A 2-way Out-of-order Issue Superscalar RISC-V Pipeline With An Issue Queue Of Size 32

Team
SuperScary

Design Group
Yuxuan Zhang (yz2580) - Master
Sen Lin (sl3773)

Verification Group
Jonathan Chang (jc4090) - Master
Jie-Gang Kuang (jk3735)

I. Design Overview

In this project, we are going to design a superscalar out-of-order RISC-V pipeline back-end which is capable of issuing two instructions in one single cycle. The proposed pipeline micro-architecture comprises four stages: enqueue (EQ), execute (EX), memory access (MEM), and write-back (WB). For the overall goal, two input 32-bit RISC-V instructions will be enqueued into a 32-entry issue queue (and into a load/store queue (LSQ) when there are load/store instruction); for the output, two of the completed instructions will be committed and the results are written back to the Register Files.

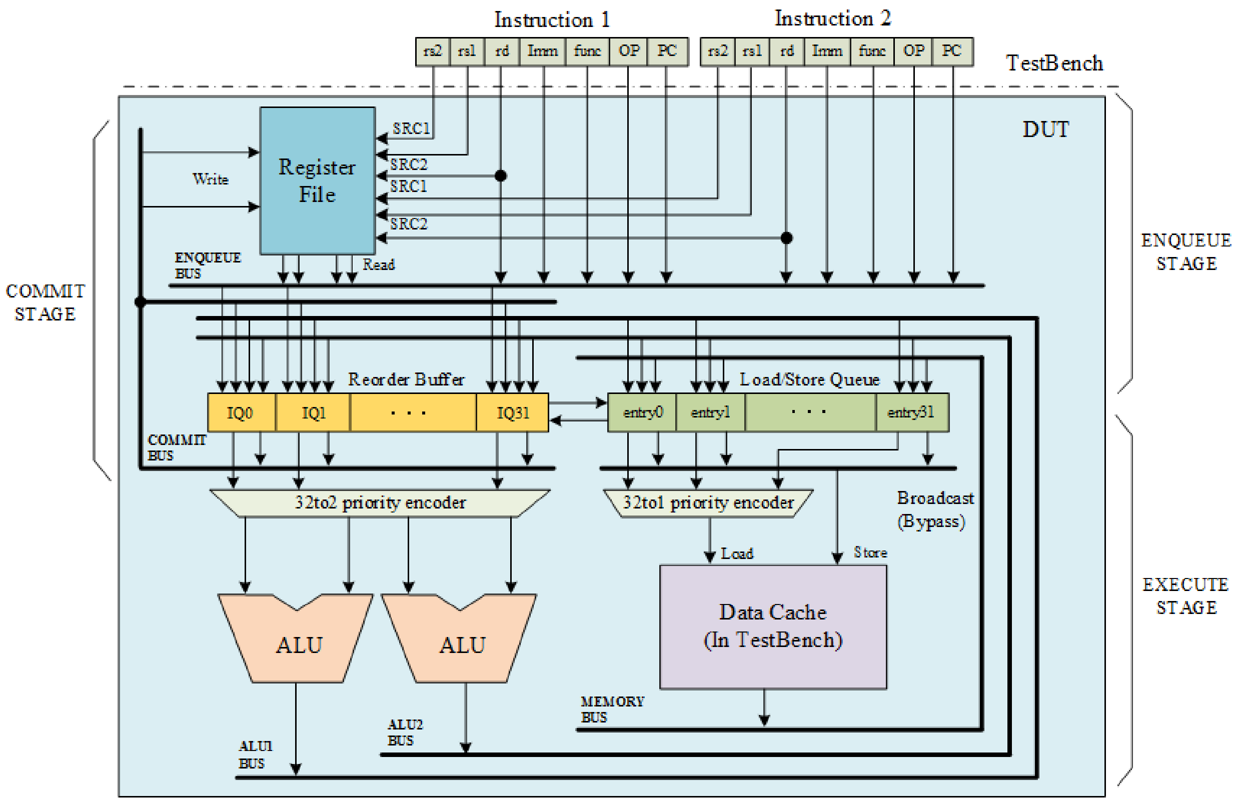

Figure 1. Overview of the system

1. Stage Description

There are four pipeline stages in our design. In this section, we briefly describe those stages proposed in our micro-architecture design.

A. Enqueue (EQ)

In our proposal, a 32-entry queue is designed to serve at the same time as an Issuing Queue (IQ) and a Reorder Buffer (ROB), named simply as ROB. At the enqueue stage, we are going to use two pointers, head and tail, always pointing to the oldest and youngest ROB entry respectively. The entire ROB is a ring buffer. It is full when both the head point and tail point to the same place and “valid” bit is asserted to “1”. It is empty when both the head point and tail point to the same place and “valid” bit is deasserted to “0”. We always want to issue the oldest and non-dependent instructions, indicating by the head pointer. If an input instruction is not of memory access and the ROB is not full, the instruction will be inserted into the tail of the ROB from Register File (which mean these instructions are the youngest); if the ROB is full, a stall signal will be set and no more instruction will be enqueued. If the instruction is of load/store category and LSQ is not full, additional to enqueueing to the ROB, it will also be enqueued to a 32-entry load/store queue (LSQ) from Register File for load/store violation detection. Also, the allocated entry in the LSQ will be recorded in the ROB.

In the process of the data and status bits being passed from Register File to ROB (process 1 and 2 defined in Figure 2), we copy the “data” and “valid” bit, defined in Figure 2, from the corresponding registers’ “data” and “valid” bit for the left and right operands of an instruction, and set the valid bit of the target register to “0” Register File.

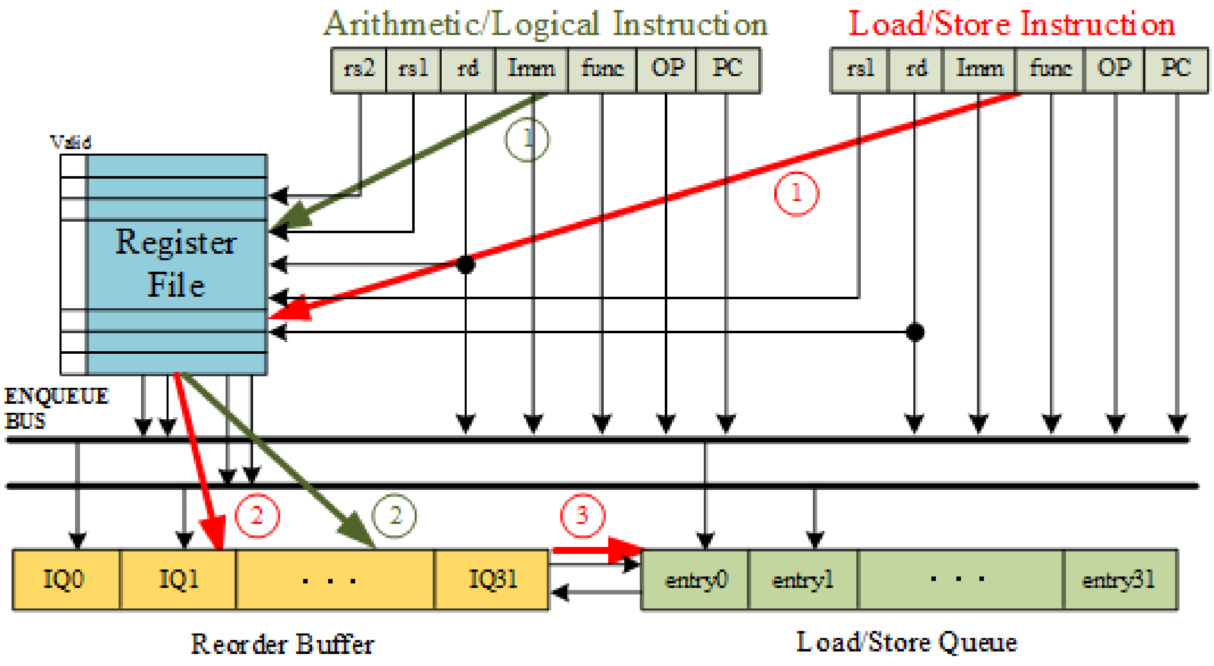

Figure 2. Instruction flow for enqueue stage

C. Execute (EX)

At the execute stage, there are two kinds of executions: ALU instructions and memory instructions. Both kinds of executions will be picked once their operands are all validated.

If a memory-access instruction is picked, the actual address will be calculated at this stage and the result will be updated in the instruction and reported to the LSQ to update its information. After the actual address is updated in the instruction, it will then be executed at the next stage, memory access, described in the following section.

For arithmetic instructions, once the two operands of an instruction are both valid, it can be sent to ALU to execute, and the result will be put on the ALU bus with the target register ID. As well, these information will be forwarded back to reorder buffer. At the same, we want to broadcast the register name just after execution and check if any instructions operands in the ROB will depend on this register and need the data. If there exists some dependence, then the value of this register will be saved into the ARGx and its “valid” bit will be set to “1”.

D. Memory Access (MEM)

When an instruction is of memory access, the memory address to be accessed has to be resolved at the execute stage. After the address is resolved, the memory-access instruction is issued. At this stage, the address will be passed to the LSQ to check whether the memory access violates the memory load/store execution order. A load/store execution order violation induces a kind of data hazards which will be further discussed in the section of I.2.A. Since it is more difficult to undo change in memory, so the store instruction only can write back to memory after committed (make sure there is no address dependency.)

The testbench environment will provides our design a memory. We also assume this memory has only one read and one write port, meaning that every clock cycle there is at most one single type of memory-access instruction to be issued, assuming that “load” and “store” can be issued together if there is no register dependence. This structure hazard can be partially resolved by instruction selection mechanism. Those that cannot be parallelized have to be executed serially. Also, when executing a load instruction, we want to check both LSQ and memory to figure out if we have chances to forward a data from LSQ. The forwarding bus between MEM and ROB servers as a MEM to EX forwarding strategy.

E. Write-back (WB)

When the execute results of ALU/Load instructions are completed, the results are written to the corresponding instructions in the ROB, and the completed instructions will be then committed (written back) to the Register File. Assume the Register File has four reading ports and 2 writing ports.

2. Design Challenges

During designing the overall issuing pipeline, there are some issues which need to be considered. In this section, we discuss two major anticipated challenges in instruction issuing pipeline design: a type of data hazard caused by memory access and a control hazard primarily induced by instructions of branch category.

A. Register Dependency - Forwarding

In pipelining, one kind of data hazard is caused by register dependence. Briefly speaking, this type of hazard occurs when the operands of to-do instructions are the results of the precedent instructions which have not been completed yet.

In the backend of the pipeline, we do not have to worry about the WAW and WAR data hazards, since Register Renaming already helped rename the dependent registers. So we only need to handle RAW hazards. We optimize our design with forwarding, which enables the data after execution to be transferred directly to ROB for the incoming instructions. Also, a load data in the LSQ can be forwarded to ROB, which resolves the register dependence between a pair of load and ALU instructions

B. Memory Access Dependency - Load/Store Queue

Another typical type of data hazard is induced by memory access disorder which violates memory data dependency. The necessary condition to this kind of issue is accessing the same address. Basically, there could be four combinations:

1. *load-after-load*: There will be no conflict between instructions no matter which one is older.
2. *load-after-store*: In the case when these memory-access instructions are executed in the program order. If the young store has same address as the later load, we could bypass the data of the store to the data of the load directly in LSQ. However, it is possible that the address of the store is not resolved after the finish of the load, meaning that the load is executed speculatively out of the program order, shown as the example below. In this case, we can issue "add" and "load" in parallel, speculating "R9" is not equal to "R1 + Imm". The value bits of "R10" in LSQ remains not committed until we know the address of "store", "R1+Imm". If they are the same, LSQ will flag a violation.
   ```
   add R1, R2, R3
   store [R1+ Imm], R7
   load [R9], R10
   ```
3. *store-after-store*: In the case of execution these instructions out of the program order, a violation could be happening if the two stores has the same memory address. These memory-access instructions have to be committed in the program order, which is ensured by the mechanism of the commit of a store instruction in LSQ. Then the store is ready to be copied into memory.
4. *store-after-load*: They should be committed in the program order. If we find the two instructions dealing with the same address and are executed out of order, that means that the load instruction could not load the correct data since the store instruction has changed the memory value. In this case, there will report a violation. Fortunately, the commit mechanism of LSQ prevents this happening.

A load/store queue (LSQ) is proposed to specifically handle the dependences and orders between memory instructions. As the description above, there is no violation of load-after-load and the commit mechanism ensure that there is no store-after-store violation.

To deal with load-after-store violation, when a store instruction is ready to access memory, we need to check all the younger instructions and their addresses. If there are younger loads that have been done, we have to invalidate the instruction on-flight in the pipeline after the store and also flush the violated load data in LSQ. Then the pipeline will reverse PC and execute the load instruction again after the store commits to memory. Besides, to increase throughput of load-after-store, when a load instruction is ready to access memory, we enable LSQ to check whether there is an older store whose address is the same as the load. If there are, then we will bypass the data from the store to the load and assert the done bit of the load to “1”.

Since the instructions should always commit in the program order, in store-after-load case, the store instruction will be written into memory only if all the older loads to the same address are all executed/committed (already read from memory). Consider this commit mechanism, there will not be a store-after-load violation.

C. Branch - Queue Flushing

For branch instructions, the testbench environment acts as a front-end branch prediction process. We simply assume branch will always predict the next PC after the branch instruction. This simplification may lead to branch misprediction and we cannot know the correctness of the prediction until the EX stage of branch instruction. The branch misprediction is detected by resolving the branch execution result, obtaining the branch is taken or not, and then comparing the predicted PC with the correct PC from the result of branch execution. If there is no misprediction, everything will be fine and pipeline continues working.

However, if the next PC as a result from execution is different from the predicted PC, but the predicted instructions have been enqueued, then these instructions have to be flushed from pipeline stages and the ROB. The correct next PC will be given back to the testbench. In this case, the testbench should pass the correct PC and then the instructions corresponding to the correct PC after branch will be enqueued and continue the reset of the stages. In the 2-way design, it is also possible to issue another instruction together with a branch instruction. The way to resolve branch misprediction is the same. Notice that we should not write back any data or access memory which is computed after the branch instruction until the correctness of the branch instruction is confirmed.

Additionally, in the special cases when a memory-access instruction under prediction is issued in parallel with a branch instruction or after a branch instruction, since the correct next PC of branch is resolved after execution stage, the mispredicted memory-access instructions also need to be flushed out of LSQ.

II. Unit Level Interfaces

In this section, we identify the unit-level components of the proposed design and specify interfaces of each units. As demonstrated in *Figure 1.*, our design are primarily composed by units of Instruction Decoder, Register File, ROB, LSQ and Function Unit. *Figure 3.* shows the the connection of datapath and control signals of all units.

