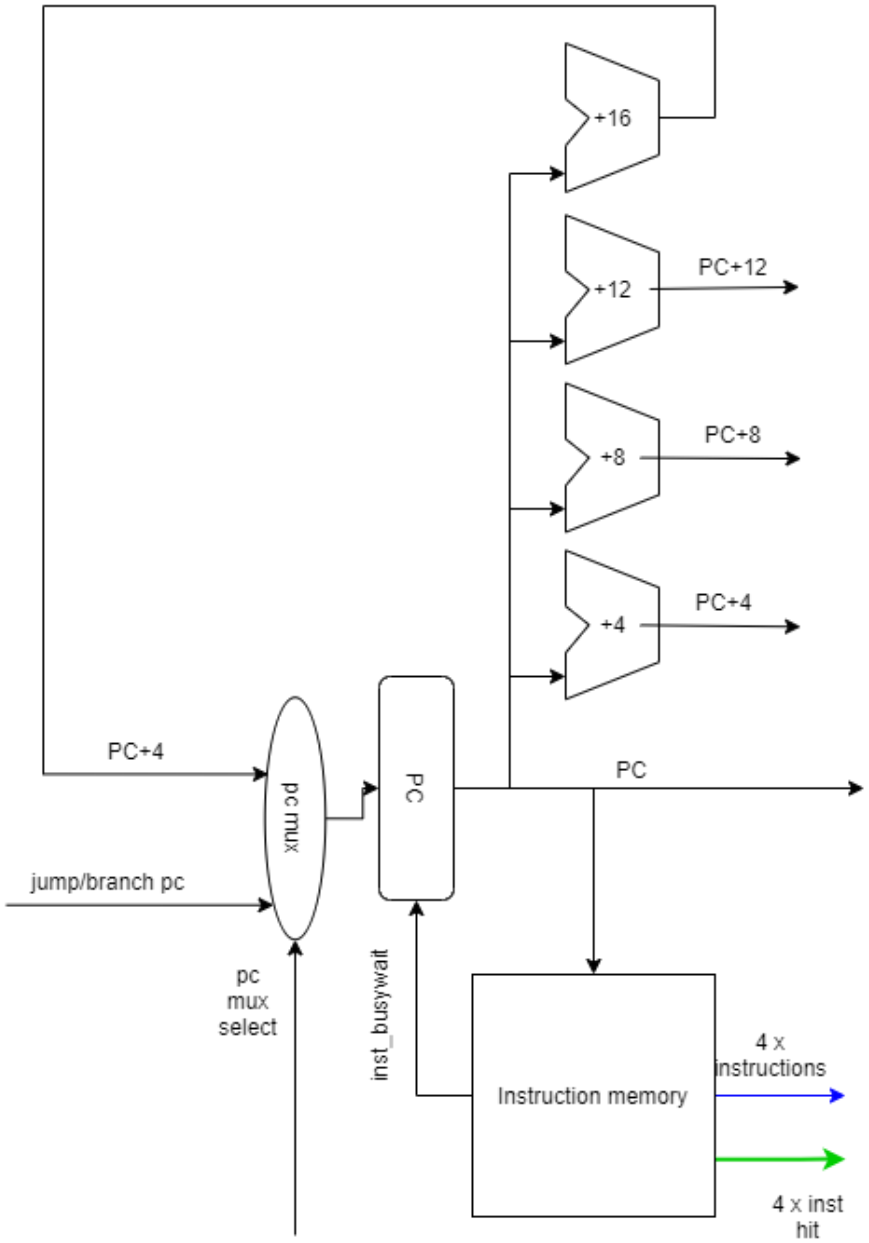
In the new design, we have 4 instruction window.
To execute each instruction we need the following outputs from fetch unit,

1. Whole 32-bit instruction
2. It's address PC
3. Next instruction address
4. Inst hit signal

Therefore to fetch 4 instructions, we need 16 outputs (4 outputs per instruction) to be fetched in a single clock cycle.

But here we thought to remove the next PC address from the fetch unit and calculate 'next PC' using 'PC' in the execution stage with the use of an additional adder because having another 32 bits along with every instruction will consume huge memory space in the reservation buffer, dispatch unit and also pipeline. It is true that having an adder reduce only 32 bits in pipeline but when it comes to the reservation buffer it frees 32 X 4 times of bits. same goes with the dispatch unit.

Therefore the total outputs are now reduced to 12.



Differences introduced to each module

Program counter module:

- Now program counter increases by 16 instead of 4.
- Program counter starts from -16.

