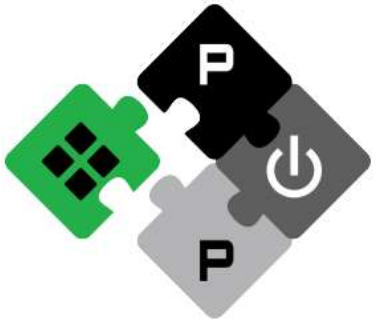


PULP PLATFORM

Open Source Hardware, the way it should be!

A look inside an Application Processor: CVA6

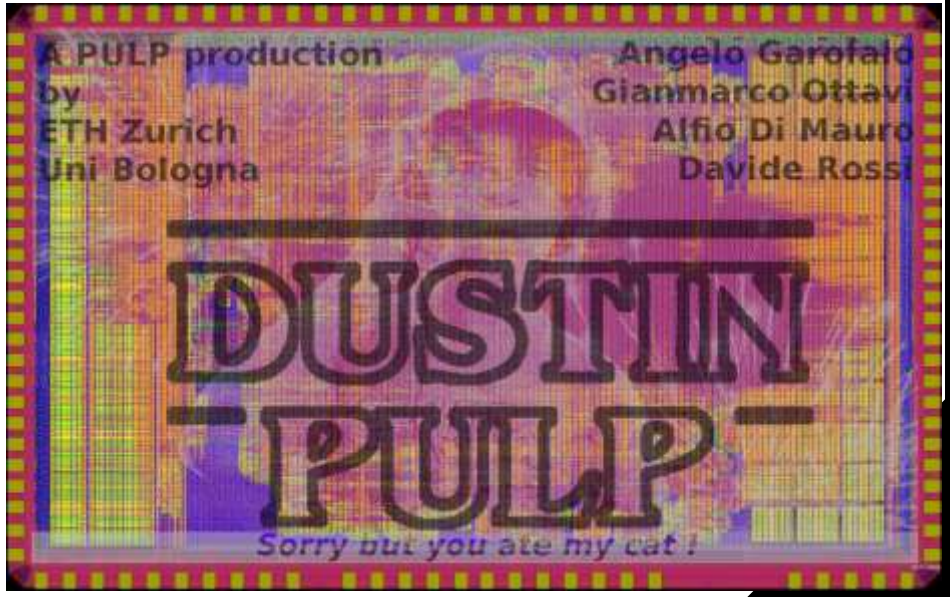
Gianmarco Ottavi <Gianmarco.ottavi2@unibo.it>

ETH zürich

 http://pulp-platform.org  @pulp_platform  https://www.youtube.com/pulp_platform

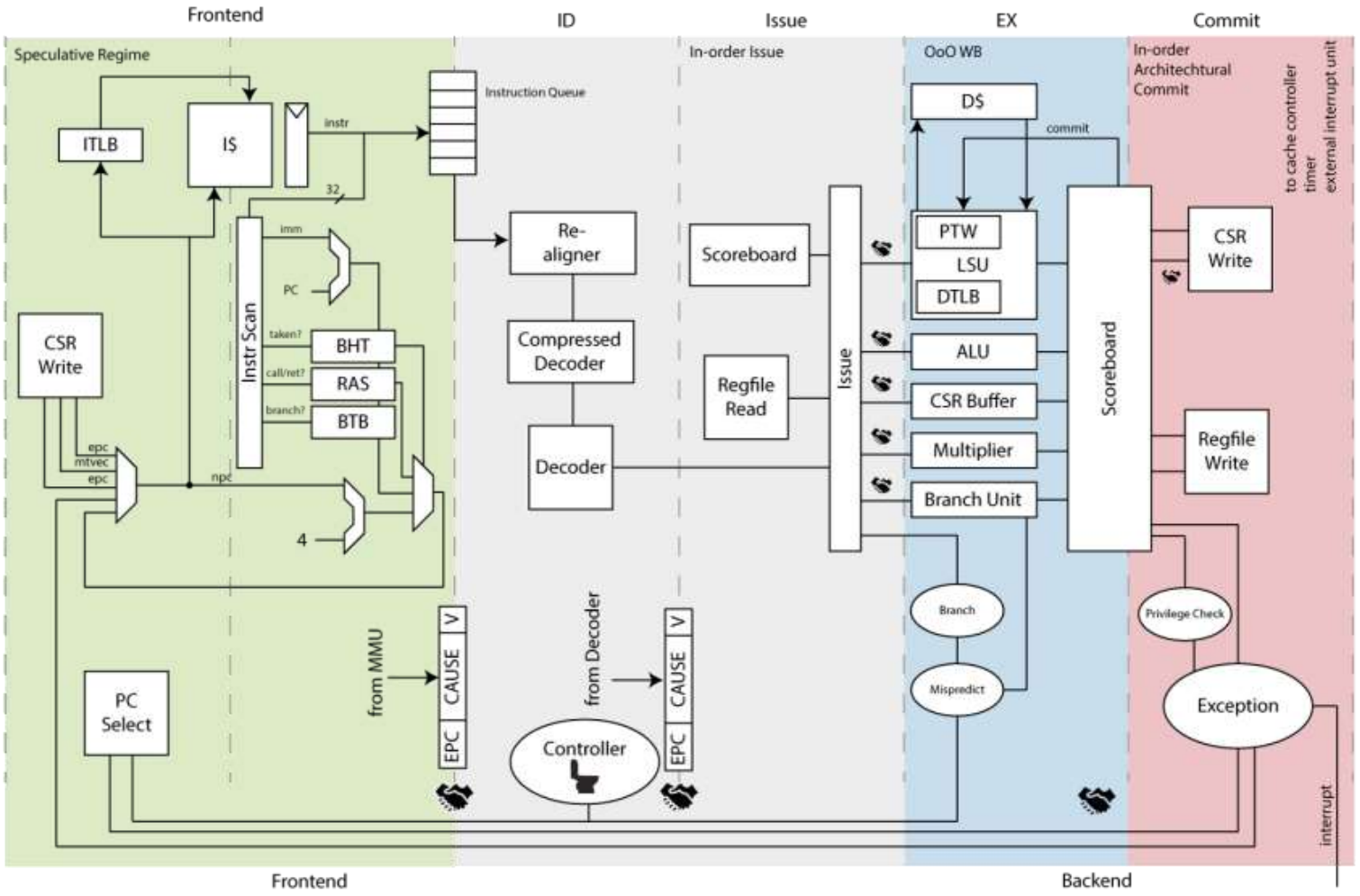
CVA6 is open-source

- You can find all the RTL of the Core at:
  - https://github.com/openhwgroup/cva6
- CVA6 is just one of the core born from the PULP Project
  - https://pulp-platform.org/
  - https://github.com/pulp-platform
- From RTL to Silicon



ETH zürich

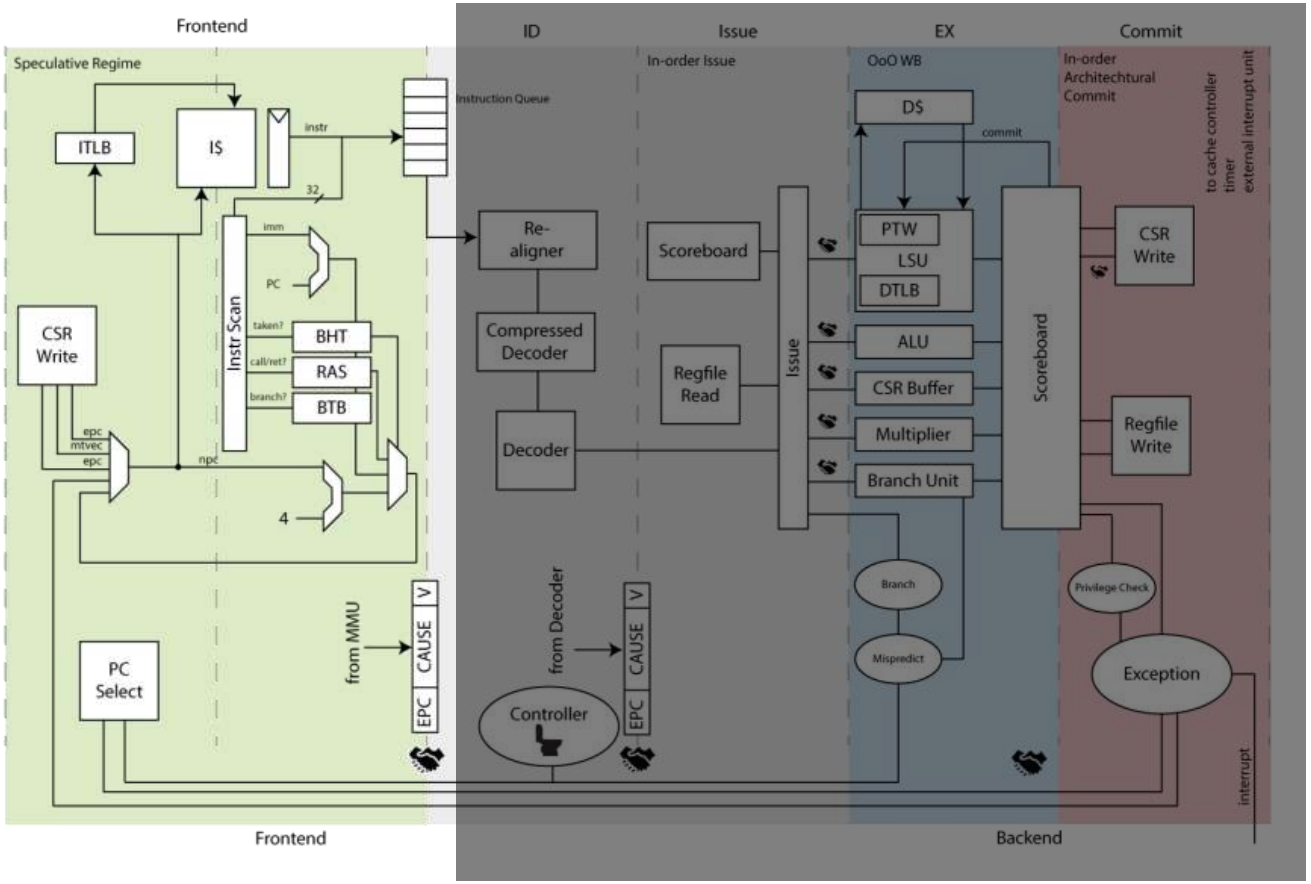
A look Into CVA6



- RISC-V ISA RV64IMAC(F)
- M,S,U Privilege level Spec.
  - Can run linux-like OS (Application class processor).
- In-Order Single Issue with 6 stages of Pipeline

F. Zaruba, L. Benini -- DOI: 10.1109/TVLSI.2019.2926114

A look Into CVA6: Frontend



PC Generation:

- Sequential Fetch
- Miss Prediction Recovery
- Exception
- Debug
- CSR

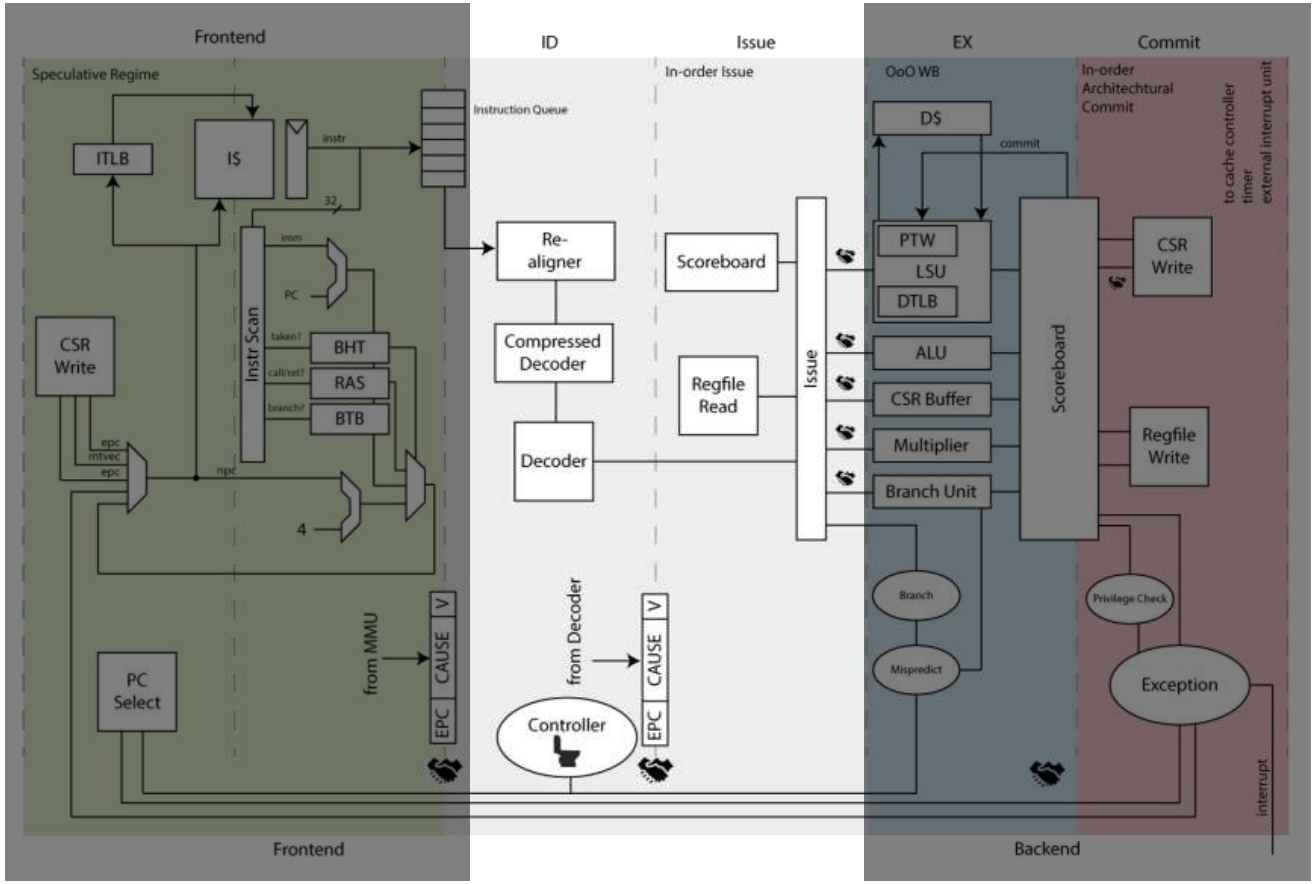
Instruction Fetch:

- Pre-Decoding for Branch Prediction
- Instruction Cache
  - Virtually Indexed and Physically Tagged
  - Pipelined to mitigate long propagation delays of memory macros

Frontend Decoupled with backend via Instruction queue

F. Zaruba, L. Benini -- DOI: 10.1109/TVLSI.2019.2926114

A look Into CVA6: Backend



Decode Stage:

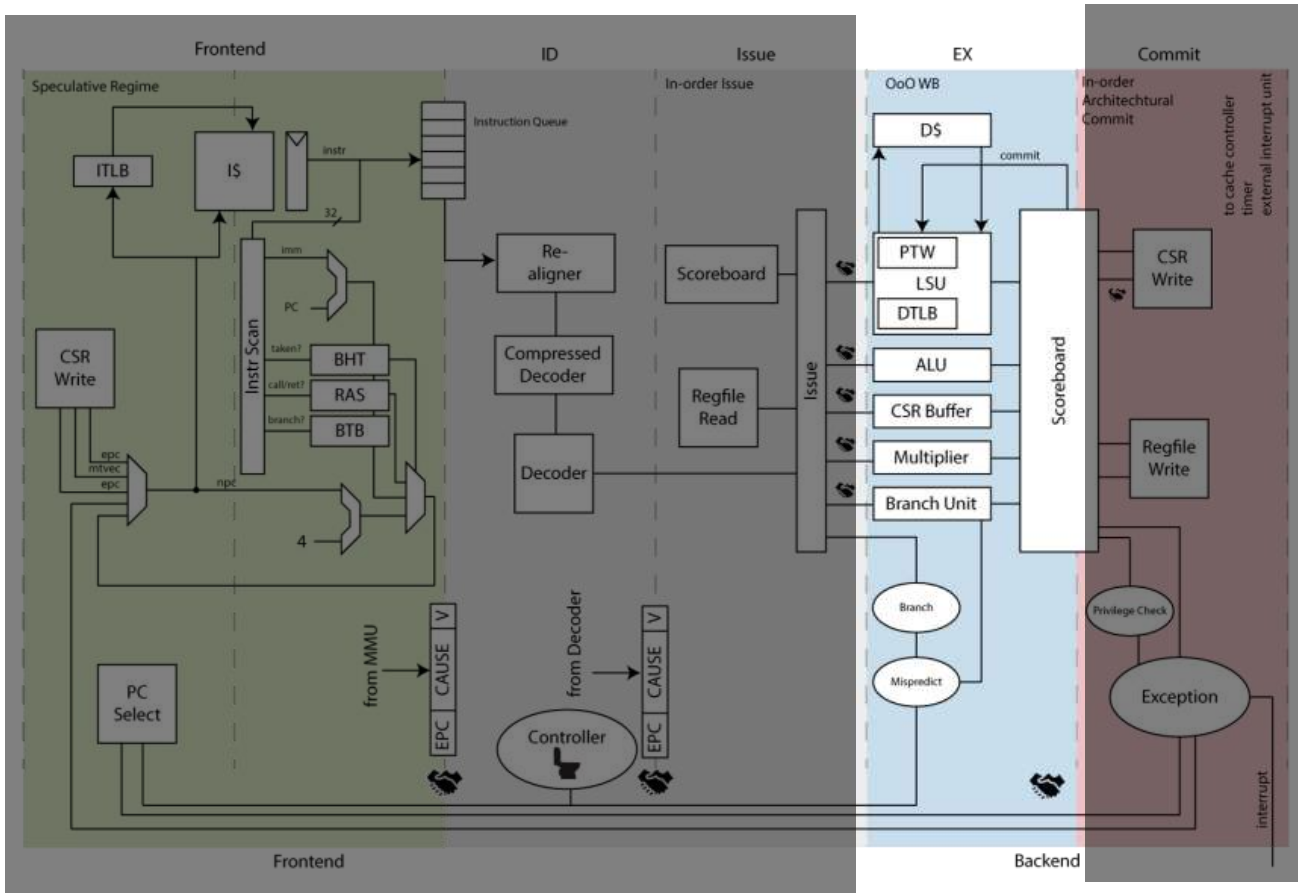
- Compressed decoder
- Decoder

Issue Stage:

- Renaming
  - Lightweight 1-bit renaming scheme
- Operand Read
  - Instruction Are issued if Operands are ready
- Scoreboarding
  - Mixed Scoreboard with Reorder Buffer
  - Issued instruction are tracked for data Hazards
  - Allow Out-Of-Order completion