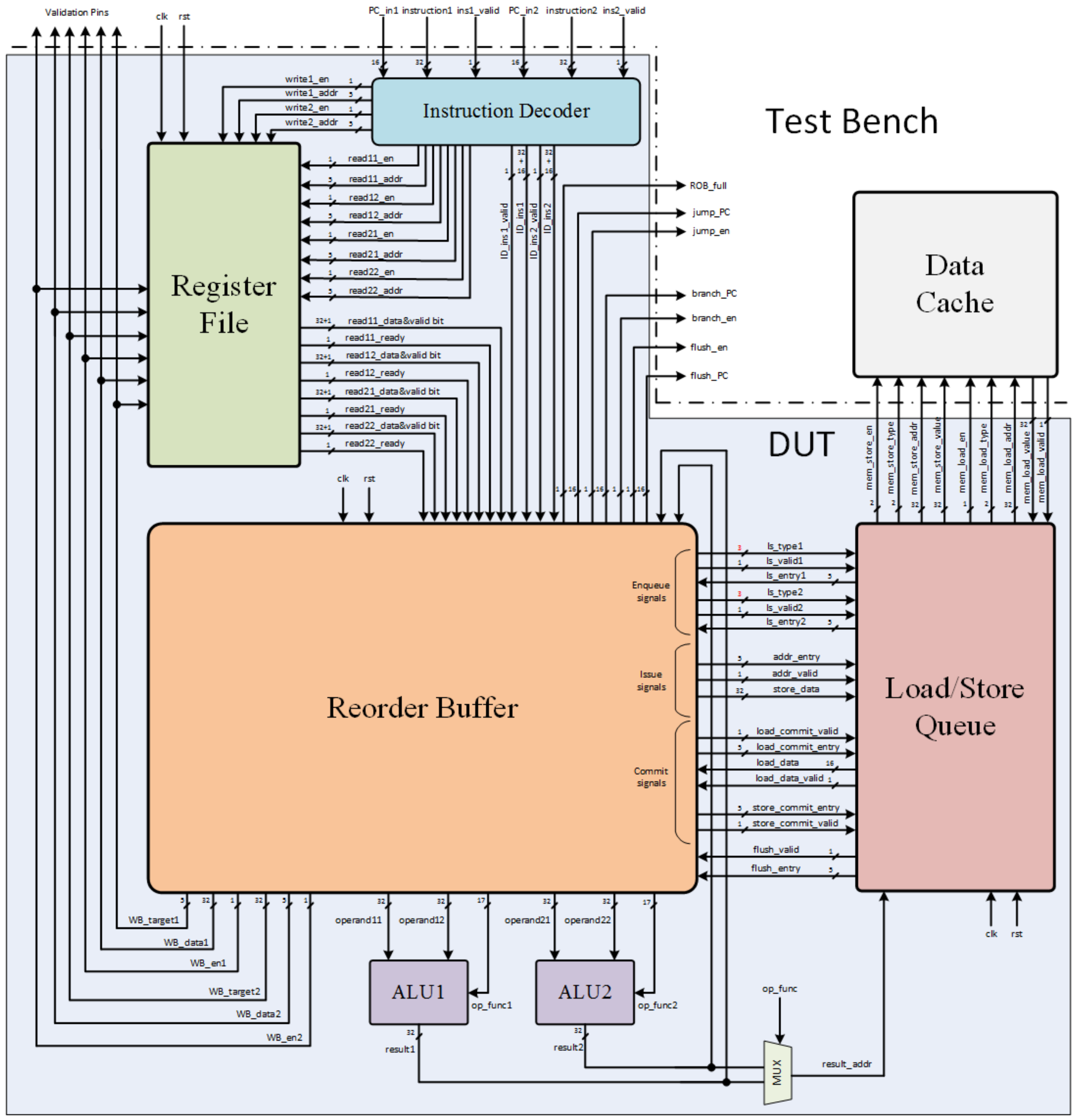

Figure 3: DUT Top-level diagram

1. Unit Level Interfaces

We specify the signal interfaces of each unit in the subsection. There are four units in our design, including Register Files, ROB, LSQ and ALU.

A. Instruction Decoder

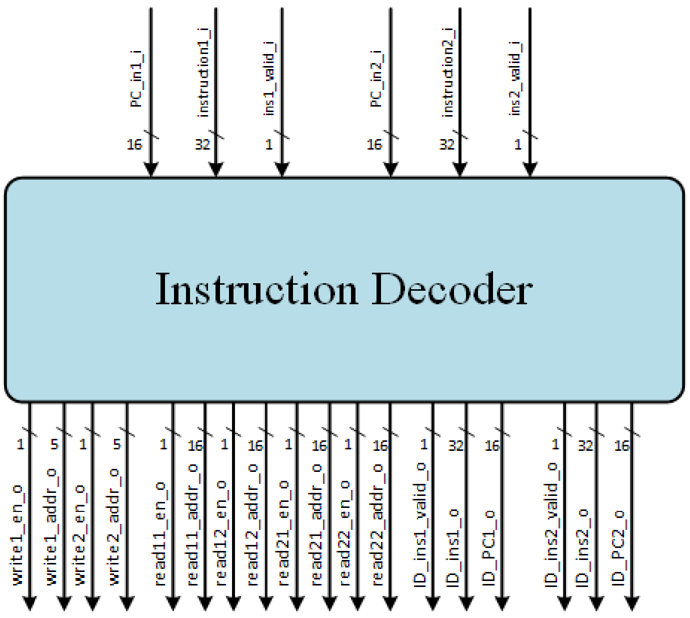

Figure 4: Instruction Decoder interface

Signal	Bit Width	Description
PC_in1_i	16	PC of instruction 1
instruction1_i	32	all input bits of instruction 1 from TB
ins1_valid_i	1	validity of instruction 1 as an input from TB
PC_in2_i	16	PC of instruction 2
instruction2_i	32	all input bits of instruction 2 from TB
ins2_valid_i	1	validity of instruction 2 as an input from TB
read11_en_o	1	read enable of left source operand of instruction 1
read11_addr_o	16	register name of left source operand of instruction 1

read12_en_o	1	read enable of right source operand of instruction 1
read12_addr_o	16	register name of right source operand of instruction 1
read11_en_o	1	read enable of left source operand of instruction 2
read11_addr_o	16	register name of left source operand of instruction 2
read12_en_o	1	read enable of right source operand of instruction 2
read12_addr_o	16	register name of right source operand of instruction 2
write1_en_o	1	enable signal to update “valid” bit in “valid” FFs by the target operand of instruction 1
write1_addr_o	5	register name of target register of instruction 1 to update the “valid” bit in “valid” FFs
write2_en_o	1	enable signal to update “valid” bit in “valid” FFs by the target operand of instruction 2
write2_addr_o	5	register name of target register of instruction 2 to update the “valid” bit in “valid” FFs
ins1_valid_o	1	validity of instruction 1
ins1_o	32	32-bit fields instruction 1
ins2_valid_o	1	validity of instruction 2
ins2_o	32	32-bit fields instruction 2
ID_PC1_o		
ID_PC2_o		
op1_i	7	opcode of instruction 1
op2_i	7	opcode of instruction 2
imm1_i	12	immediate of instruction 1
imm2_i	12	immediate of instruction 2
r11_name_o	5	register name of left operand of instruction 1
r12_name_o	5	register name of right operand of instruction 1
r21_name_o	5	register name of left operand of instruction 2

r22_name_o	5	register name of right operand of instruction 2
tar1_name_o	5	register name of target operand of instruction 1
tar2_name_o	5	register name of target operand of instruction 2
func1_o	10	function bits of instruction 1
func2_o	10	function bits of instruction 2

B. Register File Interfaces

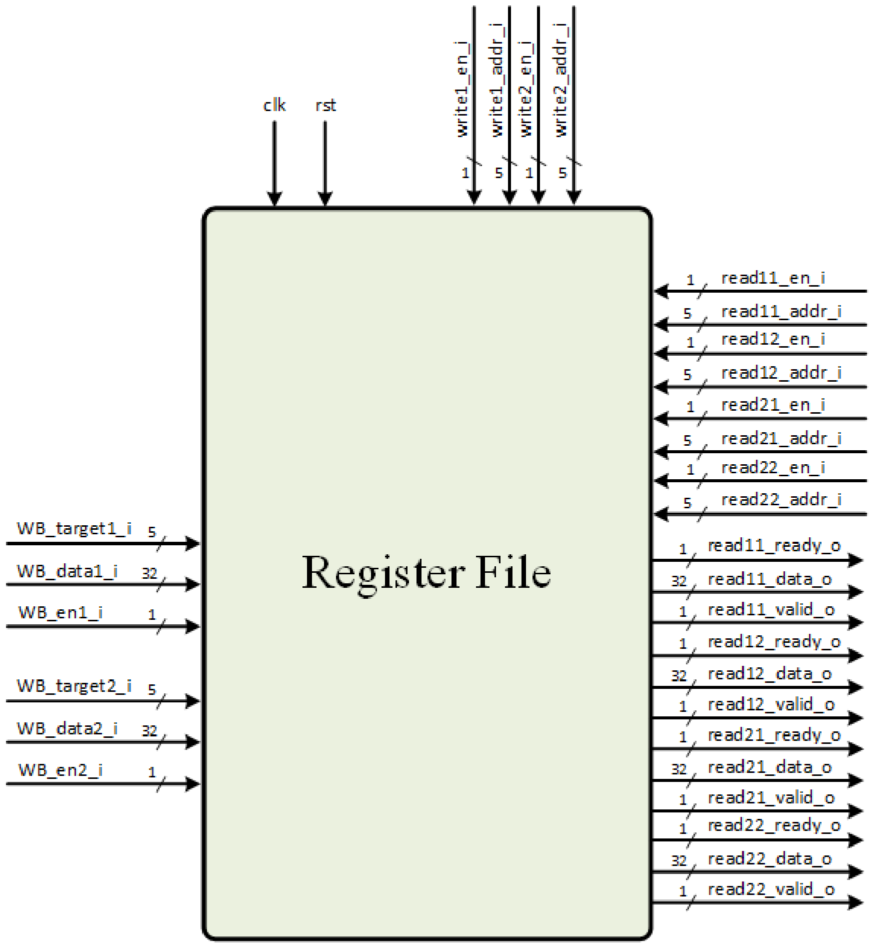

Figure 5: Register File interface

Signal	Bit Width	Description
clk	1	clock signal
rst	1	synchronized reset with active high
read11_en_i	1	read enable of left source operand of instruction 1
read11_addr_i	5	register name of left source operand of instruction 1
read12_en_i	1	read enable of right source operand of instruction 1
read12_addr_i	5	register name of right source operand of instruction 1
read21_en_i	1	read enable of left source operand of instruction 2
read21_addr_i	5	register name of left source operand of instruction 2
read22_en_i	1	read enable of right source operand of instruction 2
read22_addr_i	5	register name of right source operand of instruction 2
read11_data_o	16	data of left source operand of instruction 1
read11_valid_o	1	read “valid” of left source operand of instruction 1
read11_ready_o	1	read signal ready of left source operand of instruction 1
read12_data_o	16	data of right source operand instruction 1
read12_valid_o	1	read “valid” of right source operand of instruction 1
read12_ready_o	1	read signal ready of right source operand of instruction 1
read21_data_o	16	data of left source operand of instruction 2
read21_valid_o	1	read “valid” of left source operand of instruction 2
read21_ready_o	1	read signal ready of reft source operand of instruction 2
read22_data_o	16	data of right source operand instruction 2
read22_valid_o	1	read “valid” of right source operand of instruction 2
read22_ready_o	1	read signal ready of right source operand of instruction 2
wb_en1_i	1	write enable of target operand of instruction 1
wb_target1_i	5	register name of target operand of instruction 1

wb_data1_i	16	write data of target operand of instruction1
wb_en2_i	1	write enable of target operand of instruction 2
wb_target2_i	5	register name of target operand of instruction 2
wb_data2_i	16	write data of target operand of instruction 2
write1_en_i	1	enable signal to update “valid” bit in “valid” FFs by the target operand of instruction 1
write1_addr_i	5	register name of target register of instruction 1 to update the “valid” bit in “valid” FFs
write2_en_i	1	enable signal to update “valid” bit in “valid” FFs by the target operand of instruction 2
write2_addr_i	5	register name of target register of instruction 2 to update the “valid” bit in “valid” FFs

C. ROB Interfaces

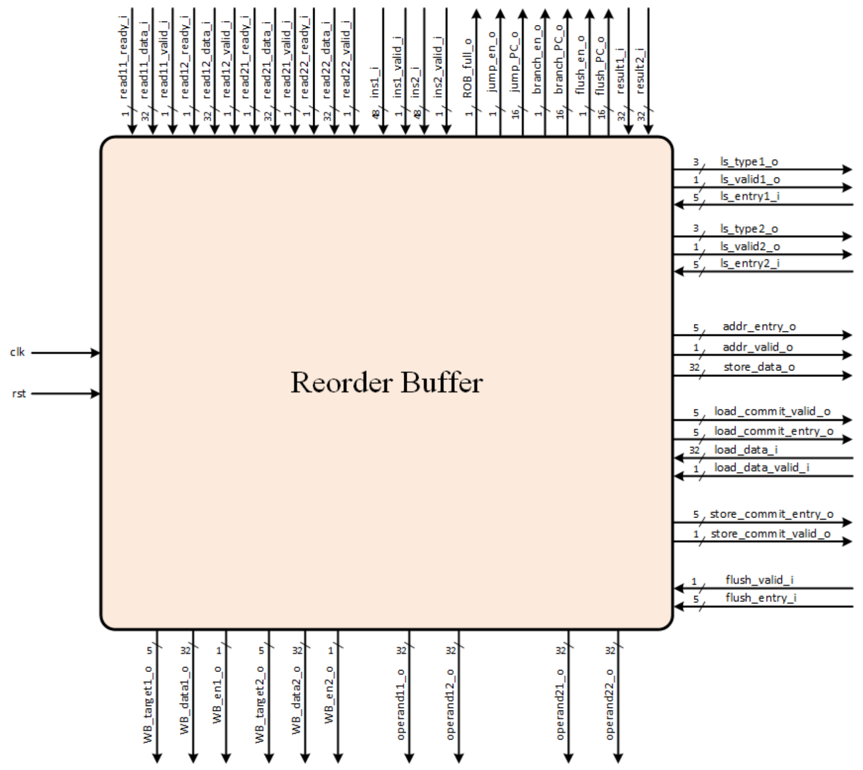

Figure 6: Reorder Buffer interface

Sigal	Bit Width	Description
clk	1	clock signal
rst	1	synchronized reset with active high
ins_valid1_o	1	validity of instruction 1
ins1_o	32	32-bit fields instruction 1
ins_valid2_o	1	validity of instruction 2
ins2_o	32	32-bit fields instruction 2
pc1_i	16	PC of instruction 1
pc2_i	16	PC of instruction 2
op1_i	7	opcode of instruction 1
op2_i	7	opcode of instruction 2
imm1_i	12	immediate value of instruction 1
imm2_i	12	immediate value of instruction 2
r11_name_i	5	register name of left operand of instruction 1
r12_name_i	5	register name of right operand of instruction 1
r21_name_i	5	register name of left operand of instruction 2
r22_name_i	5	register name of right operand of instruction 2
tar1_name_i	5	register name of target operand of instruction 1
tar2_name_i	5	register name of target operand of instruction 2
func1_i	10	function bits of instruction 1
func2_i	10	function bits of instruction 2
read11_data_i	16	read data of left operand of instruction 1
read11_valid_i	1	read “valid” of left operand of instruction 1
read11_ready_o	1	read signal ready of left source operand of instruction 1