Instruction cache:

- Instruction cache will get pc value and fetch 4 instructions ahead starting from the given PC value.

Adder module:

- Now adder keeps 4 adders in it.
- When program starts each adder will add +4 , + 8 , +12 and + 16 in each clock cycle.

2. Decode Stage

From previous design

In the decode stage, we created control signals for each instruction. There were 18 bits as control signals for a particular instruction. These control signals were generated using the control unit.
Then we had 1 register file which had the capability of reading two address simultaneously and asynchronously and write to one address synchronously.
Wire extension module was used to sign extend immediates.
Then we had an adder to generate branch PC for the branch prediction unit.

New design

In the new design we have,

1. Super Scalar Control unit
2. Superscalar Wire extension module
3. 4 adders to calculate branch PCs for each address.

Differences introduced to each module

Superscalar control unit:
Now we have to generate control signals for 4 instructions so we replicate the previous control unit 4 times so that each control unit generate 18 control signals for each instruction assigned to it.

Superscalar wire extension module:
This module consists of 4 units of the previous wire extension module. Which generate all the immediate values needed for each instruction. Therefore there will be 4 outputs with 16 inputs.

Adders:
No particular difference in adders instead of having 4 of them to support 4 instructions.

3. Dispatch Stage

This stage is a new stage for this superscalar CPU design. And does not contain any module inherited from the previous CPU design.

Here we have,

1. Dispatch unit
2. 2 register files
   i. Architecture Register file
   ii. Renaming Register file
3. Reorder buffer

3.1 Dispatch unit

Dispatch unit is responsible for several important things,

1. Identify which instruction belongs to which reservation buffer and the validity
2. Ready the instruction to give to the reservation buffer
3. Store instructions in reorder buffer in-order they arrive
4. Renaming destination register
5. Issue the instructions to the reservation station

1. Identify which instruction belongs to which reservation buffer

This get done using control signals from control unit. Then the dispatch unit decides to which this instruction should get stored.validity will be checked using ‘inst hit’ signal. If not valid instruction will get discarded.

2. Ready the instruction to give to the reservation buffer

Here the dispatch unit reads the ARF and RRF according to a procedure that we explain later in this document.

3. Store instructions in reorder buffer in-order they arrive

To do this dispatch unit will find a free slot using busy bit. Then store PC, branch, jump bits, branch address in the reorder buffer.

4. Renaming destination register

In our design we rename every destination register we receive. Here what we do is in our dispatch unit when it get the destination register address from the instruction it go to the ARF and set the corresponding busy bit to 1. Then it

goes to the Renaming Register File and find a register which have it’s busy bit zero. Then get its TAG (tag means the address of that register in RRF) and store it in ARF in the TAG position. Then also this Tag will be send to the reorder buffer to store it in there.

5. **Issuing instruction to reservation station**

This step is not yet finalized how to do.

This 5 steps are done for all the 4 instructions received in 1 clock cycle with the order they arrive. Step 1,2,3 can happen parallelly while 4 and 5 should happen sequentially.

3.2 Register files

Two register files were introduced to overcome the name dependency.
Each register file supports 8 address reading simultaneously and asynchronously.
But Register writing happens 1 per 1 clock cycle in the finishing stage and committing stage.
Since this is a synchronous operation we have to give a clock. But in this time we do not give the base clock for this. We planned to give the clock which created by increasing frequency by 4 times than base clock (so that period will be 0.25 times the base clock).
This clock was given instead of the base clock because our committing stage has this altered clock too. We write to registers in this stage. so the clocks should be the same.

3.2.1 Architecture register file(ARF)

This register file is the main register file in the processor.
It has the following structure in storing data,

R1	Data	Busy	TAG
R2	Data	Busy	TAG
R3	Data	Busy	TAG
R4	Data	Busy	TAG

The previous design we had only data in the register file but now as shown in the above figure, we have a busy bit and Tag additional to the data. Busy bit indicates whether this particular is used in register renaming. If busy is 1 that means this register does not contain the most recent updated data. 4 bit Tag corresponds to the address of the renaming register. This means tag acts as the mapping from the renaming register file to the architecture register file.

3.2.2 Renaming register file(RRF)

The renaming register file is another register file addition to the ARF. This register file has the following structure in storing data.

r1	Data	Valid	Busy
r2	Data	Valid	Busy
r3	Data	Valid	Busy
r4	Data	Valid	Busy

Here we have valid bit which indicates the data in the particular register in RRF can be used as most recent updated data for a particular register address that maps to the renaming register file.
The busy bit indicates that the corresponding register in RRF is in use by a previous instruction or not.
Note: these bits will be explained again in this document please read along.