F. Zaruba, L. Benini -- DOI: 10.1109/TVLSI.2019.2926114

A look Into CVA6: Backend

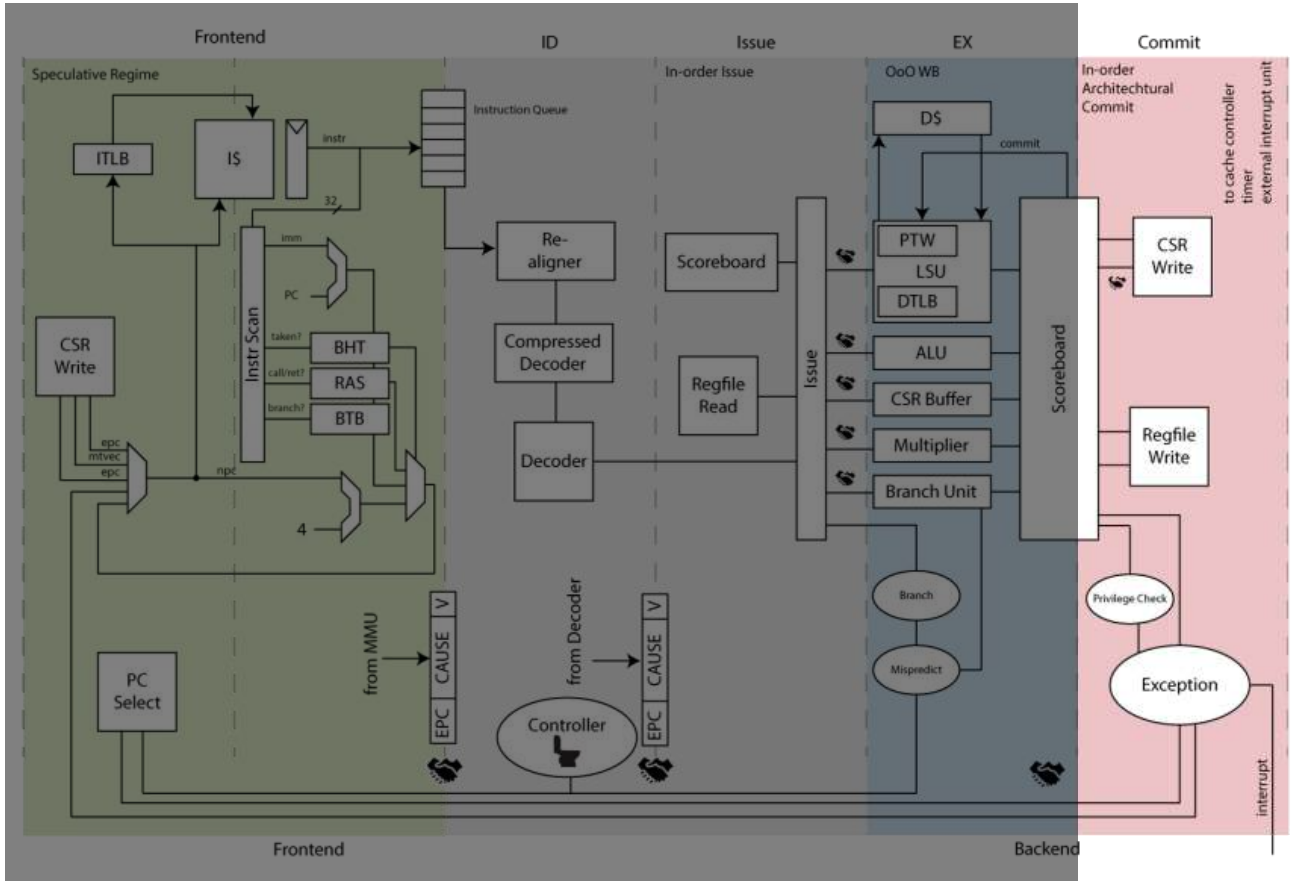


Ex Stage:

- ALU
- FPU
- Multiplier/Integer Div
  - 2 Stage/Iterative
- LSU with 3 Master Ports
  - Load
  - Store
  - PTW
- Data Cache 2 Stages Pipeline
- Branch Unit

F. Zaruba, L. Benini -- DOI: 10.1109/TVLSI.2019.2926114

A look Into CVA6: Backend



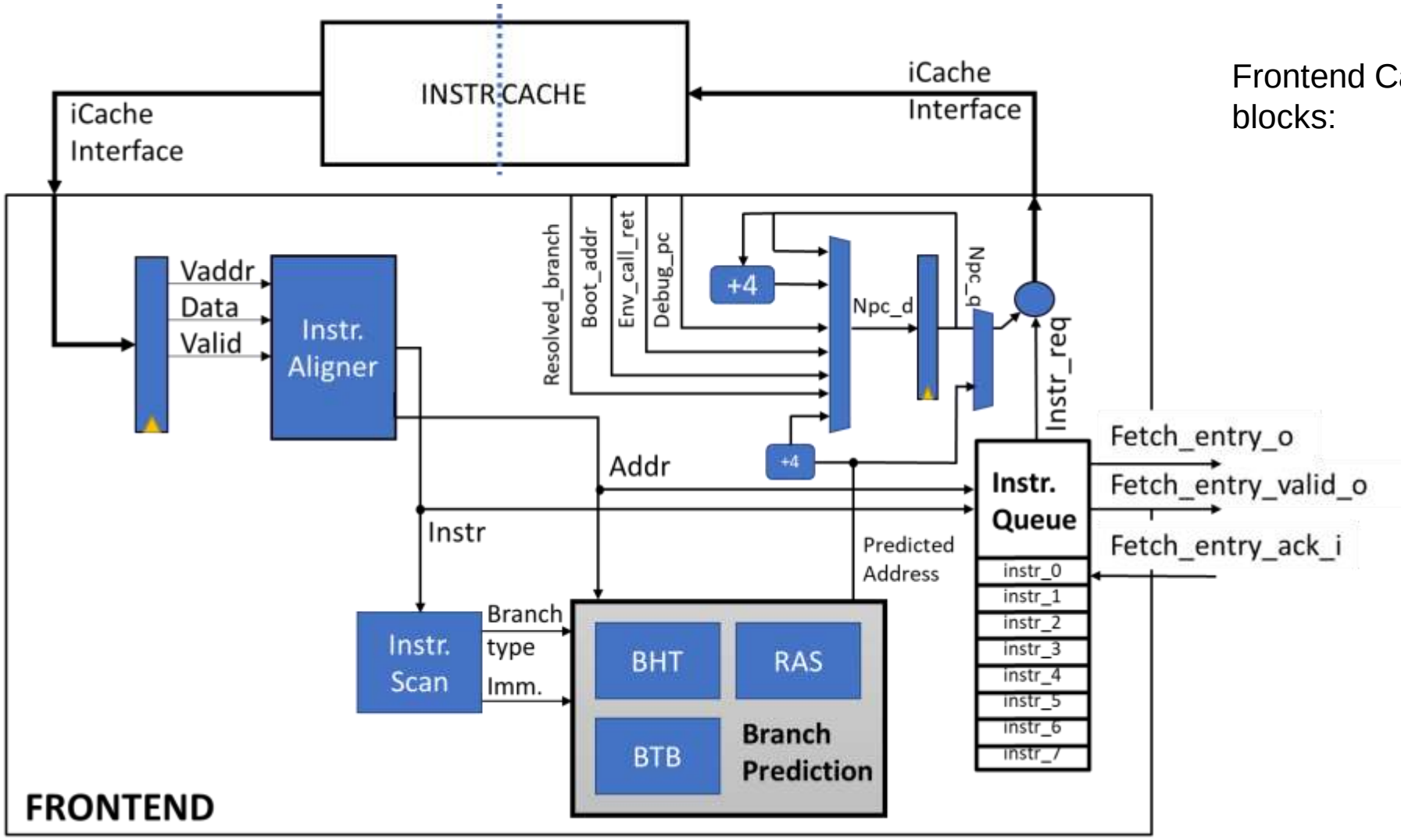
Commit Stage

- Commit Completed instruction In-Order
  - Register File
  - Main memory

External Interface to main memory AXI4 Compliant

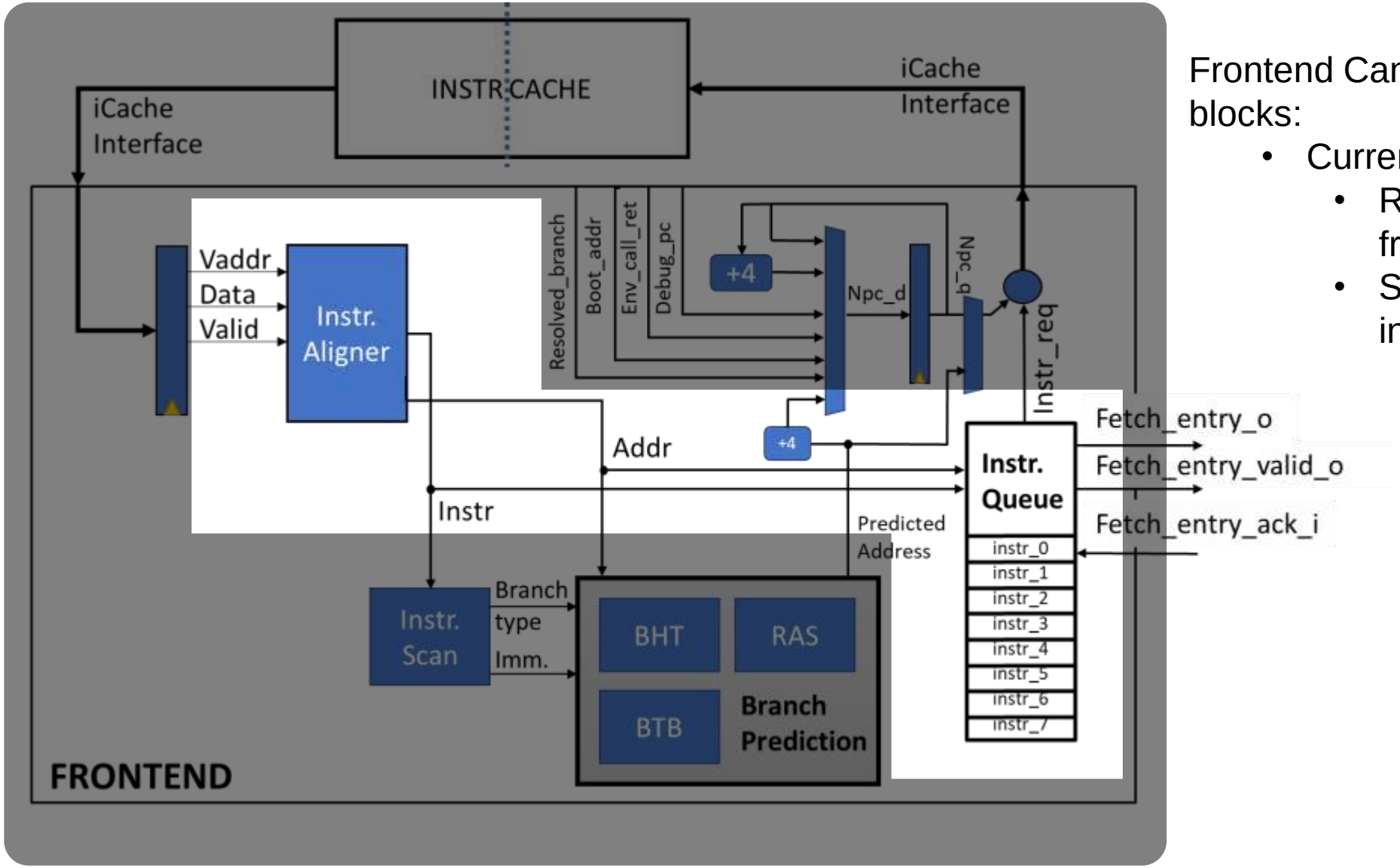
F. Zaruba, L. Benini -- DOI: 10.1109/TVLSI.2019.2926114

Deep Dive: Frontend



Frontend Can be divided in 3 main blocks:

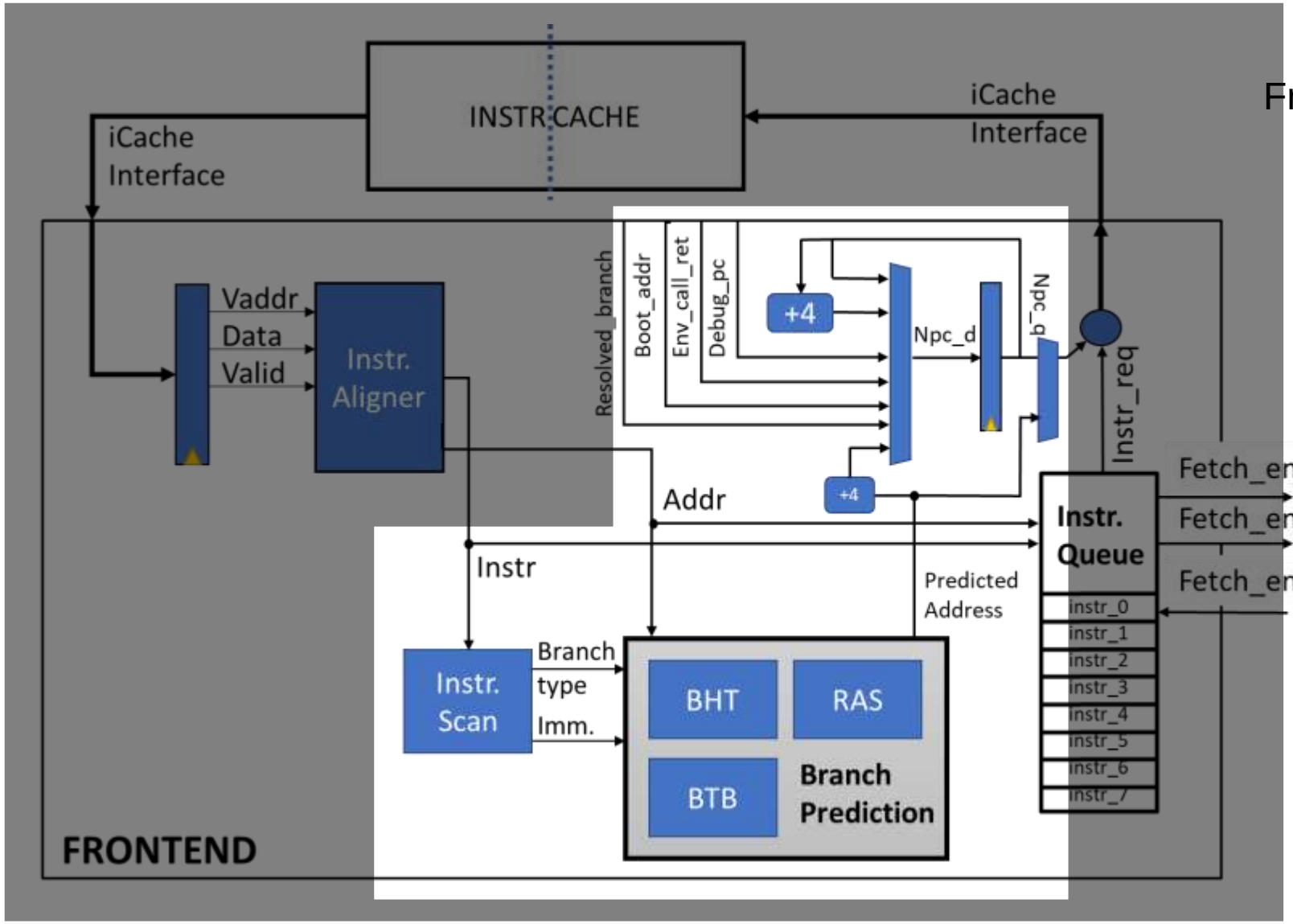
Deep Dive: Frontend



Frontend Can be divided in 3 main blocks:

- Current Instruction
  - Realign Sampled Instruction from Instruction cache
  - Store the current instructions into instr. queue;

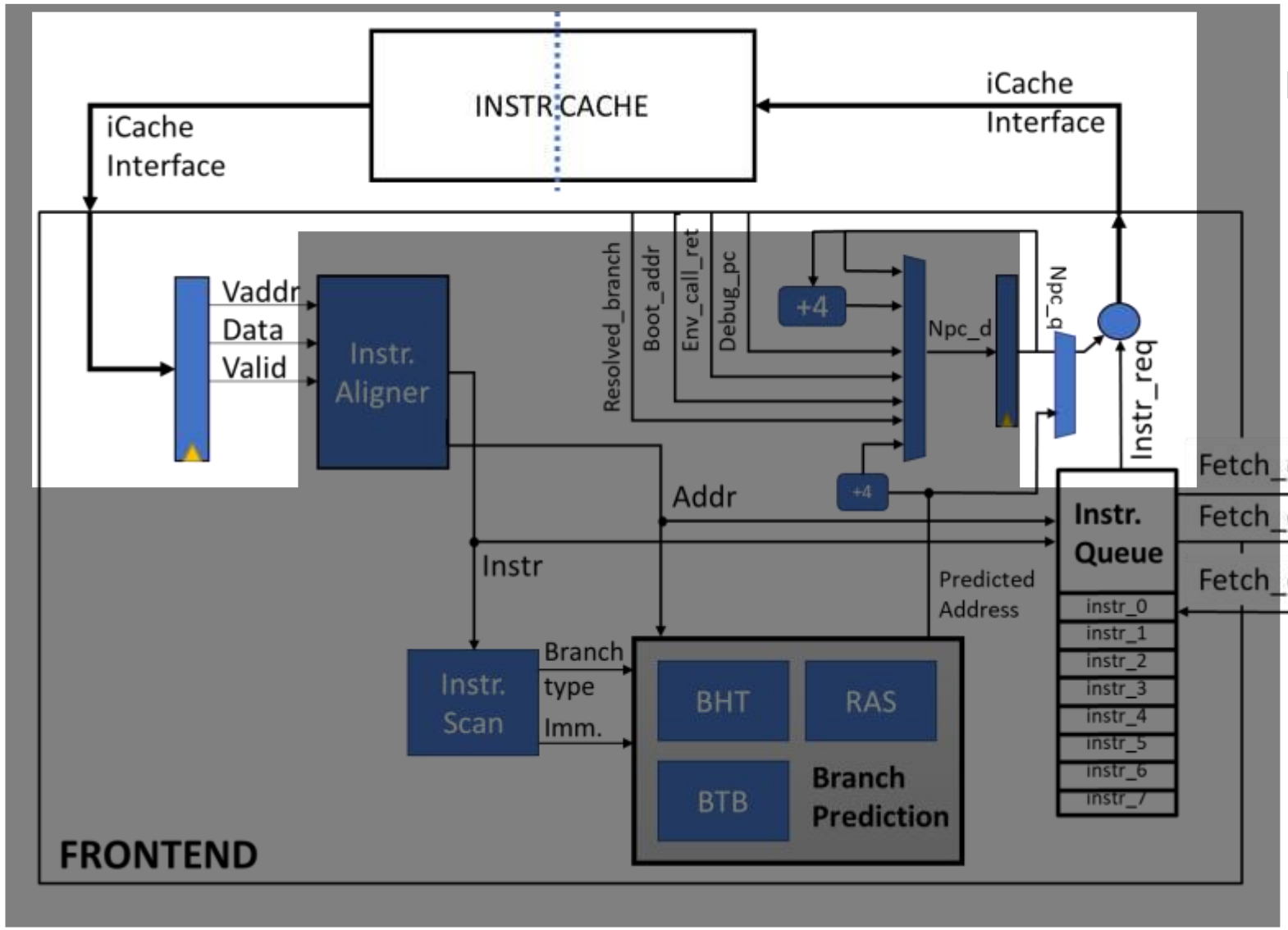
Deep Dive: Frontend



Frontend Can be divided in 3 main blocks:

- Current Instruction;
  - Realign Sampled Instruction from Instr. cache;
  - Store the instruction into instr. queue;
- Next Program Counter Logic:
  - Compute next PC depending on:
    - Booting;
    - Exceptions;
    - Debugging;
    - Branch;
    - Recovery from bad Prediction;
    - Sequential Fetching.

