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

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.

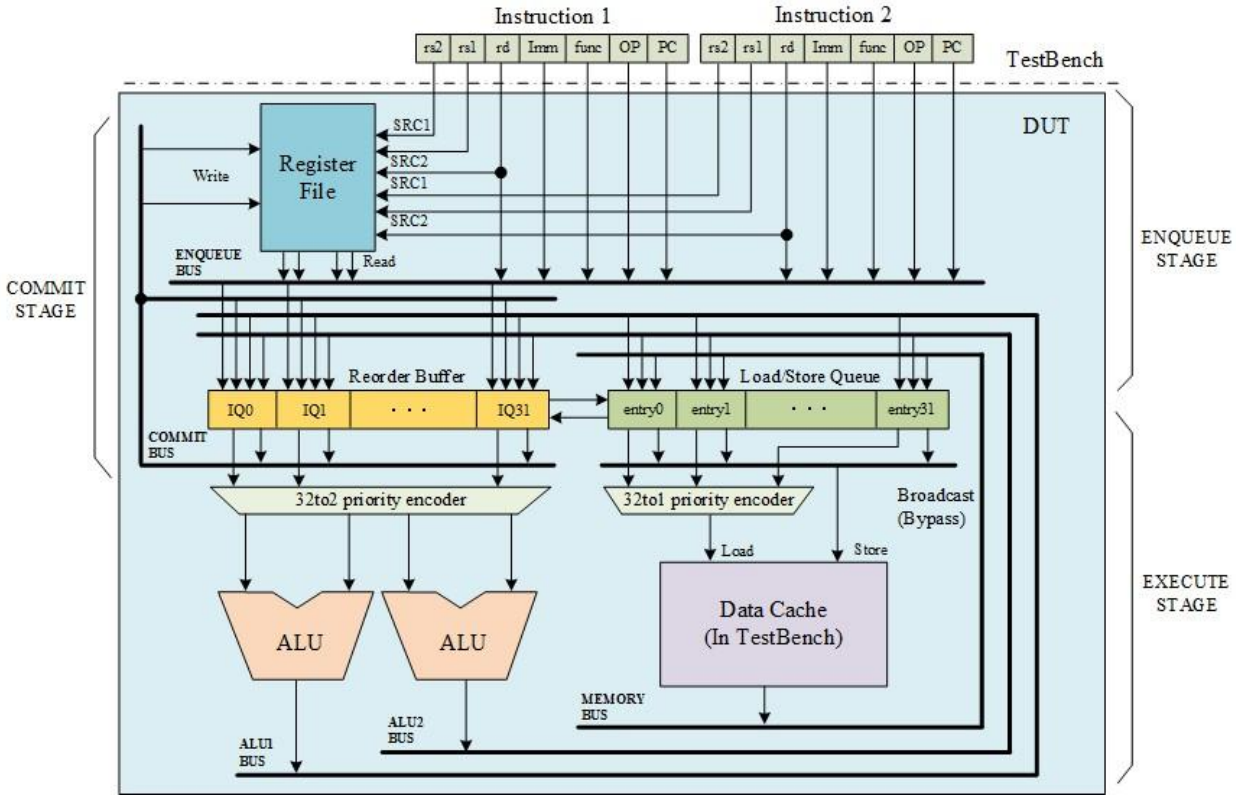

Figure 1. Overview of the system

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.