read12_data_i	16	read data of right operand of instruction 1
read12_valid_i	1	read “valid” of right operand of instruction 1
read12_ready_o	1	read signal ready of right source operand of instruction 1
read21_data_i	16	read data of left operand of instruction 2
read21_valid_i	1	read “valid” of left operand of instruction 2
read21_ready_o	1	read signal ready of left source operand of instruction 2
read22_data_i	16	read data of right operand of instruction 2
read22_valid_i	1	read “valid” of right operand of instruction 2
read22_ready_o	1	read signal ready of right source operand of instruction 2
WB_en1_o	1	write-back data valid of target operand of instruction 1
WB_target1_o	5	write-back register name of target operand of instruction 1
WB_data1_o	16	write-back data of target operand of instruction 1
WB_en2_o	1	write-back data valid of target operand of instruction 2
WB_target2_o	5	write-back register name of target operand of instruction 2
WB_data2_o	16	write-back data of target operand of instruction 2
operand11_o	16	first operand data to ALU 1
operand12_o	16	second operand data to ALU 2
op_func1_o	17	op(7-bit) and function(10-bit) applied to ALU 1
result1_i	16	data/address/next PC after computation of ALU 1
operand21_o	16	first operand data to ALU 2
operand22_o	16	second operand data to ALU 2
op_func2_o	17	op(7-bit) and function(10-bit) applied to ALU 2
result2_i	16	data/address/next PC after computation of ALU 2
ls_type1_o	1	indication if the instruction 1 is load or store
ls_valid1_o	1	validity of type bit for instruction 1

ls_entry1_i	5	entry index of instruction 1 in LSQ
ls_type2_o	1	indication if the instruction 2 is load or store
ls_valid2_o	1	validity of type bit for instruction 2
ls_entry2_i	5	entry index of instruction 2 in LSQ
addr1_entry_o	5	entry index to instruction 1 whose address is returned to ROB
addr1_valid_o	1	validity of address returned to instruction 1
addr2_entry_o	5	entry index to instruction 2 whose address is returned to ROB
addr2_valid_o	1	validity of address returned to instruction 2
load_commit1_valid_o	1	enable signal to LSQ notifying load instruction 1 to commit
load_commit1_entry_o	5	entry index of load instruction 1 in LSQ whose target operand is transferred to ROB when commit
load_data1_i	16	target operand data of load instruction 1 which is committed
load_data1_valid_i	1	valid signal notifying ROB of incoming load data of load instruction 1
load_commit2_entry_o	5	entry index of load instruction 2 in LSQ whose target operand is transferred to ROB when commit
load_data2_i	16	target operand data of load instruction 2 which is committed
load_data2_valid_i	1	valid signal notifying ROB of incoming load data of load instruction 2
store_data_o	16	source operand data of a store instruction which is committed
store_commit_entry_o	5	entry index of a store instruction in LSQ which is committed
store_commit_valid_o	1	validity of the operand data and entry index of a store instruction which is committed

flush_i	1	flush signal caused by load-after-store violation
branch_PC_o	16	correct next PC after a branch instruction execution
branch_en_o	1	valid signal notifying TB the incoming correct next PC
jump_PC_o	16	informing TB next PC after execution jump instruction
jump_en_o	1	valid signal notifying TB the incoming next PC after jump
rob_full_o	1	backpressure to TB
flush_PC_o	16	informing TB next input PC after flush
flush_en_o	1	valid signal notifying TB the return PC after violation

D. LSQ Interfaces

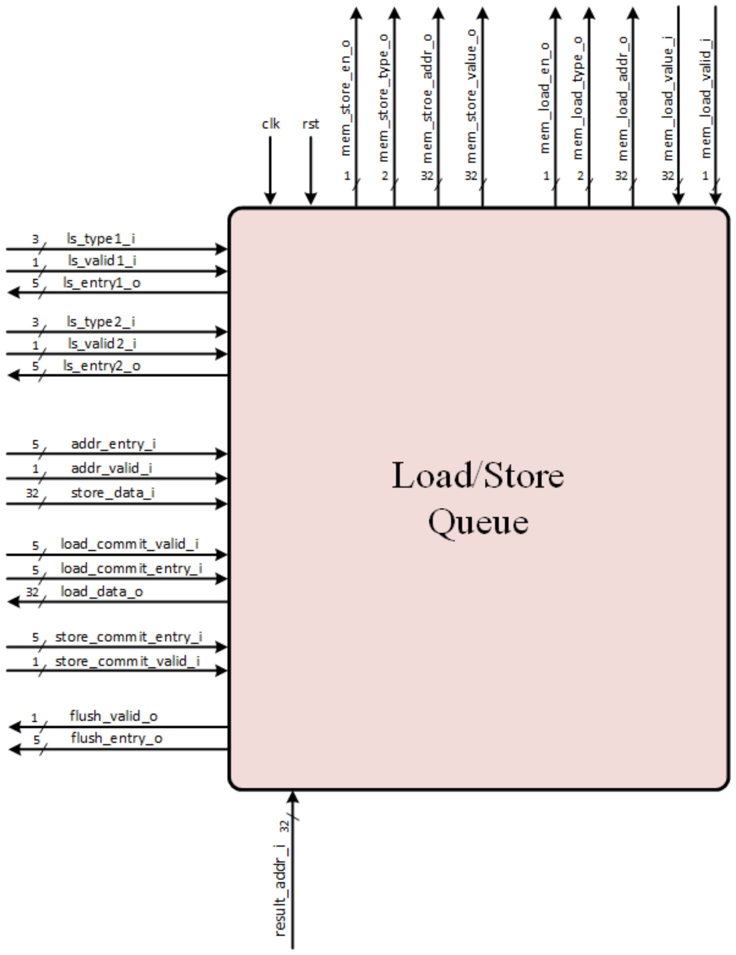

Figure 7: Load/Store Queue interface

Signal	Bit Width	Description
clk	1	clock signal
rst	1	synchronized reset with active high
result1_i	16	executed address value returned to instruction 1 in LSQ
result2_i	16	executed address value returned to instruction 2 in LSQ
ls_type1_i	1	indication if instruction 1 is load or store
ls_valid1_i	1	validity of type bit for instruction 1
ls_entry1_o	5	entry index of instruction 1 in LSQ
ls_type2_i	1	indication if instruction 2 is load or store
ls_valid2_i	1	validity of type bit for instruction 2
ls_entry2_o	5	entry index of instruction 2 in LSQ
addr1_entry_i	5	entry index to instruction 1 whose address is returned
addr1_valid_i	1	validity of address returned to instruction 1
addr2_entry_i	5	entry index to instruction 2 whose address is returned
addr2_valid_i	1	validity of address returned to instruction 2
load_commit1_valid_i	1	enable signal to LSQ notifying instruction 1 to commit
load_commit1_entry_i	5	entry index of load instruction 1 in LSQ whose target operand is transferred to ROB when commit
load_data1_o	16	target operand data of load instruction 1 which is committed
load_data1_valid_o	1	valid signal notifying ROB of incoming load data of load instruction 1
load_commit2_valid_i	1	enable signal to LSQ notifying instruction 2 to commit
load_commit2_entry_i	5	entry index of load instruction 2 in LSQ whose target operand is transferred to ROB when commit
load_data2_o	16	target operand data of load instruction 2 which is

		committed
load_data2_valid_o	1	valid signal notifying ROB of incoming load data of load instruction 2
store_data_i	16	source operand data of a store instruction which is committed
store_data_entry_i	5	entry index of a store instruction in LSQ which is committed
store_data_valid_i	1	validity of the operand data and entry index of a store instruction which is committed
flush_o	1	flush signal caused by load-after-store violation
mem_signal_o	1	indication of load or store of an instruction to data cache when commit
mem_addr_o	16	address to access for a load or store instruction
mem_store_value_o	16	data commited into memory
mem_load_value_i	16	data in memory sent to LSQ

E. ALU interfaces

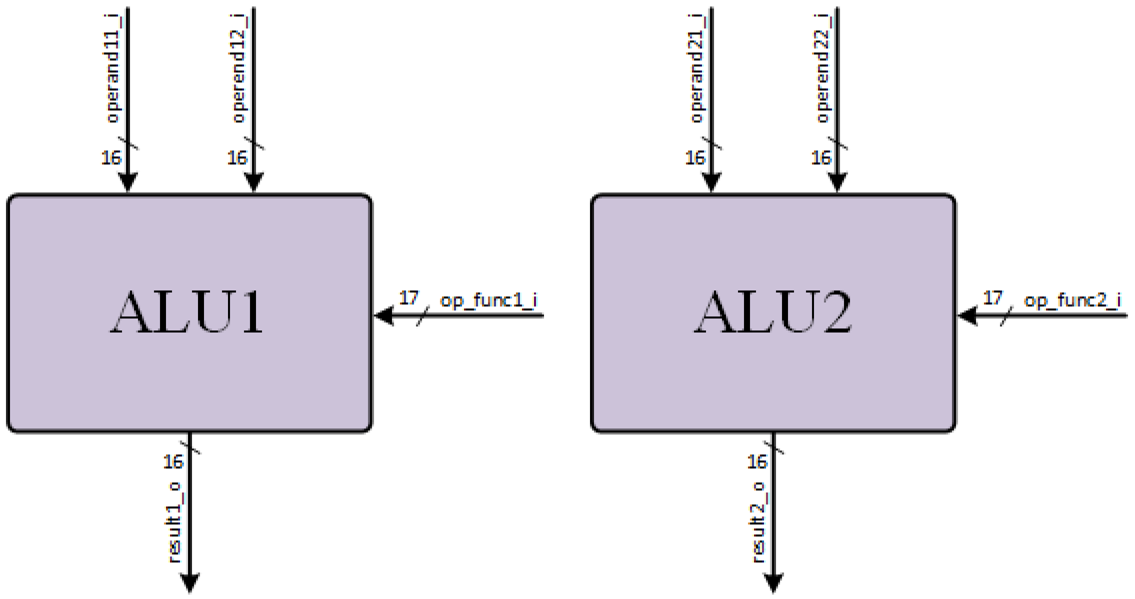

Figure 8: ALUs interface

Signal	Bit Width	Description
operand11(12)_i	16	inputs (register data or immediate) to ALU1
operand21(22)_i	16	inputs (register data or immediate) to ALU2
op_func1_i	17	opcode and function applied to ALU1
op_func2_i	17	opcode and function applied to ALU2
result1_o	16	operation result of ALU1
result2_o	16	operation result of ALU2

2. Unit-level Communications

In this subsection, the interactions among each unit in different stages are illustrated based on the signals defined in the previous subsection.

A. Instruction Decoder, Register File, ROB and LSQ Communications at Enqueue Stage

When there are slots available in ROB, one or two instructions are ready to be enqueued into ROB. The incoming instructions with an asserted valid bit each are decoded in the Instructions Decoder. The decoded PCs, opcodes, immediates, source operand register names and possible target register names are directly copied into the respective fields in the ROB. Other parts of signals after the decoder are sent to Register File, including necessary inputs to read data from Register File, such as read enable and read address. The register name of each operand of an instruction serves as input to Register File with an "enable" signal for each register name, indicating the address to index the register that needs to be read is valid. At the same clocking cycle, the value corresponding to that address is returned, with a valid bit notifying ROB to receive the data. There are at most 4 operands that needs the data.

Also, target register name of an instruction is sent to Register File's "valid" FFs to update the dependencies between the target operand register and source operand registers of the upcoming instructions. Initially, all of the valid bits in valid FFs of Register File corresponding to each register are all "1", indicating there is no data dependencies. To ensure the data dependencies among instructions, when we want to enqueue an instruction, the decoder also inputs the target register name to Register File, updating the valid bit to the respective Register FIle into "0", so that the later instruction whose source operands want to read the register will see a dependency to that target operand register. After an source operand reads the valid bit, it will be reflected in valid bit of each operand in ROB. There is a special case to determine the dependencies among the multiple instructions that are sent to ROB at the same time, since "valid" bits are not able to be updated immediately in this case. So we need to add judgement int ROB before instruction enqueue.

Immediately, all instructions are enqueued in ROB, all of the memory-access instructions are copied to LSQ in program order. At this stage, only the type bit needs to be copied to LSQ, and then a 5-bit entry index of LSQ will be returned to ROB. There will be at most two memory-access instructions sent to LSQ.

B. ROB, ALU and LSQ Communications after Execution Stage

After two (at most) instructions are issued by ROB to ALUs, for the arithmetic and logic instructions, the results of target operand are calculated and returned back to ROB. The target register names will be broadcasted to each entry of ROB, seeking for the dependent source operands of the instructions that have not been executed. If there are dependent source operand are found, these executed values are written to the respective ROB fields and valid bits will also be set high. The results are also updated back to the executed instructions, with their "E" and "D" bits asserted.

For the memory-access instructions, signals are complex. an ALU calculates the explicit address of a load or store instruction. The address is written back to ROB "result". It will be transferred immediately to LSQ from ALU. Meanwhile, ROB sends the entry index bits corresponding the memory-access instruction in LSQ and its valid signal. Also, LSQ uses this address to search among addresses of the other instructions in LSQ, looking for a potential load-after-store violation or a load bypass.
Consider a load instruction. If an older store instruction with the same address found in LSQ, the load instruction will get the data of "store" immediately of the same address. When the address value is visible in LSQ the next cycle, the load can communicate with data cache (TB). LSQ initiates the communication by sending 1-bit memory type signal, 16-bit memory address, with a valid bit of the load instruction. Immediately after that, the value corresponding to the memory address is returned to LSQ, with a ready bit notifying LSQ. Then LSQ will set "DV" bit to "1" of the corresponding data. When to commit load instruction, ROB will check the validity of data of corresponding entry from LSQ. If the data is valid, the load could be committed. Otherwise the load could not be committed this cycle.

If the address belongs to an instruction in LSQ which is a store instruction, at the same time when the address value is sent to LSQ, its "E" bit in ROB is asserted high, indicating the store address is ready. When the store data is valid, the store could be committed. ROB will sent store data with store entry to LSQ, then LSQ will store the data into data cash correspond to the address.

III. Subunit Partitioning And Interfaces, Test Harness Structure

In this section, the subunit design partitioning and interfaces are presented in the first subsection. In the following subsection, we will introduce the test harness structure of our testbench.

1. Subunits and Interfaces

A. Register File

Figure 9 demonstrates the subunits of Register File. Since our register file not only store the data but also the "valid" information to determine the data dependencies, the subunits are a 32-entry x 16-bit data FF arrays and a 32-entry x 1-bit "valid" FFs. Their entry indexes are corresponding. The control logic will control the read and write data flows between unit-level interfaces and subunit-level interfaces. But there is no signal connection between the two subunits.

