ARCHITECTURE OVERVIEW OF I-CLASS PROCESSOR

INTRODUCTION

This documents includes information pertaining primarily to architecuteral overview of I-Class processor of Shakti family.

I Class or Industrial Class of processor aimed at performance seeking commercial grade applications in mobile and desktop environments. The processor has been designed using Bluespec System Verilog(BSV). The design has been modularized to enable a plug-and-play kind of environment that lets us to modify the parts of the design with ease. Heavy parameterization is also incorporated into the design to facilitate quick design space exploration.

FEATURES

- **64-bit, 8 stage pipeline.**

I Class is 64-bit variant of shakti processor and features a pipeline of depth of 8 stages. The 8 stages of the pipeline being Fetch, Decode, Rename(Map), Wakeup, Select, Drive, Execute, Commit. Each of the pipeline stages take utmost a single cycle to execute.

- **Based on RISC-V ISA.**

I Class processor supports all standard instructions based on RISC-V ISA.

  - RV32I and RV64I (Integer Base Instructions).
  - RV32M and RV32M (Multiplier and Divider instructions).

- **Out-of-Order execution.**

Each instruction has its operand registers renamed in-order but are selected for issue to execution units Out-of-order. Finally, instructions commit in-order.

- **Explicit Register Renaming**:

Register Renaming is done through merged register file approach. Merged register file stores both the architectural register values and speculated values. The number of architectural registers are 32 and the number of physical registers are 64. A buffer (register alias table) that maintains the map from architectural registers to physical registers.

- **Tournament Branch Predictor.**

The type of Branch Predictor used speculative branching is Tournament Branch predictor. Tournament Branch Predictor has Bimodal and Global predictors contend between each other.

- **Functional Units.**

The Functional Units are parameterised in the design. Current design uses 2 Arithmetic and Logical Units, 1 Branch Unit, a Load Store Unit.

- **L1 Cache**

A fully parameterized I-Cache and D-Cache use physical address for both index and tag (PIPT). Each cache is of size of 32KB. The cache is fully parameterised in terms of the size of the cache, associativity, number of blocks within a cache line, number of sets, etc. The caches are implemented using BRAMs provided in the Bluespec library. These BRAMs have a direct correlation to the FPGA based Block RAMs, thus making translation to an FPGA based design easy.

- **Dual Issue**

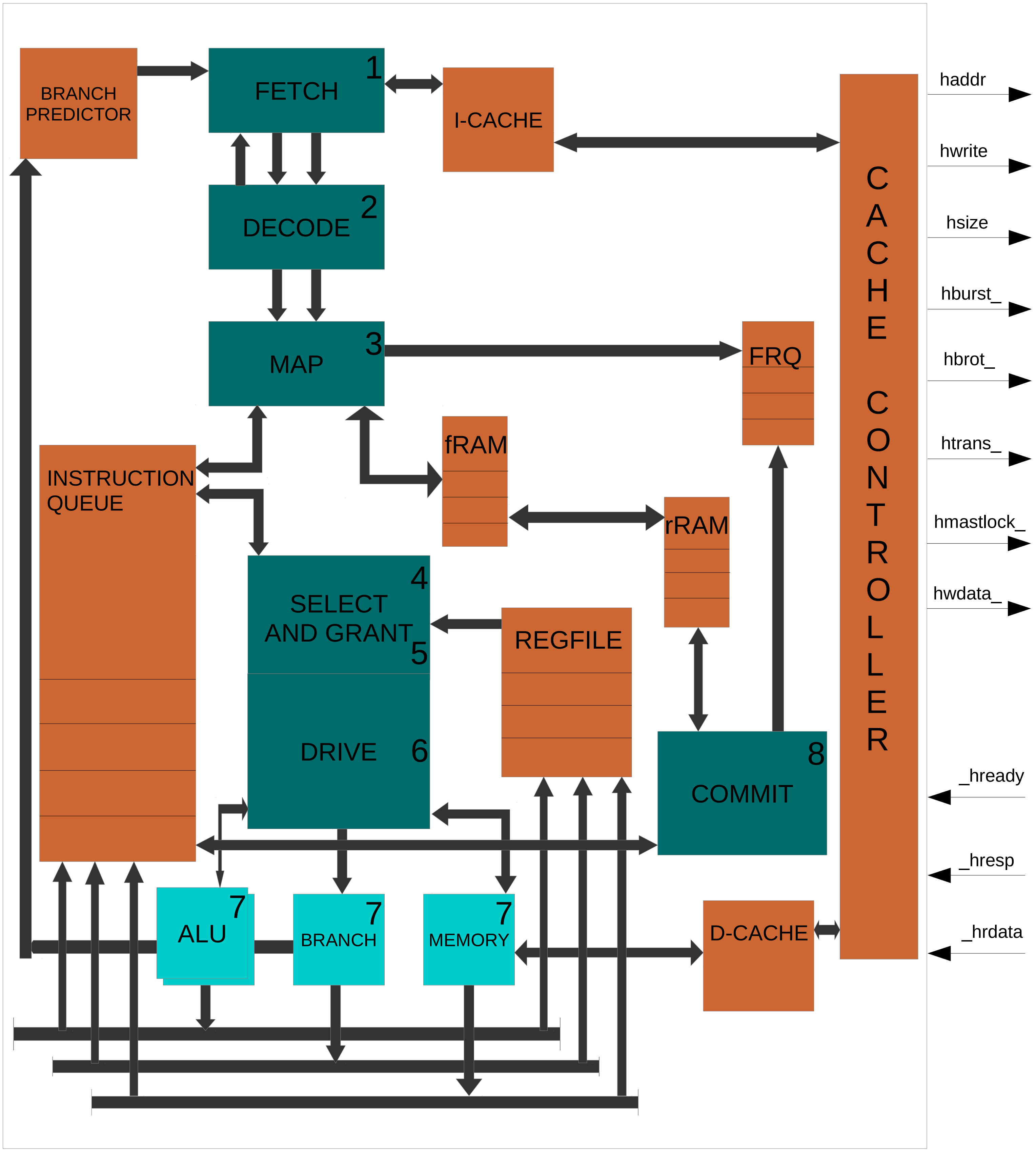
This variant of I Class processor supports Dual Issue. This variant supports no hardware for address translation or memory protection.

LIST OF SUPPORTED RISC-V INSTRUCTIONS

1	LUI	arithmetic_unit.bsv
2	AUIPC	arithmetic_unit.bsv
3	JAL	branch_unit.bsv
4	JALR	branch_unit.bsv
5	BEQ	branch_unit.bsv
6	BNE	branch_unit.bsv
7	BLT	branch_unit.bsv
8	BGE	branch_unit.bsv
9	BLTU	branch_unit.bsv
10	BGEU	branch_unit.bsv
11	LB	memory_unit.bsv
12	LH	memory_unit.bsv
13	LW	memory_unit.bsv
14	LBU	memory_unit.bsv
15	LHU	memory_unit.bsv
16	LWU	memory_unit.bsv
17	LD	memory_unit.bsv
18	SB	memory_unit.bsv
19	SH	memory_unit.bsv
20	SW	memory_unit.bsv
21	SD	memory_unit.bsv
22	ADDI	arithmetic_unit.bsv

23	ADDIW	arithmetic_unit.bsv
24	SLTI	arithmetic_unit.bsv
25	SLTIU	arithmetic_unit.bsv
26	XORI	arithmetic_unit.bsv
27	ORI	arithmetic_unit.bsv
28	ANDI	arithmetic_unit.bsv
29	SLLI	arithmetic_unit.bsv
30	SLLIW	arithmetic_unit.bsv
31	SRLI	arithmetic_unit.bsv
32	SRLIW	arithmetic_unit.bsv
33	SRAI	arithmetic_unit.bsv
34	SRAIW	arithmetic_unit.bsv
35	ADD	arithmetic_unit.bsv
36	ADDW	arithmetic_unit.bsv
37	SUB	arithmetic_unit.bsv
38	SUBW	arithmetic_unit.bsv
39	SLL	arithmetic_unit.bsv
40	SLLW	arithmetic_unit.bsv
41	SLT	arithmetic_unit.bsv
42	SLTU	arithmetic_unit.bsv
43	XOR	arithmetic_unit.bsv
44	SRL	arithmetic_unit.bsv
45	SRLW	arithmetic_unit.bsv
46	SRA	arithmetic_unit.bsv
47	SRAW	arithmetic_unit.bsv
48	OR	arithmetic_unit.bsv
49	AND	arithmetic_unit.bsv
50	MUL	arithmetic_unit.bsv
51	MULH	arithmetic_unit.bsv
52	MULSHU	arithmetic_unit.bsv
53	MULHU	arithmetic_unit.bsv

BLOCK DIAGRAM



🟦 Turquoise rectangles indicates modules that carry pipeline stages. The number on top right-corner indicate the position in the pipeline stage.

