Indian Institute of Technology Bombay

EE 705: VLSI Design Lab

Implementation Of A Superscalar CPU Design

May 20, 2021

Student Name	Roll Number
Shubham Kar	180070058
Garaga VVS Krishna Vamsi	180070020
Rupak Kundu	203074008
Anurag Agarwal	203070056
Alok Kumar	204076011

Overview

Superscalar CPU design implements the idea of having more than 1 parallel pipeline in a single CPU architecture. We have tried to implement a simple superscalar architecture with a width of 2 pipelines using a MIPS like Instruction set architecture.
We fetch two instructions at a time in the Fetch Unit. We then decode the instructions two at a time and pass the control bits to the Dispatch Unit. The Dispatch Unit then passes the instructions to the respective Reservation Stations for the ALU, FP, Memory or the Branch Execution units. The Reservation Station manages the dispatching of the instructions stored in its buffer to the respective execution units. The outputs of the execution unit is next fed to the Common Data Bus(CDB) which is continuously monitored for forwarding the corresponding valid outputs of the execution units back to the Reservation Stations and the Dispatch Unit. The CDB is also monitored by the Reorder Buffer to determine if an in-flight finally gives a valid result or not. The Reorder Buffer operates as a queue wherein the new instructions are introduced at the tail end of the queue and the head of the queue is incremented(by maximum of two instructions in a single clock cycle) whenever the results of those instructions become valid. These are then written back into the Register File/Memory in the Completion stage which is partly handled by our Dispatch unit itself(for Register Writeback) and the Store Buffer(for Memory Writeback).
We now go stage by stage to describe each stage's work and way of implementation in detail.

Fetch Stage

The Fetch Unit requires an input of the Program Counter. This Program Counter is fed to the Fetch Unit via a Multiplexer which is responsible for selecting the next program counter depending on the branch execution that takes place at various stages in the whole superscalar pipeline.
We have included a 1-bit branch memory to include some sort of branch prediction in our pipeline. When the program counter is read at the rising clock edge, the fetch unit searches for the program counter in the Branch History Table which is a part of the Fetch Unit. It forms a sort of Branch History Cache. If the program counter is available in the cache, then the branch is taken and the next program counter is passed on as the destination Program Counter stored in the Branch History Cache. A prediction bit is set for the corresponding branch instruction and the Branch History Table Location for that particular Branch Instruction is sent along with the instruction to the Decode Stage. If the Program Counter is not present and it is a Branch Instruction, then a spot is reserved for the Branch Instruction in the Branch History Table and the branch is not taken. The prediction bit is set as 0 and the location of the reserved Branch History Cache index is sent along with the instruction to the Decode Stage. The location is sent because once the branch instruction execution completes in either the Execution Stage or the Decode stage, the result can be inspected by the Fetch Unit at every rising clock edge to check whether the prediction was right or not. If it was not, then the current destination program counter for that particular branch instruction is overwritten and if the branch instruction was not present in the cache before, the branch history row for that instruction is filled with the executed result.
For instructions which are not branches, nothing special is done and the instructions are just passed as it is to the Decode Stage. The Fetch Unit takes two 2 inputs from the Decode Stage and one input from the Branch Execution Unit. This is to help it update the Branch History Table once a branch instruction is evaluated in either of these stages. This helps us in Bypassing the Branch instruction result being forwarded from the ROB itself as Branch instruction need not be evaluated in an in-order-fashion as the speculation bit and branch tag takes care of it everywhere.

Decode Stage

The Decode Stage is responsible for two broad jobs. One of the job is to decode the instructions and send the corresponding register data and the architectural register tags for the destination and the operand registers along with the immediate operand(if any). It also sends control bits to be used by the different stages further in the pipeline. The other job is to check whether the instruction received is a jump or a jump-and-link instruction. If they are, then they are evaluated in the Decode Stage itself and the evaluated output is then sent back to the Fetch Stage in the next cycle to be used by the Fetch Unit to get the next Program Counter.
For decoding of instructions, we use the firt 6 bits of the Instructions as opcode and based on the opcode, we decide the instruction and then provide the control bits. For opcode "000000", we will need the Function field, the last 6 bits of the Instruction to decide the instruction. The mapping from opcodes to instruction are shown below.