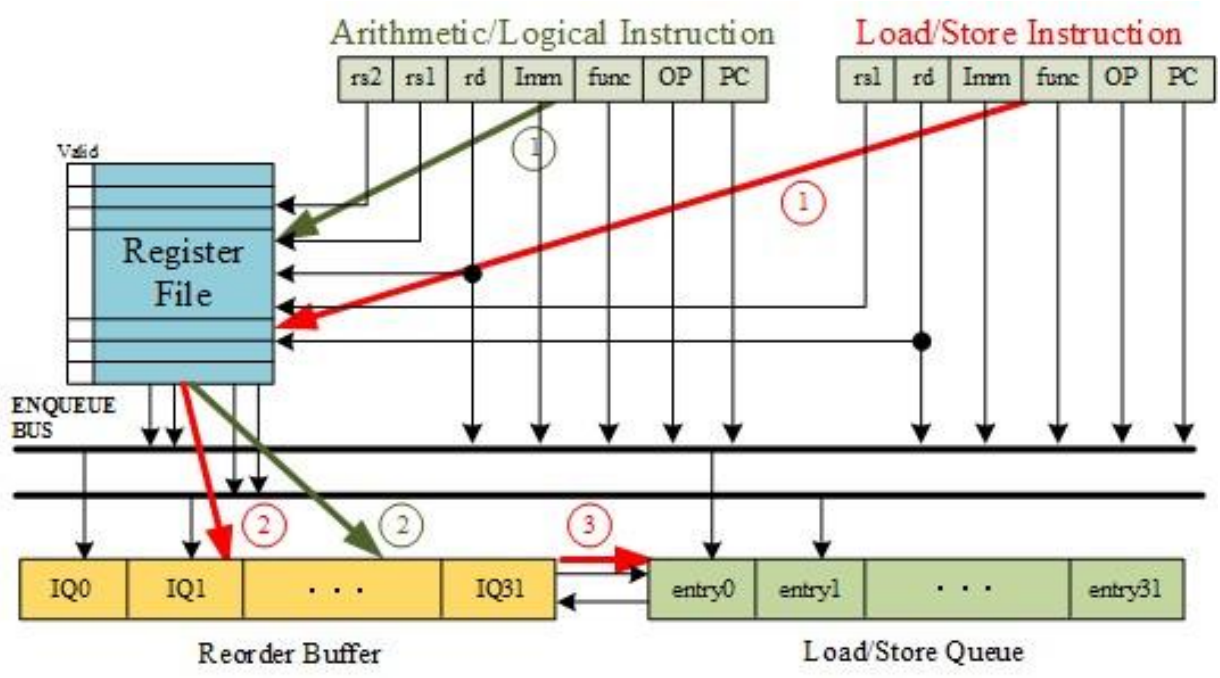

Figure 2. Instruction flow for enqueue stage

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.