How a typical register read happen using ARF and RRF

Let's say we need to read register **R1** from the register file.
Dispatch unit will go to the ARF requesting the value of R1.

There can be several scenarios in ARF,

Scenario 1:- busy bit is 0
No issue the dispatch unit can read the corresponding data value from ARF right away.

Scenario 2:- busy bit is 1
Now this means 2 things.

1. This address is a destination address for a instruction that executed before
2. The updated value is not yet received to the ARF. which means 2 things again
    a. It can be either in transit in mid way in execution stage

b. Or it completed execution but not yet committed.

Now we have to go to the RRF to get this data value. We use the TAG value stored along with data in ARF. using that we can find the particular mapped register in RRF to this register in ARF.

In the RRF we have valid bit. Using this Valid bit we can identify whether the previous instruction that used this address as destination address is

- ❖ Finished but not yet committed or
- ❖ Neither finished nor committed

Sub scenario 1: valid bit is 1
it means the previous instruction is finished but not yet committed.

This means we can use this value. therefore we give this value as the read value for the dispatch unit. This is analogous to the forwarding method in previous RISCV CPU implementation which was used to avoid data hazards.

Sub scenario 2: valid bit is 0
It means the previous instruction which has this address as the destination address is still in transit and not yet finished. So nobody knows its value at this point. Therefore we cannot help in this scenario. We have to stall the previous modules until the data arrives or follow a speculative method.
Therefore what we do is instead of stalling, we give the 4 bit TAG to the dispatch unit instead of 32-bit data with the intention to give the most recent updated data as soon as the previous instruction finishes. (The dispatch unit will include this tag in the Tag space in the reservation buffer. This will be explained later)

With this reading operation is finished.

How a typical register write happen

1. ***Writing to ARF***
   This write operation happens at the commit stage and this operation will be explained there. It will be more clear then. In ARF, writing happen to one address one time

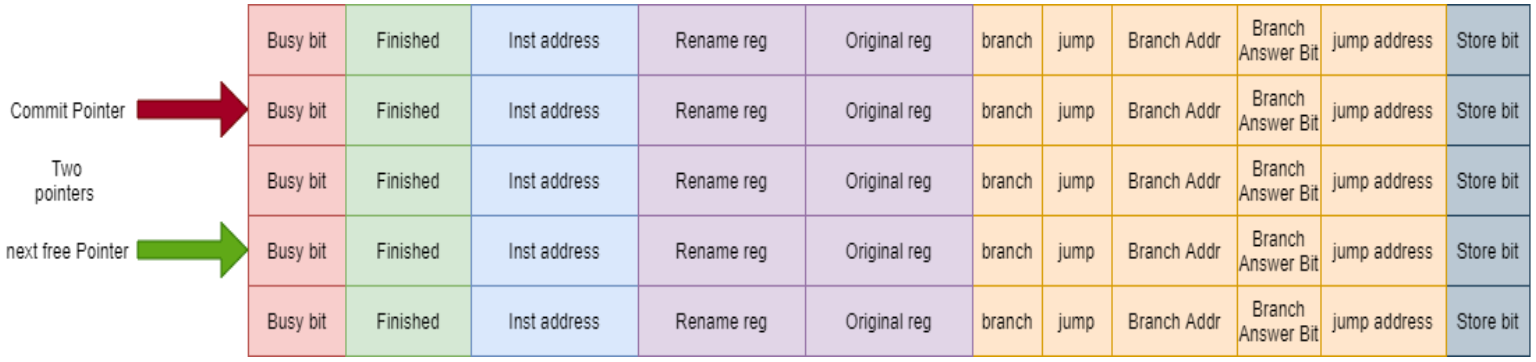
2. ***Writing to RRF***
   This writing operation happens at the finish stage. This also will be explained in that particular stage.In RRF, writing can happen to two addresses simultaneously. There is no way that these two addresses will be the same at any given point. As it is explained in the register renaming operation section in dispatch unit, a particular register in RRF is not reusable until busy bit deassert. This deassertion takes place in the commit stage and will be explained there.

Implementation

We give all the 8 addresses from the pipeline to the ARF. then ARF read 8 addresses parallely. Here first we read a busy bit which incurs less time than reading all 32-bit data. And therefore we can use it as a control signal to a multiplexer. Actually, there are 8 busy bit signals corresponding to each address and also there are 8 multiplexers too. Each multiplexer has two 32 bit inputs and 1 32 bit output. One of that inputs are data values from ARF and the other input is data value from RRF. We select either to forward ARF data to the dispatch unit or RRF data to the dispatch unit using the busy bit outputs in ARF. also we give 8 tags from ARF and 8 valid bits from RRF to the dispatch unit too.