🟦 Indicates the Functional Units. Multiple squares piled upon each other indicate multiple functional units.

🟧 Orange Rectangle indicates modules that have storage structures and rectangle with horizontal lines indicate storage structures with no combinational logic or rules.

- Arrow Heads indicate the data flow(wires in general). Multiple Arrow heads into a module indicate duplicates of same type of data flow into the module. The overlap of wires is the indication the data flow of a wire is latest when compared to wire underneath it.

Ports from RISC-V_master:

S.No	Port Name	Size	Direction	Description
1	haddr_	32	Output	`address going to the bus`

2	hwrite_	1	Output	`indicate whether read or write operation`
				`(0 for read and 1 for write).`

3	hsize_	3	Output	`size of the byte transfer.`
				`(0=byte, 1=halfword, 2=word.)`

4	hburst_	3	Output	`burst type.`
				`(0=single, 1=incr, 2=wrap4, 3=incr4)`

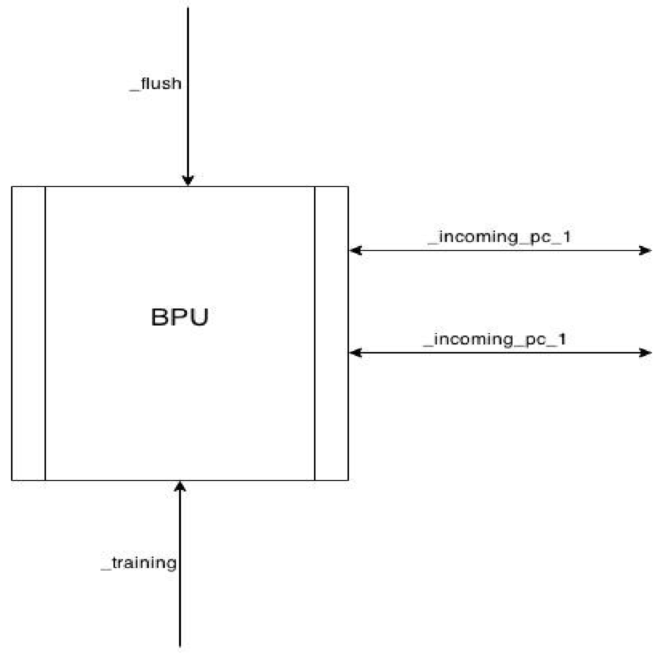
5	hprot_	4	Output	`0th bit=0 opcode fetch. 0th bit=1 data access.`
				`(other bits not required)`

6	htrans_	2	Output	`11 when doing burst transfer.`
				`10 when doing single transfer.`
				`00 when not doing anything.`
				`Default to 00.`

7	hmastlock_	1	Output	`This is highly useful when performing atomic operations`
8	hwdata_	64	Output	`data to be written in case of write operation`
9	_hready	1	Input	`when high indicates the completion of the transfer`
10	_hresp	1	Input	`when low indicates the status as okay. Else ERROR`
11	_hrdata	64	Input	`the 32bit data read from the bus.`

The modularized description of the core follows. The first 8 modular descriptions

MODULAR DESIGN OF BRANCH PREDICTOR.

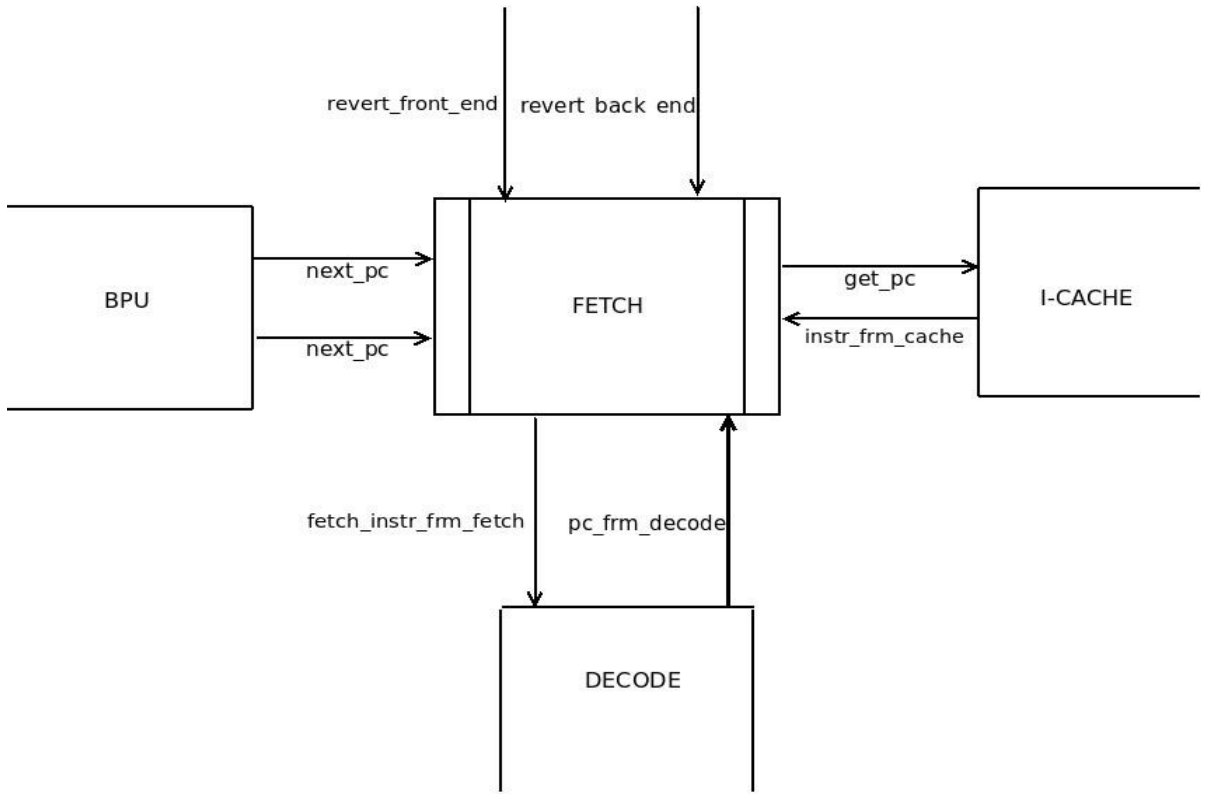


This diagram represents module bpu and it's interaction with other modules . The module bpu has following memory elements :

Memory Element	**Name**	**Type(in BSV) or Fields : Bits**	**Size(**bits**)**
Register	`rg_global_idx(4)`	Bit#(4)	4
Register_array	`rg_select(16)`	Bit#(2)	32
Register_array	`tb_tag(16)`	Bit#(61)	976
Register_array	`tb_bimodal_state(16)`	Bit#(2)	32
Register_array	`tb_global_state(16)`	Bit#(2)	32
Register_array	`tb_branch_addr(16)`	Bit#(64)	1024

The branch predictor being used is a Tournament Branch Predictor. It uses two branch predictor models : bimodal and global branch predictor. Both are 2-bit branch prediction models and have a branch predictor table of length 16 which stores a tag of 60 bits. The Branch predictor table gets the actual Jump data, whether branch taken or not and pc value if taken, as training data from the branch unit. The training data obtained from branch unit is updated in the Branch Target Buffer(BTB) i.e `tb_branch_addr and tb_tag`. The branching decision is trained in two types of branch predictors as mentioned earlier. For every Program Counter(PC) sent by the fetch unit, based on the slot indexed in rg_select, one of the two branch predictors is selected for prediction. Branch predictor module returns predicted PC and prediction status(branch taken or not) to fetch unit.