There are 27 control signals, most of which are redundant. The meaning of each of these control signals are shown below. The 27 control signals are mapped according to these. Apart from these control signals, decode stage also provides I1_Valid signals and I2_Valid signals, Branch tags and speculative bits. These Branch tags and speculative bits are also updated based on the outputs from ROB once the branch instruction get resolved.

Opcode	Function	Instruction
000000	100000	Add
000000	100010	Sub
000000	101010	Slt
000000	110000	Mul
000000	110001	Div
000000	100100	And
000000	100101	Or
000000	100110	Xor
000000	001000	jr
000000	000010	Srl
000000	000011	Sra
000000	000000	Sll
001000		Addi
001010		slti
001100		Andi
001101		Ori
001110		Xori
100011		Lw
101011		Sw
000010		Jump (J)
000001		bltz
000100		beq
000011		jal
000110		FP Add
000111		FP Mul

Location	Control signal	of bits	Meaning
26	ALU_instr	1	1 when its an ALU instruction
25	FP_instr	1	1 when its an FP instruction
24	Mem_instr	1	1 when its an Mem instruction
23	Br_instr	1	1 when its an BR instruction
These four signals are used to select the Execution unit			
22	R_type	1	1 for R type instructions
21	Imm_type	1	1 for Immediate type instructions
20	RegWrite	1	Write enable for the register file
19:18	RegDst	2	00 if dest rt, 01 if dest is rd, 10 for jal, $31
17:16	RegInSrc	2	00 for Memory unit output, 01 for ALU unit output 10 for Incremented PC, 11 for FP unit output
15	ALUsrc	1	'0' for Register file output, '1' for Immediate operand
14	Add_sub	1	0 for Add, 1 for Sub
13:12	Logic_ctrl	2	for 00 for And, 01 for Or, 10 for Xor
11:9	alu_outp_control	3	shift_outp when "000", slt_outp when "001" add_outp when "010", logic_outp when "011" mult_outp when "100", div_outp when "101"
8:7	Shift_control	2	00 - no shift, 01 - sra, 10 - sll, 11 - srl
6	FP_mul/add	1	0 for FP add, 1 for FP mul
5	DataRead	1	Enable signal for Memory Read
4	DataWrite	1	Enable signal for Memort Write
3:2	BrType	2	00 - no branch, 01 - beq, 11 - bltz
1:0	PCSrc	2	00 -Incremented PC, beq, bltz, 01 -jump, jal, 10 - jr

The jump and the jump-and-link instruction is handled in a special way by the decode stage. Both of these instructions are passed down to the Dispatch Stage too just as the other instructions as the bookkeeping is done at the ROB which gets these from the Dispatch Stage. The point of difference comes when the jump and the jump-and-link instructions are also evaluated in the Decode Stage itself. The evaluation output of these instructions are given as a Branch Result Bit to indicate whether the branch was mispredicted or not. This is fed back to the Fetch Unit where the Branch History Table is updated with the correctly evaluated Program Counter upon branching.

Dispatch Stage

The Dispatch Stage is crucial for the Superscalar CPU architecture. This stage is responsible for dispatching the instructions coming to it in an in-order fashion to the reservation stations. Dispatch stage receives the decoded instruction from the decode stage. In the case of 2-fetch superscalar, the dispatch stage receives two decoded instructions from decode stage. Each instruction has associated control signals and register Ids or immediate operand. The dispatch stage resolves the false dependencies by using register renaming. In our implementation, the Architectural Register File (ARF) and the Rename Register File (RRF) are a part of the Dispatch Stage itself. In our architecture there are 32 architectural registers (in ARF) and 32 rename registers in (RRF). For every instruction, dispatch stage determines if the instruction writes back to the ARF. In case of a register writeback, dispatch stage allocates a new rename register from the RRF. The register ID of the rename register is updated as the tag of the corresponding register in ARF. Any instruction which needs to further read this architectural register, must read the tagged register in RRF. This resolves any false dependencies.