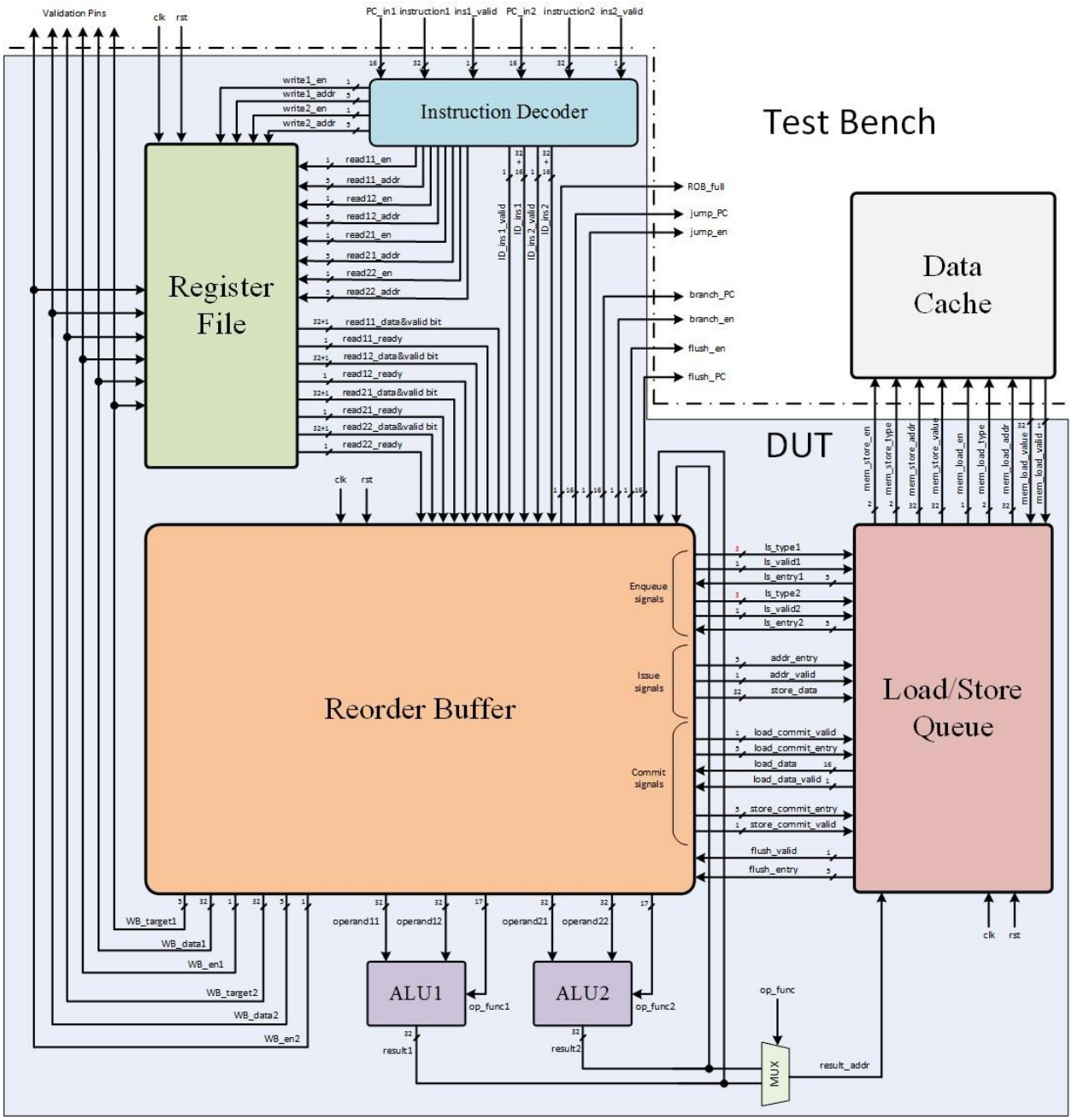

Figure 3: DUT Top-level diagram

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

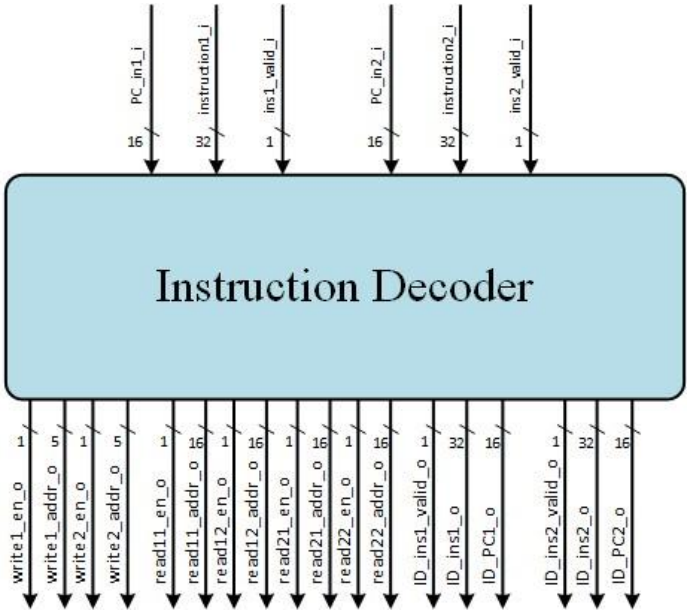

Figure 4: Instruction Decoder interface

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