MODULAR DESIGN OF INSTRUCTION FETCH PHASE



This diagram represents module Prf_fetch and it's interaction with other modules. It includes a FIFO of type **Fetched_instruction. Fetched_instruction** has following four fields:

Memory Element	**Name**	**Data Type(in BSV)**	**Fields: bits**
FIFO	`ff_fetch_0`	Fetched_instruction	thread_id : 0 program_counter : 64 instruction : 32 prediction : 2
FIFO	`ff_fetch_1`	Fetched_instruction_2	thread_id : 0 program_counter : 64 instruction : 32 prediction : 2
Register	`rg_program_counter`	Bit#(64)	64

The fetch module receives `revert_back_end` and `revert_front_end` from the main module. These are basically flush signals which when activated Fetch module will be reverted to initial state and abandons it's current operation. The Fetch module currently in operation is Dual Fetch. In this module the program counter is read from rg_program_counter and the next pc is assumed to be rg_program_counter + 4(as it is dual Fetch). The two instructions are received from instruction-cache as a 64 bit packet. Each set of PC, instruction and prediction status are enqueued into a FIFO. If the prediction of PC in rg_program_counter recorded to be taken, then data enqueued in ff_fetch_1 tagged invalid. This module receives a pair of predicted PCs from Branch Predictor module. If the first of the two PCs is predicted to be taken or PC in rg_program_counter is found to be not aligned with with double word, then rg_program_counter is updated with second of the two predicted PCs from Branch Predictor module. Else rg_program_counter is updated with the other.

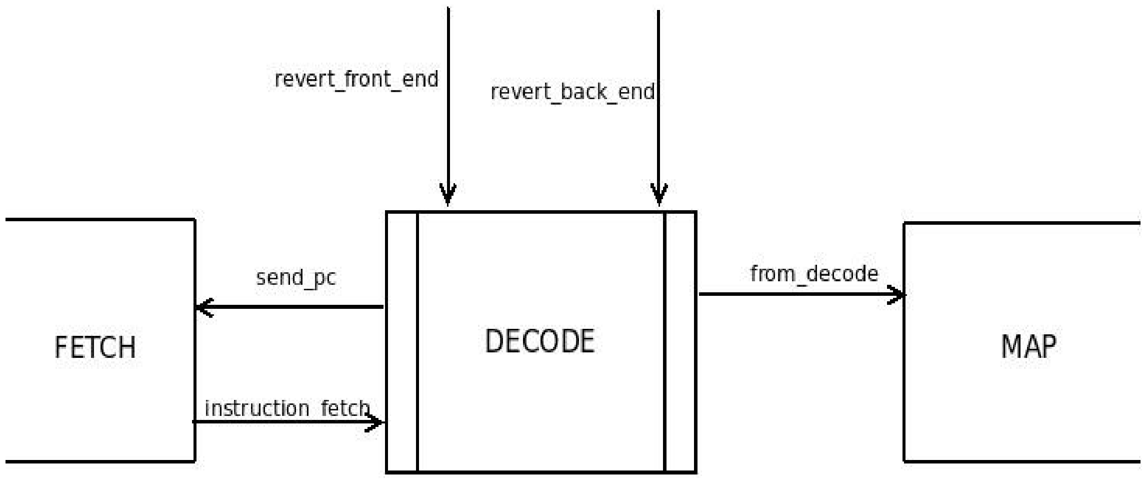
The following are the in-ports of Fetch:

Port	**Type**	**Size**(bits)
incoming_pc_bpu_1	Bpu_packet	33
incoming_pc_bpu_2	Bpu_packet	33
pc_from_decode	Maybe#(Bit#(64))	65
instr_frm_cache	Bit#(64)	64
revert_front_end	Bool	1
revert_back_end	Bool	1

The following are the out-ports of Prf_fetch

Port	**Type**	**Size**(bits)
get_pc	Bit#(64)	64
fetch_instr_frm_fetch	Fetched_instruction	100

MODULAR DESIGN FOR DECODE STAGE.



The above diagram represents a module Prf_decode. It imports Prf_decoder which implements a function(a combinational block) which returns data of **Decoded_info_type.** This function identifies the types and subtypes of instructions for each functional unit. For Memory type instructions Prf_decoder identifies if it is Load instruction or store instruction, for Arithmetic instructions Prf_decoder identifies if it is a single cycle instruction, or a multiply instruction or a division instruction and for a Branch instruction it identifies if it is conditional or unconditional. The fuction also isolates the register addresses and immediate values encoded in the instruction. The data returned by module is enqueued into a FIFO of type Decoded_instruction. **Decoded_instruction** has following fields:

Memory Element	**Name**	**Data Type(in BSV)**	**Fields : bits**
FIFO array	`fifo_decode_packet`	Decode_packet	2 x (valid : 1) 2 x (Decoded Instruction { thread_id : 0 instruction_decoded : 98 prediction : 2 program_counter : 64})

Prf_decode receives `revert_back_end` and `revert_front_end` from the top module which act as flush signals. JAL(an unconditional branch instruction) instructions are executed in the decode stage itself. The module Prf_decode returns Program Counter(PC) which is calculated from immediate for JAL instruction to Prf_Fetch. Then Prf_Decode enqueues information into the FIFO later dequeued by Prf_Map.

The following are the in-ports of Prf_Decode:

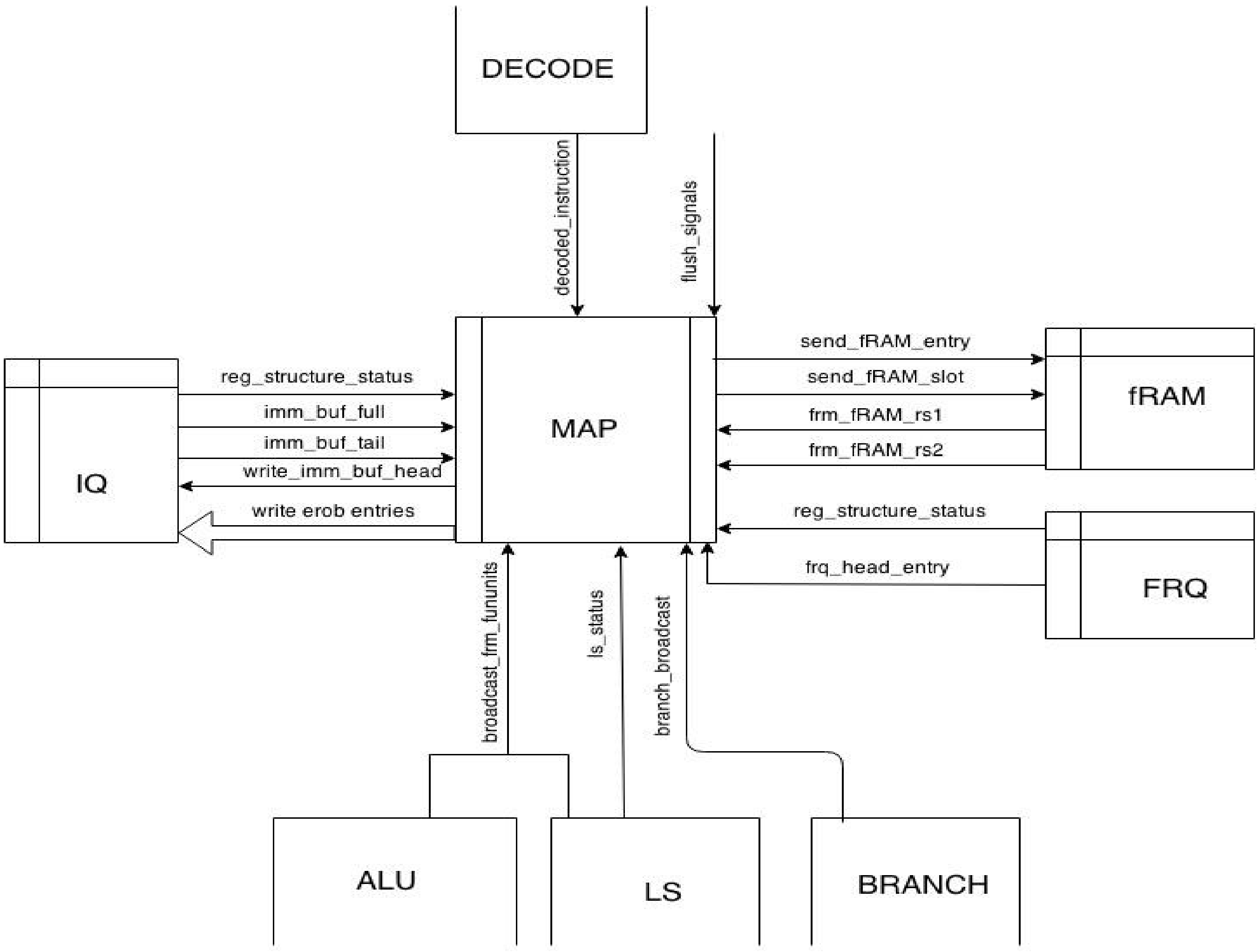
Port	**Type**	**Size**(bits)