The instruction in dispatch reads its operands from the ARF or RRF. In case operands are not available, the instruction reads the tag of the associated rename register. The dispatch stage reads the free locations of ROB.

The control signals, operands (or their tags), tag of the destination and location in ROB is then sent to the reservation station. Dispatch stage also sends the instruction and its associated architecture and rename registers simultaneously to Reorder Buffer (ROB) .

As the reservation stations are decentralized, dispatch unit sends the instruction to appropriate reservation station.

Inputs to Dispatch stage are described below.

- **From Decode stage**:Dispatch stage receives instruction, operands,control bits, source and destination registers from decode stage. Dispatch receives 2 instructions from decode, in each clock cycle.
- **From execution units**:Dispatch stage receives forwarded operands and rename register tags from the execution units.
- **From Reorder Buffer (ROB)**: Dispatch stage receives free ROB locations and commit signals. Every cycle dispatch receives 2 free locations of ROB. These locations of ROB shall be used to store the current instruction. As part of commit ROB sends the architectural register and rename register. Once commit is received, the architectural register file is updated with the result and the rename register is freed, so that it is available for subsequent instructions. Dispatch sends the ROB location along with the instruction to the appropriate reservation station. ROB location is used by ROB to update the output of execution unit into correct location in ROB.

Outputs from the dispatch unit are described below

- **To reservation stations**: Dispatch unit sends each instruction along with its associated control signals, operands (or tag of operand), operand validity bits, tag of destination and ROB location to the correct reservation station.
- **To Reorder Buffer**:Dispatch stage sends each instruction along with its control signals, Program Counter and architectural,tag registers of destination. Every instruction dispatched is also stored in ROB. Thus, instruction dispatch and ROB storage are done in order. ROB uses the information of destination register (architecture and tag registers) during commit stage.

Every execution unit forwards its result along with tag (of rename register) to dispatch. This resolves the dependencies of instructions in dispatch stage. If the operands are available either in ARF or RRF or on forwarding slots, dispatch unit sends the operand to reservation station or else the rename register tag of operand is forwarded.

Dispatch stage writes the result to corresponding rename register. The rename register is written to appropriate Architectural register, when commit is received from ROB. ROB sends the architectural register name and rename register name as part of commit. Dispatch stage writes the data of the rename register into corresponding architectural register. The instruction execution is completed once the corresponding architectural register is written.

Reservation Station

Instructions wait in the reservation station till their operands are ready. We have designed decentralized reservation stations, with each execution unit having its own reservation station. Thus there are 4 reservation stations, one each for ALU, FPU, Memory and Branch units.

The reservation station sends the instruction to execution unit once its operands are ready. Instruction are executed as soon as their operands are ready. This is out of order execution.

Reservation station performs forwarding of operands from execution units. Every execution unit upon completion of instruction execution forwards the result and tag (of rename register) to the reservation station. Any instruction, which is wait for any operand, matched its tag with the tag forwarded from the execution unit. In case of tag match, the instruction stores the result in place of the tag. An instruction is ready for execution once, all its operands are available. The instruction is then sent to the execution unit.

In our design, we have taken each reservation station to have 8 locations. Thus the processor can support 32 in flight instructions, at any point in time.

Inputs to reservation station are described below.

- **From Dispatch stage**: Each reservation gets the instruction, control bits, operands, operand validity bits, tag of destination register and ROB location of this instruction. The operand can be valid operands or they can be tags of rename register, This is indicated by the validity bit of the operand. If the operand valid bit is 0, then the operand location contains tag and the instruction waits till the matching tag is forwarded by the execution unit.
- **From execution units**: Reservation station receives forwarded operands and rename register tags from execution units. Any instruction waiting for an operand, stores the operand (if tag is matching). This resolves true data dependencies.