3.3 Reorder Buffer

This functional unit helps to track the instruction order according to programme order. So that we are using this for committing the instructions in order. Here the reorder buffer is a cyclic buffer, where we have two pointers for the next available free slot and next committing instruction slot.In our implementation we are planning to have 10 slots in the buffer



Fields in the Buffer

Busy bit:
We are asserting the busy bit when we use the particular slot otherwise it indicates that particular slot isn't used .Also we are deasserting the busy when we are flushing set of instructions.

Finished bit:
We are asserting when finished the instruction in the finish stage.

InstAdress:
Contains the PC value of the instruction.

Rename reg:
Renamed register identifier is allocated here.

Original reg:
Original register name is allocated here.

Branch and jump bits:
To identify whether the instruction is branch or jump.

Branch Addr:
We set next branching address here, in the branch Instruction.

Branch Answer Bit:
The branch result whether to branch or not would set here

jump Address:
This is jump address. Will get filled in the finished stage after ALU evaluates jump address.

Store bit:
We are asserting store bit in the store instruction

4.Execution Stage

This is the stage where all the executions take place. The modules in this stage are,

1. Reservation stations
2. Alu unit
3. Jump and Branch unit
4. Adder to resolve address for load and store
5. Adder to resolve PC+4 value for instructions

4.1 Reservation station

There are two reservation stations each of which is dedicated to ALU/Branch and Load/Store. This is what allows this SuperScalar CPU to execute Register type instructions and memory instructions parallelly.
Reservation stations are like register files which have special structure in storing instruction data until it is executed.
Each reservation station have two submodules named as,

1. Issuing unit
2. Allocation unit

Issuing unit is responsible for issuing an instruction to execute when it is ready and the allocation unit responsible for finding the next not busy slot to input a new instruction. These units keep pointers as in reorder buffer to access particular free slot or the instruction that should be issued next.

4.1.1 ALU/Branch Reservation Station

This station has the following structure in storing data

Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Ready bit	PC	Control signals

From left to right
Busy bit :
this bit is used to identify a free block in the reservation station before putting instructions to the station by the dispatch unit.

TAG 4 bits:
This is corresponding to the RRF address where the corresponding operand is supposed to stored in. this bits are used when we want to have a mapping to the RRF in order to get the most updated value finish stage.

Value 32 bits : This is the operand 1

Valid bit :
this is used to identify whether the data in value is valid. This is useful when we do not have the most recent updated data yet. At that time we fill only the Tag area and value will contain some garbage data. To identify this we set valid bit to zero.

Then we have same Tag,value and valid bits for operand 2.

Ready bit:
this indicates whether the two operands are valid. This is what is tracked by issuing submodule before issuing an instruction to execute.
Therefore every instruction which have operands that have not yet been resolved get stuck in the reservation station until the particular operand gets the most recent updated value.

PC: PC of the instruction
This is needed by some instructions in ISA and also it is needed to identify instruction order in finish stage by the reorder buffer and finish buffer.

Control bits: 18 bits of control signals that a particular instruction needs for proper execution.

This reservation station contains all the instructions which need the service of ALU in order to execute. Since this is a large portion of the considering ISA the station is large compared to Load/Store reservation station. This station can hold up to two times more instruction than Load/Store reservation station.

Also this station's issuing module is configured such that it can issue instructions which come later without considering who comes first. This is important when a not ready instruction is there and the whole system can get stalled because of that one instruction resulting in a large performance impact.

4.1.2 Load/Store Reservation Station

This reservation station has the following structure in storing data,

Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Immidiate	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Immidiate	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Immidiate	Ready bit	PC	Control signals
Busy	Tag	Value	Valid bit	Tag	Value	Valid bit	Immidiate	Ready bit	PC	Control signals

The difference is having space to store immediates which needs in addition to two operands in store and load instructions.

The other difference is this reservation station issues instructions as first in first out(FIFO). Thus we can avoid having data hazards that occur due to not-reading most recent updated data from data memory. Because of this FIFO, we can be sure that all the storing instructions that were before, in the original program order, now have passed the execution stage. So we can use them.

When reading first we look at the store finish buffer then store buffer and finally the cache. This way we can obtain the most recent updated values for a particular address. Store finish buffer contains the most recent set of store instructions but not committed yet and store buffer contains the store instructions which are committed but not yet written to the cache.