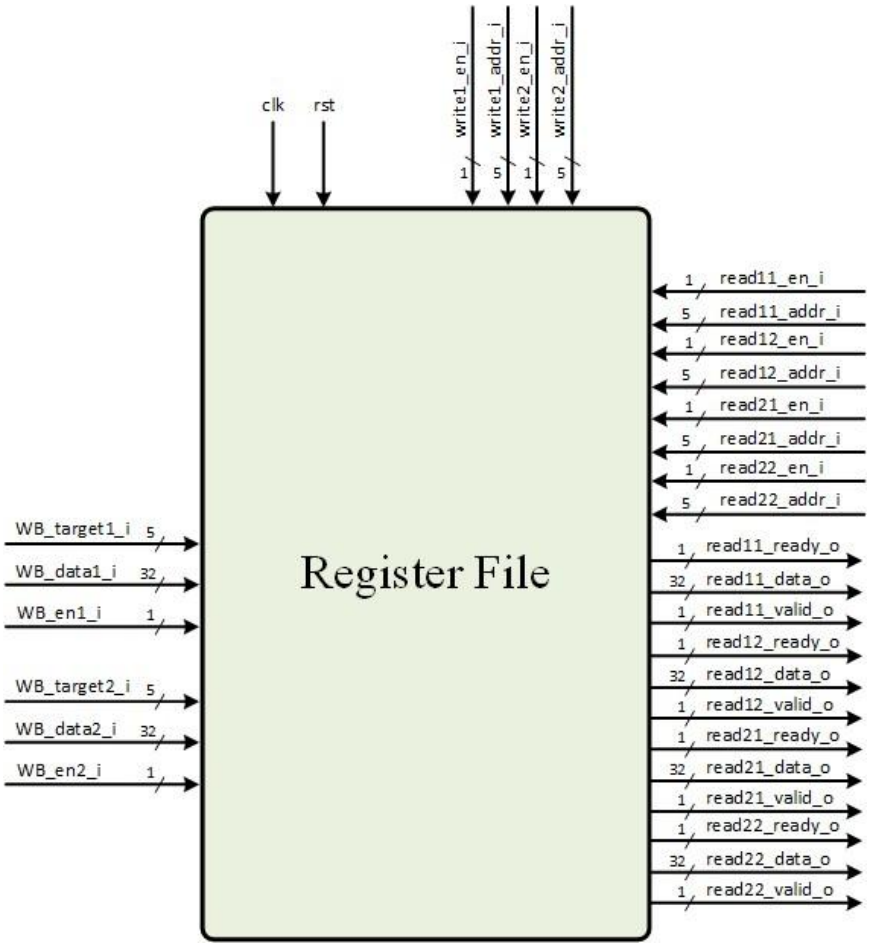

Figure 5: Register File interface

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

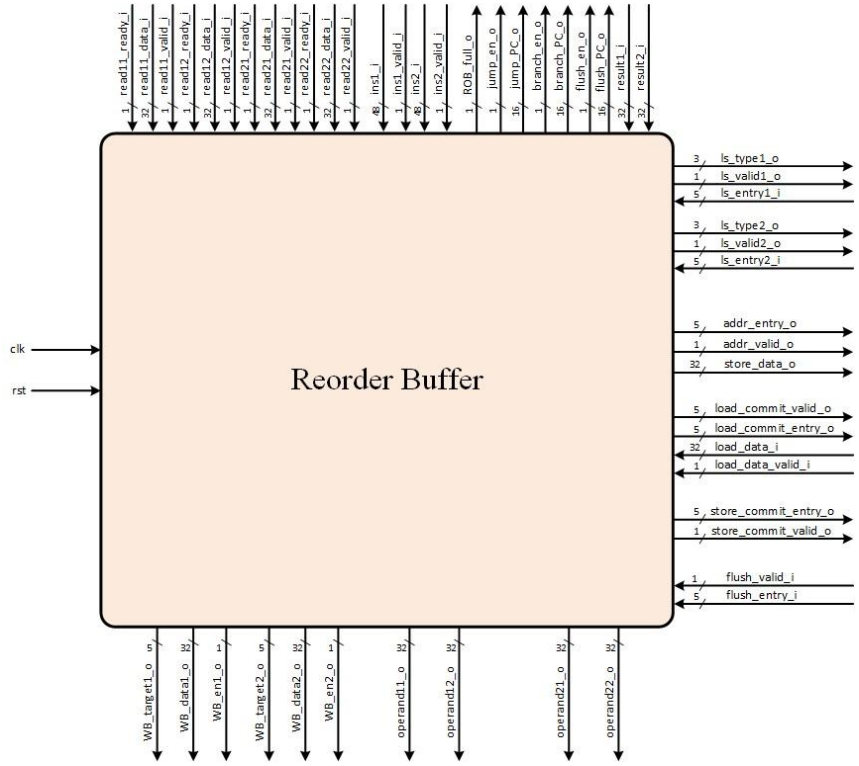

Figure 6: Reorder Buffer interface

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