Deep Dive: Frontend



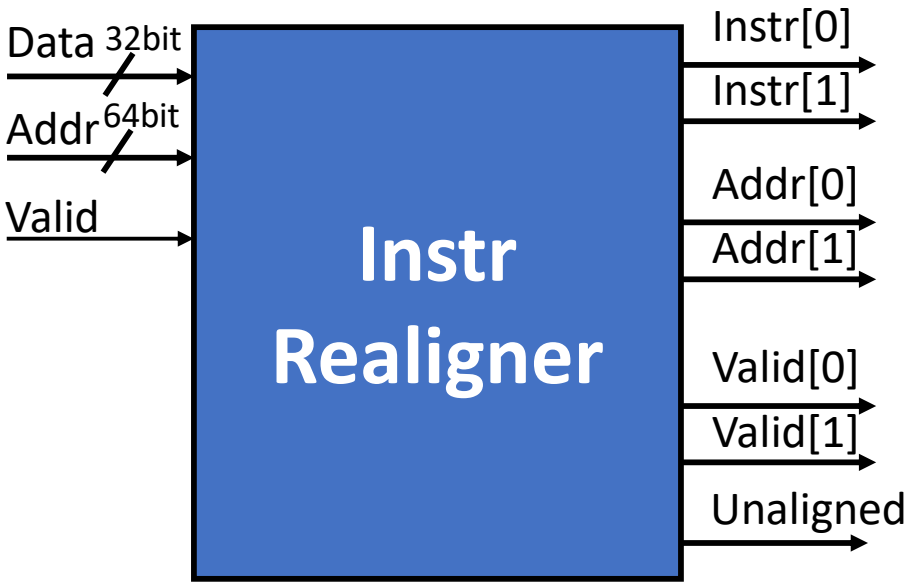
Frontend Can be divided in 3 main blocks:

- Current Instruction;
  - Realign Sampled Instruction from Instr. cache;
  - Store the instruction into instr. queue;
- Next Program Counter Logic:
  - Compute next PC depending on:
    - External Inputs;
    - Branches on current instr;
    - Recovery from bad Prediction;
    - Exception;
    - Debugging;
    - Booting;
- Cache interface:
  - Request Side;
  - Response Side;

Why do we need an istruction realigner?

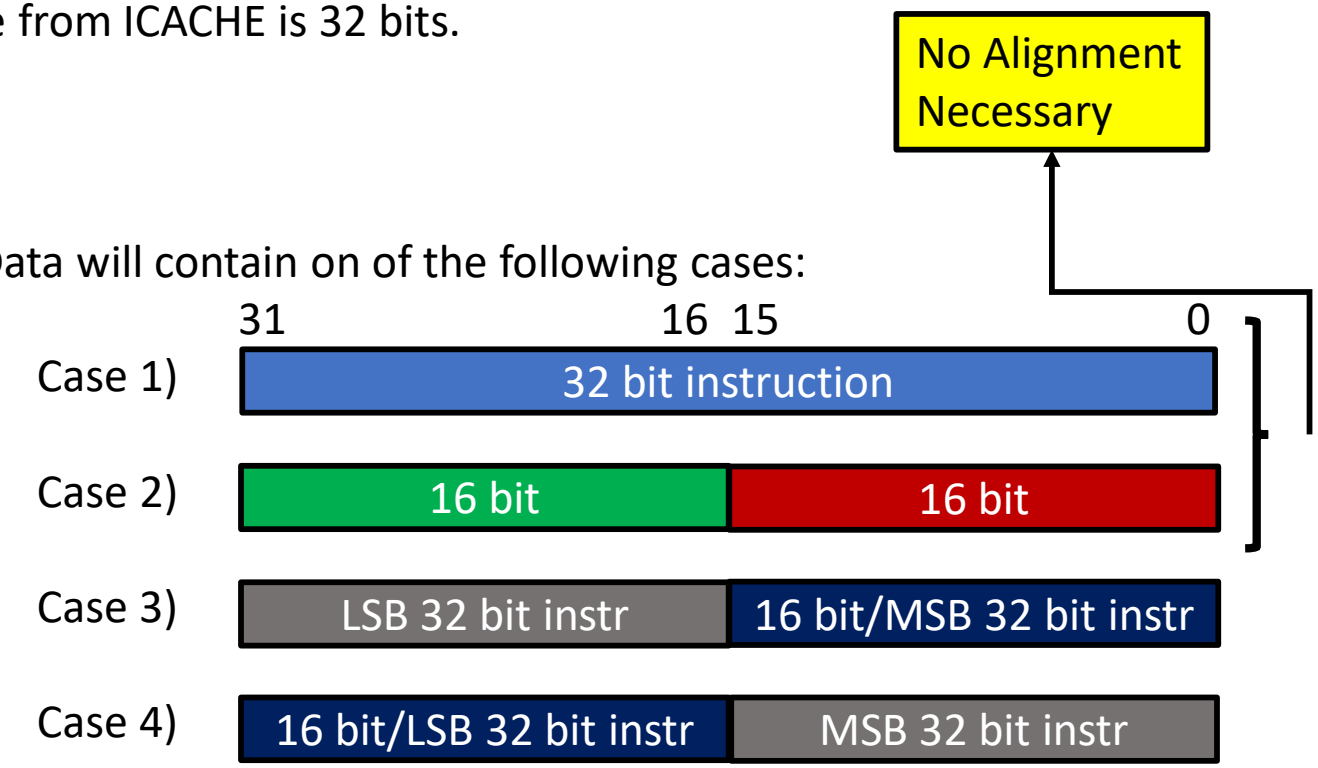
CVA6 supports Compressed Instructions and data interface from ICACHE is 32 bits.

- Compressed Instructions ~70% of total instruction.
- On 32bit fetch width equals to ~1.5 instr per fetch.

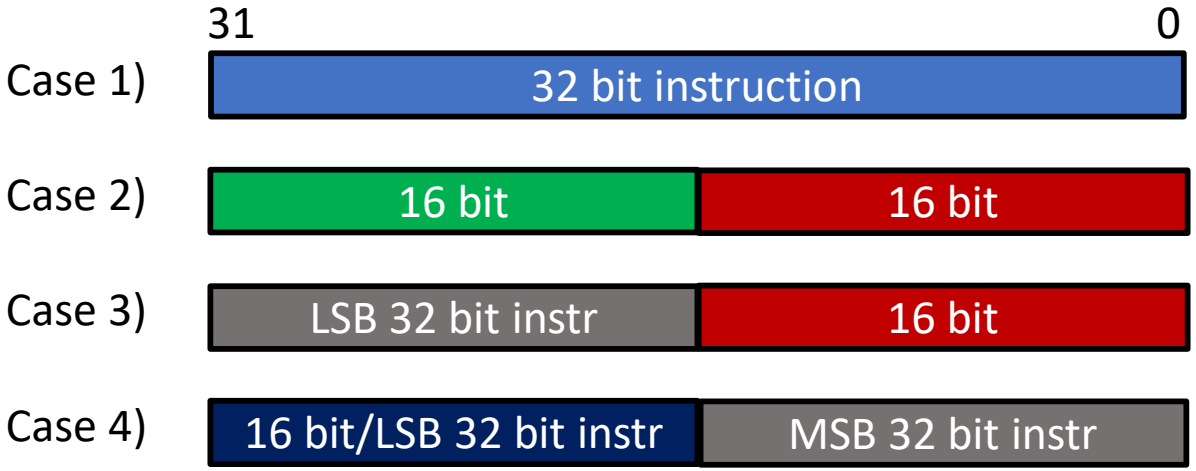


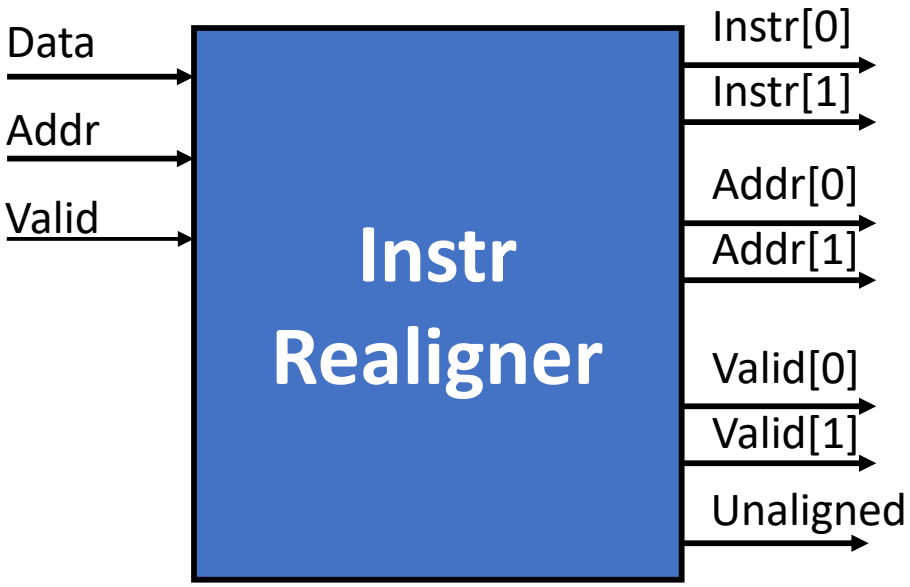
Non Compressed instruction have the first 2 bits = 1 ➡ Check bits [1:0] and [17:16]

- Data will contain on of the following cases:

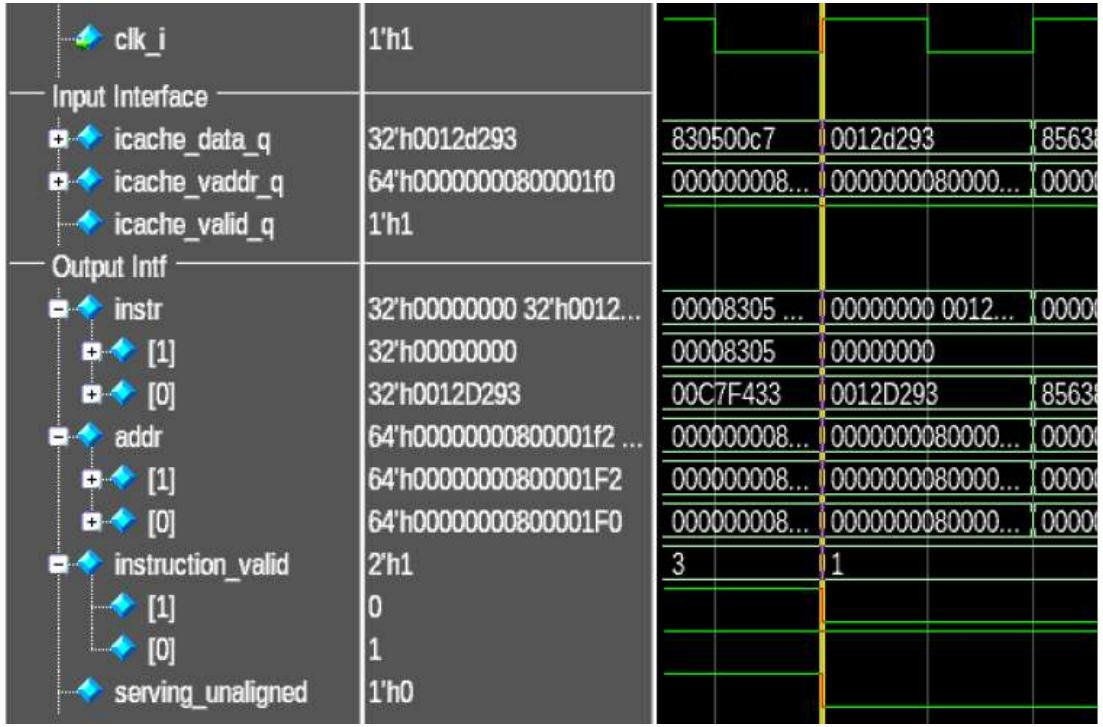


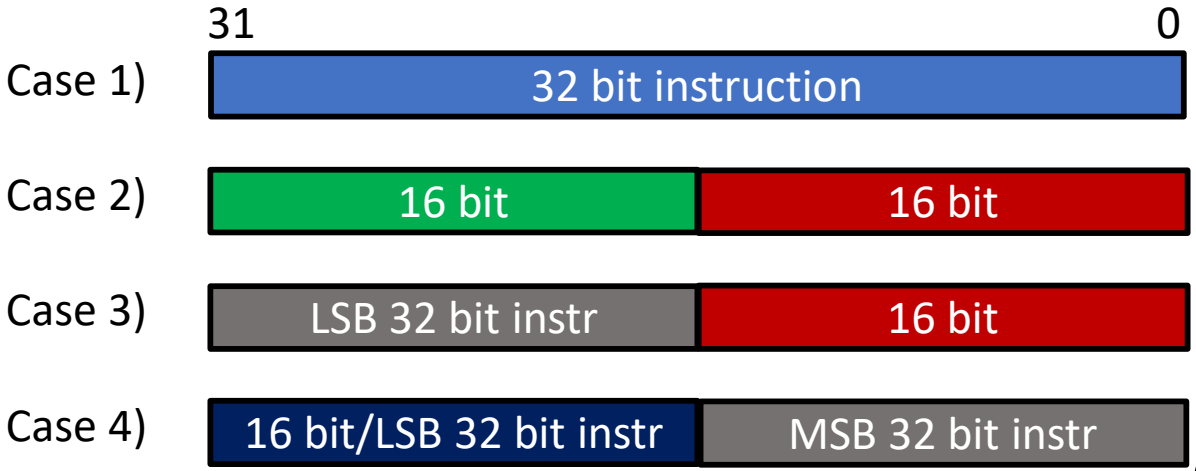
- Addr corresponds to the fetch address of the presented data.
- Valid tells if the data that is presented at the interface is usable.

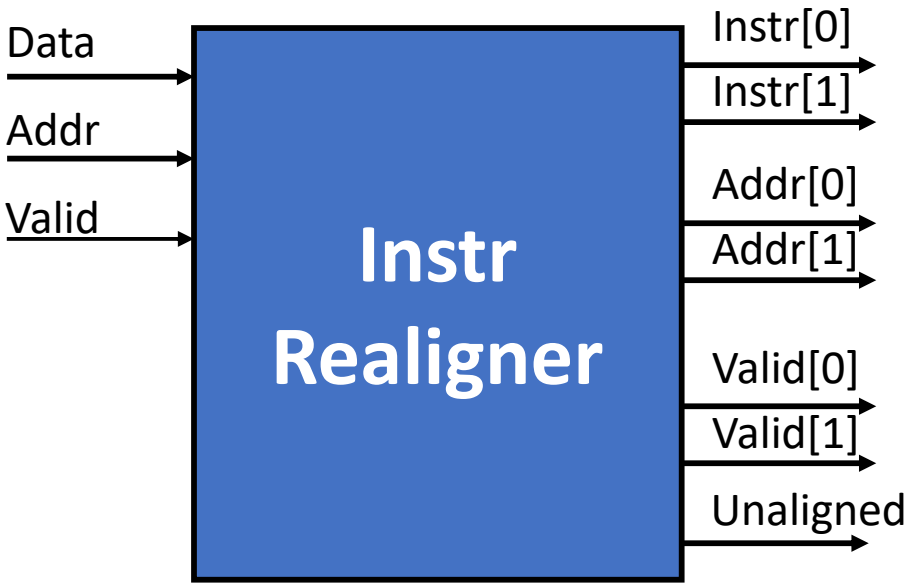

31
0
Case 1)
32 bit instruction
Case 2)
16 bit
16 bit
Case 3)
LSB 32 bit instr
16 bit
Case 4)
16 bit/LSB 32 bit instr
MSB 32 bit instr


Data
Addr
Valid
Instr Realigner
Instr[0]
Instr[1]
Addr[0]
Addr[1]
Valid[0]
Valid[1]
Unaligned

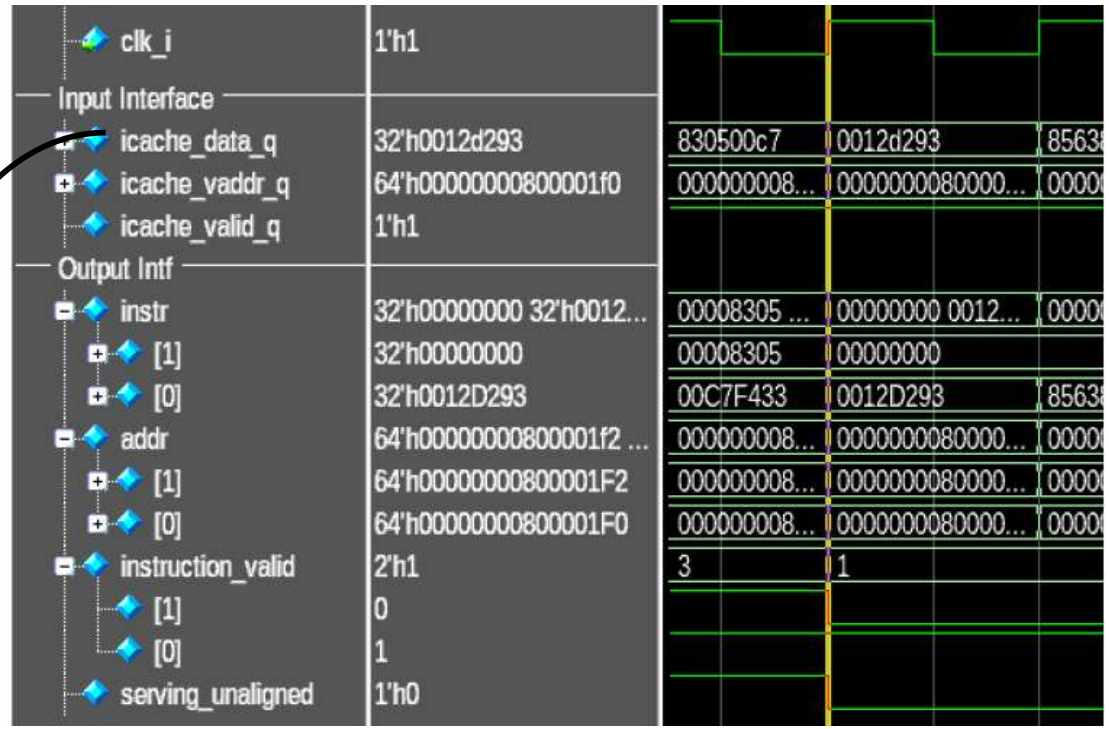
Case 1)


clk_i
1'h1
Input Interface
icache_data_q
32'h0012d293
830500c7
0012d293
icache_vaddr_q
64'h000000000800001f0
000000008...
0000000080000...
icache_valid_q
1'h1
Output Intf
instr
32'h00000000 32'h0012...
00008305 ...
00000000 0012...
[1]
32'h00000000
00008305
00000000
[0]
32'h0012D293
00C7F433
0012D293
addr
64'h000000000800001f2 ...
[1]
64'h00000000800001F2
[0]
64'h00000000800001F0
instruction_valid
2'h1
3
1
[1]
0
[0]
1
serving_unaligned
1'h0