If the reservation buffer get filled it indicate it by setting stall signal to the dispatch unit. Then the dispatch unit will indicate it to the previous modules (pipeline register and the PC). We designed it like this instead of directly informing the fetch unit to stall because stalling can happen at a single reservation station at a time but doing so may stall the whole operation even though the other reservation station need not stalling. But when Having it like this we can let the other reservation station to operate even when one reservation station is stalled. if the next coming instruction is an instruction that needs to go to the stalled reservation station, we cannot help we have to stall the whole fetching from the beginning.

ALU unit: same Alu unit we used in RISC-V pipeline processor no difference in particular

Branch and Jump unit: this is also the same as the previous implementation in the pipelined processor.

Adder: additional 2 adders are used to calculate the store/load address and PC+4 values.

5. Finish Stage

After the execution all the read data from cache, calculated values from ALU, the data needed to store in memory, PC of instructions, control signals needed are stored in the pipeline register. There are space for all of this for at most two instructions which are executed parallelly.

The module in this stage,

1. Finish module
2. Store finish buffer

At the next clock cycle, the finish module send the two PC values from the finished instructions to the the reorder buffer.
Using this PC values reorder buffer gets updated as these instructions get finished. Then the reorder buffer send the corresponding renaming register file address to the finish stage. Then the finish module send the received data from ALU operation or load operation along with the RRF address to Renaming register file to write to it.

Writing operation in RRF

When finish module send the data with address to RRF the register writing happen at the next clock edge. Since finish stage deal with two instructions simultaneously the RRF should support 2 register writings at the same time. This two registers will not be same at any given point. After writing operation the valid bit of the corresponding block set to 1 indicating that the RRF gets written with most recent updated value so that the next instructions can use it for their calculations.

At the same time, this updated values are given to the two reservation stations with RRF addresses so that they can compare the addresses and update the value in the instructions. Then they also set the valid bit to 1 in their block so that the instruction then can pass on to execute. This is why the tag value we stored in the reservation station is important.

If the instruction does not involve register writing at the end, such as store instructions or branch instructions procedure is slightly different.

If the instruction is a store instruction,

Then instead of writing to RRF as the previous stage, the finish buffer will write to the Store finish buffer along with the PC.

Store finish buffer contains data as in the following structure,

Data	Address	PC
Data	Address	PC
Data	Address	PC
Data	Address	PC

This is where data waits until the particular instruction gets committed in the commit stage. Since this buffer contains the most recent store instructions, this will be the first priority in the Load operation.

If the instruction is a branch instruction,

Then in addition to the PC, branch results are also sent to the reorder buffer. Then the reorder buffer will get updated its finish state and branch state too. There is nothing to write to, in these instructions.

If the instruction is a jump instruction,

Then in addition to the PC, jump PC is also sent to the reorder buffer. Then the reorder buffer will get updated its finish state and jump address too.

6. Commit Stage

This stage includes two modules,

1. Commit module
2. Store buffer

6.1 Commit module

Commit module is responsible for two main tasks,

1. **Migrate data from RRF to ARF**.
2. **Migrate values from store finish buffer to store buffer**.

Commit module has a clock of 4 times faster than the base clock. At each positive clock edge, it goes to the reorder buffer find the block which pointed by committing pointer. Check whether it's a jump or branch. if not, get the renaming register address and original register address. Then write to the ARF getting value from RRF.

Writing to ARF

After retrieving addresses commit module set necessary control signals to read RRF using it’s address and write to ARF using its address. After writing busy bit of RRF is set to zero. Then check the RRF address is equal to the TAG stored in the corresponding ARF register.

- If equals, then the busy bit in ARF is set to zero
- If not, it will not set to zero.

The Tag stored in ARF and the RRF address from reorder buffer mismatch means there is some other instruction that not yet committed has the same ARF address as its destination address. So that register renaming process changed the tag for that second instruction and if we set the busy bit to zero in this scenario the next coming instruction may think that this is the most recent updated value while the actual updated value is not yet committed. That is why we need to check tag matches before updating the busy bit in ARF.

If the instruction is a store instruction,

Then there is no any migration from RRF to ARF. commit unit just get that instruction from store finish buffer and put it into store buffer. Store buffer contains store instructions that are committed but have not yet get written to the cache. There is also a synchronization unit that handles the

synchronization between storing and writing to the cache. This operations could not happen at the same time. Therefore at the time that one store gets to the execution stage (that means no reading), so this time window will be used by the synchronization unit to allow the store buffer to write to the cache. If store buffer is filled, then the reservation station for the load/store will receive a stall signal from the synchronization unit and the cache-store will take place afterward.