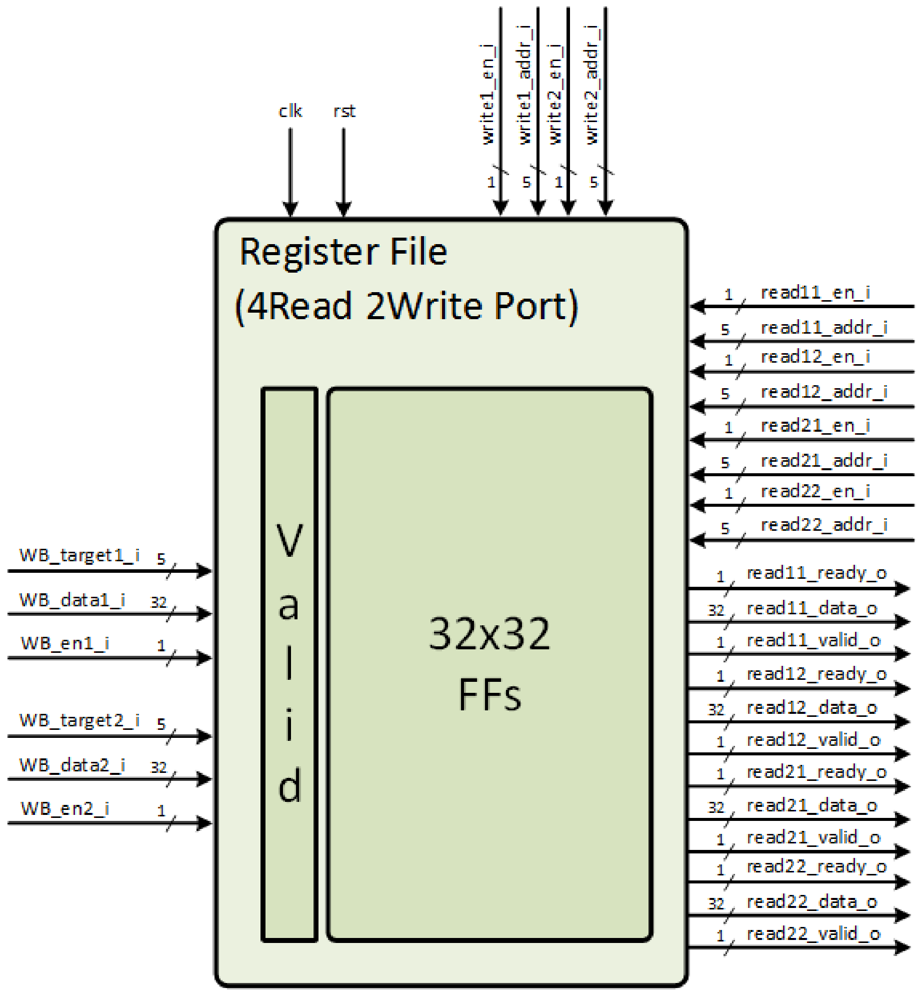

Figure 9: Register File subunit diagram

B. Reorder Buffer

In reorder buffer unit, as shown in Figure 10, we use 32-entry x 120-bit width data FFs and CAM to store all the information from the instruction decoder and Register File: the PC value; opcode, function bits and immediate value in the instruction; the data (ARG0, ARG1) read from the register file and the corresponding register name (src0, src1). If the data is valid, set the valid bits (v0, v1) into "1s", and vise versa. If the instruction is load or store, we will record the entry index where this instruction is located in the LSQ; If the instruction has been issued, the result will be stored into "Result" field and the target register entry name "Target". The "V" bit will be set to "1" if this instruction is valid, which is used when there needs a "flush". The "D" bit represents for a "done" bit, which will be set to "1" when an instruction is ready to commit. The "E" bit determines if an instruction has been executed or not. The "M" bit determines a memory-access instruction.

Since our CAM and FFs are designed as FIFO structure, there are two pointers, head and tail, to record the age of an instruction. The head pointer always points to the oldest instruction. We require the instruction commit always starts from here, which guarantees committing in program order. As well, the tail pointer always points to the next slot of the youngest instruction. So if there are new upcoming instructions, they will be stored into the slots where the tail pointer points and allocated using two 1-to-32 demultiplexers. Also, we use a 5-bit counter to check if the ROB is full or not: simply plus one if there is an incoming instruction, minus one if an instruction has been committed and deallocated. When the counter reaches 32, a "full" signal will be outputted.

To issue two instructions per cycle out of order, we use a 32-to-2 priority encoder to find the first two instructions whose source operands are both valid with the priority from head to tail. After the instructions are issued, their "E" are asserted high, not considered to be issued again.

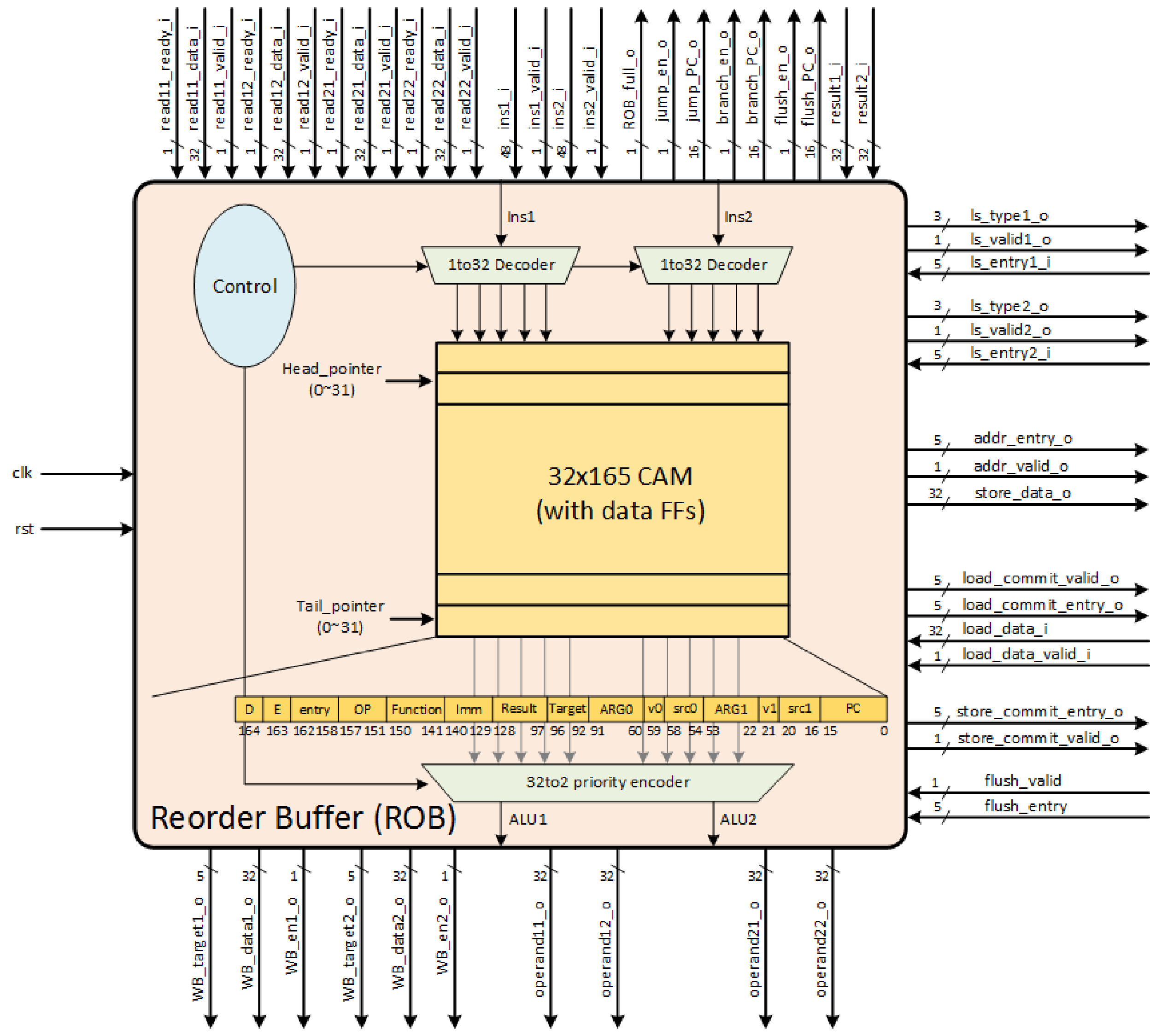

Figure 10: Reorder Buffer subunit diagram

C. Load/Store Queue

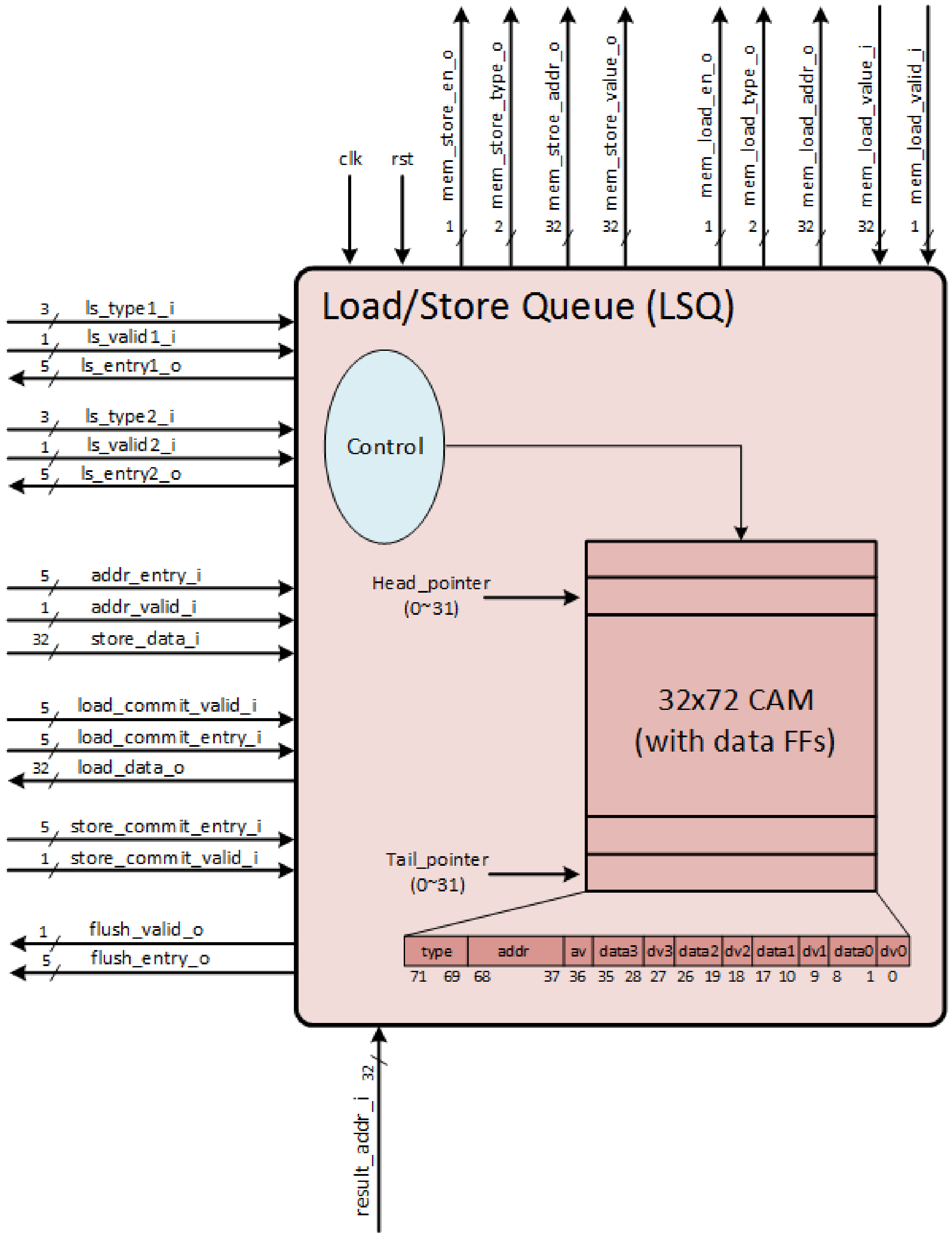

Figure 11: Load/Store Queue subunit diagram

Figure 11 presents the subunit of LSQ. The LSQ is a FIFO-structured CAM and FF arrays. The logic organization of this structure is shown. “B” is a 32-entry x 1-bit CAM, used in a load instruction to indicate a bypass from an older “store” to this load instruction. “T” is a 32-entry x 1-bit FFs to determine a load or store type. “Address” field is a 32-entry x 16-bit CAM which is able to be searched by upcoming executed address. “Av” is a 32-entry x 1-bit FFs to indicate the address it valid. “Data” field is a 32-entry x 16-bit FF array. “Dv” is 32-entry x 1-bit FFs, determining the validity of the data. The data flow communication between LSQ and ROB are controlled by a control logic, which is implemented by lower-level hardware structures.

3. Test Harness Structure

With the top-level interface, the testbench program will be connected with the design under test (DUT). The TB I/O signals are defined in the previous section. The testbench generates the random signal as input for the DUT, collects the outputs of the DUT, and then compares the results with the golden model each clock cycle.

We develop several classes to handle different tasks in the testbench.

A. Environment Class

The environment class is to set up the environmental parameters of the testbench, including cycles, random seeds, signal coverage and probability controls, etc.

B. Transaction Class

The transaction class is designed to randomize the test input signal for the DUT and the golden model. The data generated randomly by the instance of this class will be driven into the DUT and the golden model before the clocking.

C. Golden-model Class

The golden-model class emulates the behaviors of the DUT, compare and check the results with the outputs of the DUT. Before the clocking, the random signals generated by the transaction instance are driven into the input tasks/functions of this class, and compute the anticipated outputs of the DUT. After the clocking, the outputs of the DUT are collected via the top-level interface by the checker tasks/functions to check the results.

D. Memory Class

Since we are not going to actually compile and utilize the memory (data cache). We construct a memory class to approximate the real memory module. To simplify the condition, we define that it always takes one clock cycle to access the simulated memory.

IV. Microarchitecture Design

We demonstrate the interfaces of all the subunits in each unit and all the control logic in detail. We will use tables and pseudo codes to characterize them.

1. Instruction Decoder

Figure 4 shows the unit-level interfaces of Instruction Decoder. Since it is a unit composed only by combinational logic, the interfaces connection within this unit depends on the logic inside. Therefore, we use pseudo codes to describe the logic here. To avoid redundancy, we only shows the logic of one instruction decoding.

```
 assign dec_rob.ins1_valid = ins1_valid_i;
 //write to ROB
assign dec_rob.ins1 = instruction1_i;
assign dec_rob.PC1 = PC1_i;

assign  dec_rf.write1_addr = instruction1_i[11:7];
assign  dec_rf.read11_addr = instruction1_i[19:15];
assign  dec_rf.read12_addr = instruction1_i[24:20];

 case(instruction1_i[OP_WIDTH-1:0])
 R_TYPE_OP: begin
    //read data control and write *valid* control
    dec_rf.write1_en = '1;
    dec_rf.read11_en = '1;
    dec_rf.read12_en = '1;
     end
 I_TYPE_OP: begin
    dec_rf.write1_en = '1;
    dec_rf.read11_en = '1;
    dec_rf.read12_en = '0;
     end
 I_TYPE_OP: begin
    dec_rf.write1_en = '0;
    dec_rf.read11_en = '1;
    dec_rf.read12_en = '1;
     end

 U_TYPE_OP: begin
    dec_rf.write1_en = '1;
```

```
        dec_rf.read11_en = '0;
        dec_rf.read12_en = '0;
         end
     endcase
```

The pseudo code shown above describes different cases of enable or valid signals to ROB, because the different instructions may have different formats.