Outputs of reservation station are described below.

- **To execution units**: Reservation station sends the instruction, control bits, destination register tag, ROB location to the execution unit. The instruction, control bits and operands are used by the execution units as desired. Destination register tag is used for forwarding result and ROB location is used by ROB to store the instruction.

The instructions which are ready are sent to execution unit ahead of instructions dispatched earlier, thus leading to data driven execution.

Execution Units

ALU

The ALU executes the normal arithmetic and logical instructions like add, addi, and, andi etc. along with div and mult instruction. The normal arithmetic instructions use a 32-bit adder which uses four 8-bit Kogge Stone Adders in a ripple carry fashion. The div and mult instructions are inferred using behavioral statements in the hardware description. The list of instructions it executes is shown in Table 1.

Instruction	Instruction Type	Example	Dataflow
add	R-Type	add R1, R2, R3	R3 $\leftarrow$ R1 + R2
sub	R-Type	sub R1, R2, R3	R3 $\leftarrow$ R1 - R2
slt	R-Type	slt R1, R2, R3	R3 $\leftarrow$ 1 if R1 < R2 else 0
mult	R-Type	mult R1, R2, R3	R3 $\leftarrow$ R1 * R2
div	R-Type	div R1, R2, R3	R3 $\leftarrow$ R1 / R2
and	R-Type	mult R1, R2, R3	R3 $\leftarrow$ R1 and R2
or	R-Type	mult R1, R2, R3	R3 $\leftarrow$ R1 or R2
xor	R-Type	mult R1, R2, R3	R3 $\leftarrow$ R1 xor R2
addi	I-Type	mult R1, R2, Imm	R2 $\leftarrow$ R1 + Imm
slti	I-Type	mult R1, R2, Imm	R2 $\leftarrow$ 1 if R1 < Imm else 0
andi	I-Type	mult R1, R2, Imm	R2 $\leftarrow$ R1 and Imm
ori	I-Type	mult R1, R2, Imm	R2 $\leftarrow$ R1 or Imm
xori	I-Type	mult R1, R2, Imm	R2 $\leftarrow$ R1 xor Imm

Table 1: Instruction executed by ALU

Floating Point Unit

The Floating Point Unit is responsible for executing instructions such as Floating Point Addition and Multiplication. The list of instructions it executes is shown in Table 2:

Instruction	Instruction Type	Example	Dataflow
fadd	R-Type	fadd R1, R2, R3	R3 $\leftarrow$ R1 + R2
fmult	R-Type	fmult R1, R2, R3	R3 $\leftarrow$ R1 * R2

Table 2: Instruction executed by FPU

Memory Unit

It is a two-pipeline Execution Unit which calculates the memory address corresponding to the memory instruction in the first pipeline stage and then accesses the memory in case of a load word instruction in the second pipeline stage. The output of this execution unit is a 32-bit memory address output and a 32-bit operand output, both to the Common Data Bus. The Operand output is the data to be written back to the destination register in case of load word and it is the data to be stored at the calculated memory address location in case of a store word instruction. This Unit also stores the output in the store buffer in case a store word instruction completes its execution. This is to prevent writing to the memory in an out-of-order fashion and is only written back to memory upon successful commit from the ROB. The list of instructions it executes is shown in Table 3.

Instruction	**Instruction Type**	**Example**	**Dataflow**
lw	I-Type	lw R1, R2, Imm	R2 $\leftarrow$ mem(R1 + Imm)
sw	I-Type	sw R1, R2, Imm	mem(R1 + Imm) $\leftarrow$ R2

Table 3: Instruction executed by FPU

Branch Execution Unit

This unit evaluates branch instructions which require reading register values and not only seeing the PC value of the instruction like bltz, beq and jr. This unit has a prediction bit as an input to indicate whether the branch was taken or not in the fetch unit. It also takes in the branch tag of that instruction as an input. If the branch was mispredicted, then a Branch Result bit is enabled else its disabled. The branch tag along with the Result bit is packed together into a bus output of the same size as the Rename Register Tag and is sent to the Reorder Buffer which in turn sends it to the Fetch, Decode, and the Dispatch Unit. The Branch Result Bit calculated after evaluating the branch is used by the respective units to flush the instructions which are not in-flight without getting registered in the ROB. The list of instructions it executes is shown in Table 4.