If the instruction is a branch or jump,

Then the commit stage will forward the value stored in the branch address or jump address block to the fetch unit and make all the busy bits to 0 in the reorder buffer. Also the reset signals was sent to the store finish buffer and finish module to discard all the uncommitted instructions. And also in the ARF and the RRF, all the busy bits will be zeroed. Indicating any uncommitted register file writes to get discarded.

Instruction Execution Paths for Different Instruction Types

1. How ALU Instruction get executed

- Instruction is fetched through the fetch unit of four instruction window with four instruction packet.
- Instruction is decoded in decode stage and generates source and destination registers and particular command signals according to func3 value and opcode. Same time branch address will calculate asynchronously but not related to this instruction
- In dispatch, check whether the alu stage is idle and weather there is a free space in particular reservation buffer. If not, the dispatch unit will stall the previous stages. Also same time dispatch unit checks for reorder buffer and if there is no free slot, the previous stage will be stalled.If there is a free slot, dispatch unit puts a slot asserted as issued. Also at the same time the dispatch unit read the source registers from ARF. If the ARF busy is asserted for a particular reg address, then lookup RRF through the tag. If not, data will be taken from ARF.When we check for RRF, first we check for valid bit of RRF.If valid bit is asserted, data is taken from RRF otherwise the RRF tag will be taken and set as not valid in reservation slot.If only the both source register values are valid, ready bit is asserted in reservation slot. Also then we look ARF for destination register and we tried to find a free RRF register. If we don't find one, dispatch unit will stall as earlier.Then free RRF tag is put in to particular ARF tag field and update related reorder fields.Also RRF busy bit is asserted and valid bit will deasserted..Finally Dispatch unit put this reservation slot into particular reservation station with control signal field and PC fileld.
- Then the issuing unit of the reservation buffer issues the slot in reservation when the ready bit is asserted.
- Alu stage finished its execution out of order.
- At the finished stage, the finish unit finds the related reorder slot of the finished slot using pc and asserted finished bit.Also same time the resulting value will be updated to corresponding RRF field,using the rename reg tag value in reorder slot and asserting the valid bit .Also we are sending wakeup signal for reservation station for acknowledge of availability of particular RRF reg value.

- When the committing pointer of the reorder buffer comes to a particular reorder buffer slot, the instruction will be committed. At this stage the earlier written RRF value writes to ARF and deasserted the ARF busy . Same time check for particular ARF tag and reorder buffer rename reg tag. IF only that are equal we deasserted the ARF busy signal

2. Branch Instruction

- Instruction is fetched through the fetch unit of four instruction window with four instruction packet.
- Instruction is decoded in decode stage and generates source registers and particular command signals according to func3 value and opcode. Same time branch address will be calculated asynchronously by the adder parallel to this unit.
- In dispatch, check whether the alu stage is idle and whether there is a free space in particular reservation buffer. If not, the dispatch unit will stall the previous stages. Also same time dispatch unit checks for reorder buffer and if there is no free slot, the previous stage will be stalled.If there is a free slot, dispatch unit puts a slot asserted as issued and speculated.Also branching address is updated to reorder buffer. Also at the same time the dispatch unit read the source registers from ARF. If the ARF busy is asserted for a particular reg address, then lookup RRF through the tag. If not, data will be taken from ARF.When we check for RRF, first we check for valid bit of RRF.If valid bit is asserted, data is taken from RRF otherwise the RRF tag will be taken and set as not valid in reservation slot.If only the both source register values are valid, ready bit is asserted in reservation slot.dispatch put the slot into reservation station.
- When both source operands are ready, issue unit of the reservation station issues the instruction.
- Execution stage executes branch instruction and the result is produced.
- At the finish stage, the finish stage marks the instruction as finished in the reorder buffer. Also we store the result in the particular reorder slot.
- When a particular reorder slot comes to the commit stage, we check whether speculation is correct or not.If correct we continue the current execution flow. If not correct ,we deasserted the busy bits of next

reorder slots.Also we send the branch address to the PC for next execution.

3. Load Instruction

- In load instructions also follow the same pattern in the fetch,decode and dispatch stages as discussed earlier.
- Here,the issue unit in the reservation station is issuing instructions in program order to eliminate the same memory address hazards.
- In the execution stage, we check the store finish buffer and store buffer for value first by giving a memory address. If we found we take priority order as store finish buffer and store buffer. If we don't find an address, we are going to cache and take the value from memory.
- After that, in the finish and commit stages,we are following the same procedure as we discussed earlier.