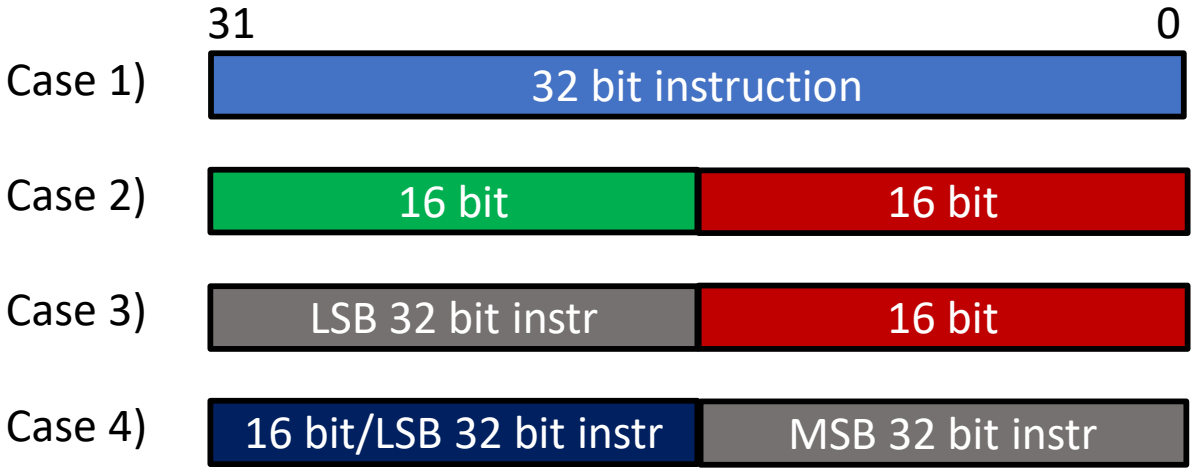


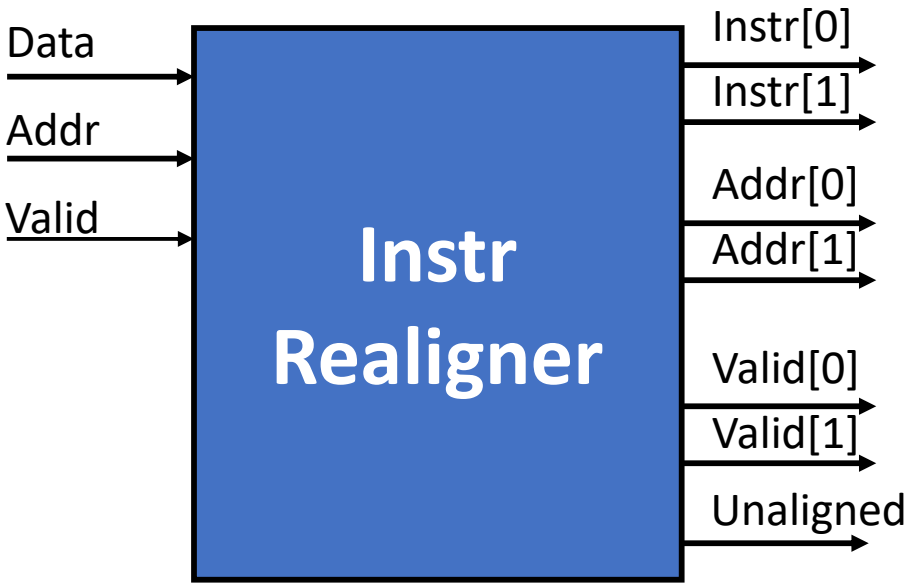
Case 1)



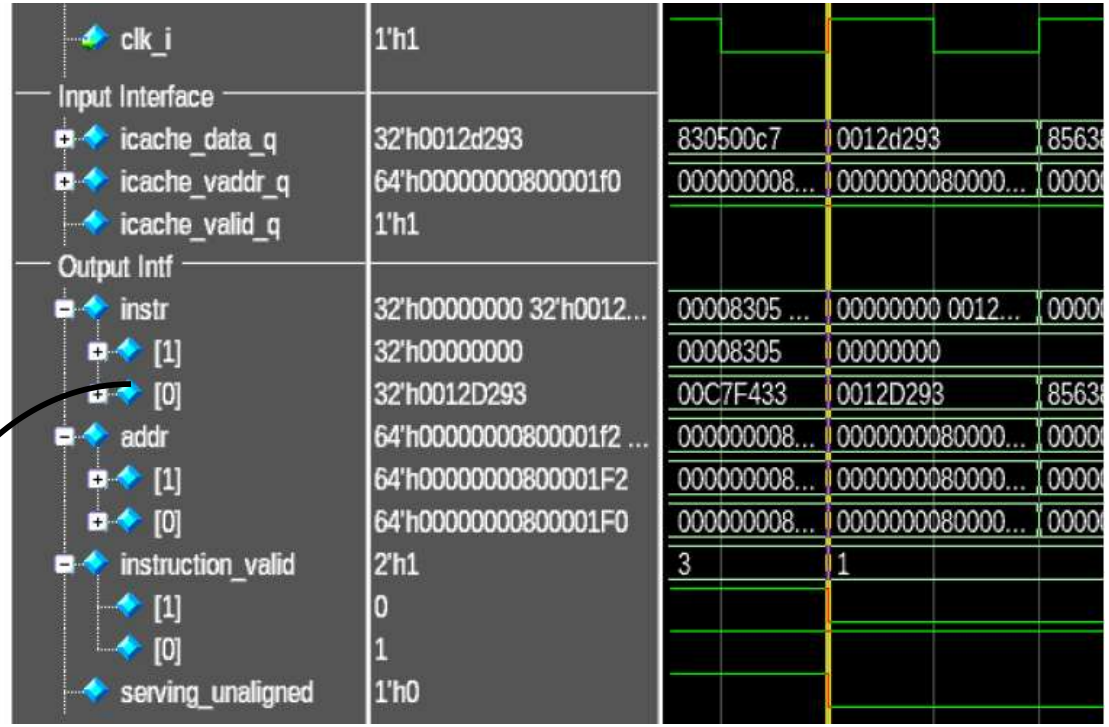
0x0012 d293 → Data[1:0] = 11

Non Compressed Instructions





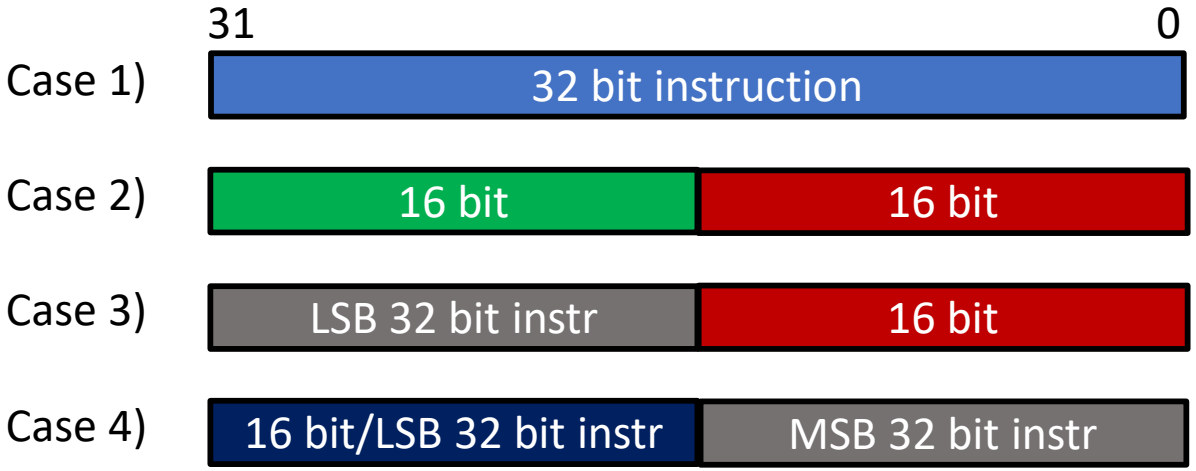
Case 1)

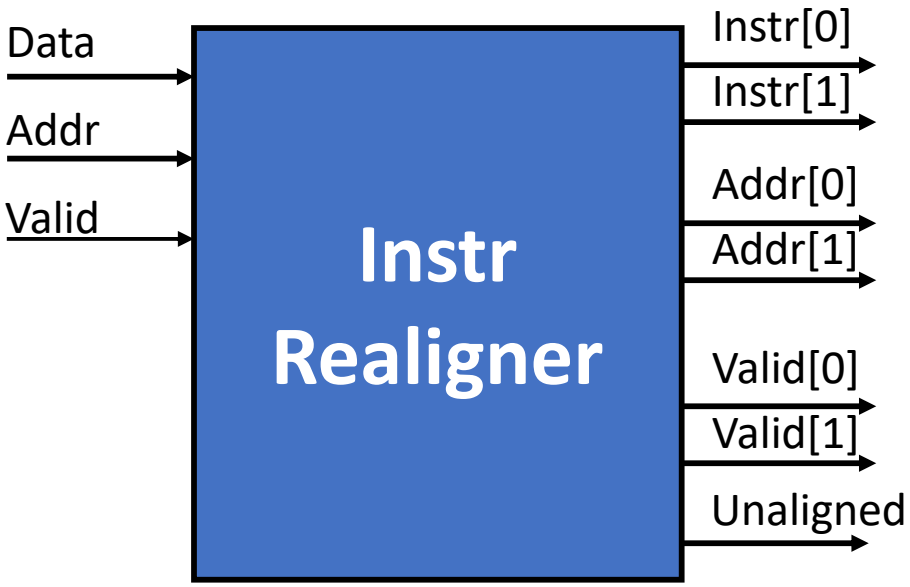


0x0012 d293 → Data[1:0] = Data[17:16] = 11

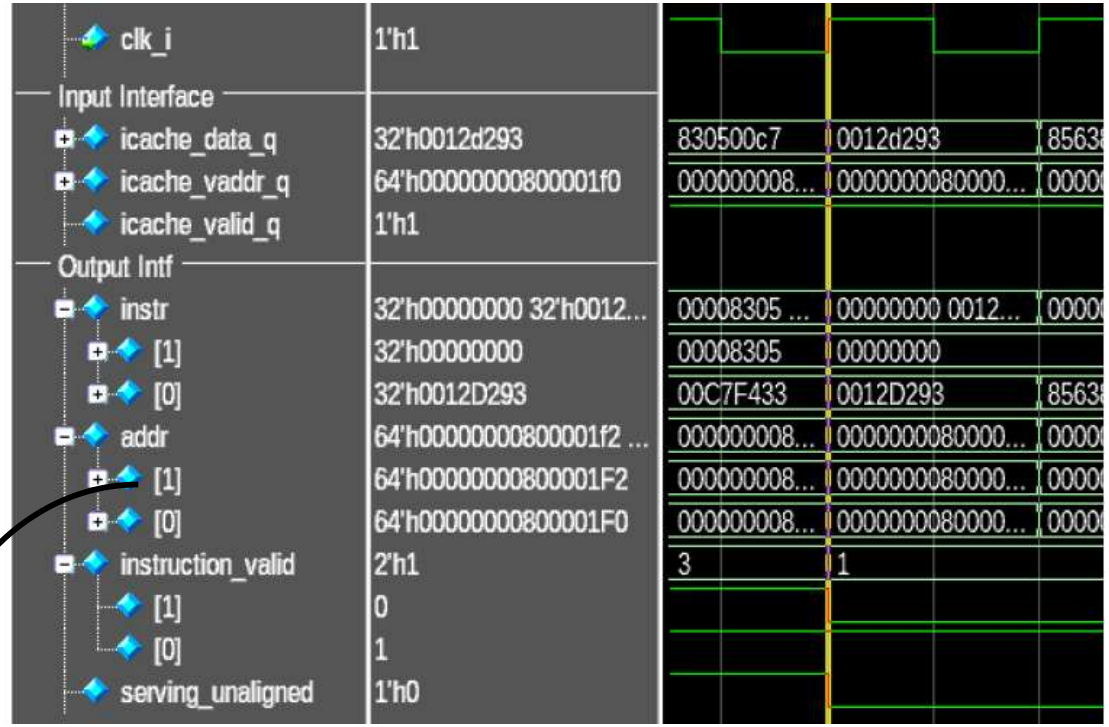
Non Compressed Instructions

All data is put into Instr[0]





Case 1)

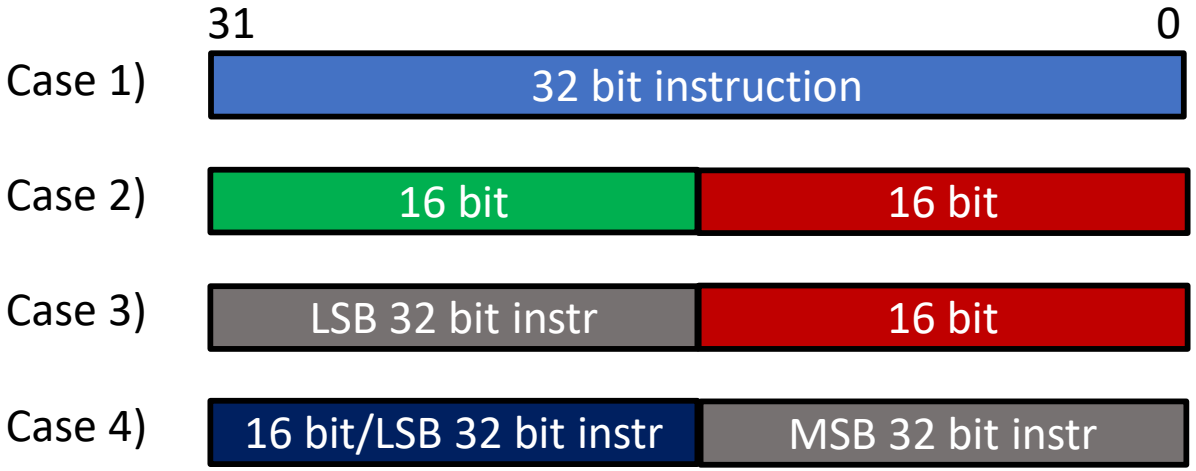


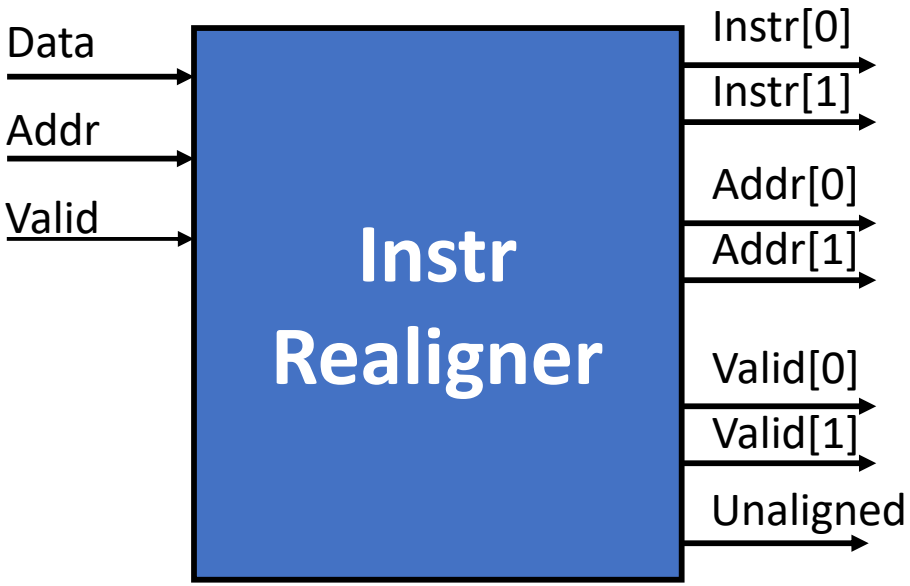
0x0012 d293 → Data[1:0] = Data[17:16] = 11

Non Compressed Instructions

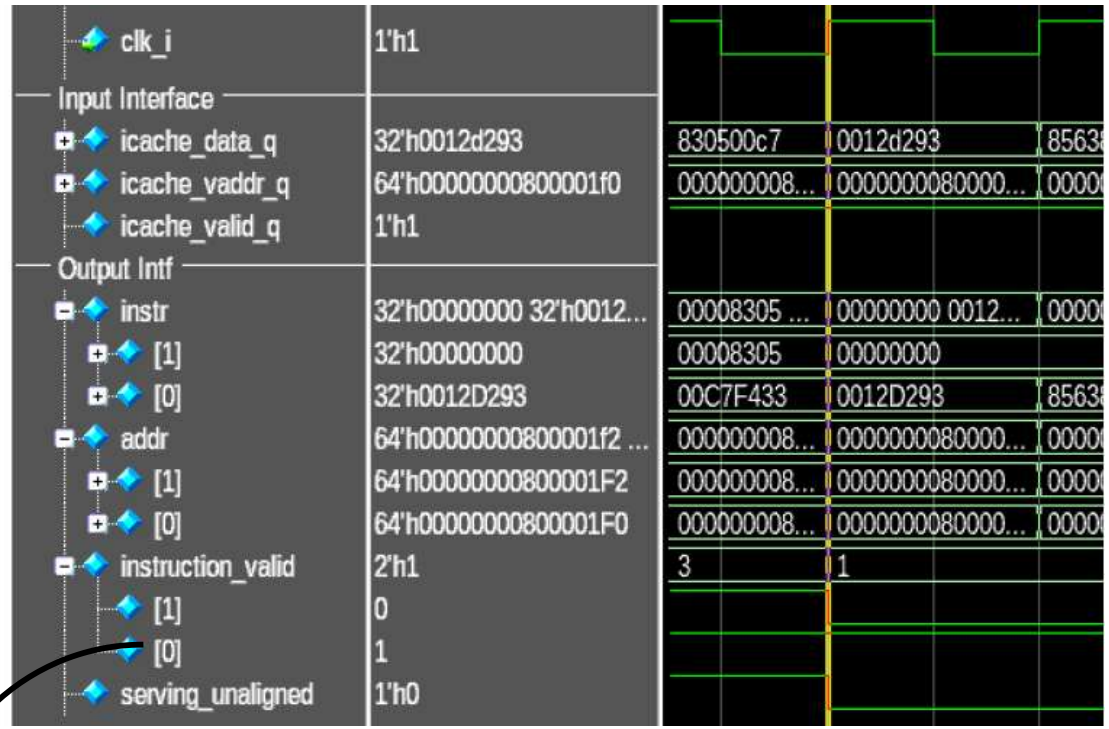
All data is put into Instr[0]

We have both Addr[0] and [1] with valid addresses





Case 1)