2. Register File

A. Microarchitecture Specification and Pseudo Code

As shown in Figure 9, there's two subunits in Register File. One is 32 1-bit "valid" bit FFs and the other one is 32 x 32-bit FF arrays used as storage elements. Considering the interfaces we defined the Section II, the "valid" bit FFs are built with 4-read-port and 4-write-port RAMs and the FF arrays are built with 4-read-port and 2-write-port RAMs. The RAM we use here can be partitioned into MUXes, FFs and Decoders, which are lower-level submodules and are not displayed here.

```
interface:
 ifc_dec_rf.rf dec,
 ifc_rob_rf.rf rob
module ram_4r_2w //4 read 2 write ram for store data
(.write1_data_i(rob.WB_data1),
 .write2_data_i(rob.WB_data2),
 .write1_addr_i(rob.WB_target1),
 .write2_addr_i(rob.WB_target2),
 .read11_addr_i(rob.read11_addr),
 .read12_addr_i(rob.read12_addr),
 .read21_addr_i(rob.read21_addr),
 .read22_addr_i(rob.read22_addr),
 .write1_en_i(rob.WB_en1),
 .write2_en_i(rob.WB_en2),
 .read11_en_i(dec.read11_en),
 .read12_en_i(dec.read12_en),
 .read21_en_i(dec.read21_en),
 .read22_en_i(dec.read22_en),
 .read11_data_o(read11_data),
 .read12_data_o(read12_data),
 .read21_data_o(read21_data),
      .read22_data_o(read22_data),
      .read11_ready_o(read11_ready_data),
      .read12_ready_o(read12_ready_data),
      .read21_ready_o(read21_ready_data),
      .read22_ready_o(read22_ready_data)
       );
     module ram_4r_4w //4 read 4 write ram for “valid”
           ( .write11_data_i('0),//write “0” when enqueue
      .write12_data_i('0),//priority of write “valid” defined in
FF cell, in case of data conflicts
      .write21_data_i('1),//write “1” when write back
      .write22_data_i('1),
      .write11_addr_i(dec.write1_addr),
      .write12_addr_i(dec.write2_addr),
      .write21_addr_i(rob.WB_target1),
      .write22_addr_i(rob.WB_target2),
      .read11_addr_i(rob.read11_addr),
      .read12_addr_i(rob.read12_addr),
      .read21_addr_i(rob.read21_addr),
      .read22_addr_i(rob.read22_addr),
      .write11_en_i(rob.write1_en),
      .write12_en_i(rob.write2_en),
      .write21_en_i(rob.WB_en1),
      .write22_en_i(rob.WB_en2),
      .read11_en_i(dec.read11_en),
      .read12_en_i(dec.read12_en),
      .read21_en_i(dec.read21_en),
      .read22_en_i(dec.read22_en),
      .read11_data_o(read11_valid_bit),
      .read12_data_o(read12_valid_bit),
      .read21_data_o(read21_valid_bit),
      .read22_data_o(read22_valid_bit),
      .read11_ready_o(read11_ready_valid),
      .read12_ready_o(read12_ready_valid),
      .read21_ready_o(read21_ready_valid),
      .read22_ready_o(read22_ready_valid)
       );

     rob.read11_ready = read11_ready_data && read11_ready_valid;
     rob.read12_ready = read12_ready_data && read12_ready_valid;
     rob.read21_ready = read21_ready_data && read21_ready_valid;
     rob.read22_ready = read22_ready_data && read22_ready_valid;


          //corner_case: data dependency among two in-flight
instructions at enqueue stage

       if(dec.write1_addr == rob.read21_addr)
rob.read21_valid_bit = '0;
       else rob.read21_valid_bit = read21_valid_bit;
       if(dec.write1_addr == rob.read22_addr)
rob.read22_valid_bit = '0;
       else rob.read22_valid_bit = read22_valid_bit;

      //corner_case: write and read data/"valid" consistency
      if(rob.WB_target1 == rob.read11_addr) begin
        rob.read11_valid_bit = '1;
        rob.read11_data = rob.WB_data1;
       end
      else begin
        rob.read11_valid_bit = read11_valid_bit;
        rob.read11_data = read11_data;
        end

      if(rob.WB_target1 == rob.read12_addr) begin
        rob.read12_valid_bit = '1;
        rob.read12_data = rob.WB_data1;
       end
      else begin
        rob.read12_valid_bit = read12_valid_bit;
        rob.read12_data = read12_data;
        end

      if(rob.WB_target1 == rob.read21_addr) begin
        rob.read21_valid_bit = '1;
        rob.read21_data = rob.WB_data1;
       end
      else begin
```

```
    rob.read21_valid_bit = read21_valid_bit;
    rob.read21_data = read21_data;
    end

  if(rob.WB_target1 == rob.read22_addr) begin
    rob.read22_valid_bit = '1;
    rob.read22_data = rob.WB_data1;
   end
  else begin
    rob.read22_valid_bit = read22_valid_bit;
    rob.read22_data = read22_data;
    end


  if(rob.WB_target2 == rob.read11_addr) begin
    rob.read11_valid_bit = '1;
    rob.read11_data = rob.WB_data2;
   end
  else begin
    rob.read11_valid_bit = read11_valid_bit;
    rob.read11_data = read11_data;
    end

  if(rob.WB_target2 == rob.read12_addr) begin
    rob.read12_valid_bit = '1;
    rob.read12_data = rob.WB_data2;
   end
  else begin
    rob.read12_valid_bit = read12_valid_bit;
    rob.read12_data = read12_data;
    end

  if(rob.WB_target2 == rob.read21_addr) begin
    rob.read21_valid_bit = '1;
    rob.read21_data = rob.WB_data2;
   end
  else begin
    rob.read21_valid_bit = read21_valid_bit;
    rob.read21_data = read21_data;
    end

  if(rob.WB_target2 == rob.read22_addr) begin
    rob.read22_valid_bit = '1;
    rob.read22_data = rob.WB_data2;
   end
  else begin
    rob.read22_valid_bit = read22_valid_bit;
    rob.read22_data = read22_data;
```

As we can see, the unit-level interfaces connect to the two RAMs, with enable signals, addresses and data.

B. Corner Cases and Solutions

a. “valid” bit write contention

There is one corner case when an instruction is being deallocated, meaning “0” is written to the “valid”. At the same time, an instruction is being enqueued and writing the “valid” into “0”. If the address they are going to write in “valid” is the same potentially, there will be contention. Here we determine writing “0”s has higher priority than writing “1” in “valid”. Hence, in the FF cell, we will define the priority.

Cycle Time	Description
Cycle 1	ROB not full; “0” corresponding to target register name written to “valid”; “1” corresponding to target register name whose instruction is going to be deallocated. Also, the two names are the same.
Cycle 2	The “valid” bit should be “0” when visible.

b. write and read data/“valid” consistency

Another corner case is when an instruction is being deallocated and another instruction is being enqueued, reading and writing data or “valid” at the same address could happen. The data or “valid” being written is not visible until the next cycle.

Cycle Time	Description
Cycle 1	Target register broadcast when committed in ROB and also is written into Register File with data & “valid”; an upcoming instruction being enqueued

	reads Register File for data & "valid". If its source register name is the same as the target register name, data & "valid" should be bypassed directly from the target register to source register.
Cycle 2	The bypassed source register data and valid bit visible in the ROB.

c. data dependency among two in-flight instructions at enqueue stage

Possibly, there are data dependencies among two instructions which are being enqueued in parallel, but "valid" FF is not able to reflect the dependencies so fast. So we have to compare the them to prevent this case. As we always assume instruction 1 before instruction 2 in terms of PC, we then compare the dependencies between two sources registers names of instruction 1 with target register name of instruction 2. If there is a match, meaning there is a data dependency, we have to asserted the corresponding "valid" bits into "0"s into ROB.

```
          if   (dec.write1_addr == rob.read21_addr)
rob.read21_valid_bit = '0;
          else rob.read21_valid_bit = read21_valid_bit;
          if   (dec.write1_addr == rob.read22_addr)
     rob.read22_valid_bit = '0;
          else rob.read22_valid_bit = read22_valid_bit;
```

3. ALU

Figure 8 presents the unit-level interfaces of ALU. It is composed by combinational logics. Since we can only execute at most one instruction each cycle, there are some logics of the after ALUs' results to input address to LSQ, if any.

```
     //one ALU
     input [OPRAND_WIDTH-1:0] operand1_i, operand2_i,
     input [OP_FUNC_WIDTH-1:0] op_func_i,
     output [OPRAND_WIDTH-1:0] result_o
      case(op_func_i)
      //R_type and I_type
       ADD,ADDI,LB,LH,LW: result_o = operand1_i + operand2_i;
       SUB: result_o = operand1_i - operand2_i;
       SLL,SLLI: result_o = operand1_i << operand2_i[4:0];
       SLT,SLTI: result_o = {{(OPRAND_WIDTH-1){'0}},res};
       XOR,XORI: result_o = operand1_i ^ operand2_i;
       SRL,SRLI: result_o = operand1_i >> operand2_i[4:0];
       SRA,SRAI: result_o =
{operand2_i[4:0]{operand1_i[OPRAND_WIDTH-1]}, (operand1_i>>
operand2_i[4:0])[32-operand2_i[4:0]:0]} ;  //?
```

```
    OR,ORI: result_o = operand1_i || operand2_i;
    AND,ANDI: result_o = operand1_i && operand2_i;
   //S_type
    BEQ: result_o = (operand1_i == operand2_i);
    BNE: result_o = (operand1_i != operand2_i);
    BLT: result_o = (operand1_i < operand2_i);
    BGE: result_o = (operand1_i < operand2_i);
    SH,SB,SW: result_o = operand1_i + operand2_i;
  //U_type
    JAL:result_o = operand1_i + 1;
    JR: //do nothing
  endcase

    assign operand1 = operand1_i;
    assign operand2 = operand2_i;
  //comparator
       comparator_signed
        (
       .operand1_i(operand1),.operand2_i(operand2),
       .res_o(res) //set 1 if rs1 < rs2, otherwise
        );
```

This is a sub-unit of signed comparator.

```
       //signed comparator
       input [DATA_WIDTH-1:0] operand1_i, operand2_i,
       output res_o //set 1 if rs1 < rs2, otherwise

        if(operand1_i[DATA_WIDTH-1] <
operand1_i[DATA_WIDTH-1]) res_o = '1;//rs1 = 0 && rs2 = 1
       else if(operand1_i[DATA_WIDTH-1] >
operand1_i[DATA_WIDTH-1]) res_o = '0;
       else begin
          for(int i=DATA_WIDTH-2;i>=0;i--)
          if(operand1_i[i]<operand2_i[i]) begin
         res_o = '1;
         break;
          end
          else if(operand1_i[i]>operand2_i[i]) begin
         res_o = '0;
        break;
         end
         else res_o = '0;
         end
```

This is a control module by selecting “address” as “result” to LSQ.

```
input [OPRAND_WIDTH-1:0] result1, result2,
input [OP_FUNC_WIDTH-:0] op_func1, op_func2;
output [OPRAND_WIDTH-1:0] address;

if(op_func1 == Memory-access instructions) address = 
result1;
else if(op_func2 == Memory-access instructions) address = 
result2;
```

4. ROB

A. Microarchitecture Specification and Pseudo Code

In this section, we shows the microarchitecture specification of our ROB and the pseudo codes corresponding to each stages.

Note:Instantiate CAMs: V (32*1), D (32*1), E (32*1), B (32*1), M (32*1), entry (32*5), OP (32*8), Func (32*12), Imm (32*12), Result (32*32), Target (32*5), ARG0 (32*32), v0 (32*1), src0 (32*5), ARG1 (32*32), v1 (32*1), src1 (32*5), PC (32*16);

Combine v0 v1 OP func arg0 arg1 imm target PC entry CAMs and visible each other.

a. Head and Tail FSM

Here shows the truth table of head and tail control logic of our FIFO-like CAM or RAM structures. Normally, in1_en_i and in2_en_i are two enable signals of enqueuing instructions; out1_en_i and out2_en_i are two enable signals of committing instructions; tail_inc, tail_inc2, head_inc1 and head_inc2 indicate head and tail increment steps, meaning head and tail index increases by one or two.

in1_valid_i	in2_valid_i	tail_inc1	tail_inc2
0	0	0	0
0	1	1	0
1	0	1	0

1	1	0	1

out1_valid_i	out2_valid_i	head_inc1	head_inc2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

However, there are special cases when it comes to branch misprediction, jump and load violation flush. Here we show the detailed implementation of these special cases handling strategies. As specified before, when a load violation happens, flush_valid signal flush_index are returned are read by ROB. When flush_valid signal is asserted high, ROB will search flush_entry using entry_CAM in ROB. We want to find the index of the violated “load” and move the tail pointer to matched entry. For the branch and jump flush, we use enable signal to indicate time to flush in our design and move the pointer to the pointer after the branch and jump. We assume two branches or jumps or jump and branch are not allowed to be issued in parallel.

```
    //tail
    //LSQ violation flush
    entry_cam_search1_en = flush_valid;
    entry_cam_search1_data = flush_entry;

    //branch misprediction flush
    op_cam_search1_en = branch_en;
    op_cam_search1_data = first matched entry;
    op_cam_search2_en = branch_en;
    op_cam_search2_data = second matched entry;

    //jump flush
    op_cam_search3_en = jump_en;
    op_cam_search3_data = first matched entry;
    op_cam_search4_en = jump_en;
    op_cam_search4_data = second matched entry;
    if (op_cam_search1_index == branch || op_cam_search3_index
== jump)
```

```
          tail_temp1 =  first matched entry;
     else if (op_cam_search2_index == branch ||
op_cam_search4_index == jump)
          tail_temp1 =  second matched entry;
     else
          tail_temp1 = tail;
     if (entry_cam_search_valid)
          tail_temp2 = entry_cam_search_index;
     else
          tail_temp2 = tail;

     //LSQ flush load and branch/jump happens at the same time
     if (tail < tail_temp1)
          tail_temp3 = tail + 32 - tail_temp1;
     else
          tail_temp3 = tail - tail_temp1;

     if (tail < tail_temp2)
          tail_temp4 = tail + 32 - tail_temp2;
     else
          tail_temp4 = tail - tail_temp2;

     if(tail_temp3 < tail_temp4)
          tail = tail_temp4;
     else if (tail_temp3 >= tail_temp4)
          tail = tail_temp3;

     //Normal increment and decrement
     else if both insN_valid_i is 1
          if tail+2>31
               tail <= tail+2-32;
     else
                 tail <= tail+2;
     else if only ins1_valid_i is 1
           if tail+1>31
                tail <= tail+1-32;
     else
                tail <= tail+1;
     else;
```

For the head pointer, we use a variable which indicate the number of instructions which are committed in this cycle, so we use the head this cycle and number of commits this cycle to determine where the head goes the next cycle. ROB full signal is asserted when the index in ROB reaches 29, for there is possibly at most two instructions being enqueued. So in the worst case, a "stall" will be returned to TB when there are 32 entries, which is safe.

```
//head
if head+comcnt > 31
   head <= head + comcnt - 32;
else
   head <= head + comcnt;

//full_cnt
If both insN_valid_i is 1
     full_cnt = full_cnt + 2 -  comcnt;
else if only ins1_valid_i is 1
full_cnt = full_cnt + 1 -  comcnt;
else
full_cnt = full_cnt  -  comcnt;
assign full = (full_cnt >= 29) ? 1 : 0;
```