4. Store Instruction

- In store instructions also follow the same pattern in the fetch,decode and dispatch stages as discussed earlier.
- Here we set the store bit at the dispatch stage for recognition as a store instruction also issuing from the reservation station is happening in the program order.
- At the finish stage we are writing to finish store buffer with the value , PC and memory address.
- When the instruction reaches the commit stage, we write particular data and memory address into the finished buffer.
- By the synchronize unit, these values are written to memory in order when the memory accessing is idle.

Problems Encountered and Solutions Given

- **How to secure program order?**

  We are using a recorder buffer to maintain the program order .And also store finish buffer and rename register file helps to store the finished values until committing.

- **How to resolve from wrong speculations?**

  We use speculation in the branch instructions. Here we speculate as the branch is not taken and continue the program executions. When the particular branch finishes we store the result in the reorder buffer and also we have already marked the particular instruction as a branch. So that when the instruction reaches the commit stage, we check the correctness of speculation. If the speculation is wrong we flush(Reset) the reservation stations, RRF and store finish buffer. Also we asserted as not busy for the next reorder slots. Same time we are using the calculated branch address in the particular reorder slot, to fetch the next instruction.

- **How do you resolve memory address hazards?**

  When we consider out of order execution. There can be a possibility that we have two instructions that are store instruction and load instruction in actual programme order. In out of order execution, there could be a possibility to finish the load instruction first and next store instruction since we are storing at the end of the committing stage. If the two instructions refer to the same memory addresses then we are reading wrong values and hazard occurred.

  This case could be worse when multiple load and store instructions are referred to the same address. So that we take a design decision to handle this by issuing load and store instructions only in program order from the reservation stations.Then the hazard is going to be eliminated

- **How to eliminate structural hazards in memory accessing?.**

  Here we are using the same cache for memory load and store. But at the same time load is happening in the execution stage of the load/store path and the store is happening at the end of the committing stage by a separate unit. So that there is a possibility of structural hazard when both are trying to access the cache at the same time.So we are designing a separate commit store buffer for store the committed store instruction values. So that we don't need to access the memory to store at all times. We are giving priority for load instructions so that they can access the memory. When the store buffer fills 6/7 potion of its size we are going to store in memory and acknowledge the load process so that the memory bus is busy.

- **How to synchronize load and store memory accessing?**

  When we are doing load instruction, there is an issue where we have the most updated value for that particular address. We handle that by parallel checking the finish store buffer and store buffer. Then we are giving priority to the finish store buffer. If the requested memory address is available in those. We are taking according to our priority. If not we are going to memory and fetch the values.

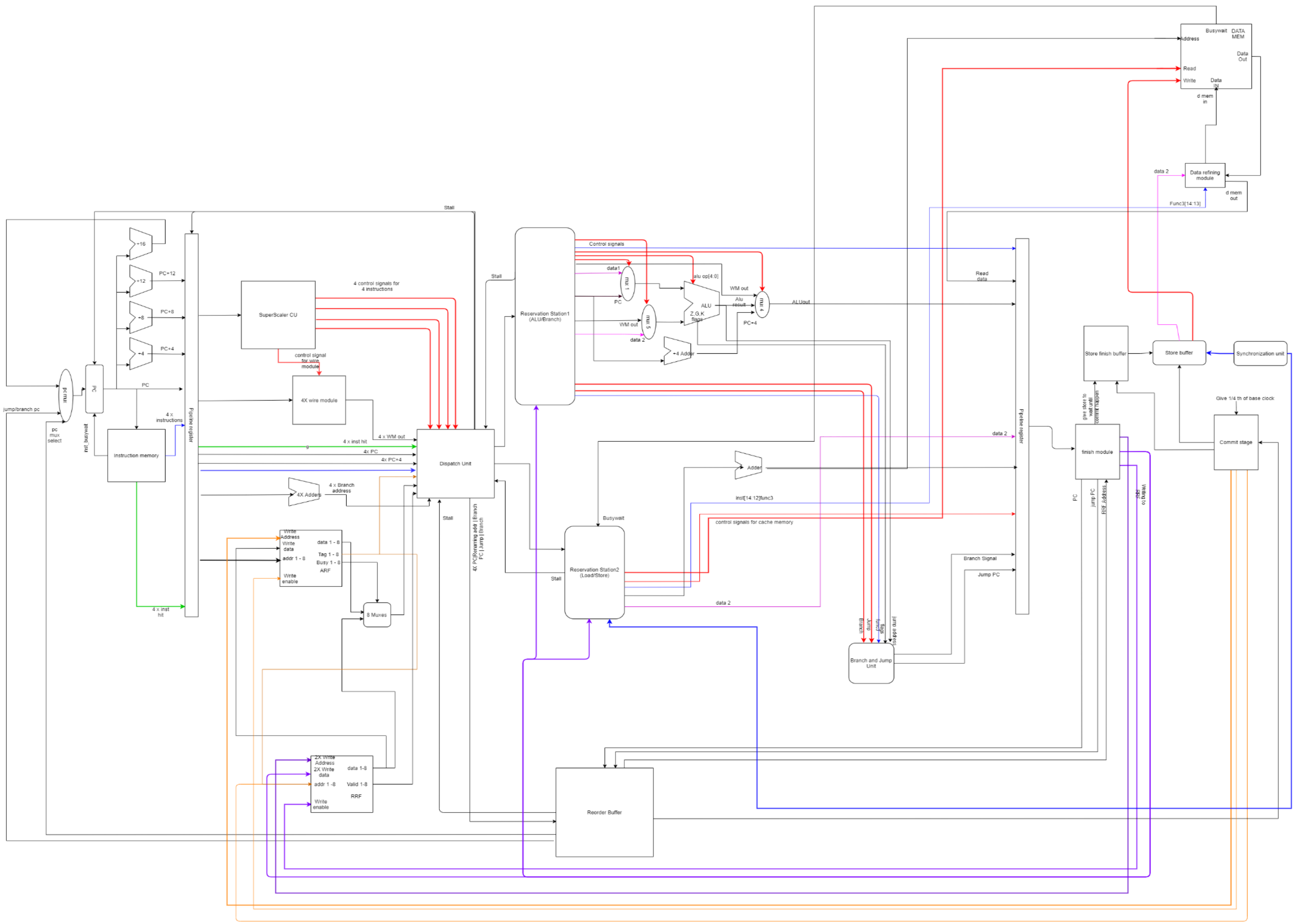
- **How to eliminate the commit stage bottleneck?**