0x0012 d293 → Data[1:0] = Data[17:16] = 11

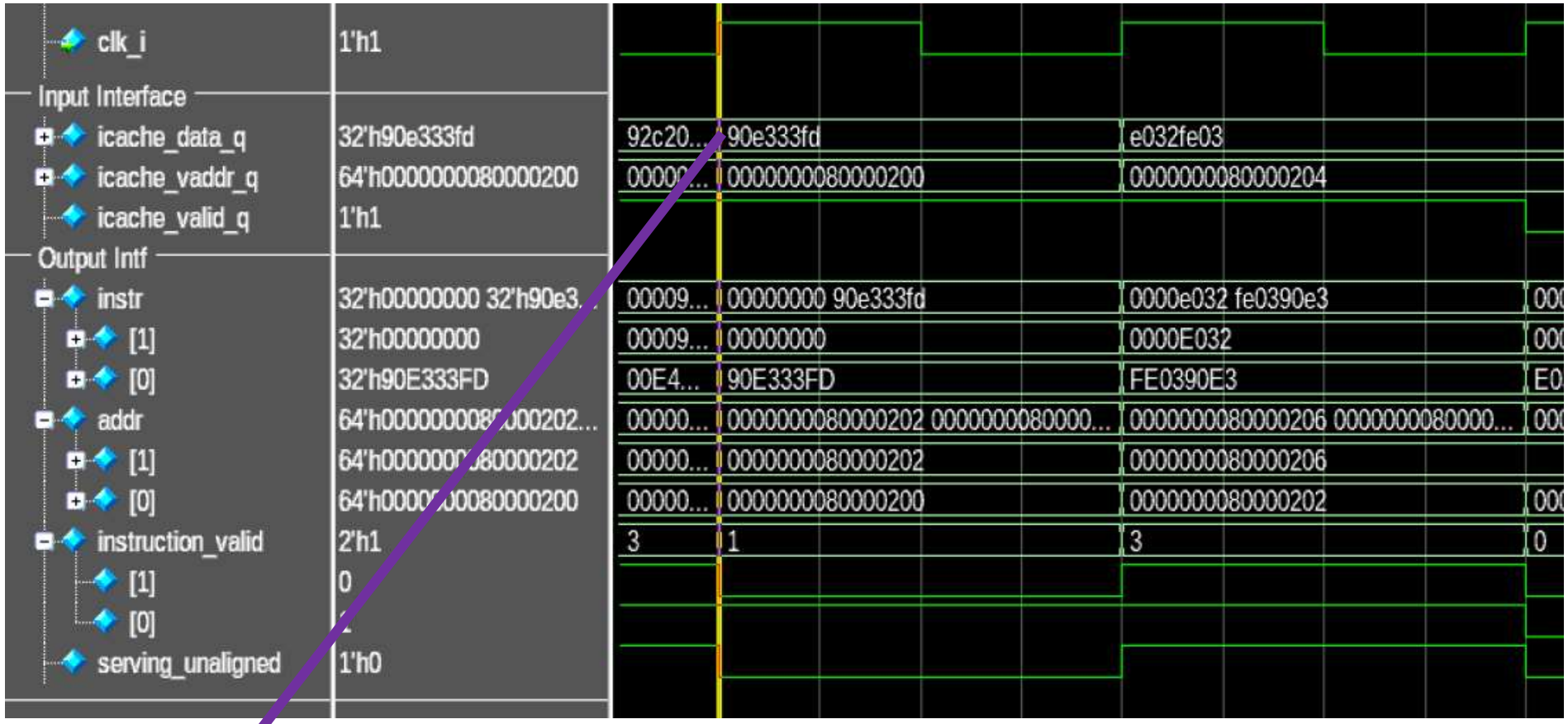
Non Compressed Instructions

All data is put into Instr[0]

We have both Addr[0] and [1] with valid addresses

But only valid[0] = 1 → Subsequent block in the Frontend will consider only instr[0]

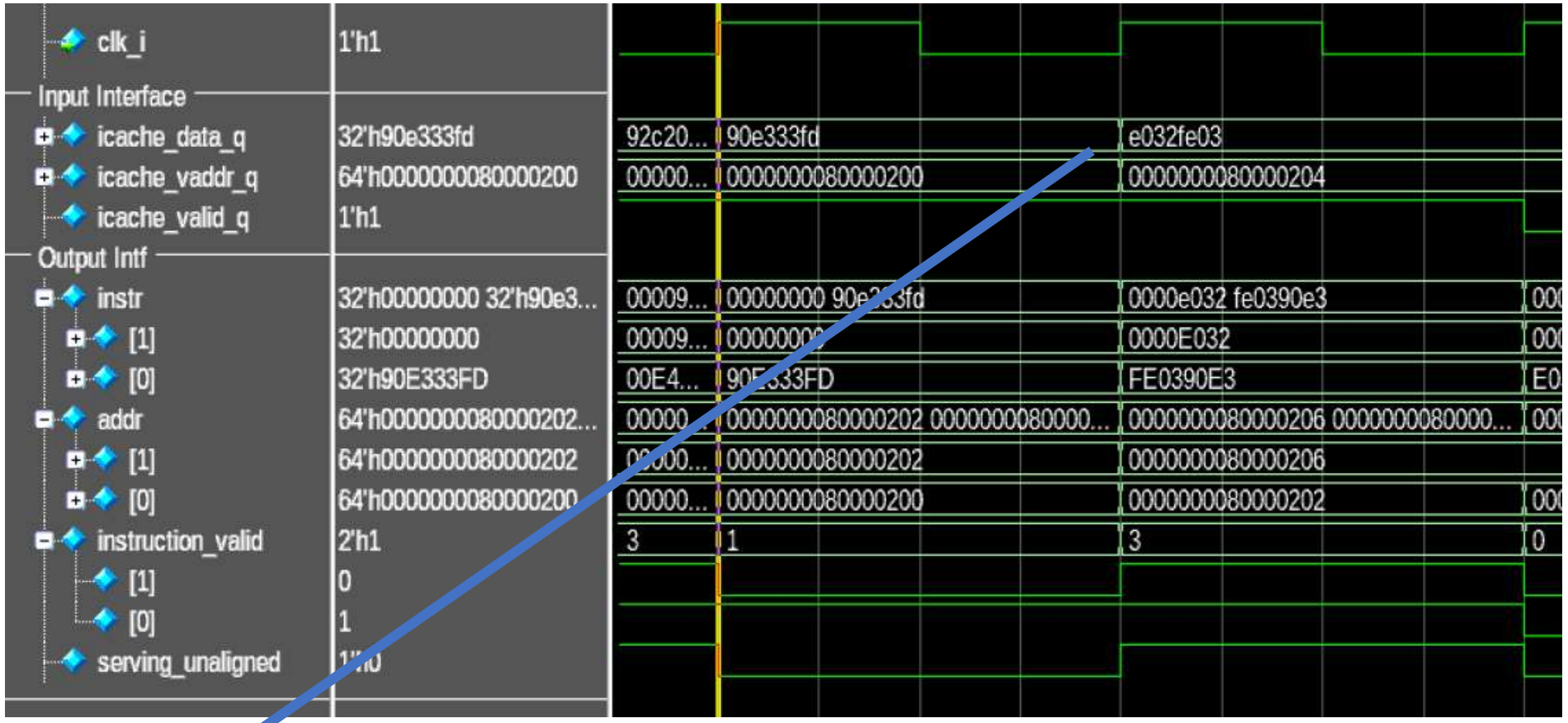
Case 3)



0x90E3 33FD → 0xD = 1101 (Compressed)
0x3 = 0011 (Not Compressed)

→ Store the Upper 16 bits into an inside register and serve the Non-Compressed instruction the next Cycle

Case 3)

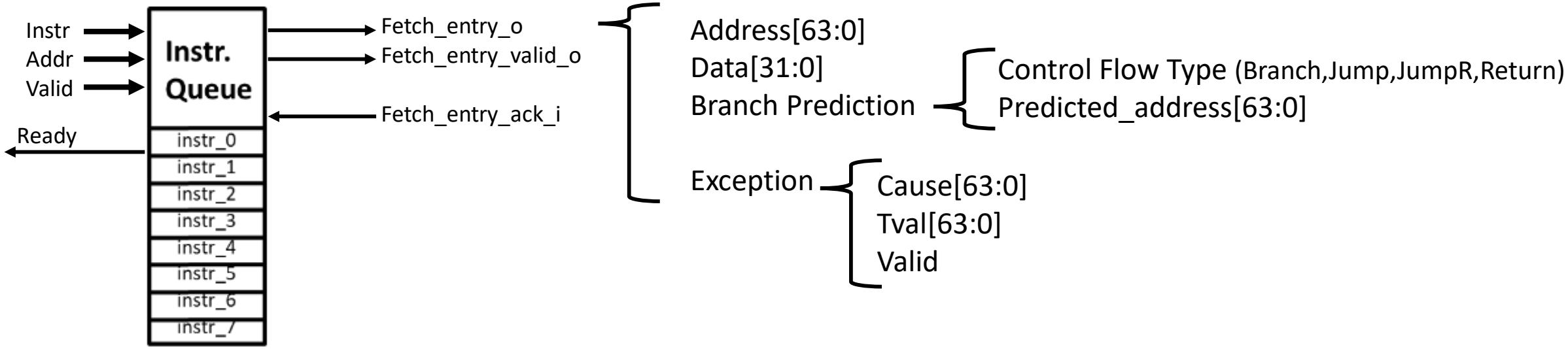


0xE032 FE03 → Upper 16bits Compressed!

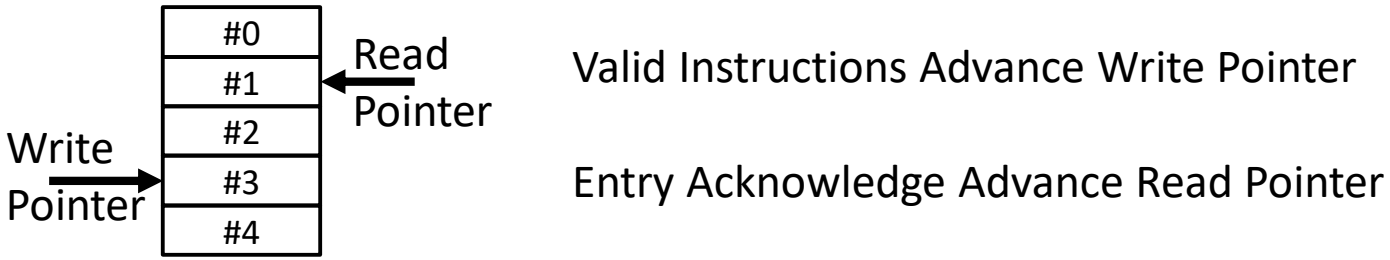
→ We can serve a compressed and the 32bit unaligned instruction in the same Cycle

Pushing into Instruction Queue

- Valid Instructions can be Pushed into the instruction Queue

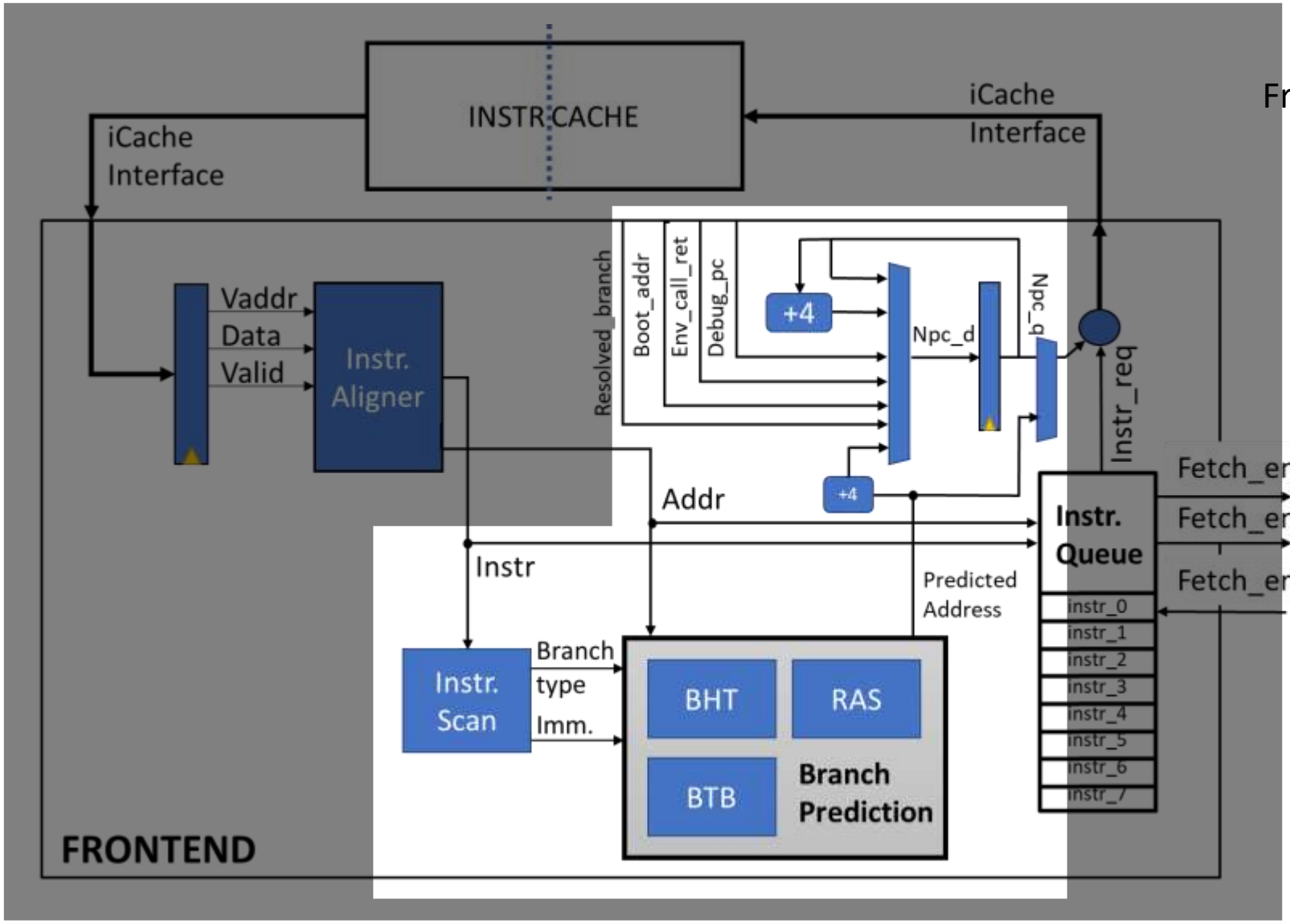


- Form of circular Buffer: pointer for write and read.



- If the Queue becomes full → ready = 0 stops request to cache

Next Program Counter Logic



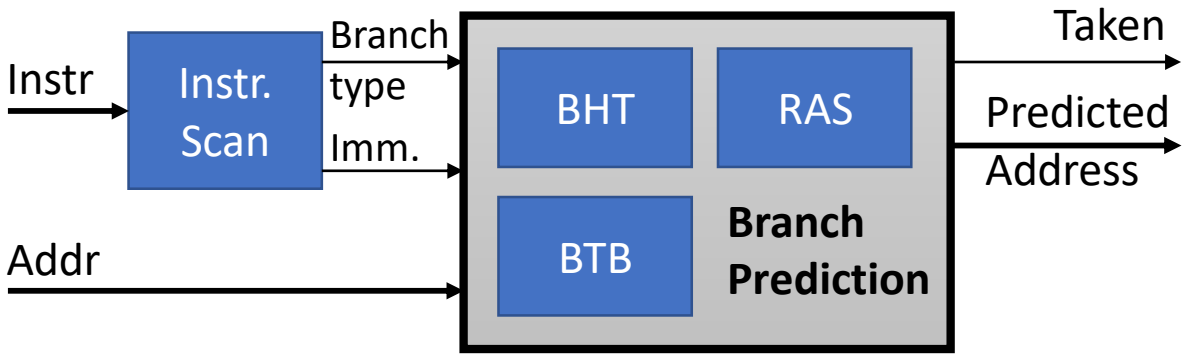
Frontend is divided in 3 main blocks:

- Current Instruction;
  - Realign Sampled Instruction from Instr. cache;
  - Store the instruction into instr. queue;
- Next Program Counter Logic:
  - Compute next PC depending on:
    - Booting;
    - Exceptions;
    - Debugging;
    - Branch;
    - Recovery from bad Prediction;
    - Sequential Fetching.

Branch Prediction

CVA6 implements Dynamic Branch Prediction:

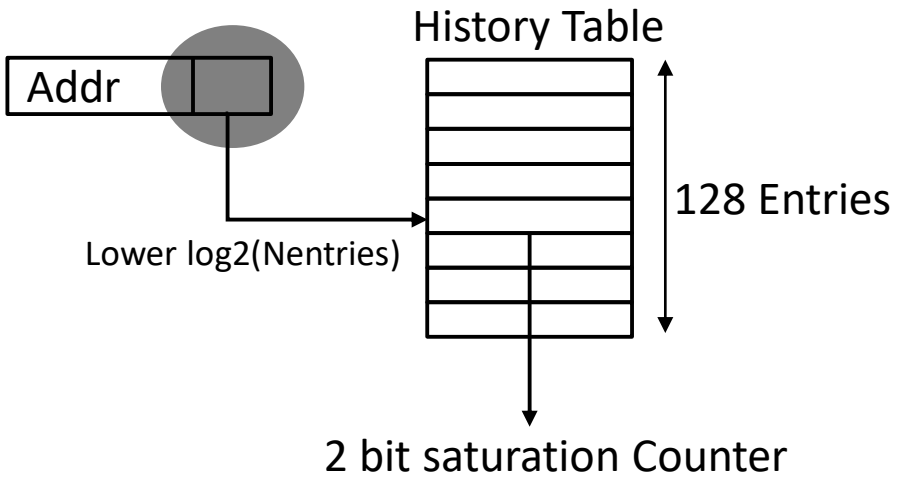
- Branch History Table (BHT): Used for predicting Branches (BEQ, BNE, etc).
- Branch Target Buffer (BTB): Used for predicting JALR.
- Return Address Stack (RAS): Used for predicting returns from function calls.

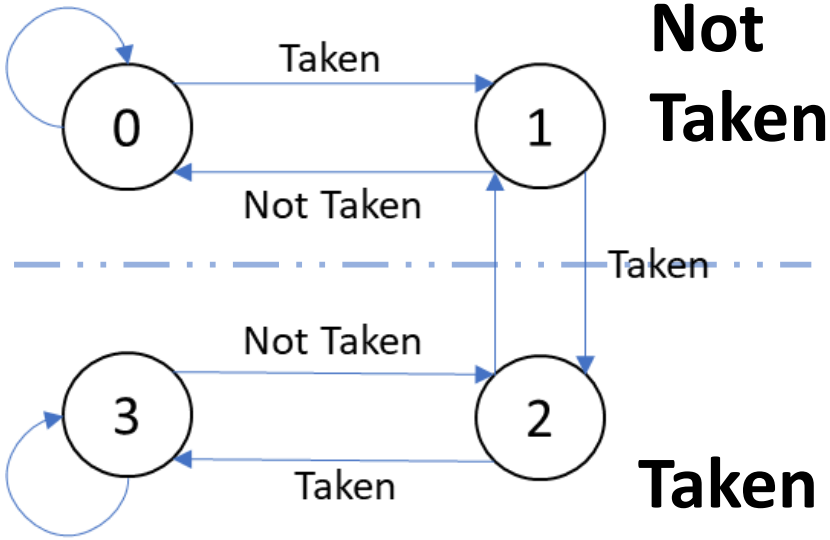


Instruction Scan Pre-decodes the instructions to control branch prediction:

- Extract Type of branch → Select Predictor.
- Extract Immediate → Calculation of predicted Address.