Instruction	**Instruction Type**	**Example**	**Dataflow**
beq	R-Type	beq R1, R2, label	Goto label if R1 = R2 else don't
bltz	R-Type	bltz R1, label	Goto label if R1 < 0 else don't
jr	R-type	jr R1	Goto Address stored in R1

Table 4: Instruction executed by FPU

Reorder Stage

The Reorder Buffer has the job of book keeping the instructions. The structure of the Reorder Buffer is that of a circular queue with a head and last pointer pointing to the head and tail ends of the queue respectively.
Once an instruction arrives at the Dispatch stage, it is also sent to the Reorder Buffer. If the instruction is valid, the instruction details are stored at the tail end of the Reorder Buffer. The tail pointer is incremented accordingly and the next two free locations in the Reorder Buffer are sent back to the Dispatch Stage to help it pipeline the next two in-flight instructions. The details of each instruction stored in the Reorder Buffer is shown below:

- **Program Counter of the instruction**: Helps in mere bookkeeping of the instruction details. Is used by the Branch Execution Unit as well.
- **Speculation Bit**: Whenever an instruction is taken after a Branch prediction in the Fetch Unit, the instruction becomes speculative in nature. Enabling this bit implying that the instruction is speculative and should not be committed unless this speculation is resolved.
- **Branch Tag**: Whenever an instruction is speculative, a branch tag is associated with it implying that the instruction was speculated under the branch instruction x with Branch Tag x. If $x_1 < x_2$, it implies that the execution of first branch removes speculation for the the instruction with tag x_1 but not for the one with tag x_2 but then, the branch tag of x_2 is reduced by 1 to indicate that a parent branch was evaluated. However, evaluation of second branch instruction removes speculation for none of the instructions but the speculation for both the instructions depend only on the parent branch x_1.
- **Control Bits**: The Control Bits are used to check whether an instruction needs to write back into the register file or not. If set, than writeback signal is sent upon the commit of that particular instruction. If not, no such signal is sent and the garbage value sent as output from ROB to register file are ignored.
  This is also helpful when a store instruction needs to be committed. In that case, we need to send the Store Commit signal and the location of the store instruction in the store buffer back to the store buffer in order to finally perform a memory writeback.
- **Architectural Register Tag**: The Architectural register tag for the instruction that was sent by the dispatch stage is sent back to it upon committing an instruction to check for the next rename register tag.

- **Rename Register Tag**: The Rename Register Tag is provided to the architectural register used in the instruction if it needs a register writeback. This is sent to the reorder buffer and needs to be checked upon writeback to accordingly update the contents of the register. Therefore, it is provided by the ROB upon the instruction commit.
- **Valid Bit**: It is disabled once instructions arrive at ROB from the Dispatch Stage. Its enabled when the particular instruction gets executed in one of the execution units. This information is gathered by the ROB from the CDB at each rising clock edge.
- **Busy Bit**: This is set whenever the instruction is in flight in the execution unit. It is disabled when the instruction finally gets resolved. It is the complement of the valid bit if the instructions only go through the Execution units once. However, it is not so if a single instruction can loop back to the reservation stations to complete another half of its execution. This facility is not used in this project but is maintained for future practicality.
- **Flush Bit**: This is set whenever an instruction needs to be flushed from the ROB. This is usually the case when a speculative instruction gets speculated as wrong. Then, that instruction needs to the flushed. This bit helps in doing that.
- **Store Buffer Location**: These set of bits are used to store the store buffer location received from the Memory Execution Unit for a sw instruction. Whenever a store word instruction has to be committed, a store commit signal will be sent from the ROB to the Store Buffer and along with that the Location stored in the ROB corresponding to that store instruction will also be sent. This helps the Store Buffer in sending the correct memory address and the data to be stored in the memory.