B. **Enqueue Stage**

In this stage, we will enqueue two instructions at most from Instruction Decoder or Register File to ROB. At the same time, memory-access instructions are decoded and allocated into LSQ with their types, and return their entries in LSQ. The number of write ports these CAMs are two to enable two instructions are enqueued at the same time at "tail" and "tail + 1". The content being enqueued corresponds to the logic organization in Figure 10. The N here ranges from 1 to 2.

```
//N = 1-2
//enqueue PC
PC_cam_wN_en = insN_valid_i;
PC_cam_w1_index = tail;
PC_cam_w2_index = tail+1;
PC_cam_wN_data = insN_pc;

//enqueue src0
src0_cam_wN_en = insN_type != U ? insN_valid_i : 0;
```

```
src0_cam_w1_index = tail;
src0_cam_w2_index = tail+1;
src0_cam_wN_data = insN_src0;

//enqueue src1
src1_cam_wN_en = ( insN_type == (R || U) ) ? insN_valid_i;
src1_cam_w1_index = tail;
src1_cam_w2_index = tail+1;
src1_cam_wN_data = insN_src1;

//enqueue v0
v0_cam_wN_en = readN1_ready_i;
v0_cam_w1_index = tail;
v0_cam_w2_index = tail+1;
v0_cam_wN_data = readN1_valid_i;

//enqueue v1
v1_cam_wN_e
v1_cam_w1_index = tail;
v1_cam_w2_index = tail+1;
v1_cam_wN_data = readN2_valid_i;

//enqueue ARG0
arg0_cam_wN_en = readN1_ready;
arg0_cam_w1_index = tail;
arg0_cam_w2_index = tail+1;
arg0_cam_wN_data = readN1_data_i;

//enqueue ARG1
arg1_cam_wN_en = readN2_ready;
arg1_cam_w1_index = tail;
arg1_cam_w2_index = tail+1;
arg1_cam_wN_data = readN2_data_i;

//enqueue Target
target_cam_wN_en = insN_valid_i && insN_type != S;
target_cam_w1_index = tail;
target_cam_w2_index = tail+1;
target_cam_wN_data = writeN_addr;
```

```
//enqueue Imm
if(insN_valid_i)
case (insN_type)
R:   immN_valid = '0;
     immN_data = '0;
I:   immN_valid = '1;
     immN_data = ins1_i[31:20];
S:   immN_valid = '1;
     immN_data = {ins1_i[31:25], ins1_i[11:7]};
U:   immN_valid = '1;
immN_data = ins1_i[31:20];
endcase

imm_cam_wN_en = immN_valid;
imm_cam_w1_index = tail;
imm_cam_w2_index = tail+1;
imm_cam_wN_data = immN_dataN;

//enqueue func
func_cam_wN_en = insN_valid_i && insN_type != U;
func_cam_w1_index = tail;
func_cam_w2_index = tail+1;
func_cam_wN_data = func(need to judge);

//OP
op_cam_wN_en = insN_valid_i;
op_cam_w1_index = tail;
op_cam_w2_index = tail+1;
op_cam_wN_data = opN;

//entry
entry_cam_wN_en = ls_validN;
entry_cam_w1_index = tail;
entry_cam_w2_index = tail+1;
entry_cam_wN_data = ls_entryN_i;

//enqueue D
d_cam_wN_en = insN_valid_i;d_cam_w1_index = tail;
```

```
d_cam_w2_index = tail+1;
d_cam_wN_data = 0;

//enqueue LSQ
if (ins1_valid && ins1_type)
case(ins1_type)
LW:   ls_valid1 = '1;
     ls_type1 = 3b'010;
LH:   ls_valid1 = '1;
     ls_type1 = 3b'001
LB:   ls_valid1 = '1;
     ls_type1 = 3b'000;
SW:   ls_valid1 = '1;
     ls_type1 = 3b'110;
SH:   ls_valid1 = '1;
     ls_type1 = 3b'101;
SB:   ls_valid1 = '1;
     ls_type1 = 3b'100;
default: ls_valid1 = '0;
     ls_type1 = 3b'111;
endcase

else if (ins1_valid && ! ins1_type && ins2_valid)
case(ins2_type)
LW:   ls_valid1 = '1;
     ls_type1 = 3b'000;
LH:   ls_valid1 = '1;
     ls_type1 = 3b'001
LB:   ls_valid1 = '1;
     ls_type1 = 3b'010;
SW:   ls_valid1 = '1;
     ls_type1 = 3b'100;
SH:   ls_valid1 = '1;
     ls_type1 = 3b'101;
SB:   ls_valid1 = '1;
     ls_type1 = 3b'110;
default: ls_valid1 = '0;
     ls_type1 = 3b'111;
endcase
```

```
if (ins2_valid &&  ls_valid1)
case(ins2_type)
LW:  ls_valid2 = '1;
     ls_type2 = 3b'000;
LH:  ls_valid2 = '1;
     ls_type2 = 3b'001
LB:  ls_valid2 = '1;
     ls_type2 = 3b'010;
SW:  ls_valid2 = '1;
     ls_type2 = 3b'100;
SH:  ls_valid2 = '1;
     ls_type2 = 3b'101;
SB:  ls_valid2 = '1;
     ls_type2 = 3b'110;
default: ls_valid2 = ‘0;
     ls_type2 = 3b’111;
endcase

else
     ls_valid2 = ‘0;
     ls_type2 = 3b’111;
```

C. Issue Stage

The pseudo code of issue stage are presented as follows. Basically, we look for two instructions from “head” to “tail” to issue, according to their “v0”, “v1” and opcode information. Those which are matched will be sent to ALU and put the corresponding register names of target operands to target bus for broadcast, according to the instruction types. For the issue logic, we have to make “v0”, ”v1” and “OP” CAMs visible to each other, meaning that we want to use the information of three combined to determine which is ready to be issued. Also, if there is a “load” or “store” going to be issued, we want to send the address to the “load” or “store” with its entry index in LSQ and the “store” data, if it is a “store”.

```
//search for the first issued instruction
for(head to tail)

if(first matched found)
     issue1_valid[] = 1;
     break;
```

```
if (op_data[] == R type && v0_data[] && v1_data[])
     operand11_o = arg0_data[];
     operand12_o = arg1_data[];
     op_func1 = {op_data, func_data};
     target_bus1 = target_data[];
else if (op_data[] == I type && v0_data[])
     operand11_o = arg0_data[];
     operand12_o = imm_data[];
     op_func1 = {op_data, func_data};
     target_bus1 = target_data[];

else if (op_data[] == S type && v0_data[] && v1_data[])
     if(op_data[] == branch)
          operand11_o = arg0_data[];
          operand12_o = arg1_data[];
          op_func1 = {op_data, func_data};
          target_bus1 = 0;
     else
          operand11_o = arg0_data[];
          operand12_o = imm_data[];
          op_func1 = {op_data, func_data};
          target_bus1 = 0;

else if (op_data[] == U type) //jump
          operand11_o = PC_data[];
          operand12_o = 1;
          op_func1 = {op_data, func_data};
          target_bus1 = 0;
     else
          operand11_o = 0;
          operand12_o = 0;
          op_func1 = 0;
          target_bus1 = 0;

// pc-alu
if (op_data[] == branch && v0_data[] && v1_data[])
     pc_alu_operand1 = PC_data[];
     pc_alu_operand2 = imm_data[];
```

```
          if(result1) branch_en = 1;
                    branch_pc = pc_alu_result;
          else branch_en = 0;
               branch_pc = 0;
     else if (op_data[] == jump)
           pc_alu_operand1 = PC_data[];
           pc_alu_operand2 = imm_data[];
           jump_en = 1;
           jump_pc = pc_alu_result;

     else
          pc_alu_operand1 = 0;
          pc_alu_operand2 = 0;


     //issue to LSQ
     if (op_data[]  == load && v0_data[])
          addr_entry = matched entry;
          addr_valid = 1;
          store_data = 0;
     else if (op_data[] == store && v0_data[])
          addr_entry = matched entry;
          addr_valid = 1;
          store_data = arg1_data[];
     else
          addr_entry = 0;
          addr_valid = 0;
          store_data = 0;

end for

for(first matched issue entry to tail)

     if(second matched found)
     issue2_valid == 1;
     break;

     if (op_data[] == R type && v0_data[] && v1_data[])
          operand21_o = arg0_data[];
          operand22_o = arg1_data[];
          op_func2 = {op_data, func_data}
          target_bus2 = target_data[];

     else if (op_data[]  == I type && v0_data[])
          if(first matched issue == load or store && op_data[]
== load)
               operand21_o = 0;
                          operand22_o = 0;
               op_func2 = 0;
               target_bus2 = 0;
          else
               operand21_o = arg0_data[];
                          operand22_o = imm_data[];
               op_func2 = {op_data, func_data}
               target_bus2 = target_data[];

     else if (op_data[] == S type && v0_data[] && v1_data[])
          if(op_data[] == store)
               if (first matched issue == load or store)
                    operand21_o = 0;
                    operand22_o = 0;
                    op_func2 = 0;
                    target_bus2 = 0;
               else
                    operand21_o = arg0_data[];
                    operand22_o = imm_data[];
                    op_func2 = {op_data, func_data}
                    target_bus2 = 0;
          else
               if (op_data[first matched issue] == branch ||
                  op_data[first matched issue] == jump)
                    operand21_o = 0;
                    operand22_o = 0;
                    op_func2 = 0;
                    target_bus2 = 0;
               else
                    operand21_o = arg0_data[];
                    operand22_o = arg1_data[];
                    op_func2 = {op_data, func_data}
                    target_bus2 = 0;

     else if (op_data[] == U type) //jump
          if (op_data[first matched issue] == jump ||
             op_data[first matched issue] == branch)
               operand21_o = 0;
               operand22_o = 0;
               op_func2 = 0;
               target_bus2 = 0;
          else
               operand21_o=PC_data[];
               operand22_o=imm_data[];
               op_func2 = {op_data, func_data}
               target_bus2 = 0;
          else
               operand21_o=0;
               operand22_o=0;
               op_func2 =0
               target_bus2 = 0;

     // pc-alu
     if (op_data[] == branch && v0_data[] && v1_data[])
          if (first matched issue != branch && first matched
issue ! = jump)
               pc_alu_operand1 = PC_data[];
               pc_alu_operand2 = imm_data[];
               if(result2) branch_en = 1;
               branch_pc = pc_alu_result;

     else if (op_data[] == jump)
          if (first matched issue != branch && first matched
issue != jump)
           pc_alu_operand1 = PC_data[];
           pc_alu_operand2 = 1;
           jump_en = 1;
           jump_pc = pc_alu_result;

     //issue to LSQ
```

```
if (first matched issue == load or store)
     addr_entry = 0;
     addr_valid = 0;
     store_data = 0;
else
if (op_data[]  == load && v0_data[])
     addr_entry = matched entry;
     addr_valid = 1;
     store_data = 0;
else if (op_data[] == store && v0_data[])
     addr_entry = matched entry;
     addr_valid = 1;
     store_data = arg1_data[];
else
     addr_entry = 0;
     addr_valid = 0;
     store_data = 0;
end for

seq: match1_entry
     match1_entry = first_match_entry;
seq: match1_entry
     match2_entry = second_match_entry;

result_cam_w1_en = issue1_valid;
result_cam_w1_index = match1_entry
result_cam_w1_data = result_bus1;
D[match1_entry] = 1;
result_cam_w2_en = issue2_valid;
result_cam_w2_index = match1_entry
result_cam_w2_data = result_bus2;
D[match2_entry] = 1;
```

D. Broadcast After Execution and When Commit

We take two kinds of bypass and broadcast into consideration in ROB. One is done after execution, target operands of the executed instructions are broadcasted to bypass the results. Another is done when an instruction is committed and its target will be broadcast to search for a potential bypass to invalid source operands in case there come instructions whose source operands have dependencies with previous instructions which have been executed into ROB.

```
      seq:result_busN
      result_busN = alu_resultN_i;

      for(head to tail)
      if (target_bus1 == srcN[] && vN[] == 0) (N = 0 or 1)
           bcast_issue1[] = 1;
      else
           bcast_issue1[] = 0;

      if (target_bus2 == srcN[] && vN[] ==0) (N = 0 or 1)
                bcast_issue2[] = 1;
      else
           bcast_issuet2[] = 0;

      if (WB_en1_o)
           bcast_commit1[] = 1;
      else
           bcast_commit1[] = 0;
      if (WB_en2_o)
           bcast_commit2[] = 1;
      else
           bcast_commit2[] = 0;
      end for

      for (genvar i = 0 ; i < 32 ; i++ )
      ff_vN(.write_en1(bcast_issue1[i]), .write_data1(1),
           .write_en2(bcast_issue2[i]), .write_data2(1),
           .write_en3(bcast_commit1[i]), .write_data3(1),
           .write_en4(bcast_commit2[i]), .write_data4(1),
      );
      ff_argN(.write_en1(bcast_issue1[i]),
           .write_data1(result_bus1),
           .write_en2(bcast_issue2[i]),
           .write_data2(result_bus2),
           .write_en3(bcast_commit1[i]),
 .write_data3(WB_data1_o),
           .write_en4(bcast_commit1[i]),
      .write_data4(WB_data2_o),
      );
      end for
```

E. Commit Stage

We present the pseudo code of our commit policy. We assume there are only at most two instructions committed in one clock cycle. For the two memory-access instructions, only one "load" and one "store" combinations are allowed to be committed at the same time, only one otherwise. Every cycle, we search from the instruction at "head", check its instruction type and then further check instruction at "head + 1".

```
    if(D[head]==1)
    if (op_data[head] == load)
                case(func[head])
                    LH :    WB_data1_o =
{16{load_data_i[15]},load_data_i[15:0] };
                    WB_target1_o = target[head];
    LB : WB_data1_o = {24{load_data_i[7]},load_data_i[7:0]};
         WB_target1_o = target[head];
    LHU : WB_data1_o = {16'b0, load_data_i[15:0] };
          WB_target1_o = target[head];
    LBU : WB_data1_o = {24'b0, load_data_i[7:0]};
          WB_target1_o = target[head];
    Default: WB_data1_o = load_data_i;
             WB_target1_o = target[head];
    endcase
    else
         WB_data1_o = result[head];
         WB_target1_o = target[head];

WB_en1_o = (D[head] && op_data[] != S)? 1:0;

if (head + 1 > 31)
    head2 = head+1-32;
else
    head2 = head+1

if(D[head] == 1 && D[head2] == 1)
    if (op_data[head] != load && op_data[head2] == load)
         case(func[head2])
    LH : WB_data2_o = {16{load_data_i[15]},load_data_i[15:0]}
         WB_target2_o = target[head2];
    LB :  WB_data2_o = { 24{load_data_i[7]},load_data_i[7:0] };
```

```
            WB_target2_o = target[head2];
      LHU : WB_data2_o = {16'b0, load_data_i[15:0]}
           WB_target2_o = target[head2];
      LBU : WB_data2_o = { 24’b0, load_data_i[7:0] };
            WB_target2_o = target[head2];
     Default: WB_data2_o = load_data_i;
             WB_target2_o = target[head2];
      else
                WB_data2_o = result[head2];
                WB_target2_o = target[head2];
      end if

         //WB_en2_o
         if (D[head] && D[head2])
         if (op_data[head] == load && op_data[head2] == load)
                WB_en2_o = 0;
            else if(op_data[head2] == S)
                WB_en2_o = 0;
            else
                WB_en2_o = 1;
      else
           WB_en2_o = 0;

 //commit count
      if (!D[head])
           comcnt = 0;
      else if (D[head] && !D[head2])
           comcnt = 1;
      else
           if (op_data[head] == load && op_data[head2] == load)
                comcnt = 1;
           else if (op_data[head] == store && op_data[head2] ==
      store)
                comcnt = 1;
           else
                comcnt = 2;

 //commit with LSQ
      if (D[head] && op_data[head] == load)
           load_commit_entry_o = entry[head];
      else
           load_commit_entry_o = entry[head2];
      if ((D[head] && op_data[head] == load)  ||
           (D[head] && op_data[head] != load && D[head2] &&
 op_data[head2] == load) )
           load_commit_valid_o = 1;
      else
           load_commit_valid_o = 0;

      if (D[head] && op_data[head] == store)
           store_commit_entry_o = entry[head];
      else
           store_commit_entry_o =  entry[head2];

      if ( (D[head] && op_data[head] == store) ||
          (D[head] && op_data[head] != store && D[head2] &&
 op_data[head2] == store) )
      store_commit_valid_o = 1;
 else
      store_commit_valid_o = 0;
```

5. LSQ

In this section, we demonstrate the microarchitecture of our LSQ. Since we allow only one memory-access instruction to be issued in one clock cycle and want to take all kinds of load and store instructions into consideration for bypass, we always want load instructions to get data from memory speculatively and bypass from “store” to “load” when the “store” needs to be committed. Also, this “store” only bypass the data to the first matched “load” after it searches all of the younger “loads”. For the rest of matched “loads”, if any, we then assert a violation and flush them. There are several benefits if we use this scheme.