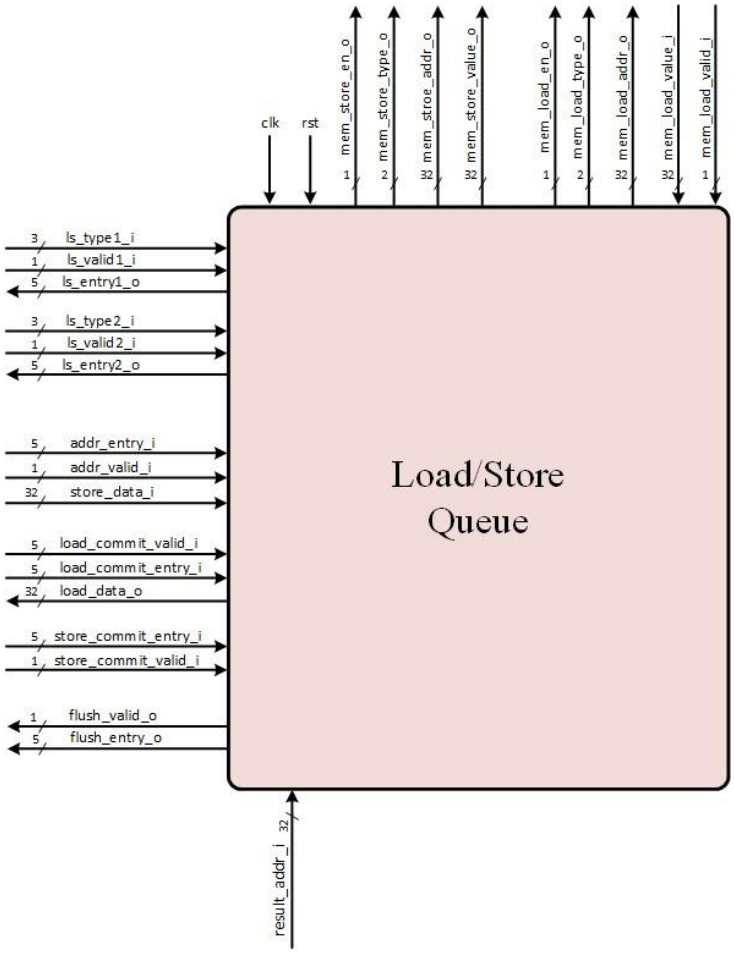

Figure 7: Load/Store Queue interface

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

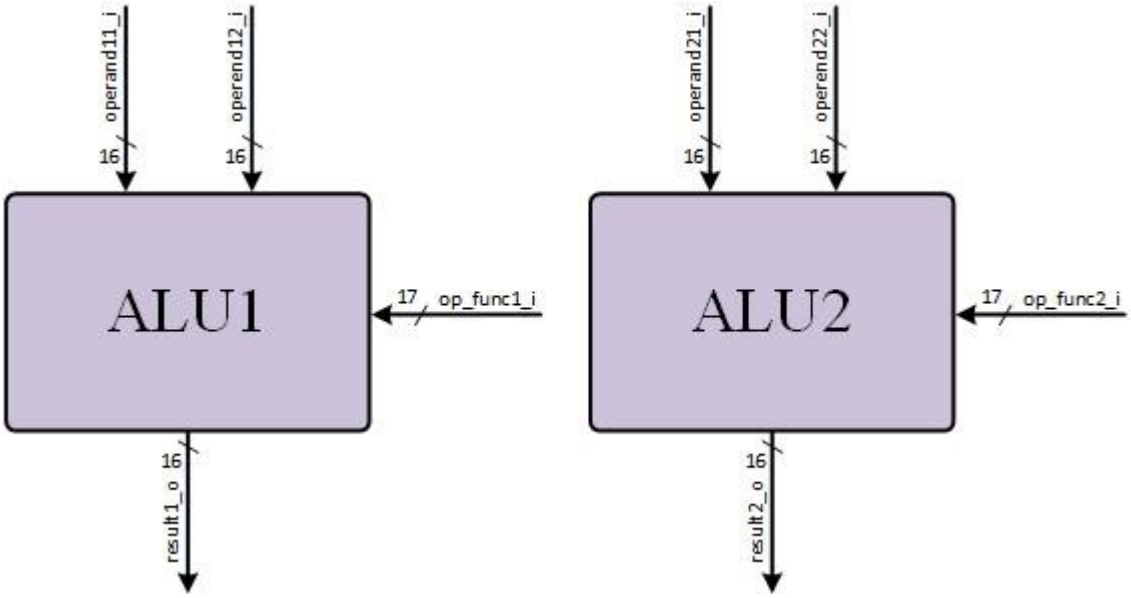

Figure 8: ALUs interface

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.