Whenever a new instruction needs to be committed, a scan is made by the ROB to check whether the head pointer points to an instruction which is meant to be flushed or not. If yes, then the head pointer is incremented such that no such instructions are present in the queue. If not, then the first two instructions are committed if both of them are valid, not busy and not speculative. Else no commit occurs in the clock cycle. These procedures help maintain the in-order commit of the instructions despite the presence of out-of-order execution in the superscalar pipeline.
The Flush bits and the Speculation Bits for the instructions are also sent to the reservation stations so that the stations can stop the instructions from executing incase they are speculative and are dependent on a branch. This is done to make our implementation easier as then the forwarding slots from the execution units need not check for an instruction getting flushed or not. This however makes our implementation slow whenever a branch comes in the code as the branch needs to be evaluated in order.
The ROB also sends a FLUSH bit for each instruction getting committed in a single cycle to the Dispatch stage. The Dispatch stage uses these bits for checking whether the committed instruction has to be flushed and its rename register freed or vice versa.
Whenever ROB encounters a Branch Execution Valid signal, it passes this along to the Fetch, Decode and the Dispatch units so that the instructions not yet registered in the ROB can be invalidated and the Fetch unit can start fetching instructions from the correct PC location in case the Branch was mispredicted. If it was not mispredicted, then the Fetch Unit simply keeps on fetching instructions normally(two at a time).

Completion Stage

The Completion Stage is mainly concerned with register or memory writeback. Register writeback occurs through ROB in the Register Files stored at the Dispatch Stage. Memory writeback occurs using the store buffer via a store commit signal sent by the Reorder Buffer upon the commit of a memory store instruction.

Register Writeback

Whenever a register writeback needs to occur, the Architectural and the Rename Register Tags are sent by the ROB to the Dispatch stage along with a signal indicating writeback needs to occur. This Rename Register Tag is then compared with the existing assigned register tags in the Rename Register File and the data is updated for that particular Register. If the Rename Register Tag is void, then it implies that the Architectural Tag itself was used for the destination register. The updates are made accordingly. If there are no pending assignments for a particular architectural register in the Rename Register File, then the corresponding architectural register(identified by the tag sent by ROB) is overwritten with the executed instruction output.

Memory Writeback

Whenever a memory writeback needs to occur, a store commit signal is enabled by the ROB and the Store Buffer location for that particular instruction(obtained from the memory unit itself upon execution) is sent to the Store Buffer wherein the data and the memory address stored at the sent location is written back to the memory. This data and memory address was stored into the Store Buffer by the Memory Unit upon successful execution. It was just waiting there for a successful commit to occur in the ROB.

Simulation

A sample simulation is shown for a simple program using load word, store word and simple ALU instructions. The simulation for Branch and FPU instructions are shown in the submitted videos.
The code ran is shown in the below listing:

```
add R1, R2, R3
lw R0, R4, 5
addi R3, R2, 30
and R1, R2, R4
xor R0, R4, R5
sw R0, R1, 38
```

The value stored in R0 is 0, R1 is X"3F800000", R2 is X"40000000", R3 is X"40400000", R4 is X"40800000", and R5 is X"40A00000". The simulated screen shots are shown in Figures 1 and 2.