- All kinds of store instructions can be bypassed, if any, with a simple implementation.
- Only one bypass channel is needed. We flushed the second and more matched loads, since this case happens in low possibilities.
- Only two “search” are needed. Easy to be implemented.

A. Microarchitecture Specification and Pseudo Code

Table shows the behavior of LSQ in terms of life cycles. We start with enqueue stage and end up with commit stage.

Cycle Time	Description
Cycle 1	Two instructions at most are sent to LSQ with instruction type at enqueue stage

Enqueue	with control bit at enqueue stage. The instructions are allocated into LSQ in program order using “tail” and “tail + 1” as index.
Cycle 2 Execution	One of them is issued, if “load”, write its address after ALU to LSQ to its entry. At the same time, the address is sent to memory, wants to read data; if “store”, write its address like “load” and also pass the data to be stored from ROB to LSQ.
Cycle 3 Memory	For “load”, reading data from the memory and write to Data_cam. Loading data from memory will be no conflicts, since we use dual port memory for one write and one read. Writing into Data_cam will also be no conflicts, since we use dual write port cam, one for “store” and one for “load”. For “store”, no operations.
Cycle 4 or more Commit	Since ROB determines when to commit instructions, a commit request is sent to LSQ when an instruction is ready to be committed. For a “load”, the data corresponding to the input entry is returned immediately to ROB. For a “store”, it sends data into memory and searches younger “loads” to check if there is a chance to bypass at the same time. We bypass the data to the first matched load and flush other loads after, for the limited number of bypass channels. Meanwhile, data is transferred into memory. It is possible to commit one “load” and one “store” the same cycle.

We demonstrate the pseudo code here stage by stage.

Note: ls_type (instruction type): Load Byte, Load Half, Load Word, Store Byte, Store Half, Store Word. We use 3 bits as index: *100* (Store Byte), *101* (Store Half), *11x* (Store Word); *000* (Load Byte), *001* (Load Half), *01x* (Load Word)

Instantiate 5 CAMs, each with 2-write and 2-read ports: *type_cam* (32*3), *addr_cam* (32*32), *addr_valid_cam* (32*1), *dataN_cam* (32*8), *dataN_valid_cam* (32*1), where N from 3 to 0.

a. Head and Tail Pointer Control // we don’t care whether LSQ is full, since we do protection in ROB, if ROB desn’t overflow, LSQ couldn’t overflow.

```
//sequential logical head, tail.
//tail
If both ls_validN_i is 1
     If tail+2>31
          tail = tail+2-32;
     else
          tail = tail+2;
     If only is_valid1_i is 1
     If tail+1>31
          tail = tail+1-32;
     else
          tail = tail+1;
     else;
//head
if load_commit_valid and store_commit_valid is 1
     if head+2>31
          head=head+2-32;
     else
          head=head+2;
if only one of load_commit_valid or store_commit_valid is 1
     if head+1>31
          head=head+1-32;
     else
          head=head+1;
     else;
```

b. Enqueue Stage Allocation

Here *type_cam_wN_* and *type_cam_rN_* denote the wires connected to the data and control signal ports of *type_cam*, where N ranges from 1 to 2, for we have 2 channels for allocation.

```
     type_cam_wN_en = ls_validN_i;
     type_cam_w1_index = tail;
     type_cam_w2_index = ( (tail+1)>31 ) ? (tail+1-32) :
(tail+1);
     type_cam_wN_data = ls_typeN_i;
     ls_entryN_o = type_cam_wN_index;

     av_cam_w1_en = ls_validN_i;
     av_cam_w1_index = tail;
     av_cam_w1_data = 0;
```

c. Execution Stage Memory Access

Here N ranges from 3 to 0, for data storage in LSQ are divided into 4 8-bit sections for “load” and “store” with different data width.

```
     //read type from type_cam
     type_cam_r1_en = addr_valid_i;
     type_cam_r1_index = addr_entry_i;
```

```
//write address into addr_cam and renew av
addr_cam_w_en = addr_valid_i;
addr_cam_w_index = addr_entry_i;
addr_cam_w_data = result_addr_i;

av_cam_w2_en = addr_valid_i;
av_cam_w2_index = addr_entry_i;
av_cam_w2_data = 1;//reset 0 when enqueue

// write data into data_cam based on type
dataN_cam_w1_en = addr_valid_i;
dataN_cam_w1_index = addr_entry_i;
data3_cam_w1_data = store_data_i[31:24];
data2_cam_w1_data = store_data_i[23:16];
data1_cam_w1_data = store_data_i[15:8];
data0_cam_w1_data = store_data_i[7:0];

dvN_cam_w1_en = addr_valid_i;
dvN_cam_w1_index = addr_entry_i;
dv3_cam_w1_data = type_cam_r1_valid && (type_cam_r1_data ==
StoreWord) ? 1 : 0
dv2_cam_w1_data = type_cam_r1_valid && (type_cam_r1_data ==
StoreWord) ? 1 : 0;
dv1_cam_w1_data = type_cam_r1_valid && (type_cam_r1_data ==
StoreWord || StoreHalf) ? 1 : 0;
dv0_cam_w1_data = type_cam_r1_valid && (type_cam_r1_data ==
Store) ? 1 : 0 ;

// read data from memory if type is load
if  type_cam_r1_valid && type_cam_r1_data == Load
   mem_load_en = 1;
else
   mem_load_en = 0;

mem_load_type = type_cam_r1_data;
mem_load_addr = result_addr_i;//give the addr to mem
directly not waiting visible in LSQ

// write memory data into data_cam based on type
```

```
dataN_cam_w2_en = mem_load_valid_i;
dataN_cam_w2_index = addr_entry_i_ff;//delay addr_entry_i
for one cycle

 // bypass data write first
if (search_valid_o == 1 && search_index_o ==
addr_entry_i_ff)
dataN_cam_w2_data = mem_store_value;
(depends on the search_type_o)

else  //no need to bypass
data3_cam_w2_data = mem_load_value_i [31:24];
data2_cam_w2_data = mem_load_value_i [23:16];
data1_cam_w2_data = mem_load_value_i [15:8];
data0_cam_w2_data = mem_load_value_i [7:0];

dvN_cam_w2_en = mem_load_valid_i;
dvN_cam_w2_index = addr_entry_i_ff;
dv3_cam_w2_data = type_cam_r1_data_ff == loadWord ? 1 : 0
dv2_cam_w2_data = type_cam_r1_data_ff == loadWord ? 1 : 0;
dv1_cam_w2_data = type_cam_r1_data_ff == loadWord ||
loadHalf ? 1 : 0;
dv0_cam_w2_data = type_cam_r1_data_ff == load ? 1 : 0;
```

B. Corner Cases and Solutions

a. load memory data & store bypass data conflict

Cycle Time	Description
Cycle 1	When a “load” tries to load memory first, consider the time when data from memory after a cycle time of read is written to the “load”. At the same time, it is possible that a “store” at “head” is committed. It searches the “load” matches its address and is going to bypass the data. So there is a conflict in assigning data to the write-data port of the “load”. Here, “bypass” should have more priority and we use some logic to resolve this potential problem.
Cycle 2	The data of “bypass” is visible in FFs.

b. *addr_entry_i* Input Port Contention

Consider when a “load” wants to read memory, memory needs one cycle to process “read”. When data from memory is ready, it needs also the index of entry to the “load” which sent the request. However, at this moment, the index from the port *addr_entry_i* could potentially changed for the following instruction whose address is sent to LSQ with its entry index. So in order to avoid this contention, we have to use FFs buffer the entry for one cycle until memory gives data to the “load”.

Cycle Time	Description
Cycle 1	The address of a “load” is resolved after ALU then sent to LSQ and to memory in parallel, with its entry index via *addr_entry_i*.
Cycle 2	The address of another instruction is resolved after ALU then updated to LSQ with its entry index via *addr_entry_i* too; The data read from of previous “load” is ready and needs the index of the “load”, from *addr_entry_i_ff*, which is the index buffered one cycle.

c. two “stores” bypass to a “load”

Normally, there is a problem potentially when two “stores” in front of a “load” in program order have the same address. It happens that the second “store” is issued first and then the “load” and finally the first “load”. In this case, if “store” bypass the data without the determination of sequences of two “stores”, a problem will happen because of the wrong bypassed data. The sequences of three instructions is shown below (<1> -> <2> -> <3>).

```
store addr1, R1 <3>
store addr1, R2 <1>
load addr1, R3 <2>
```

In our design, since we do not bypass immediately when the address of a “load” is resolved, we can avoid this problem.

Our design also solves the problem, when the width of data going to be bypassed less than one word, shown below. Say, “SB” is issued first and then “LW” some time after. In our design, “LW” speculatively load data from memory as soon as address resolved, but the data is incorrect partially. Then, when “SB” is committed, it searches younger “loads” with the same address and bypass the higher 1 Byte to “LW”.

```
SB addr1, R2 <1>
LW addr1, R3 <2>
```

V. Verification Strategy

In this chapter, we discuss the specifications and features we verify and explain how we verify them.

1. Specification Overview

Before planning the strategy to verify the design, we first need to determine the features and specifications to verify. In validating our design, we have to verify the following specifications:

- Reset should work initially and dynamically.
- The completed instructions must be committed in order.
- The design must commit two completed instructions when there is no conflict.
- *ROB_full* must be correctly set when the ROB is full.
- The result of each instruction of arithmetic/logical operations must be correct.
    - Addition/Subtraction
    - Logical
    - Shifting
    - Comparison
- The values and addresses of load/store must be correct.
    - Only at most 1 store and 1 load can be committed each cycle.
- Data/Memory dependency should be correctly handled.
    - Data register dependency
    - Load/Store dependency
- When a serious violation is detected, *flush_en* and *flush_PC* should be correctly fed back.
- When a jump is issued, *jump_en*, *jump_PC* and *flush_PC* should be correctly set.
- When a branch is mispredicted, *branch_en*, *branch_PC* and *flush_PC* should be correctly fed back.
- The pipelining should be correct when there are invalid instructions in the instruction sequence. (halt)

In the following sections, we introduce our strategies to verify each of the specifications, explain details of our plans, and discuss anticipated problems and corner cases.

2. Reset Function

Reset is a basic function. After reset is set, there should be no write-backs and other feedbacks in the consecutive cycles as long as there is no new instructions driven into the DUT. If reset is set during the execution, the current instructions in the ROB/LSQ and the data in the register files should be all flushed.

```
IF reset THEN
     NO write-back
&&   NO memory access
&&   NO control feedback
```

3. Commission Order

One of the first things we verify is the commission order. There are three major kinds of feedbacks we could receive from the DUT: the register file write-backs, the memory store, and the branch/jump/flush/stall control signal. In the test signal generator, we will generate two instructions every cycle. When these instructions are driven into the golden model, they will be analyzed and stored in the local data structures in order. When the testbench receives either kind of the feedback signals, we have to verify the order of these feedbacks, which should be the same of the order we stored in our local data structures.

If the instructions are of register write-backs, the corresponding WB outputs should be set when the completed instructions are committed. Since we will try not to reuse the in-use destination registers unless the associated instructions are committed, we can use the destination registers as identifiers to check if the write-backs are the correct ones, compared to the instructions stored in the local data structures.

The load instruction can be identified by the destination register; however, for the memory store operation, because nothing will be written back to the register files when it is committed, we need to monitor *mem_store_en_o*, *mem_store_type_o*, *mem_store_addr_o* and *mem_store_value_o* to make sure that the store operation was correctly executed.

For branch and jump, because they will not be written back to the register files, we monitor the corresponding control signals and take the feedbacks as the commissions. When we receive the branch or jump control signals, in addition to adjusting the PC, we will also mark the corresponding branch or jump instructions as committed and remove them from our data structures. Notably, the "commission" of branch and jump need not be in-order since they are primarily of higher level execution control. When the testbench receives the branch and the jump operations, it verifies the correctness of the output and starts to send the new instructions with the new PC if necessary.

```
WHILE write-backs || mem_store_en_o
     IF write-backs THEN
          rd in queue[0] || rd in queue[1]
     ELSE /* memory store */
          mem_store_type_o in queue[0] || queue[1]
     &&   mem_store_addr_o in queue[0] || queue[1]
     &&   mem_store_value_o in queue[0] || queue[1]
```

4. 2-way Commission

To test the number of commissions, we plan two polarized scenarios. One is that there is absolutely no dependency between the instructions. For example, we only generate the instructions of addition with non-repeated registers. In this case, there should always be two instructions committed. On the other hand, we generate extremely dependent instructions or consecutive load instructions such that the DUT should only commit one instruction per cycle expectedly.

It should be worth mentioning that there are only 32 registers and r0 should be fixed to zero. In that case, to test commission without dependency could rely on the execution speed of the pipeline design. If there are not enough free registers before the upcoming instructions, the test could still result in data register dependency.

```
ASSUME INDEPENDENT
ASSERT(WB_en1 & WB_en2) every cycle
```

5. Fullness Of ROB

When the design is not able to commit two instructions, the ROB could become full. When the ROB is full, *ROB_full* signal should be set to tell the testbench to hold the instruction feeding. Since we are not going to simulate the pipeline in the testbench, we will use a counter to record the status of the ROB usage. When the counter indicates that the ROB is full, we check *ROB_full* to test if this feature is completed. The counter should be incremented when the instructions are driven into the DUT and the golden model and should be decremented according to the feedbacks from the DUT.

```
counter += # of instruction driven-in
counter -= # of instruction committed
ASSERT(ROB_full == (count >= full threshold))
```

6. Instruction Results

Most of the part of checking struction results is about checking the results of the ALU and the LSQ. Based on the opcode of the instructions, we may partition the instructions into three categories: arithmetic/logical (AL), load/store, and branch/jump. In this section, we focus on the AL operations, which are all done by the ALUs. Basically, the way to verify the AL operations are very straight: simply compute the results with the corresponding operations. We explain the details of verifications for different kinds of AL operations as follows:

```
IDENTIFY rd => CHECK rd
```

A. Addition/Subtraction

There are three instructions of this category:

- `ADD  rd, rs1, rs2` (R)
- `ADDI rd, rs1, imm` (I)
- `SUB  rd, rs1, rs2` (R)

The first three instructions are the fundamental adding and subtracting operations; whereas the last two are primarily used for address and PC calculation. Arithmetic overflow is ignored.