`revert_back_end`	Bool	1
`revert_front_end`	Bool	1

The following are the out-ports of Prf_Decode

Port	**Type**	**Size**(bits)
`instruction_fetch`	Fetched_instruction	100
`send_pc`	Bit#(64)	64
`from_decode`	Decoded_instruction	98

MODULAR DESIGN FOR MAP STAGE



The module Prf_MAP does not consist of any storage structures or registers or FIFOs. It gets decoded instruction dequeued from Prf_decode. The destination register address is assigned new value by dequeuing from FRQ at head. Based on the register addresses from decode stage, this module maps source operand addresses from fRAM. When a pair of instructions are dequeued from decode stage, each of the instruction's destination register address is renamed. The registers addresses that are available at the head of the Free Register Queue(FRQ) assigned as instruction destination register.

This module marks an instruction operand ready if there is a broadcast from functional unit to same register or if the value in corresponding register is valid. After collecting the operand information this module fills the Instruction Queue entries at tail.

The following are the in-ports to MAP:

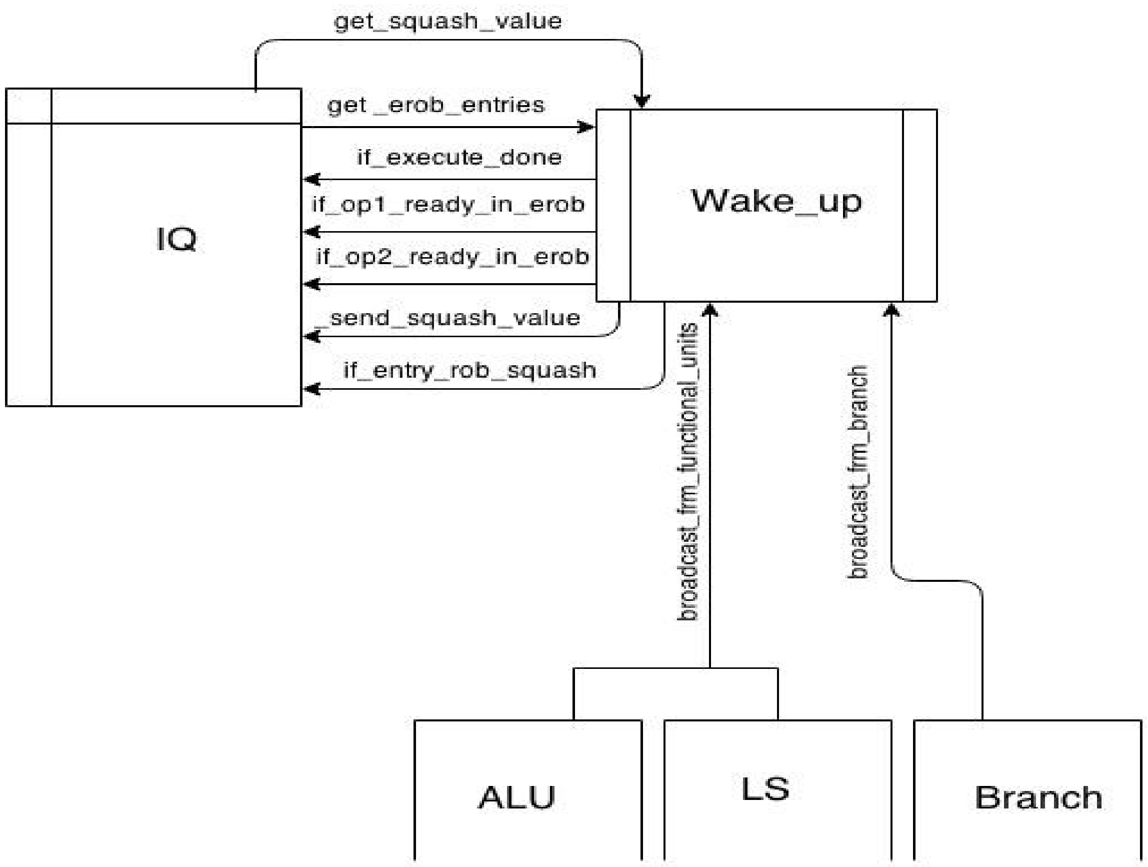
Ports	**Type**	**Size**(bits)
`decoded_instruction`	Decoded_instruction	98
`ls_status`	2 x Bool	2
`register_structure_status`	3 x Bool	3
`flush_signals`	2 x Bool	2
`broadcast_frm_fununits`	Vector#(3, Broadcast_type)	33
`broadcast_branch`	Branch_Broadcast_type	10
`prf_valid`	Vector#(64, Bool)	64
`send_imm_buf_full`	Bool	1
`imm_buf_tail`	Bit#(3)	3
`frm_fRAM_rs1`	Bit#(6)	6
`frm_fRAM_rs2`	Bit#(6)	6
`frq_head_entry`	FRQ_entry	7

Out-ports of Module Prf_MAP:

Port	**Type**	**Size**(Bits)
`send_for_fRAM_slot_rs1`	Bit#(32)	32
`send_for_fRAM_slot_rs1`	Bit#(32)	32
`write_at_tail_imm_buf`	Imm_buf_entry	33
`send_fRAM_slot`	Bit#(5)	5
`send_fRAM_entry`	Bit#(5)	5
`update_frq_head_entry`	Bool	1
`send_entry_rob`	Entry_rob_type	122
`send_entry_rob_op_1_ready`	Bool	1
`send_entry_rob_op_1_ready`	Bool	1
`send_entry_rob_execute_done`	Bool	1
`send_entry_rob_squash`	Bool	1

`send_squash_buf`	Bit#(64)	64
`send_selected_for_exec`	Bool	1
`send_update_rob_tail`	Bool	1
`allocate_load_q`	Bool	1
`allocate_store_q`	Bool	1

MODULAR DIAGRAM FOR WAKE_UP STAGE



The module of Prf_wakeup does not have any storage structures or registers or FIFOs. This module has rule that fires independent of other rules. It receives broadcasted results from the functional units and modify the entries in EROB accordingly. The intended destination register of the broadcasted result is checked with destination register of each entry in ROB. If there is a match then corresponding entry is marked to have finished execution and is ready to commit. The broadcasted destination register is also checked with source operand registers in each entry of EROB. If there is a match then corresponding operands are tagged to be valid and entries that have both the source operands valid are ready for execution.