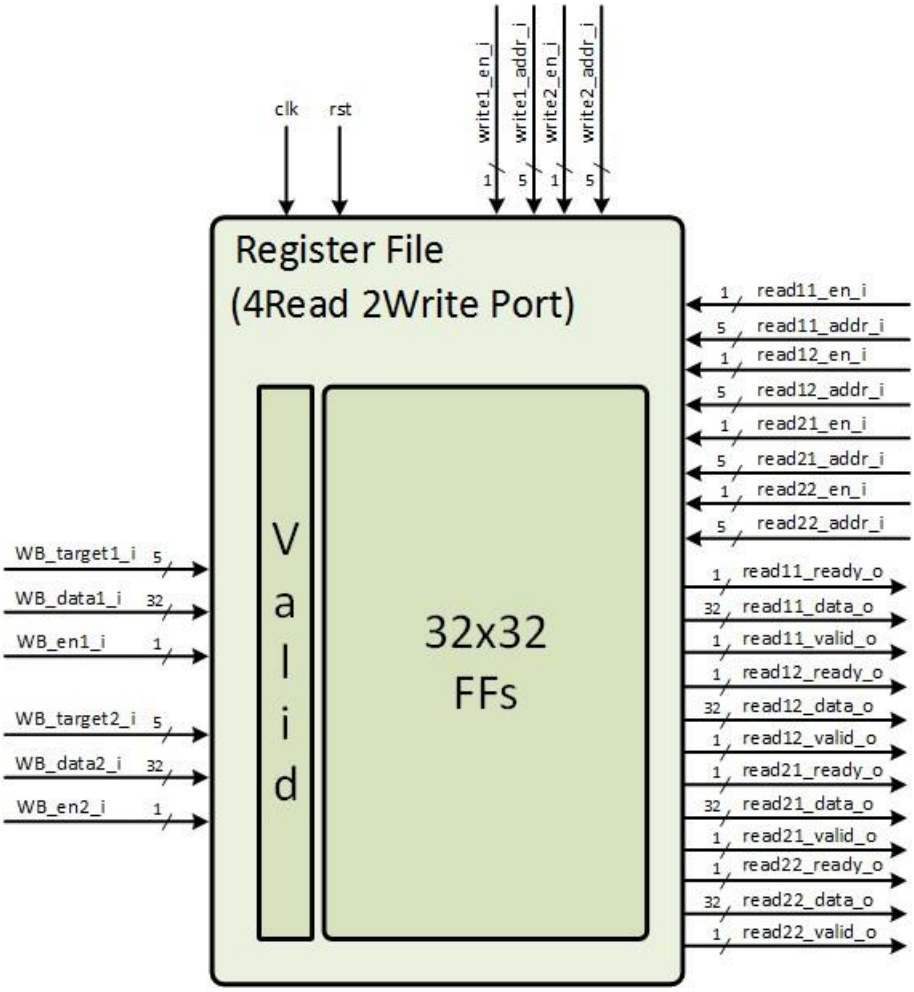

Figure 9: Register File subunit diagram

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.