B. Logical

The logical operations include:

- `XOR  rd, rs1, rs2` (R)
- `XORI rd, rs1, imm` (I)
- `OR   rd, rs1, rs2` (R)
- `ORI  rd, rs1, imm` (I)
- `AND  rd, rs1, rs2` (R)
- `ANDI rd, rs1, imm` (I)

Testing logical instructions is simple as well: operate equivalent bit operations on the parallel data in the golden model and compare the results with the commission from the DUT.

C. Shifting

The Shifting operations comprise:

- `SLL  rd, rs1, rs2` (R)
- `SLLI rd, rs1, shamt` (I)
- `SRL  rd, rs1, rs2` (R)
- `SRLI rd, rs1, shamt` (I)
- `SRA  rd, rs1, rs2` (R)
- `SRAI rd, rs1, shamt` (I)

The shifting operations will only mask the least five bits of the operand. That is, if the second operand is b100000 (32), the shifting operation will simply do nothing, since the least five bits are all zeros; if rs2/shamt is b100011 (35), the shifting operations will only shift rs1 by 3 bit. Therefore, before we compute the results of these instructions, we use 0x1F to mask the value in rs2 or the shift amount, and then shift the value in rs1 with the masked value.

D. Comparison

The comparing operations are

- `SLT  rd, rs1, rs2` (R)
- `SLTI rd, rs1, imm` (I)
- `SLTU rd, rs1, rs2` (R)
- `SLTIU     rd, rs1, imm` (I)

Since there are signed and unsigned comparison, we will test the negative number comparison for SLT/SLTI and comparison with zero for SLTU/SLTIU.

7. Load/Store Operations

The load instructions include:

- `LB   rd, rs1, imm` (I)
- `LH   rd, rs1, imm` (I)
- `LW   rd, rs1, imm` (I)

- `LBU  rd, rs1, imm` (I)
- `LHU  rd, rs1, imm` (I)

The behavior of the load operation is the same as the arithmetic/logical operation. For checking the commission of the load instruction, the write-back checker will monitor the *WB_en* signals, checking if the incoming *WB_data* is identical to the corresponding *WB_target* index in golden model register file.

The store instructions are:

- `SB   rs1, rs2, imm` (S)
- `SH   rs1, rs2, imm` (S)
- `SW   rs1, rs2, imm` (S)

For store operation, we just need to monitor *mem_store_en_o* and check if the incoming *mem_store_type_o*, *mem_store_addr_o*, and *mem_store_value_o* are identical to the computation of the corresponding instructions in the golden model. Since the store operation will be committed in order, the commission order of load/store instructions should be identical to the order of the issuing order.

The number of load/store instructions to commit is crucial too. Since there are only one read port and one write port in the memory module, only one store and one load can be committed each cycle. The pipeline should never commit two load or two store in one cycle.

8. Data/Memory Dependency Handling

There are two kinds of dependency to be dealt with: data register dependency and memory load/store dependency.

A. Data Dependency

To test the data dependency, we could generate a sequence of addition instructions. Even though the destination registers in the instructions cannot be the same, we can let the destination register of precedent instruction always be one of the operands of the succeeding instructions, such that all the instructions are dependent on the previous ones.

```
ADD R2, R1, 0
ADD R3, R2, 0
…
ADD R31, R30, 0
=> only one write-back each cycle
```

B. Memory Dependency (Bypass Policy)

The memory dependency is the case that a sequence of load/store instructions access the same memory address, especially for load after store case. There are several data type we need to test, as shown below.

I. Word to Word

```
SW R1, 0x8000
```

```
LW R2, 0x8000
=> LW write-back word (write into R2)
   must be the same as SW word (read from R1)
```
II. Word to Half
```
SW R1, 0x8000
LH R2, 0x8000
=> LH write-back halfword (write into R2)
   must be the same as upper SW halfword (read from R1)
   with the correct sign extension
```
III. Word to Byte
```
SW R1, 0x8000
LB R2, 0x8000
=> LB write-back byte (write into R2)
   must be the same as leftmost SW byte (read from R1)
   with the correct sign extension
```
IV. Half to Word
```
SH R1, 0x8000
LW R2, 0x8000
=> The upper LW write-back halfword (write into R2)
   must be the same as SH halfword (read from R1)
```
V. Half to Half
```
SH R1, 0x8000
LH R2, 0x8000
=> The LH write-back halfword (write into R2)
   must be the same as SH halfword (read from R1)
```
VI. Half to Byte
```
SH R1, 0x8000
LB R2, 0x8000
=> The LB write-back byte (write into R2)
   must be the same as upper SB halfword (read from R1)
```
VII. Byte to Word
```
SB R1, 0x8000
LW R2, 0x8000
=> The LW leftmost write-back word (write into R2)
   must be the same as SB byte (read from R1)
```
VIII. Byte to Half
```
SB R1, 0x8000
LH R2, 0x8000
=> The LH upper write-back halfword (write into R2)
```

```
   must be the same as SB byte (read from R1)
```

IX. Byte to Byte

```
SB R1, 0x8000
LB R2, 0x8000
=> The LB write-back byte (write into R2)
   must be the same as SB byte (read from R1)
```

9. Flush Due To Violations

The violation occur when there are more than one load instructions data need to bypass, since the load instruction will read from memory immediately after issue to LSQ, and we only bypass the store data to the first matched load, the data is incorrect if there exist second matched load, violation occur.

If we receive *flush_en* signal, we need to check if the returned *flush_PC* is point to the second matched load instruciton.

```
SW R1, 0x8000
...
LW R2, 0x8000
...
LW R3, 0x8000
=> The second LW instruction must be flushed because
   the data content is incorrect
```

10. Jump

The jump instruction is:

- `JAL   rd, imm`   (U)
- `JR    rd, imm`   (U)

When the jump instruction is issued, the target PC will be computed and returned to the testbench. The returned PC value can be checked by the checker of the golden model. At the same time, *branch_PC* is used as jump_flush_PC and should be returned as well to indicate the jump instruction that causes the jump. The offset for the jump function is signed. That is, the target PC could be some PC which has been executed before.

```
IF JAL THEN store PC+1 to r0 AND jump to PC+imm
=> MONITOR jump_en, jump_PC and branch_PC; CHECK r0
IF JR THEN jump to PC in r0
=> MONITOR jump_en, jump_PC and branch_PC
```

11. Branch And Misprediction

The branch instructions are shown as follows:

- `BEQ  rs1, rs2, imm` (I)
- `BNE  rs1, rs2, imm` (I)
- `BLT  rs1, rs2, imm` (I)
- `BGE  rs1, rs2, imm` (I)
- `BLTU rs1, rs2, imm` (I)
- `BGEU rs1, rs2, imm` (I)

In our design, the prediction is assumed always not taken. That is, the next instruction sent to the DUT after the branch instruction is always the instruction with the branch's PC+1. The way to verify branch operations is to monitor the *branch_en* signal, and then check if the incoming *branch_PC* value is identical to the results computed by the golden model. Furthermore, *jump_PC* is used here as well to indicate the branch PC which causes the DUT to flush for the misprediction.

Since the prediction process is simplified, the branch instruction should always take effect when the results of the comparison are true. If a branch should take effect but it does not, this fault should be detected when the consecutive instructions are committed without raising the branching flag. On the other hand, if a branch should not take effect but the branching is raised, the error branch instruction can be identified by the *flush_PC*.

```
IF BEQ && (rs1 == rs2) THEN branch to PC+imm
IF BNE && (rs1 != rs2) THEN branch to PC+imm
IF BLT && (rs1 < rs2) THEN branch to PC+imm
IF BGE && (rs1 >= rs2) THEN branch to PC+imm
IF BLTU && (rs1 < rs2) THEN branch to PC+imm
IF BGEU && (rs1 >= rs2) THEN branch to PC+imm
(ALL) => MONITOR branch_en, branch_PC and jump_PC
```

12. Correctness With Empty Instructions (Halt)

The input instructions could be given with valid bit 0, which indicates that there are currently no valid inputs to be executed. In that case, the pipelining should be correct as well. To test it, the testbench could pause sending valid instructions for some cycles or until the current instructions in the ROB are all committed.

```
RANDOMLY instruction valid = 0
```

13. Transaction Generation

The instructions are categorized into three types: arithmetic/logical, load/store, and jump/branch; the probability for each type could be set by configuration, including the probability for no instruction. In generating the testing instructions, after the types of the instructions are chosen, the instruction formats will then be decided according to the types. After the components of the instructions are generated randomly, the corresponding parts are picked to form the instructions following the instruction formats. At the same time of the construction of the instructions, the corresponding PC value will be incremented by one as our definition.

14. Overall Strategy

At the first stage, we verify the reset function. Then we will verify the functionality of the ALUs. we will generate test input instructions of addition/subtraction, logical, shifting and comparison. We verify these instructions along with commission order and numbers. At this stage, we use totally different registers for each instructions, such that there should be no dependency among the instructions and two instructions should be committed every cycle.

After the correctness of the arithmetic/logical instructions, and commission order and numbers is verified, we then verify the handling of data dependency of the instructions. Since we test data dependency handling with register dependency in the consecutive instructions, the ROB could be full after several cycles. Therefore, handling of fullness of the ROB can be verified at the same time.

At the next stage, we will check memory load/store instructions. similar to the arithmetic/logical instructions, we test load/store without register/address dependency, verifying the correctness of the instructions themselves. Then the memory dependency handling will be testified. There are several conditions discussed above to test. Once the load/store instructions are validated, we can mix the arithmetic/logical and the load/store instructions to test the design.

Jump and branch instructions are the last kinds of the instructions to test. The pipeline should function well even with continual jump or branch. After jump and branch are verified, we finally combine all the instructions in the random test validate the whole design.

The overall steps can be summarized as follows:

1. Test reset
2. Test addition first along with commission order and halt without dependency
3. Test other arithmetic instructions
4. Test logical instructions
5. Test shifting instructions
6. Test comparison instructions
7. Test data register dependency and mixed arithmetic/logical instructions
8. Test load/store instructions
9. Test memory dependency and mixed load/store instructions
10. Test mixed arithmetic/logical and load/store instructions
11. Test the jump instruction
12. Test branch instructions
13. Test mixed instruction and random conditions

VI. Performance Estimates

In this section, we estimate our design in terms of the clock frequency. Among our four-stage design, we estimate the critical path is when a “store” is determined to be committed as the second committed instruction in a cycle. Then, the entry of the “store” is returned to LSQ. LSQ sees the enable signal and knows it is a “store” will be committed. So this “store” will search from its location to “tail” in LSQ, for a potentially matched-address “load”. If there is a match, a bypass will begin and for the second or more matched “load”, a violation will be notified.

Pseudo Code	Gates Usage Estimation

if (head + 1 > 31)
  head2 = head+1-32;
else
  head2 = head+1
	Add_5
	Comparator_5
	MUX_2
	Sub_5

if ((D[head] && op_data[head] == store) ||
 (D[head] && op_data[head] != store && D[head2] &&
op_data[head2] == store))
  store_commit_valid_o = 1;
else
  store_commit_valid_o = 0;
	MUX_32_7
	Comparator_7
	And_2
	MUX_2

for (int iter = 0 ; iter < 30 ; iter++)
 if(type_cam_r2_data[store_commit_entry_i] == StoreWord);
else if(type_cam_r2_data[store_commit_entry_i] == StoreHalf);
else if(type_cam_r2_data[store_commit_entry_i] == StoreByte) ...
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2

if (av_cam_data[i] && type_cam_data[i] == LoadWord);
else if (av_cam_data[i] && type_cam_data[i] == LoadHalf);
if (av_cam_data[i] && type_cam_data[i] == LoadByte) ...
	MUX_32_3
	Comparator_3
	MUX_2
	And_2
	MUX_32_3
	Comparator_3
	And_2
	MUX_2

	MUX_32_3 Comparator_3 And_2 MUX_2

if (addr_cam_data[i] == addr_cam_r1_data)
    search_valid1[i] = 1'b1;
  else
    search_valid1[i] = 1'b0;
	MUX_2
	MUX_32_5
	Comparator_5
	Decoder_5_to_32
	FF_1

priority casez (search_valid1) ...	Priority_Encoder 32_1

if(|search_valid1) search_valid_o = 1;
 else search_valid_o = 0;
	Or_32

if(search_valid_o)
if(type_cam_r2_data == StoreWord)
else if (type_cam_data[search_index_o] == LoadHalf)
else if (type_cam_data[search_index_o] == LoadByte)
if (addr_cam_data[search_index_o] == addr_cam_r1_data)
    search_type_o = Byte2Byte;
  else
    search_type_o = None;
	MUX_2
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3

case(search_type_o) Word2Word:	And_3

for (int iter2 = 0 ; iter2 < 29 ; iter2++)
if(type_cam_r2_data[store_commit_entry_i] == StoreWord);
else if(type_cam_r2_data[store_commit_entry_i] == StoreHalf);
else if(type_cam_r2_data[store_commit_entry_i] == StoreByte)
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2
	MUX_32_3
	Comparator_3
	MUX_2

if (av_cam_data[j] && type_cam_data[j] == LoadWord);
else if (av_cam_data[j] && type_cam_data[j] == LoadHalf);
else if (av_cam_data[j] && type_cam_data[j] == LoadByte);
if (addr_cam_data[j] == addr_cam_r1_data)
    search_valid2[j] = 1'b1;
  else
    search_valid2[j] = 1'b0;
	MUX_32_3
	Comparator_3
	MUX_2
	And_2
	MUX_32_3
	Comparator_3
	And_2

	MUX_2 MUX_32_3 Comparator_3 And_2 MUX_2 Decoder_5_to_32 FF_1
case(search_valid2)	And_3

if(|search_valid2) flush_valid_o = 1;
  else flush_valid_o = 0;
	Or_32

VII. Area Estimates

In our design, we estimate the area of the proposed design unit by unit. Mainly, we list the most area-consuming components.

Subunit	Contents
Register File	1056 FFs, 8 x 32-to-1 MUX
Reorder Buffer	2304 FFs, 30 x 1-to-32 decoder, 60 x 32-to-1 MUX, 2 x 32-to-1 priority encoder
Load/Store Queue	5280 FFs, 22 x 1-to-32 decoder, 33 x 32-to-1 MUX, 2 x 32-to-1 priority encoder
ALU	4 x 32x32 adders, 6 x 32x32 comparators, 5 32x32 shifters

VIII. Bugs And Coverage

IX. References

[1] http://compas.cs.stonybrook.edu/course/cse502-s13/lectures/cse502-L9-ooo-memory.pdf
[2] http://www.ece.umd.edu/~blj/RiSC/RiSC-oo.1.pdf
[3]http://www.cs.columbia.edu/~simha/teaching/4340_fa14/lectures/Unit5.pdf

X. Document Revision History