The following are the in-ports of Prf_wakeup:

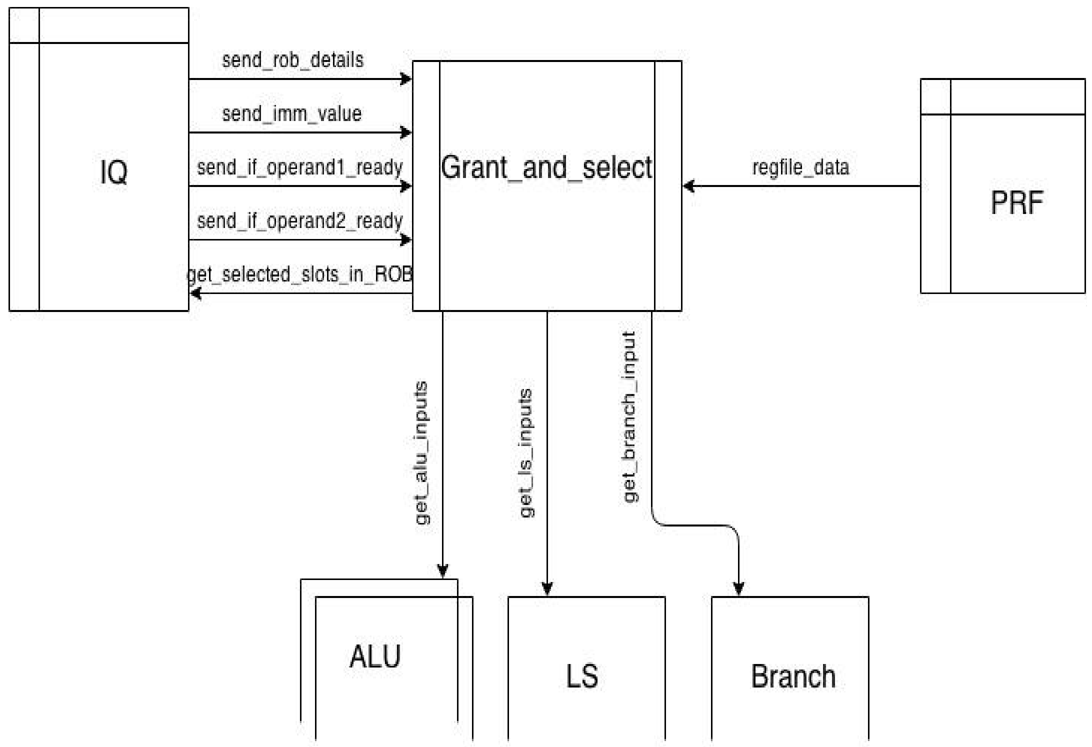
Port	**Type**	**Size**(bits)

`get_erob_entries`	Vector#(16, Entry_rob_type)	1952
`broadcast_of_functional_units`	Vector#(3, Broadcast_type)	33
`broadcast_for_branch`	Branch_broadcast_type	11
`get_squash_pc`	Bit#(64)	64

Out-ports of Prf_wakeup:

Port	**Type**	**Size**(bits)
`if_execute_done`	Vector#(16, Bool)	16
`if_op1_ready_in_erob`	Vector#(16, Bool)	16
`if_op1_ready_in_erob`	Vector#(16, Bool)	16
`if_entry_rob_squash`	Vector#(16, Bool)	16
`send_squash_value`	Bit#(64)	64

MODULAR DIAGRAM FOR GRANT AND SELECT



This module consists of 8 FIFO's. There are 2 sets of four FIFOs each. Each FIFO in a set corresponds to a functional unit(2 ALUs, 1 LS, 1 Branch). Based on the priority encoder each entries and grants obtained from the functional units are selected from the EROB and are enqueued to Data Read FIFOs(one of the two sets of FIFOs). In the next pipeline stage data is dequeued from the FIFOs . The operand values are read from Physical Register File and appended to the dequeued data. The operand values are later read from register file and and enqueued in next set of 4 FIFOs called Payload FIFOs. The data read from Payload FIFOs are sent to functional units as inputs.

FIFOs in Grant and Select:

FIFO	**Type**	**Fields : bits**	**Size(**bits**)**
`ff_alu_data_read_1`	ALU_data_read	alu_op : 5 word_flag : 1 alu_type : 2 imm_valid : 1 op_1 : 6 op_2 : 6 imm_index : 3 dest_op : 6 pc : 64 thread_id : 0	95
`ff_alu_data_read_2`	ALU_data_read	alu_op : 5 word_flag : 1 alu_type : 2 imm_valid : 1 op_1 : 6 op_2 : 6 imm_index : 3 dest_op : 6 pc : 64 thread_id : 0	95
`ff_ls_data_read`	LS_data_read	Base : 6 offset : 3 op_2 : 6 mem_q_index : 4 dest_op : 6 pc : 64 mem_type : 2 mem_size : 3 thread_id : 0	94
`ff_branch_data_read`	Branch_data_read	branch_op : 3 op_1 : 6 op_2 : 6 imm_index : 3 dest_op : 6 program_counter : 64 prediction : 2	90

		thread_id : 0	
ff_alu_payload_1	ALU_payload_type	alu_op : 5 word_flag : 1 alu_type : 2 src_1 : 64 src_2 : 64 dest_op : 6 pc : 64 thread_id : 0	206
ff_alu_payload_2	ALU_payload_type	alu_op : 5 word_flag : 1 alu_type : 2 src_1 : 64 src_2 : 64 dest_op : 6 pc : 64 thread_id : 0	206
ff_ls_payload	LS_payload_type	Base : 6 offset : 3 mem_q_index : 4 dest_op : 6 str_data : 64 mem_size : 3 thread_id : 0	208
ff_branch_payload	Branch_payload_type	src_1 : 64 src_2 : 64 dest_op : 6 imm : 64 program_counter : 64 prediction : 2 thread_id : 0	269

The following are the in-ports of Prf_select_grant:

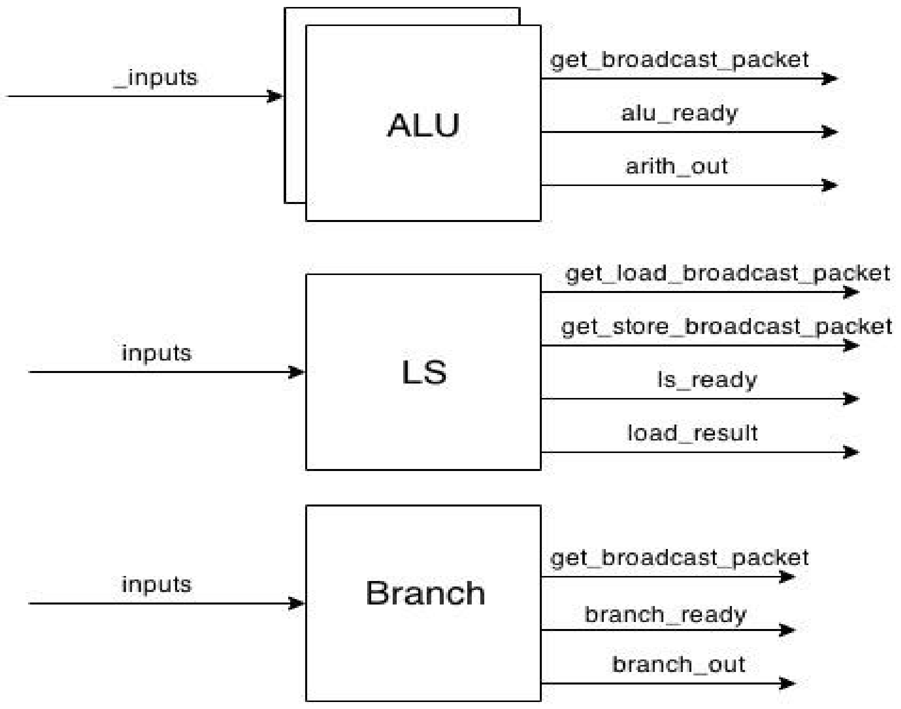
Port	**Type**	**Size**(bits)
get_rob_details	Vector#(16, Entry_rob_type)	1952
get_revert_front_end	Bool	1
get_revert_back_end	Bool	1
get_imm_value	Vector#(8, Imm_buf_entry)	512
get_if_operand1_ready	Vector#(16, 1)	16
get_if_operand2_ready	Vector#(16, 1)	16