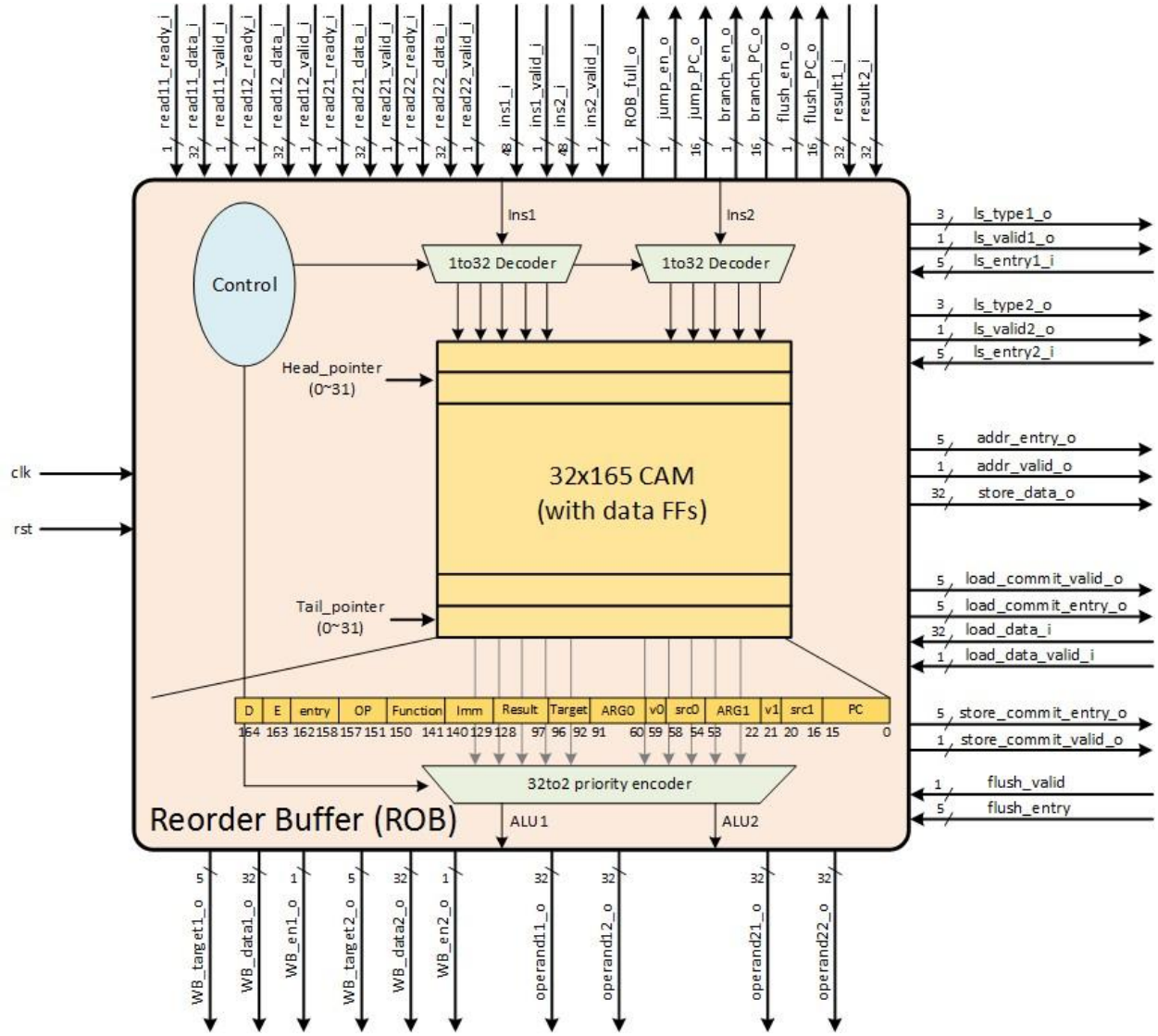

Figure 10: Reorder Buffer subunit diagram

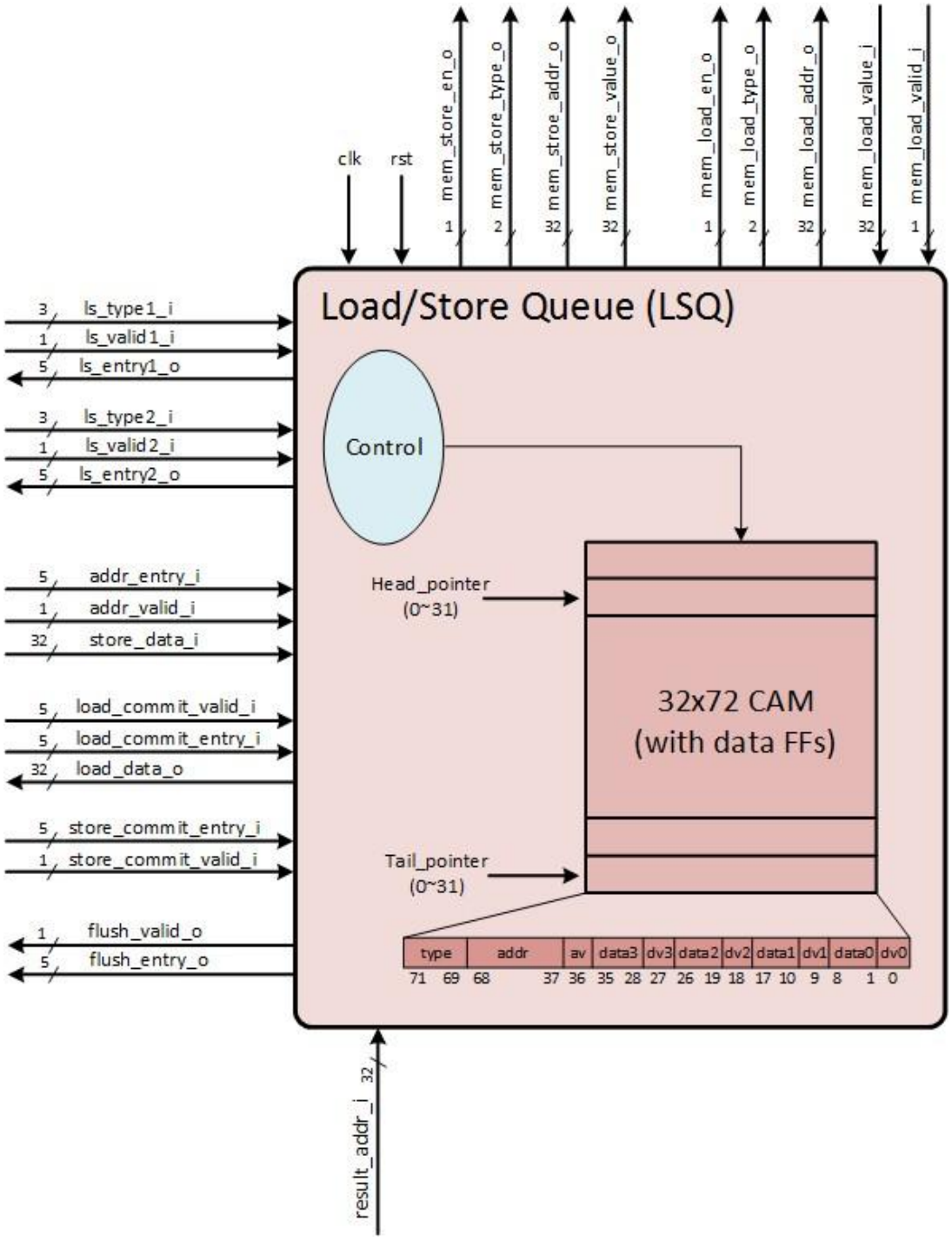
clk
rst
mem_store_en_o
mem_store_type_o
mem_stroe_addr_o
mem_store_value_o
mem_load_en_o
mem_load_type_o
mem_load_addr_o
mem_load_value_i
mem_load_valid_i
1
2
32
32
1
2
32
32
1
Load/Store Queue (LSQ)
3 ls_type1_i
1 ls_valid1_i
5 ls_entry1_o
3 ls_type2_i
1 ls_valid2_i
5 ls_entry2_o
Control
5 addr_entry_i
1 addr_valid_i
32 store_data_i
Head_pointer
(0~31)
32x72 CAM
(with data FFs)
5 load_commit_valid_i
5 load_commit_entry_i
32 load_data_o
5 store_commit_entry_i
1 store_commit_valid_i
Tail_pointer
(0~31)
1 flush_valid_o
5 flush_entry_o
type
addr
av
data3
dv3
data2
dv2
data1
dv1
data0
dv0
71 69 68 37 36 35 28 27 26 19 18 17 10 9 8 1 0
result_addr_i 32

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.

Reference

- • RISC-V BOOM (Berkeley Out-of-Order Machine)
- Rocket Chip + BOOM Integration for RISC-V OoO Design
- CVA6 (formerly Ariane) - RISC-V Out-of-Order Core
- OpenPiton + RISC-V Out-of-Order Extensions
- OpenSPL: An Open-Source OoO RISC-V Processor for Instruction-Level Parallelism Studies