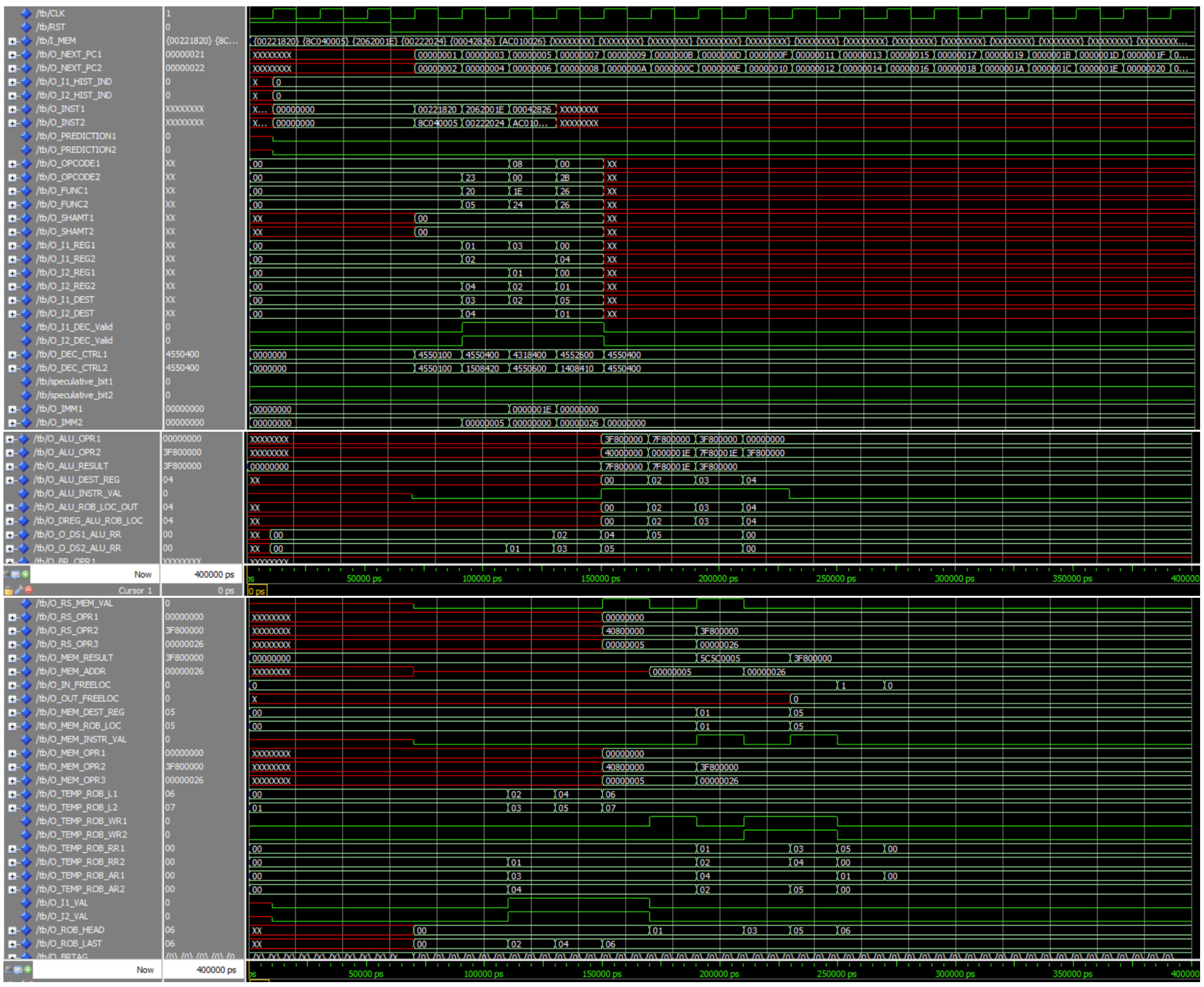

Figure 1: Simulation Results

We can observe that the instructions are correctly fetched two at a time and a fed to the decode block. There, the instructions are getting decoded two at a time and are sent to the Dispatch stage. Then the instructions are going to the respective reservation stations and a executing out of order as evident from the ALU and the load instructions getting executed together. The instructions are committed in the ROB which increases its head pointer whenever an instruction gets committed and increases its last pointer whenever a new instruction is registered into the ROB from the Dispatch stage. We can see that the register writeback happens at a one cycle delay from the ROB actually committing the instruction. This happens because of the register writeback actually occurring in the Dispatch stage. The Store word instruction upon committing enables the STORE COMMIT signal and also provides the corresponding location in the Store Buffer. This location is used to send the write enable signal, the data and the memory address to the written to the memory from the Store Buffer the next cycle. These signals are used by the Memory Unit for finally executing the Memory Writeback in order.