The following are the out-ports of Prf_select_grand:

Port	**Type**	**Size**(bits)

`send_ALU_0_inputs`	ALU_payload_type	206
`send_ALU_1_inputs`	ALU_payload_type	206
`send_LS_inputs`	LS_payload_type	208
`send_Branch_input`	Branch_payload_type	269
`send_selected_slots_in_ROB`	Vector#(16, Bool)	16

MODULARIZED DIAGRAM OF FUNCTIONAL UNITS



The three modules shown in the diagram ALU, LS_unit and branch do not have any register arrays. These modules perform execute stage of the pipeline and have huge combinational logic.
The following are the memory elements in ALU:

Memory Element	**Name**	**Type(in BSV) or Fields : Bits**	**Size**(bits)
Register	`rg_dest_address`	Bit#(6)	6

The following are the memory elements in the integer_multiplier_riscv module imported into Prf_alu:

Memory Element	**Name**	**Type(in BSV) or Fields : Bits**	**Size(**bits**)**
FIFO	`ff_stage1`	Stage1:1096 stage1_mul_type : 2 stage1_word_flag : 1 stage1_sign_bit_op1 : 1 stage1_sign_bit_op2 : 1 stage1_extra_pp : 128 thread_id : 0 destination : 6	4225
FIFO	`ff_stage2`	data_computed_stage2:256 vector_pp:2096 stage2_mul_type : 2 stage2_word_flag: 1 stage2_sign_bit_op1:1 stage2_sign_bit_op2:1 stage2_extra_pp:128 thread_id : 0 destination : 6	2491
FIFO	`ff_stage3`	data_from_stage_2 : 256 data_from_stage_3 : 256 stage3_mul_type : 2 stage3_word_flag: 1 stage3_sign_bit_op1:1 stage3_sign_bit_op2:1 stage3_extra_pp:128 thread_id : 0 destination : 6	651
FIFO	`ff_stage4`	final_rb_number : 256 stage4_mul_type : 2 stage4_word_flag: 1 stage4_sign_bit_op1:1 stage4_sign_bit_op2:1 stage4_extra_pp:128 thread_id : 0 destination : 6	395
FIFO	`ff_stage5`	unsigned_mul_output : 128 stage5_mul_type : 2 stage5_word_flag: 1 stage5_sign_bit_op1:1 stage5_sign_bit_op2:1 stage5_extra_pp:128 thread_id : 0	139

Memory Element	Name	Type(in BSV) or Fields : Bits	Size(bits)
		destination : 6	
Register	rg_ready_signal	Bool	1
Register	rg_thread_id	Bit#(0)	0
Register	rg_destination_address	Bit#(6)	6

The following are the memory elements in Module integer_divider_riscv imported into Prf_alu:

Memory Element	Name	Type(in BSV) or Fields : Bits	Size(bits)
FIFO	ff_start_buffer	_dividend : 64 _divisor : 64 _div_type : 2 _div_or_mem : 1 _word_flag :1 _dividend_sign_bit : 1 _divisor_sign_bit : 1 _thread_id : 0 _destination : 6	140
FIFO	ff_final_result	Bit#(64)	64
Register	rg_stage	partial_rem : 64 _dividend : 64 _divisor : 64 _divisor_compl : 64 div_or_rem : 1 _div_type : 2 _word_flag : 1 _dividend_sign_bit : 2 _divisor_sign_bit : 2 thread_id : 0 destination : 6	270
Register	rg_state_counter	Bit#(32)	32
Register	rg_thread_id	Bit#(0)	0
Register	rg_destination_address	Bit#(6)	6

The following are the memory elements in riscv_arithmetic_alu imported into Prf_alu

Memory Element	Name	Type(in BSV) or Fields : bits	Size(bits)
FIFO	ff_result	Bit#(64)	64
FIFO	ff_dest	Bit#(6)	6
Register	rg_thread_id	Bit#(0)	0

The following are the memory elements in module LS_unit:

Memory Element	**Name(array length)**	**Type(in BSV) or Fields : Bits**	**Size**(bits)
Register array	`store_q(16)`	Filled : 1 valid : 1 str_addr : 64 str_data : 64 mem_size : 3	133
Register array	`load_q(16)`	Filled : 1 store_mask : 4 eff_addr : 64 ld_size : 3 dest_reg : 6 load_q _index : 4	82
Register array	`load_q_forwarded(16)`	Bool	1
Register	`rg_store_q_head`	Bit#(6)	6
Register	`rg_store_q_tail`	Bit#(6)	6
Register	`rg_load_q_head`	Bit#(6)	6
Register	`rg_load_q_head`	Bit#(6)	6
FIFO	`ff_load`	load_thread_id : 0 store_mask : 16 eff_addr : 64 ld_size : 3 dest_reg : 6 load_q_index : 4	93
FIFO	`ff_store_broadcast`	Valid : 1 regfile_type : 1 dest_tag : 6 thread_id : 0	8
FIFO	`ff_load_broadcast`	Valid : 1 regfile_type : 1 dest_tag : 6 thread_id : 0	8
FIFO	`ff_load_result`	Bit#(64)	64

The following are memory elements in Prf_riscv_branch_unit :

Memory Element	**Name(array length)**	**Type(in BSV) or Fields : Bits**	**Size(**bits)
FIFO	`fifo_branch`	Valid : 1 squash : 1 dest_tag : 6 thread_id : 0	8
FIFO	`fifo_training`	Pc : 64	130

		jump_pc : 64 taken_or_not : 2	
FIFO	`fifo_next_pc`	Bit#(64)	64
FIFO	`fifo_return`	Bit#(64)	64

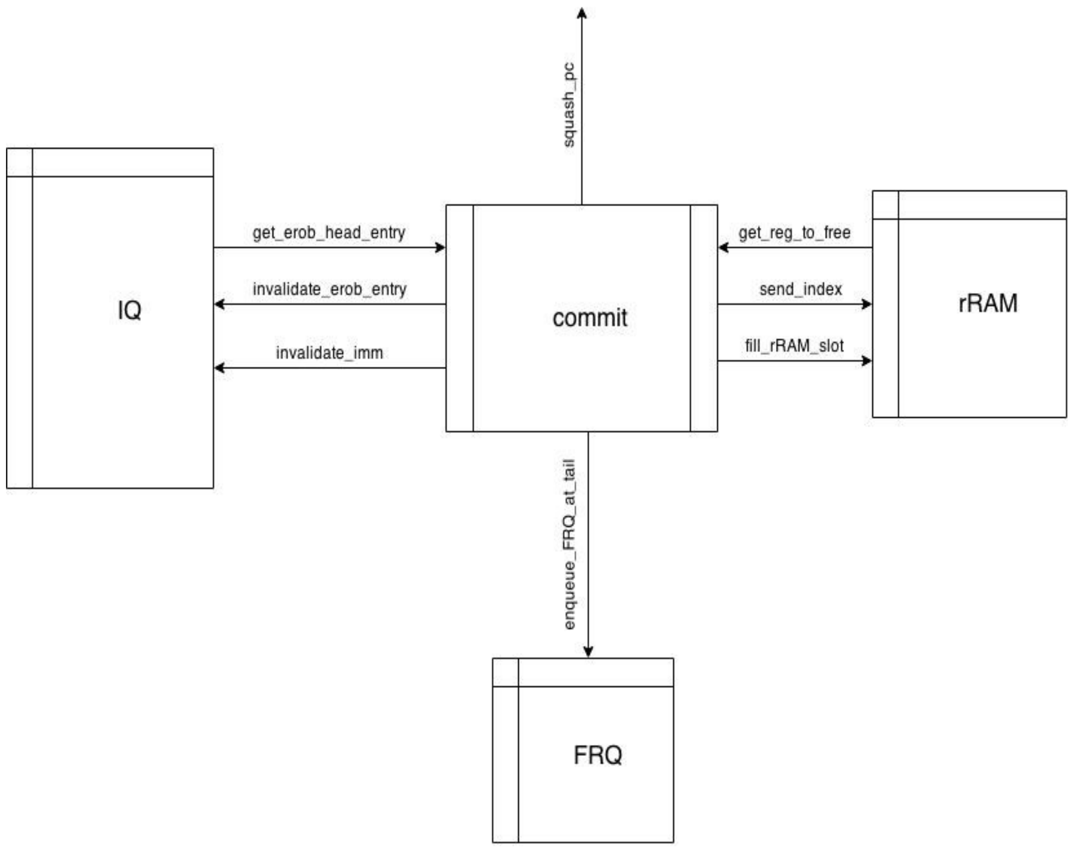
The following are the in-ports of these 3 modules:

Port	**Type**	**Size**(bits)
`_inputs(ALU)`	ALU_payload_type	206
`inputs(LS)`	LS_payload_type	208
`inputs(Branch)`	Branch_payload_type	269

The following are the out-ports of the three modules:

Port	**Type**	**Size**(bits)
`get_broadcast_packet(ALU)`	Broadcast_type	11
`get_load_broadcast_packet(LS)`	Broadcast_type	11
`get_store_broadcast_packet(LS)`	Broadcast_type	11
`get_broadcast_packet(Branch)`	Branch_broadcast_type	11
`alu_ready(ALU)`	Bool	1
`ls_ready(LS)`	Bool	1
`branch_ready(Branch)`	Bool	1
`arith_out(ALU)`	Bit#(64)	64
`load_result(LS)`	Bit#(64)	64
`branch_out(Branch)`	Bit#(64)	64

MODULAR DIAGRAM FOR COMMIT STAGE



The module Prf_commit does not have any storage structures or registers or FIFOs. Two head entries in EROB if valid are considered for commit. In case of wrong branch prediction squash(or flush) signals are activated and the instruction commit is aborted. When an instruction commits, the physical register address assigned to destination register(architectural register) of the instruction is copied to corresponding slot in rRAM and the entry in the EROB is tagged invalid and head pointer is incremented. The value that is being replaced is enqueued at the tail of FRQ. In case of squash, the data in rRAM is copied to fRAM.

The following are the in-ports of commit:

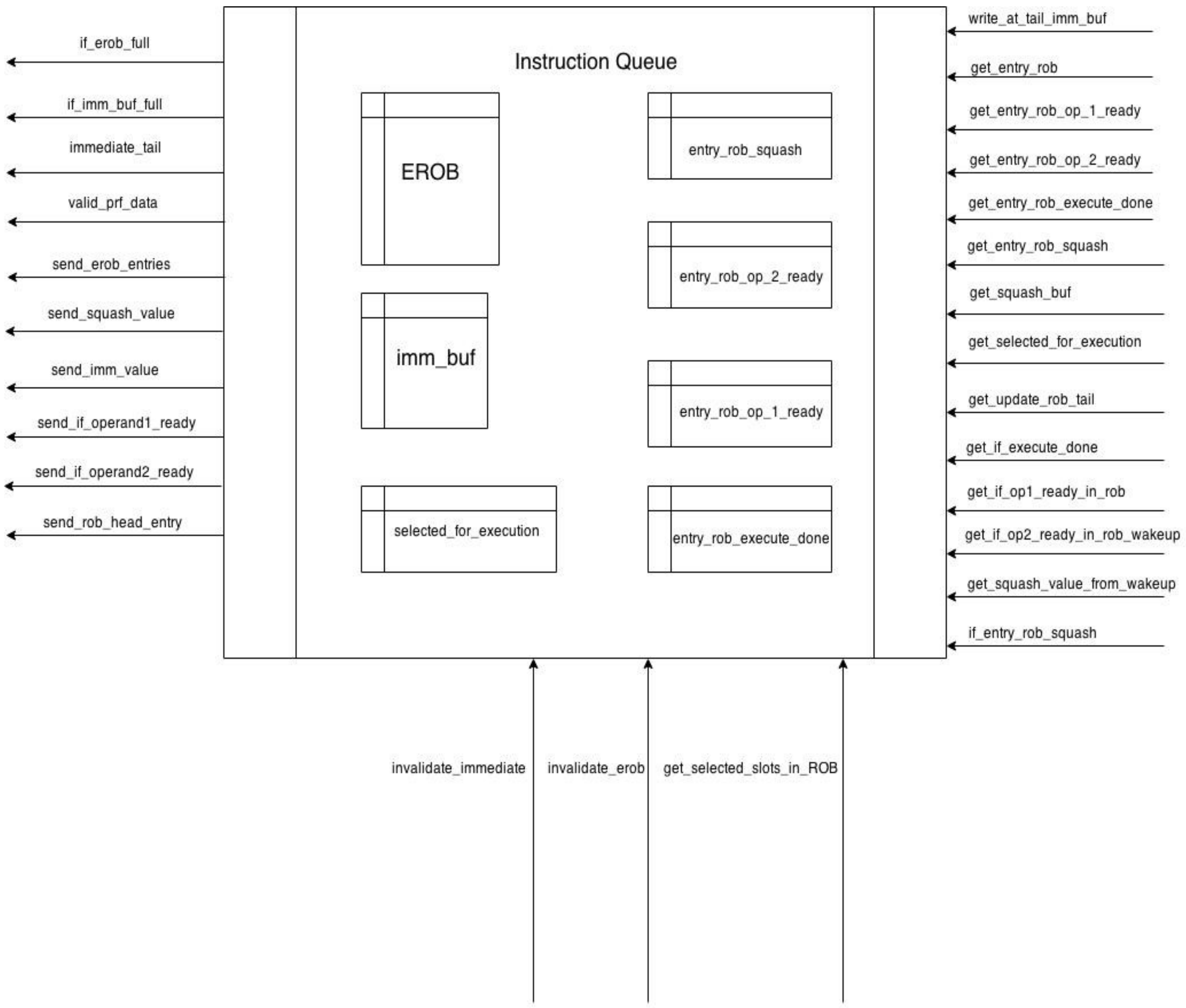
Port	**Type**	**Size**(bits)
`get_erob_head_entry`	Vector#(2, Entry_rob_type)	244
`get_reg_to_free`	Vector#(2, Bit#(6))	12

The following are the out-ports of commit:

Port	**Type**	**Size**(bits)
`invalidate_rob_entry`	Vector#(2, Bool)	2
`invalidate_imm`	Bool	2

`enqueue_FRQ_at_tail`	Vector#(2, Bit#(6))	12
`squash_pc`	Vector#(2, Bool)	2
`send_index`	Vector#(2, Bit#(5))	10
`fill_rRAM_slot`	Bit#(11)	11

MODULAR DIAGRAM FOR INSTRUCTION QUEUE



The module Prf_instruction_queue has 6 storage structures, 2 large storage structures and 4 small ones(Bool type). Entry Reorder Buffer(EROB) allocates data as soon as it is dequeued from decode stage. It consists of 16 entries each of Entry_rob_type. imm_buf allocates immediate value in the instructions if valid. It consists of 8 entries and each of Imm_buf_type. It has six other storage structures each of 16 entries and of Bool type. The following are the register structures in the module:

Register Structure	**Type**	**Size of slot**	**Array length**
EROB	Entry_rob_type	122	16
imm_buf	Imm_buf_type	64	8
selected_for_execution	Bool	1	16
entry_rob_squash	Bool	1	16
entry_rob_op_1_ready	Bool	1	16
entry_rob_op_2_ready	Bool	1	16
entry_rob_execute_done	Bool	1	16
squash_buf	Bit#(64)	64	16

The following are the registers in the module:

Registers	**Type**	**Size**(bits)
rg_erob_tail	Bit#(4)	4
rg_erob_head	Bit#(4)	4
rg_imm_buf_head	Bit#(3)	3
rg_imm_buf_tail	Bit#(3)	3

The in-ports of module Prf_instruction_queue:

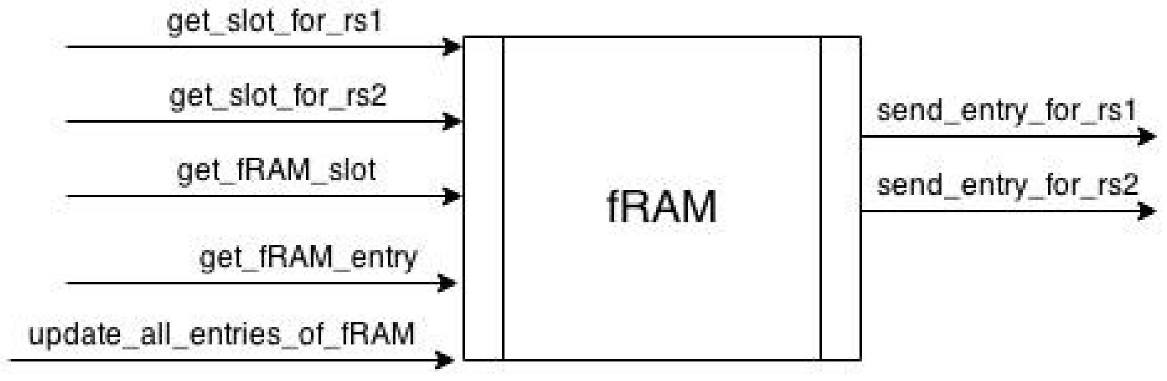
Port	**Type**	**Size(bits)**
write_at_tail_imm_buf	Imm_buf_type	64
get_entry_rob	Entry_rob_type	122
get_entry_rob_op_1_ready	Vector#(16, Bool)	16
get_entry_rob_op_2_ready	Vector#(16, Bool)	16
get_entry_rob_execute_done	Vector#(16, Bool)	16
get_entry_rob_squash	Vector#(16, Bool)	16
get_squash_buf	Vector#(16, Bool)	16
get_selected_for_execution	Vector#(16, Bool)	16
get_update_rob_tail	Bool	1
get_if_execute_done	Vector#(16, Bool)	16
get_if_op1_ready_in_rob_wakeup	Vector#(16, Bool)	16
get_if_op2_ready_in_rob_wakeup	Vector#(16, Bool)	16
get_squash_value_from_wakeup	Bit#(64) + Bit#(3)	67
if_entry_rob_squash	Vector#(16, Bool)	16
get_selected_slots_in_ROB	Vector#(16, Bool)	16
invalidate_erob	Bool	1

invalidate_immediate	Bool	1

The out-ports from the module Prf_instruction_queue:

Port	**Type**	**Size**(bits)
if_erob_full	Bool	1
if_imm_buf_full	Bool	1
immediate_ tail	Bit#(3)	3
send_squash_value	Bit#(64)	64
send_imm_value	Bit#(64)	64
send_if_operand1_ready	Bool	1
send_if_operand2_ready	Bool	1
send_rob_head_entry	Entry_rob_type	122

MODULAR DIAGRAM FOR fRAM



The module Prf_fRAM contains a array of registers of length 32 and stores a 6 bit value which corresponds to index of a physical register file. This module stores the mapping of destination registers of instructions which are just decoded. At the time of exception all the entries in the fRAM are flushed.

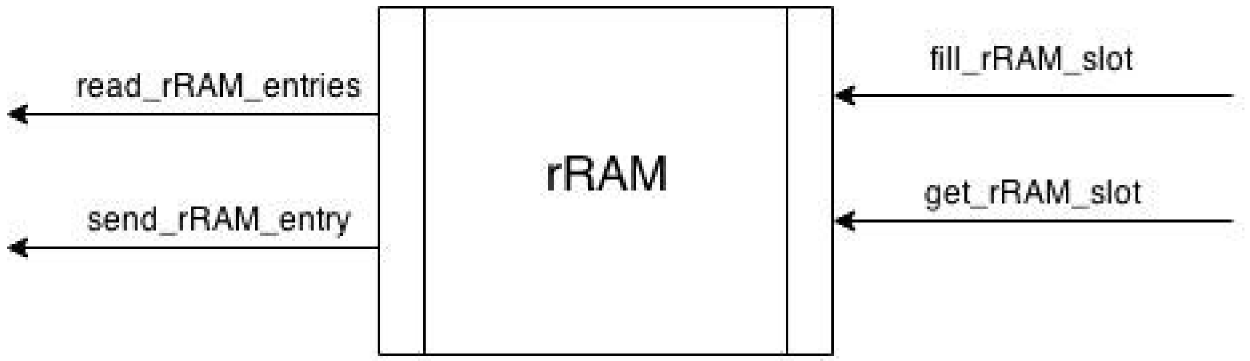
The in-ports of fRAM are as follows:

Port	**Type**	**Size**(bits)
get_slot_for_rs1	Bit#(5)	5
get_slot_for_rs2	Bit#(5)	5
get_fRAM_slot	Bit#(5)	5
get_fRAM_entry	Bit#(6)	6
update_all_entries_fRAM	Vector#(32, Bit#(6)	192

The in-ports of fRAM are as follows:

Port	**Type**	**Size**(bits)
`send_entry_for_rs1`	Bit#(6)	6
`send_entry_for_rs2`	Bit#(6)	6

MODULAR DIAGRAM FOR rRAM



This module Prf_rRAM has array of registers similar to that of fRAM but it's functionality in the processor is different from that of fRAM. This module stores mapping of destination registers of instruction that are just committed. When an instruction commits, the value in rRAM which is being replaced is send to FRQ.

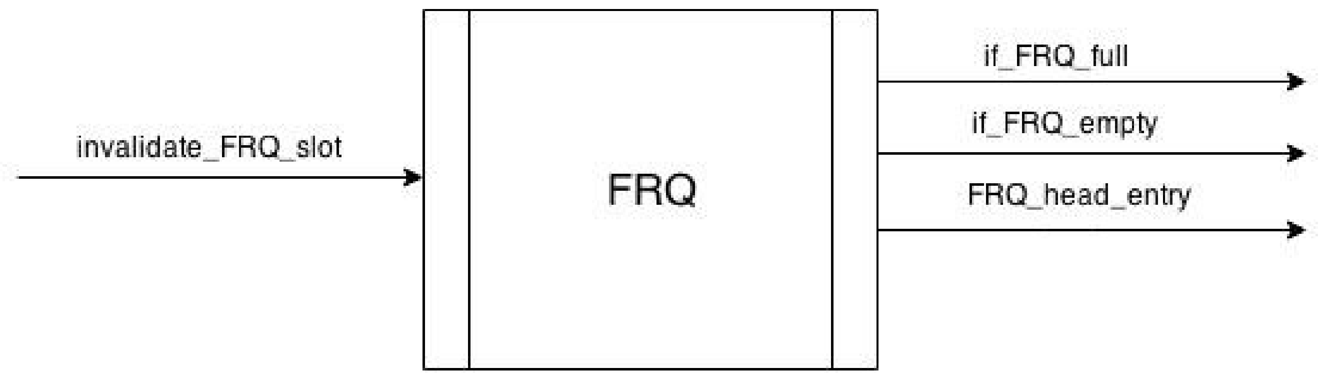
The following are the in-ports of rRAM:

Port	**Type**	**Size**(bits)
`fill_rRAM_slot`	Bit#(6) + Bit#(5)	11
`get_rRAM_slot`	Bit#(5)	5

The following are the out-ports of rRAM:

Port	**Type**	**Size**(bits)
`read_rRAM_entries`	Vector#(32, Bit#(6))	192
`send_rRAM_entry`	Bit#(6)	6

MODULAR DIAGRAM FOR FRQ



The module Prf_FRQ has an array of registers of length 64 each stores 6 bit value. There are two more registers to indicate head and tail of the queue each of 6 bits. This module indicates the number of free registers which can be used for register renaming.

The following are the in-ports of the module:

Port	**Type**	**Size**(bits)
`invalidate_FRQ_slot`	Bit#(6)	6

The following are the out-ports of the module:

Port	**Type**	**Size**(bits)
`if_FRQ_full`	Bool	1
`if_FRQ_empty`	Bool	1
`FRQ_head_entry`	Bit#(6)	6

Prf_riscv_types defines all the struct types used in the processor.