  We have four instruction window fetch units so that we are fetching 4 instructions at clock cycle. At the same time we are committing only one instruction at the clock cycle. So there is a bottleneck at the committing stage for instructions. So we are planning to use a(¼ *fetch stage clock) for committing stage clock to reduce the bottleneck and faster the committing.

References

- Computer Architecture Quantitative Approach (Section 3 Instruction-Level Parallelism and Its Exploitation) by John L. Hennessy and David A. Patterson
- https://www.cs.umd.edu/~meesh/411/CA-online/chapter/dynamic-scheduling-example/index.html
- https://www.slideshare.net/shreyabaheti/pentium-iii
- https://www.youtube.com/c/OnurMutluLectures
- https://www.youtube.com/watch?v=GxesVUkmSLA
- https://www.youtube.com/channel/UCxrGj2wisv-oeorFU_cTX4A
- https://www.youtube.com/watch?v=oVesh-w0-OI
- https://www.youtube.com/channel/UCPSsA8oxlSBjidJsSPdpjsQ
- https://www.youtube.com/user/srsarangi
- https://www.sciencedirect.com/topics/computer-science/superscalar-processor

Datapath V6



Stall
Stall
+16
+12
+8
+4
PC+12
PC+8
PC+4
PC
pc mux
jump/branch pc
pc mux select
inst_busywait
Instruction memory
4 x instructions
Pipeline register
4 x inst hit
SuperScaler CU
4 control signals for 4 instructions
control signal for wire module
4X wire module
4 x WM out
4 x inst hit
4x PC
4x PC+4
4X Adders
4 x Branch address
Dispatch Unit
Stall
Address
Write data
addr 1 - 8
Write enable
data 1 - 8
Tag 1 - 8
Busy 1 - 8
ARF
8 Muxes
2X Write Address
2X Write data
addr 1 -8
Write enable
data 1-8
Valid 1-8
RRF
Reservation Station1 (ALU/Branch)
Control signals
data1
mux 1
PC
mux 5
WM out
data 2
alu op[4:0]
ALU
Z,G,K flags
WM out
Alu result
mux 4
PC+4
ALUout
+4 Adder
Adder
Busywait
Reservation Station2 (Load/Store)
Stall
inst[14:12]func3
control signals for cache memory
data 2
Branch and Jump Unit
Branch Signal
Jump PC
Reorder Buffer
Read data
data 2
Pipeline register
Busywait
DATA MEM
Address
Data Out
Read
Write
Data IN
d mem in
Data refining module
data 2
d mem out
Func3[14:13]
Store finish buffer
Store buffer
Synchronization unit
Give 1/4 th of base clock
Commit stage
finish module
PC
Jump PC
ROB Address