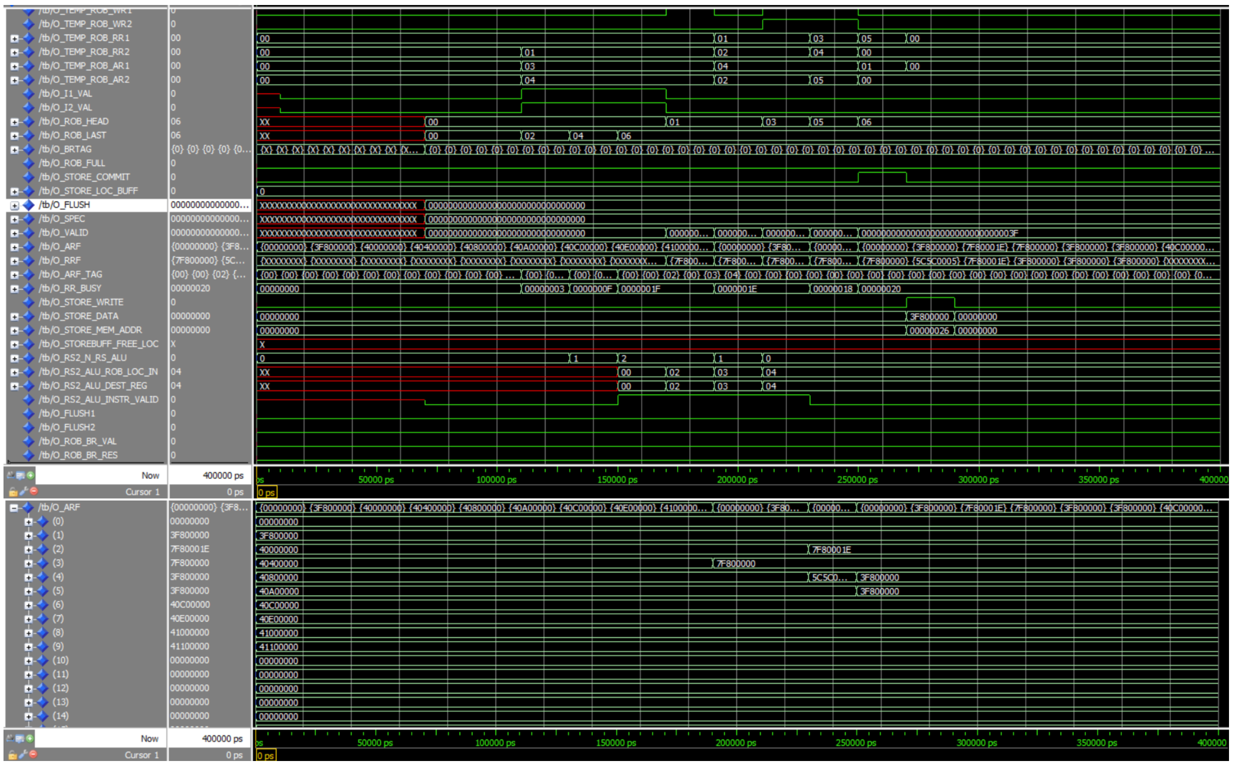

Figure 2: Simulation Results Continued

Limitations and Future aims

- Whenever a branch instruction occurs in the code, we have to wait for it to get executed first before dispatching the other instructions. This is done because if it was mispredicted, then we would have needed to forward the correct value of the registers to the reservation stations. Since the reservation station cannot access the Register file which has limited number of ports, this could not be done and therefore, we would need to wait till the speculation bits of the instructions currently waiting in the reservation stations are disabled. This decreases the parallelizability of the superscalar architecture
- In our implementation, all of our book keeping is done by the ROB itself. Therefore, there is a single point of failure in our implementation. This can be negated by distributing the work of the ROB to check for instructions to be flushed and to be speculated to the other stages as well. This will require better instruction predictors.