BHT: Bimodal

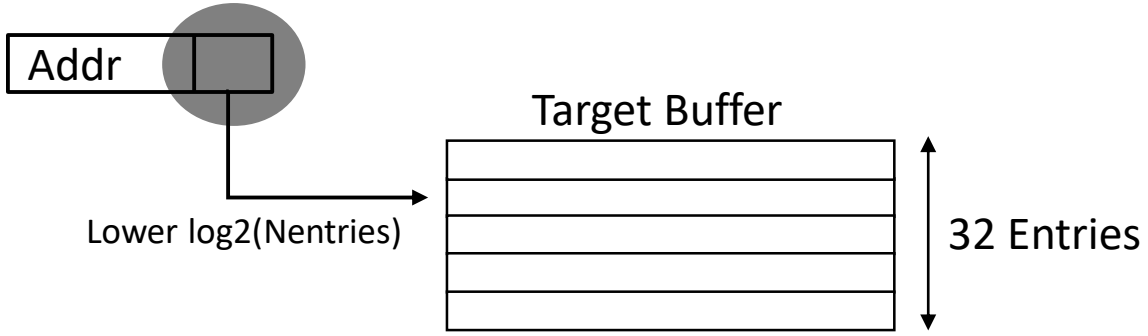




Not Taken

Taken

BTB

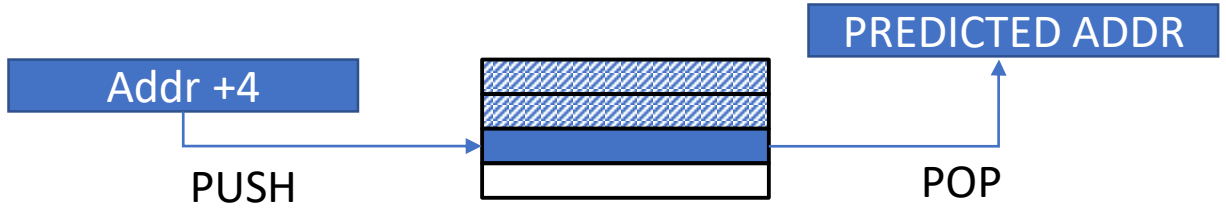


Each Buffer contains the last Address Known of the predicted JALR (that isn't a return).

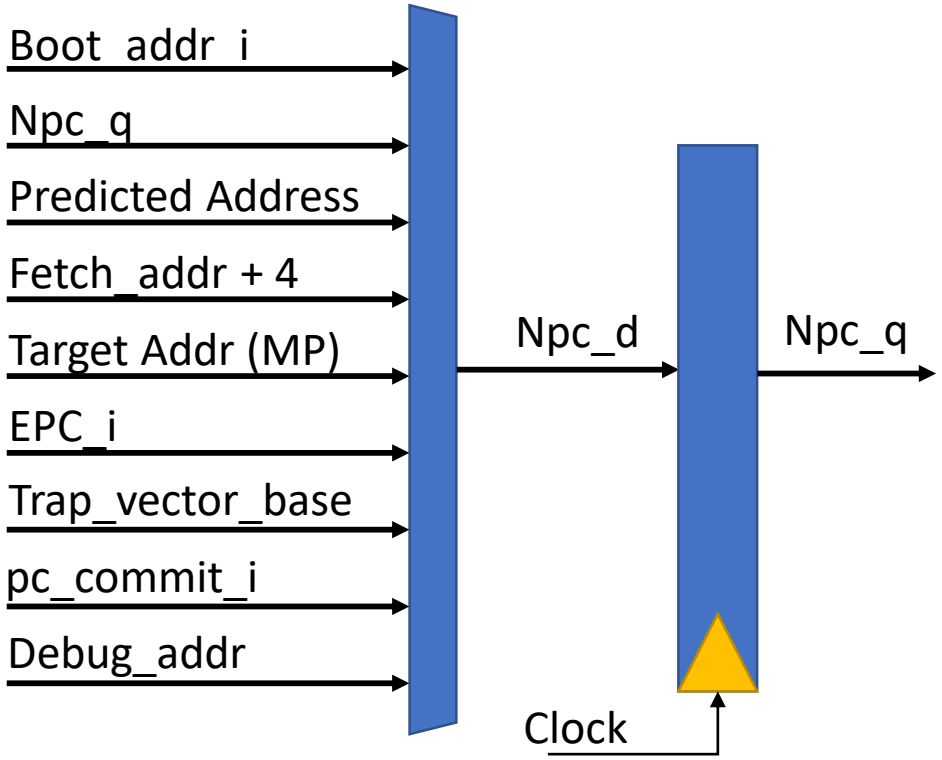
RAS

On Function Calls: PUSH → Save Return Address on the stack

On Function Return: POP → Use the popped Address as Predicted Address



Next Program Counter

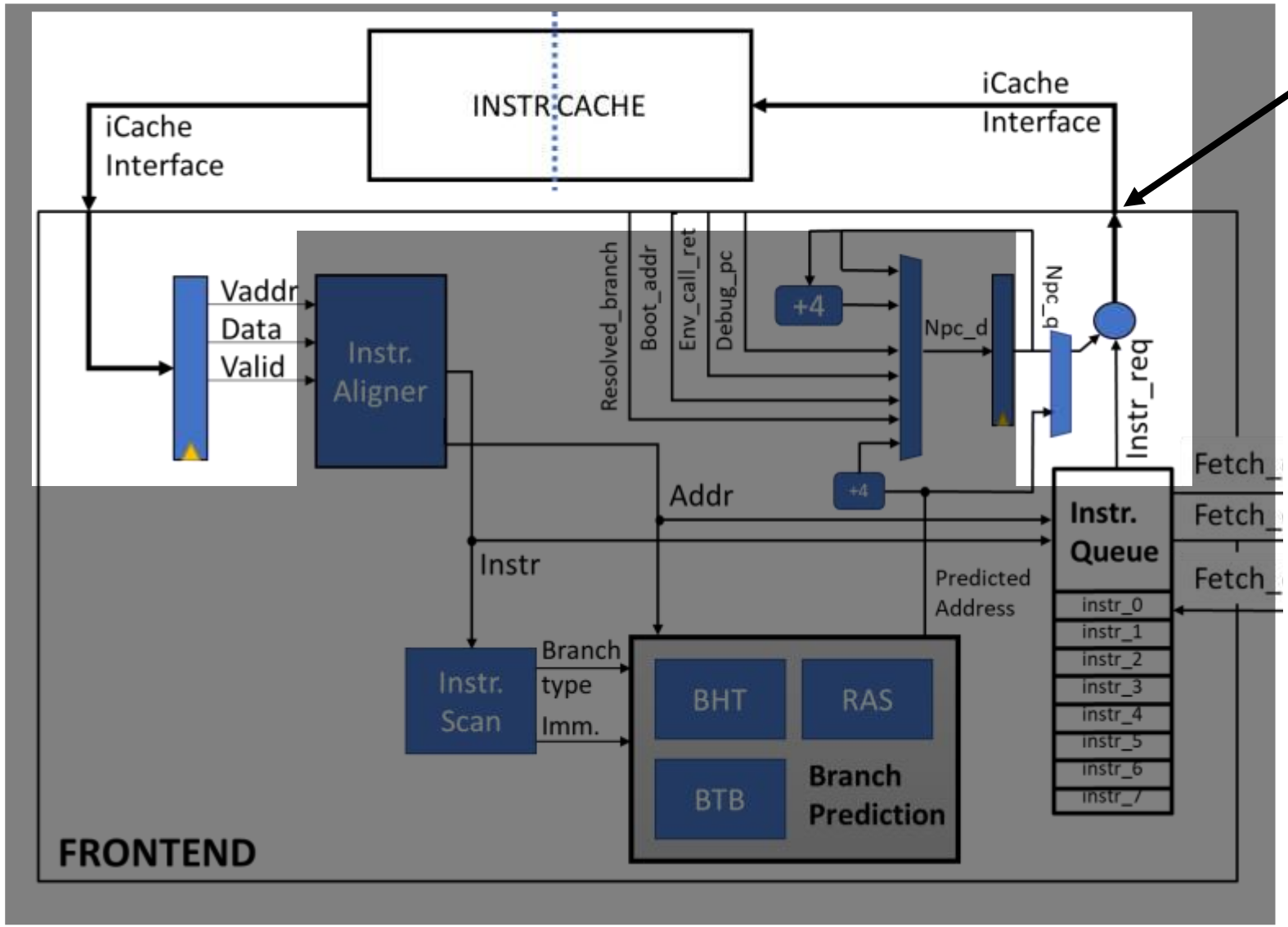


- NPC_Q → Next Program Counter at t0
- NPC_D → Next Program Counter at t1
- NPC_Q is used to decide the address of the next instruction to fetch.

SystemVerilog Code Snippet

```
always_comb begin : npc_select
  automatic logic [riscv::VLEN-1:0] fetch_address;
  // check whether we come out of reset
  // this is a workaround. some tools have issues
  // having boot_addr_i in the asynchronous
  // reset assignment to npc_q, even though
  // boot_addr_i will be assigned a constant
  // on the top-level.
  if (npc_rst_load_q) begin
    npc_d         = boot_addr_i;
    fetch_address = boot_addr_i;
  end else begin
    fetch_address    = npc_q;
    // keep stable by default
    npc_d            = npc_q;
  end
  // 0. Branch Prediction
  if (bp_valid) begin
    fetch_address = predict_address;
    npc_d = predict_address;
  end
  // 1. Default assignment
  if (if_ready) npc_d = {fetch_address[riscv::VLEN-1:2], 2'b0}  + 'h4;
  // 2. Replay instruction fetch
  if (replay) npc_d = replay_addr;
  // 3. Control flow change request
  if (is_mispredict) npc_d = resolved_branch_i.target_address;
  // 4. Return from environment call
  if (eret_i) npc_d = epc_i;
  // 5. Exception/Interrupt
  if (ex_valid_i) npc_d = trap_vector_base_i;
  // 6. Pipeline Flush because of CSR side effects
  // On a pipeline flush start fetching from the next address
  // of the instruction in the commit stage
  // we came here from a flush request of a CSR instruction or AMO,
  // as CSR or AMO instructions do not exist in a compressed form
  // we can unconditionally do PC + 4 here
  // TODO(zarubaf) This adder can at least be merged with the one in the csr_regfile stage
  if (set_pc_commit_i) npc_d = pc_commit_i + {{riscv::VLEN-3{1'b0}}, 3'b100};
  // 7. Debug
  // enter debug on a hard-coded base-address
  if (set_debug_pc_i) npc_d = ArianeCfg.DmBaseAddress[riscv::VLEN-1:0] + dm::HaltAddress[riscv::VLEN-1:0];
  icache_dreq_o.vaddr = fetch_address;
end
```

Cache Interfaces



NPC_Q will Be used for the Fetch Address of the next Instruction

Caches Interfaces Frontend – ICACHE

- Icache_dreq_o

Req: Signal Request to ICACHE.

Kill_S1: Kills Request on 1° stage.

Kill_S2: Kills Request on 2° Stage.

Vaddr: Address of Request.

- Icache_dreq_i

Ready: Cache Ready to accept requests.

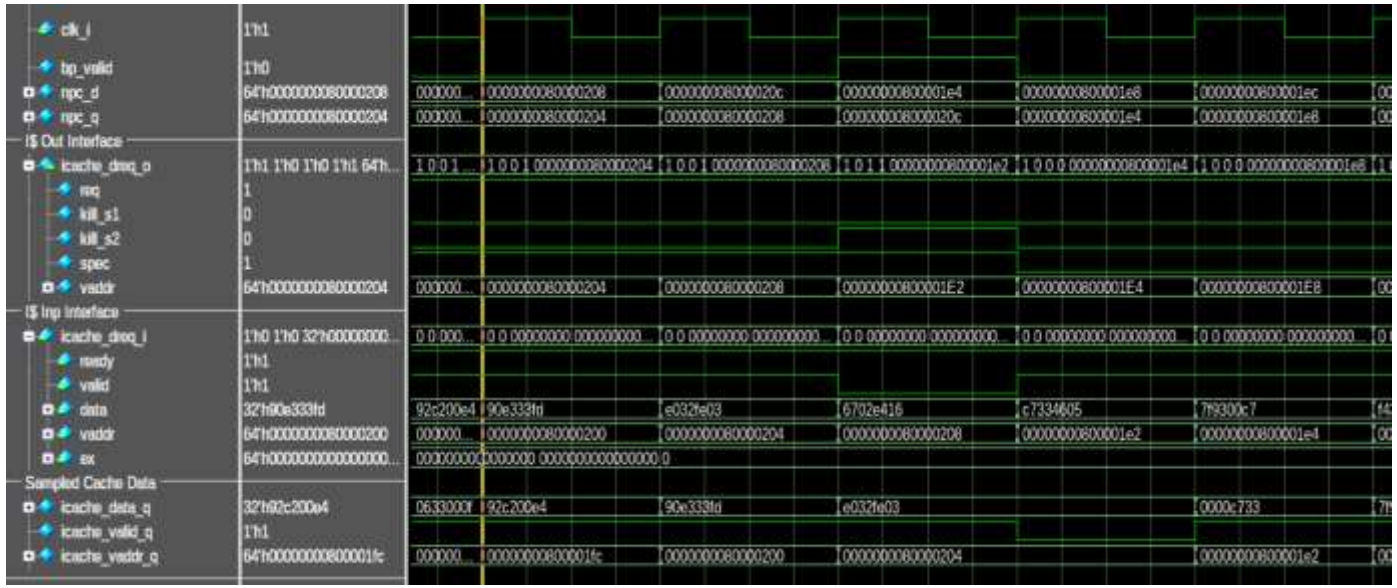
Valid: Data in the interface is Valid.

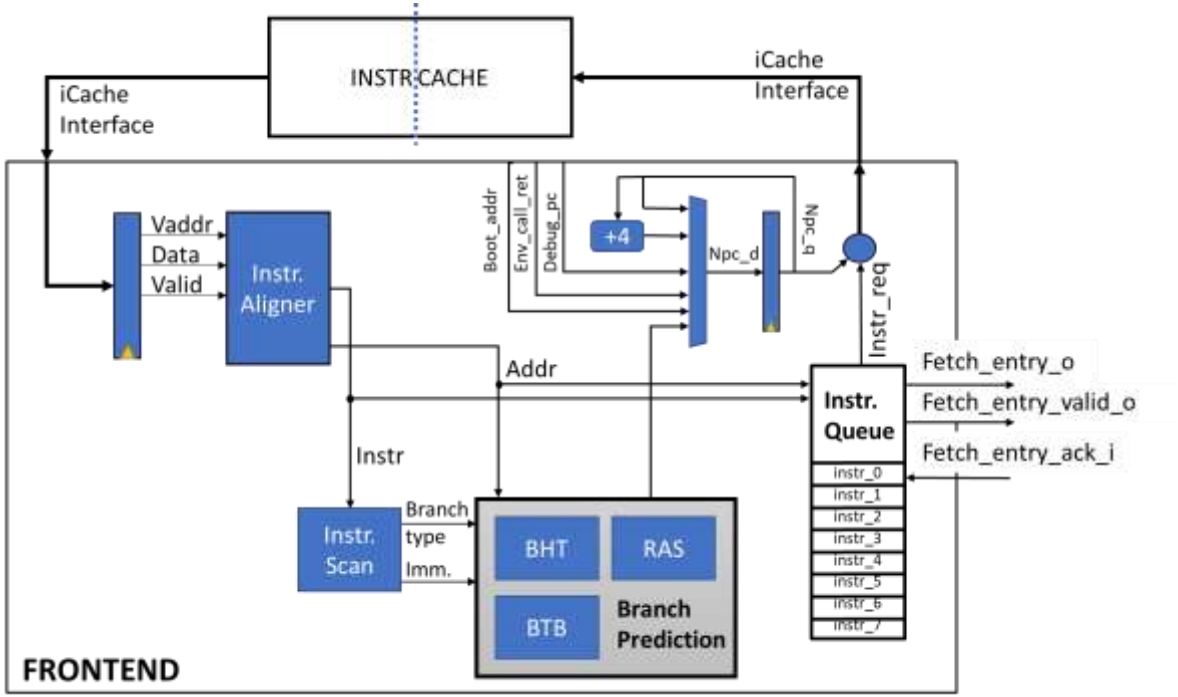
Data: Data read from ICACHE.

Vaddr: Address of the Data.

Ex: Exceptions from the request.

Cache Requests Examples

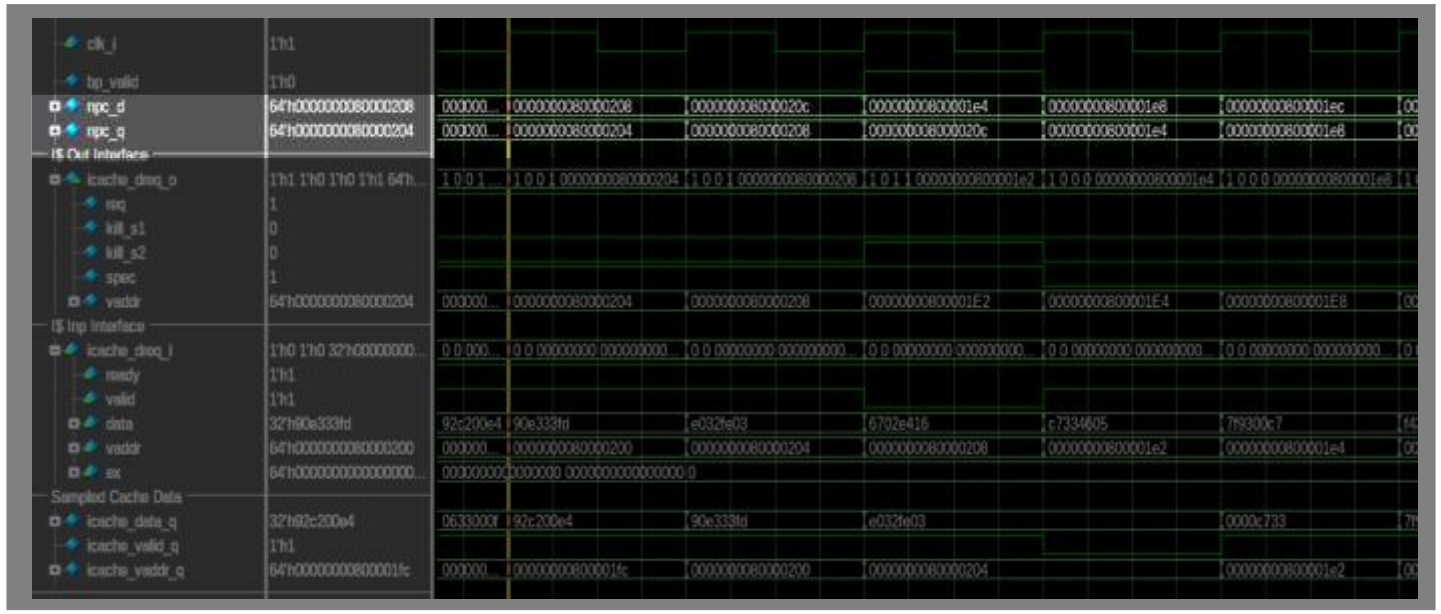


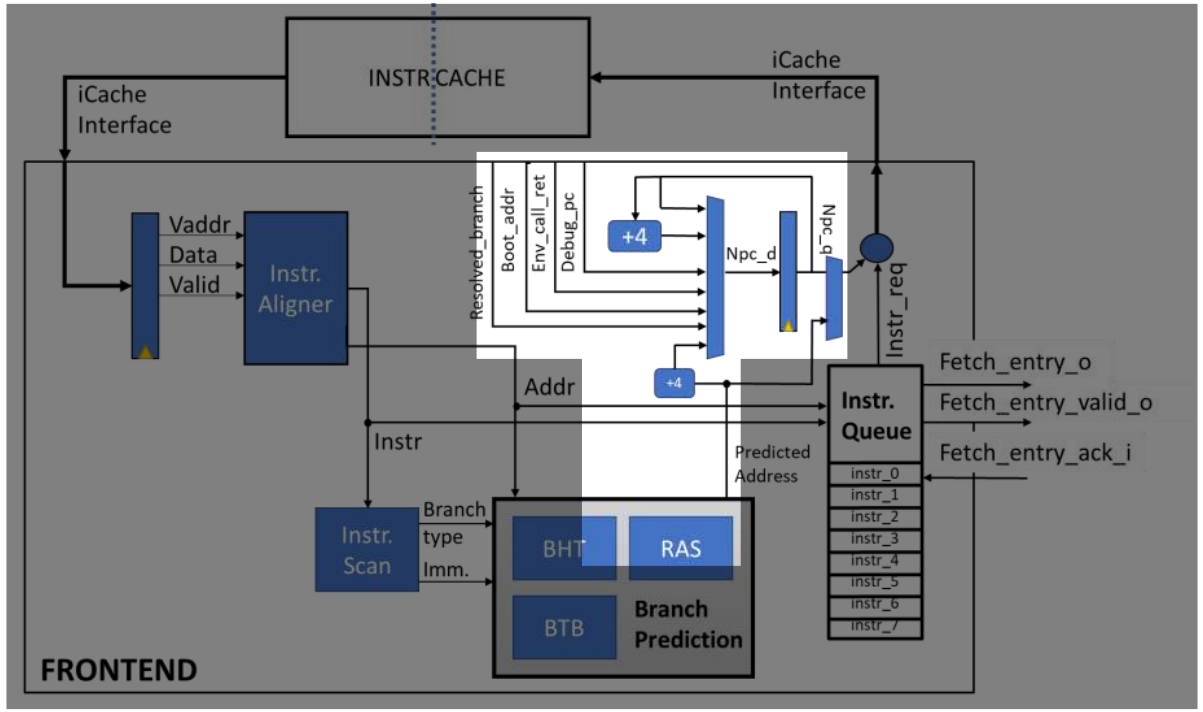


Signal Involved:

- NPC_Q & NPC_D;
- ICACHE_DREQ_O: Request Interface
- ICACHE_DREQ_I: Response Interface
- Sampled Data from ICACHE:
  - Icache_data_q
  - Icache_valid_q
  - Icache_vaddr_q

Cache Requests Examples

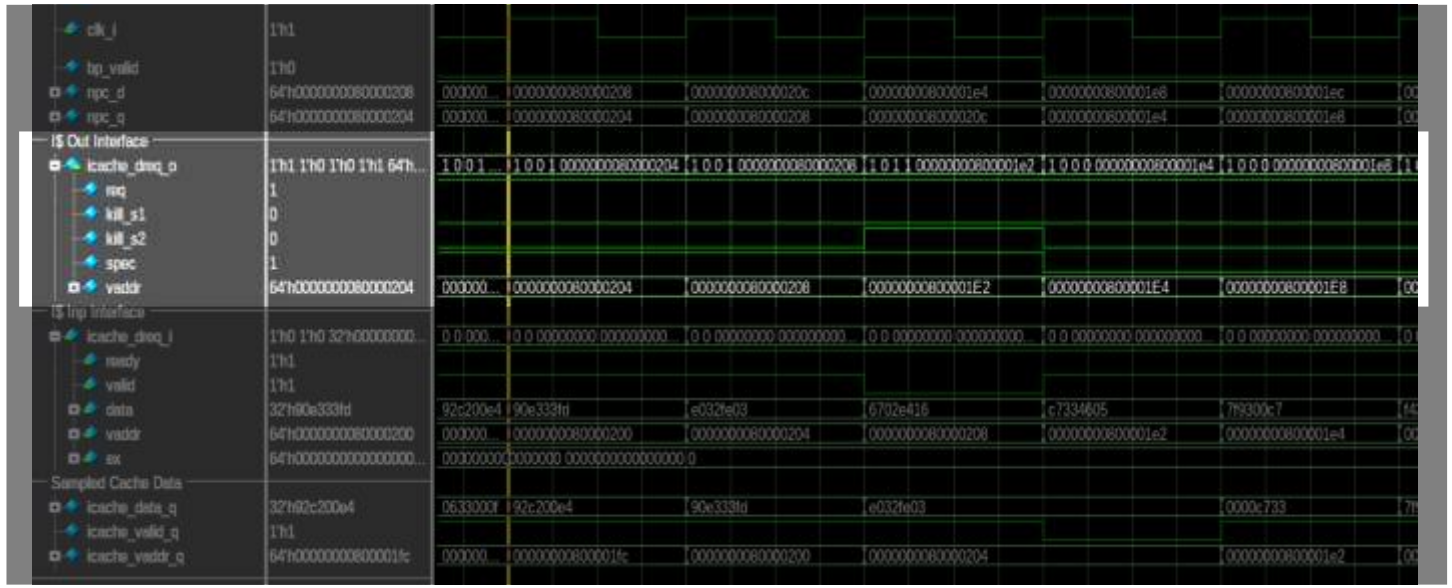


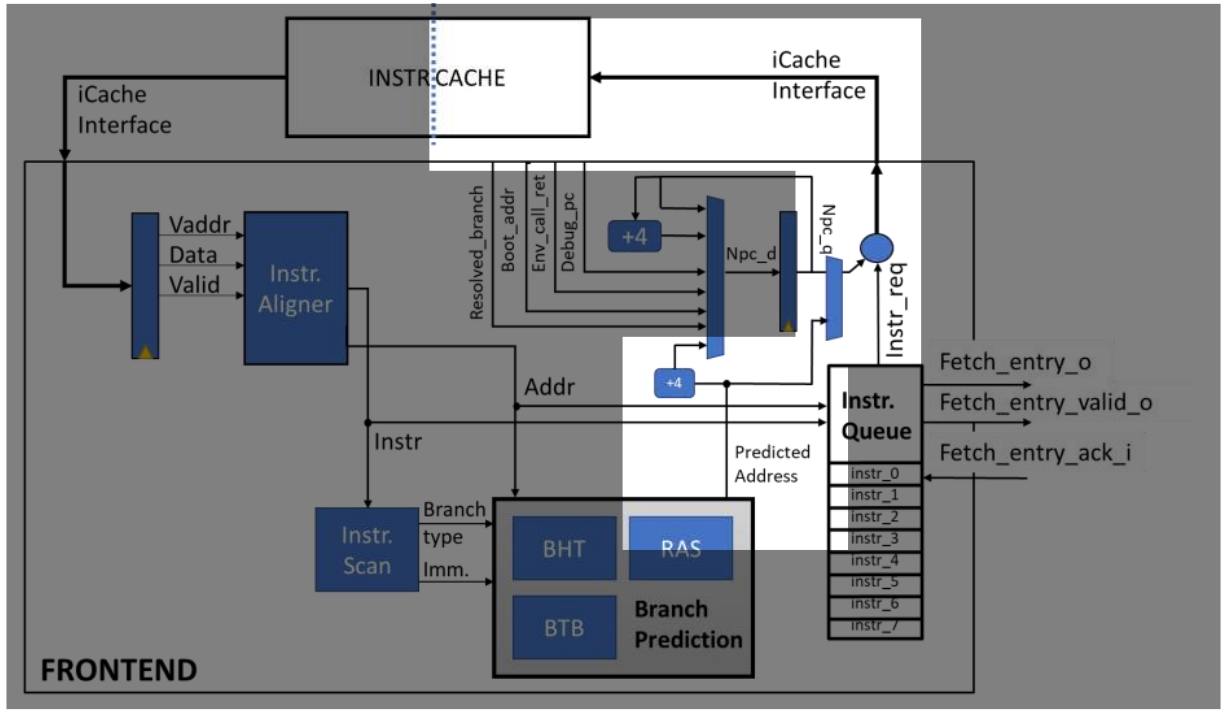


Signal Involved:

- NPC_Q & NPC_D;
- ICACHE_DREQ_O: Request Interface
- ICACHE_DREQ_I: Response Interface
- Sampled Data from ICACHE:
  - Icache_data_q
  - Icache_valid_q
  - Icache_vaddr_q

Cache Requests Examples

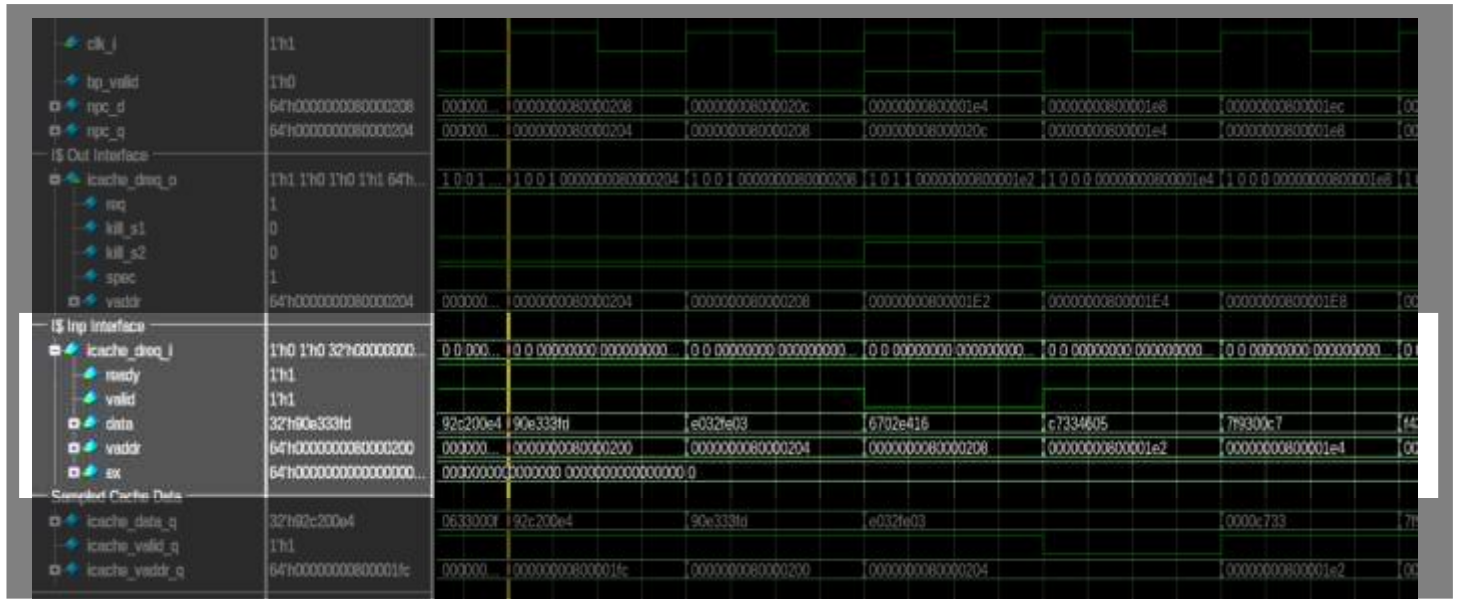


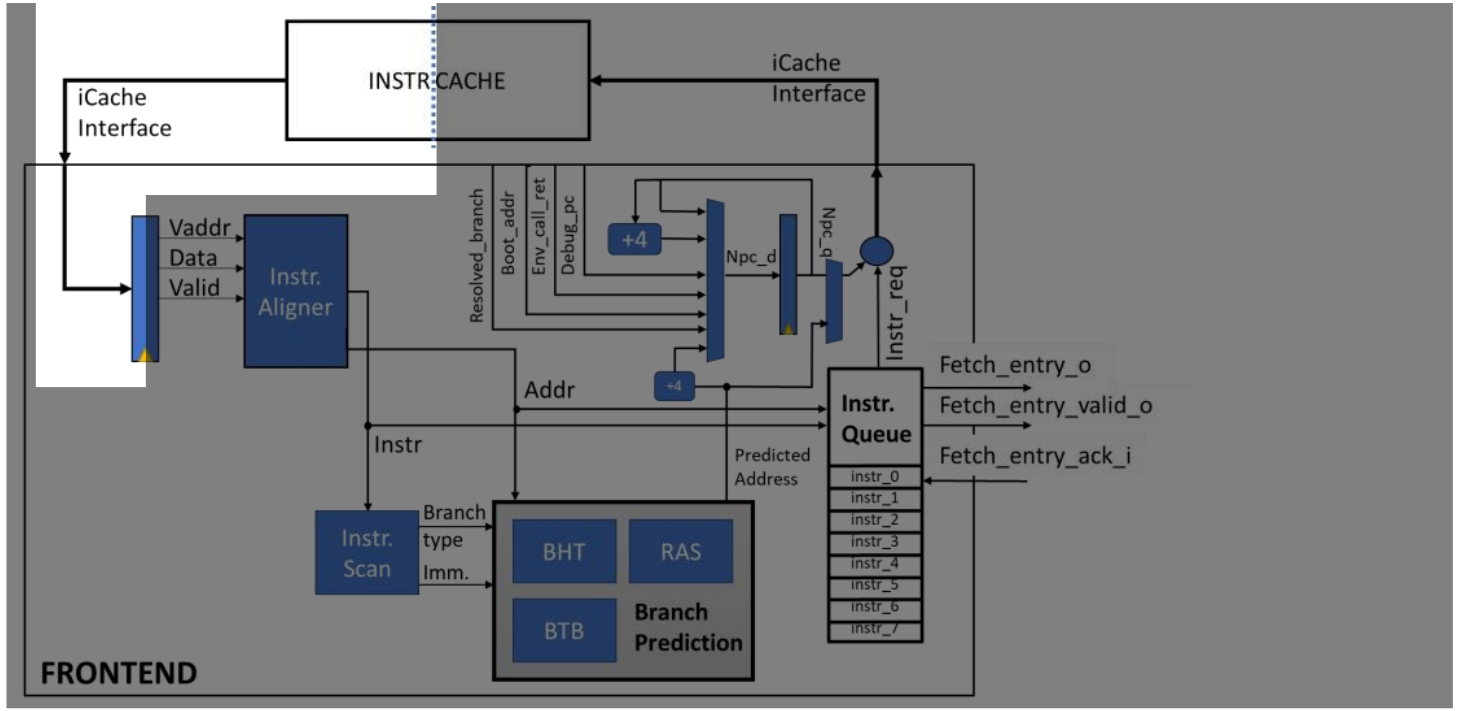


Signal Involved:

- NPC_Q & NPC_D;
- ICACHE_DREQ_O: Request Interface
- ICACHE_DREQ_I: Response Interface
- Sampled Data from ICACHE:
  - Icache_data_q
  - Icache_valid_q
  - Icache_vaddr_q

Cache Requests Examples

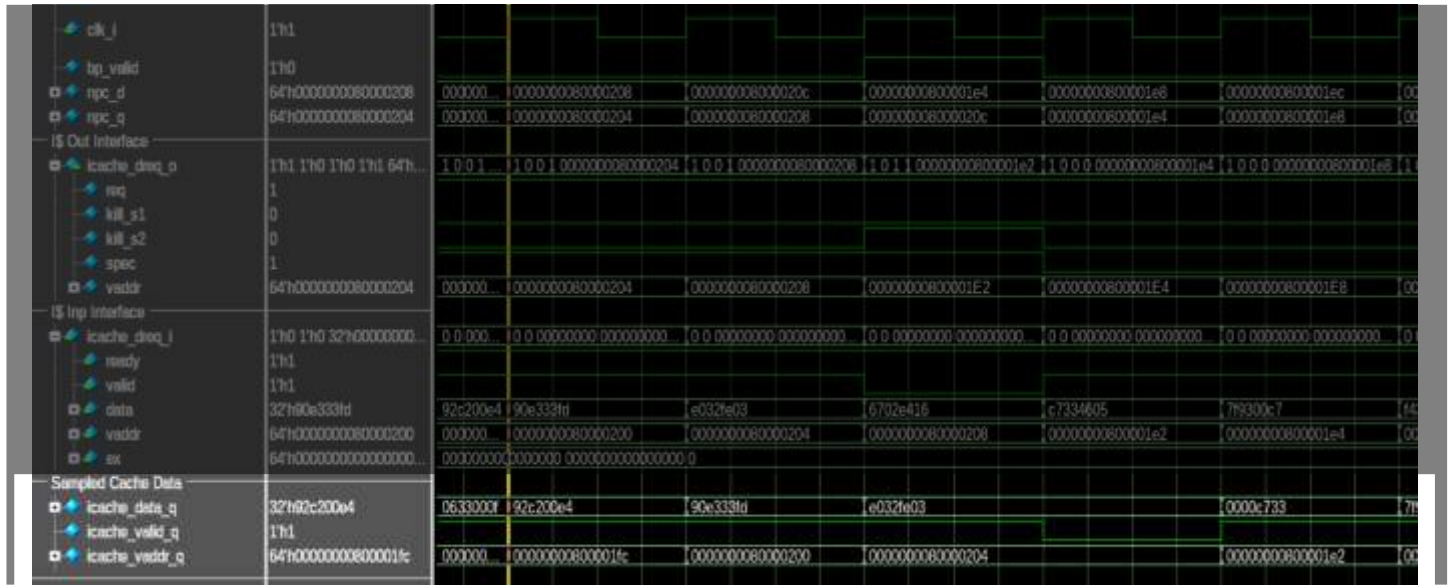


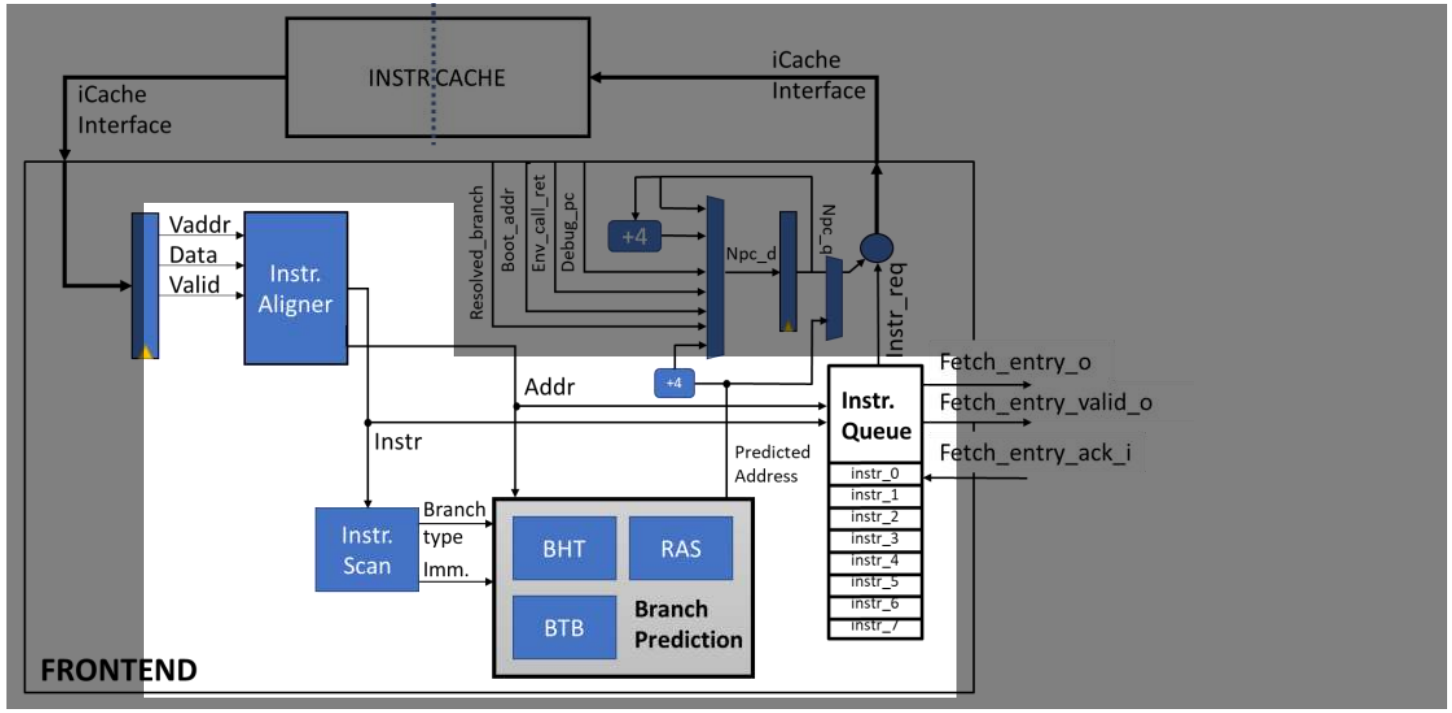


Signal Involved:

- NPC_Q & NPC_D;
- ICACHE_DREQ_O: Request Interface
- ICACHE_DREQ_I: Response Interface
- Sampled Data from ICACHE:
  - Icache_data_q
  - Icache_valid_q
  - Icache_vaddr_q

Cache Requests Examples

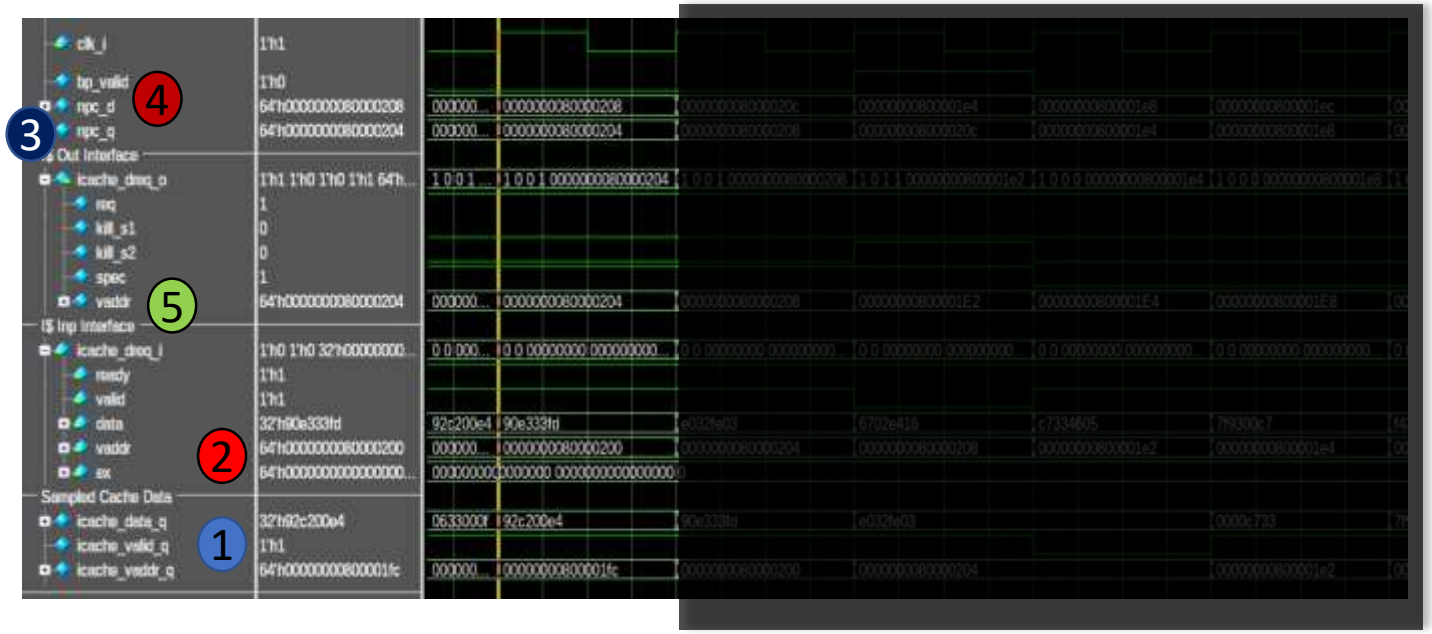


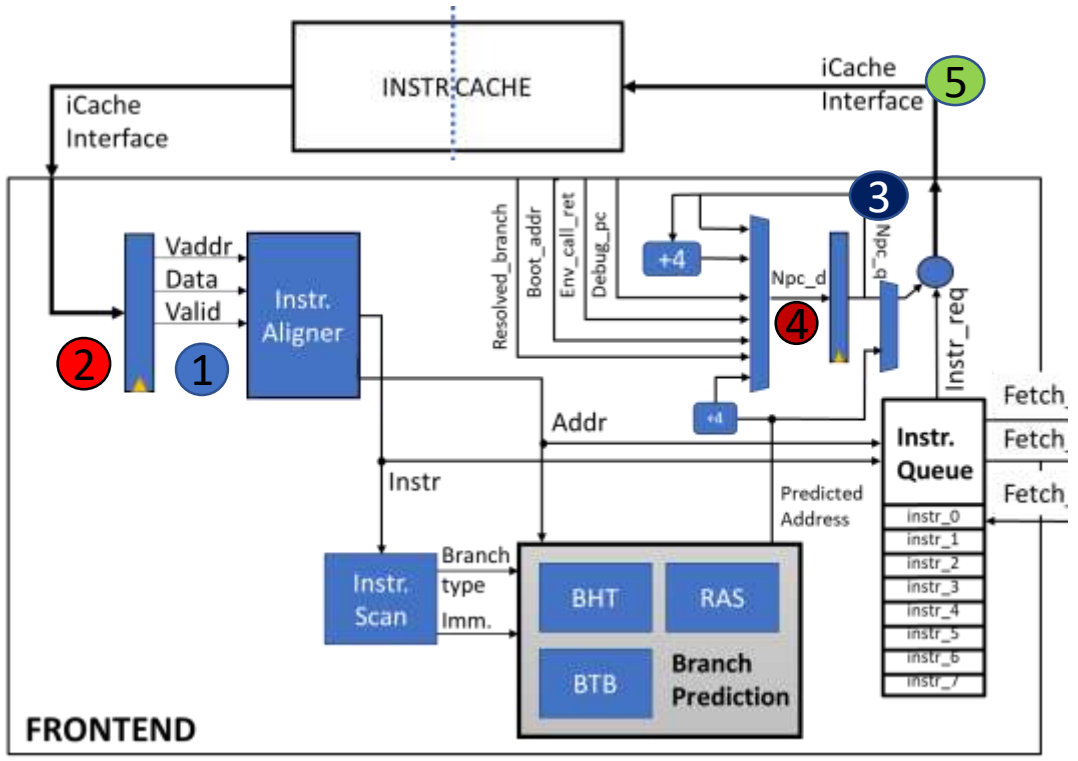


Signal Involved:

- NPC_Q & NPC_D;
- ICACHE_DREQ_O: Request Interface
- ICACHE_DREQ_I: Response Interface
- Sampled Data from ICACHE:
  - Icache_data_q
  - Icache_valid_q
  - Icache_vaddr_q

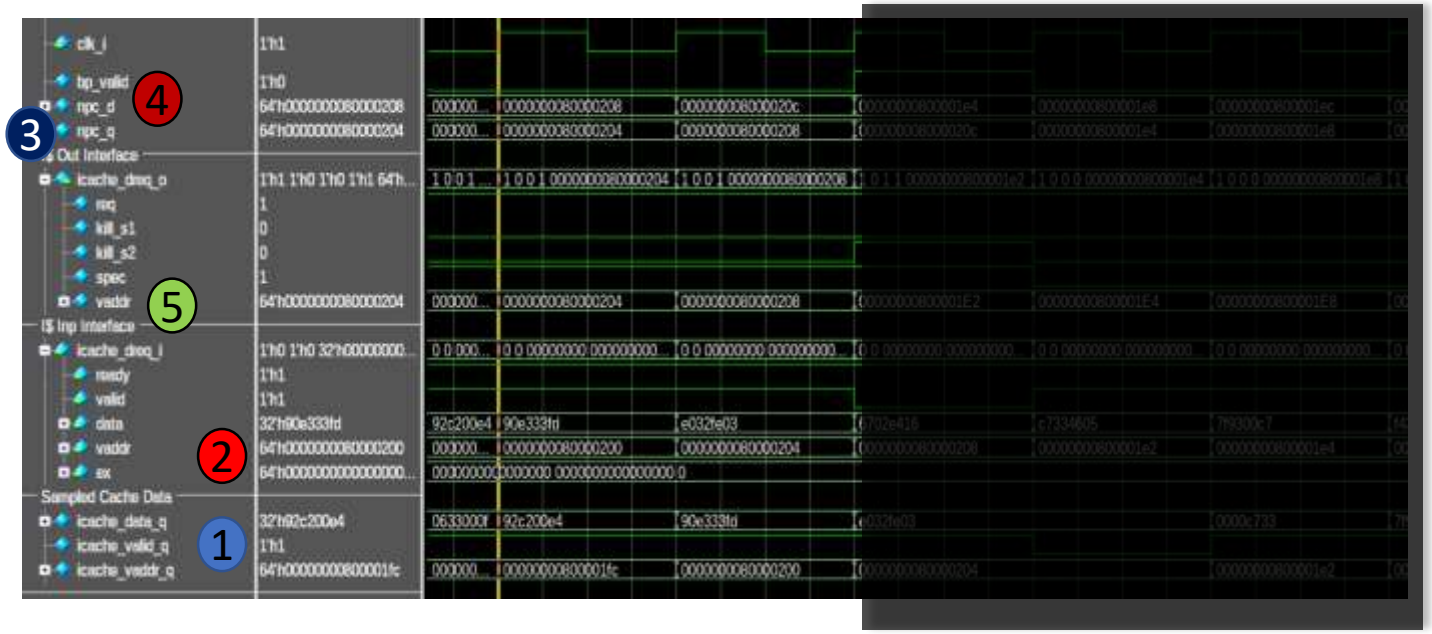
Cache Requests Examples

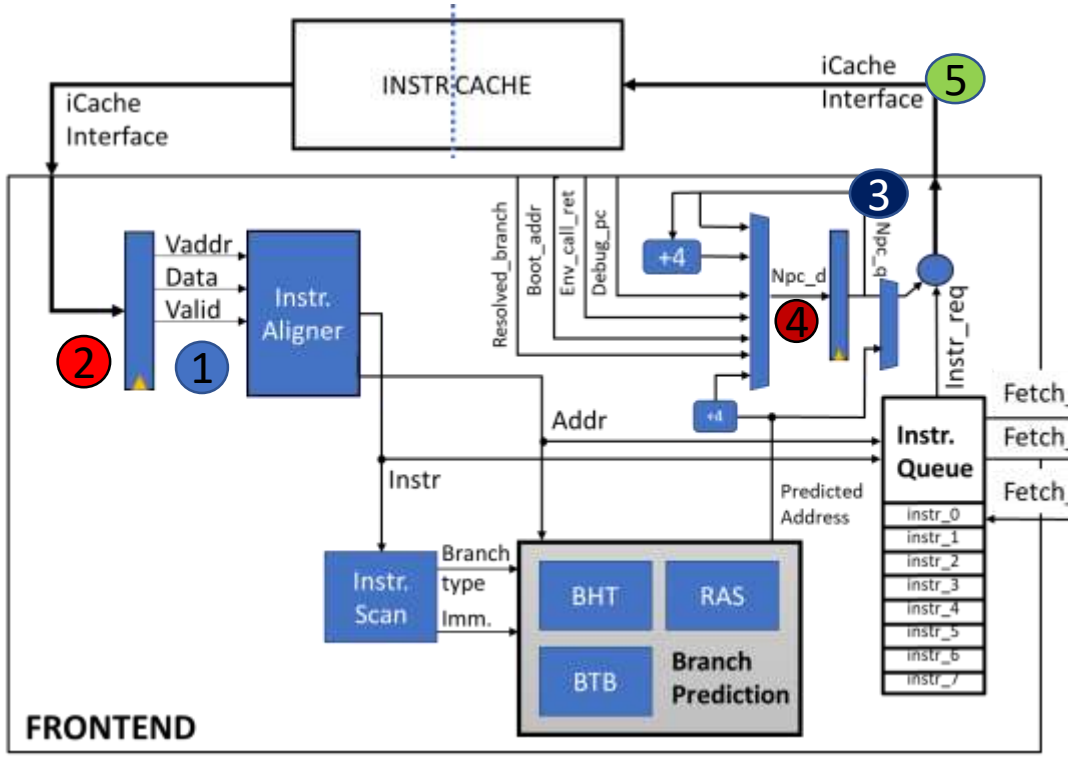




	SIGNAL	T0	T1	T2	T3	T4
1	ICACHE_VADDR_Q	0x01FC	0x0200	0x0204	0x0208	0x01E2
2	DREQ_I.VADDR	0x0200	0x0204	0x0208	0x01E2	0x01E4
3	NPC_Q	0x0204	0x0208	0x020C	0x01E4	0x01E8
4	NPC_D	0x0208	0x020C	0x01E4	0x01E8	0x01EC
5	DREQ_O.VADDR	0x0204	0x0208	0x01E2	0x01E4	0x01E8

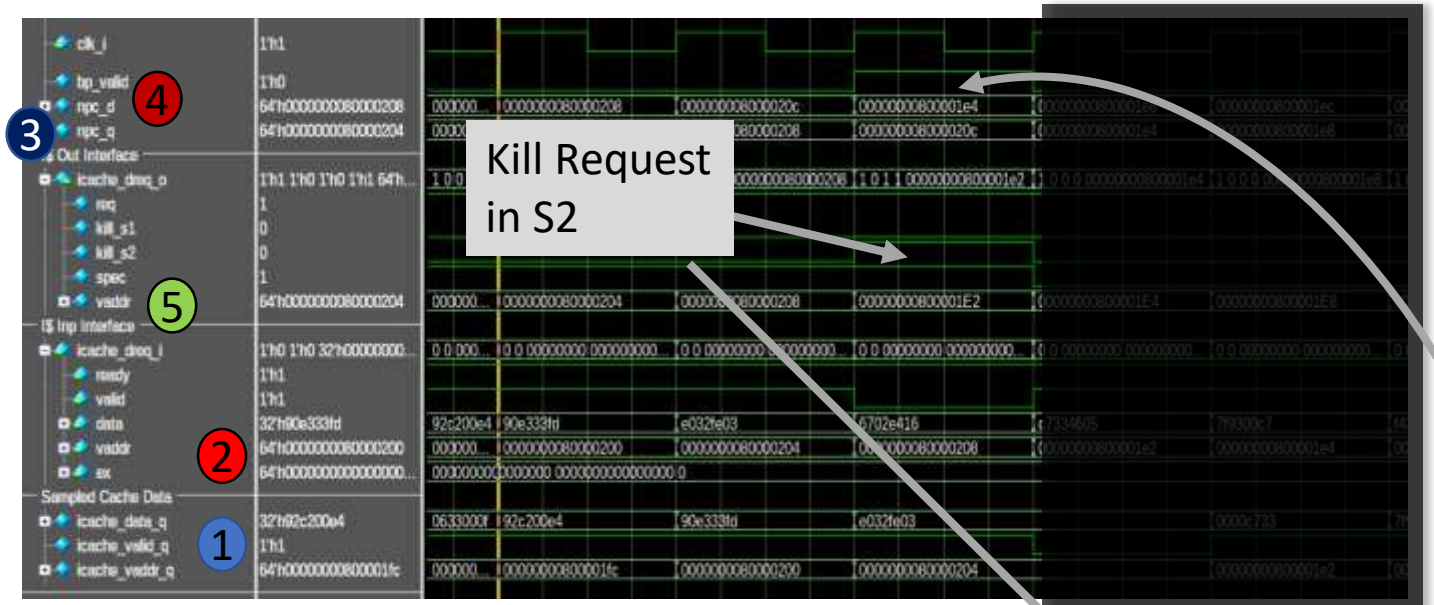
Cache Requests Examples

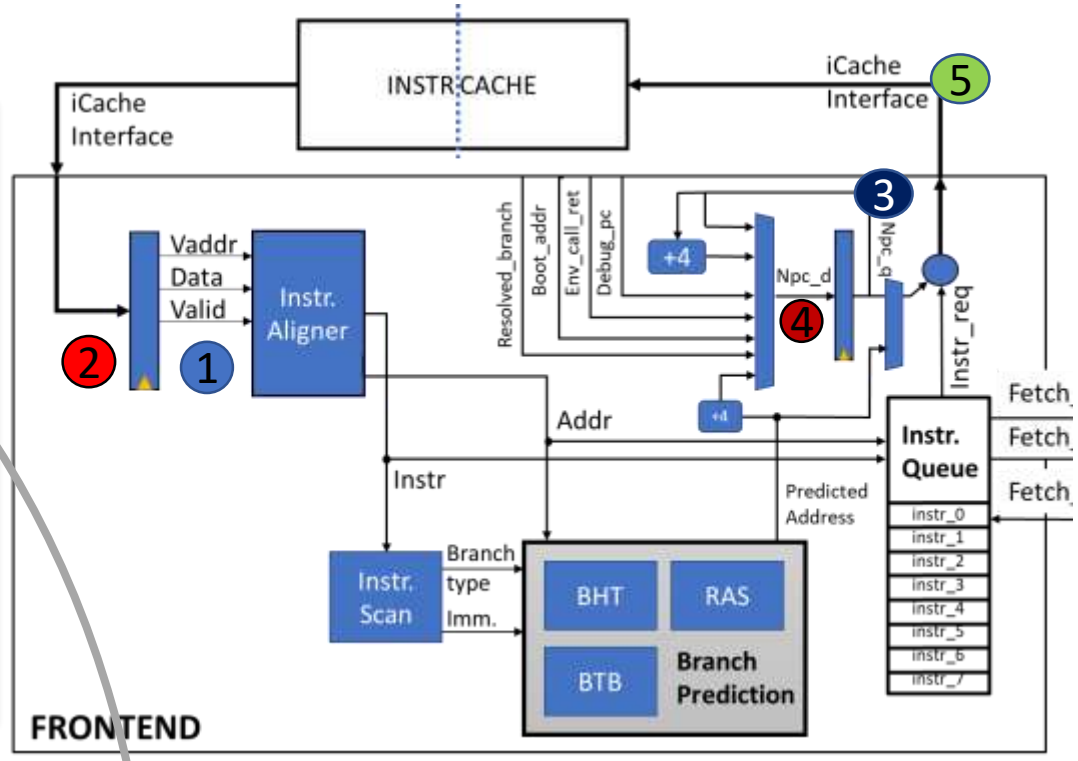




	SIGNAL	T0	T1	T2	T3	T4
1	ICACHE_VADDR_Q	0x01FC	0x0200	0x0204	0x0208	0x01E2
2	DREQ_I.VADDR	0x0200	0x0204	0x0208	0x01E2	0x01E4
3	NPC_Q	0x0204	0x0208	0x020C	0x01E4	0x01E8
4	NPC_D	0x0208	0x020C	0x01E4	0x01E8	0x01EC
5	DREQ_O.VADDR	0x0204	0x0208	0x01E2	0x01E4	0x01E8

Cache Requests Examples

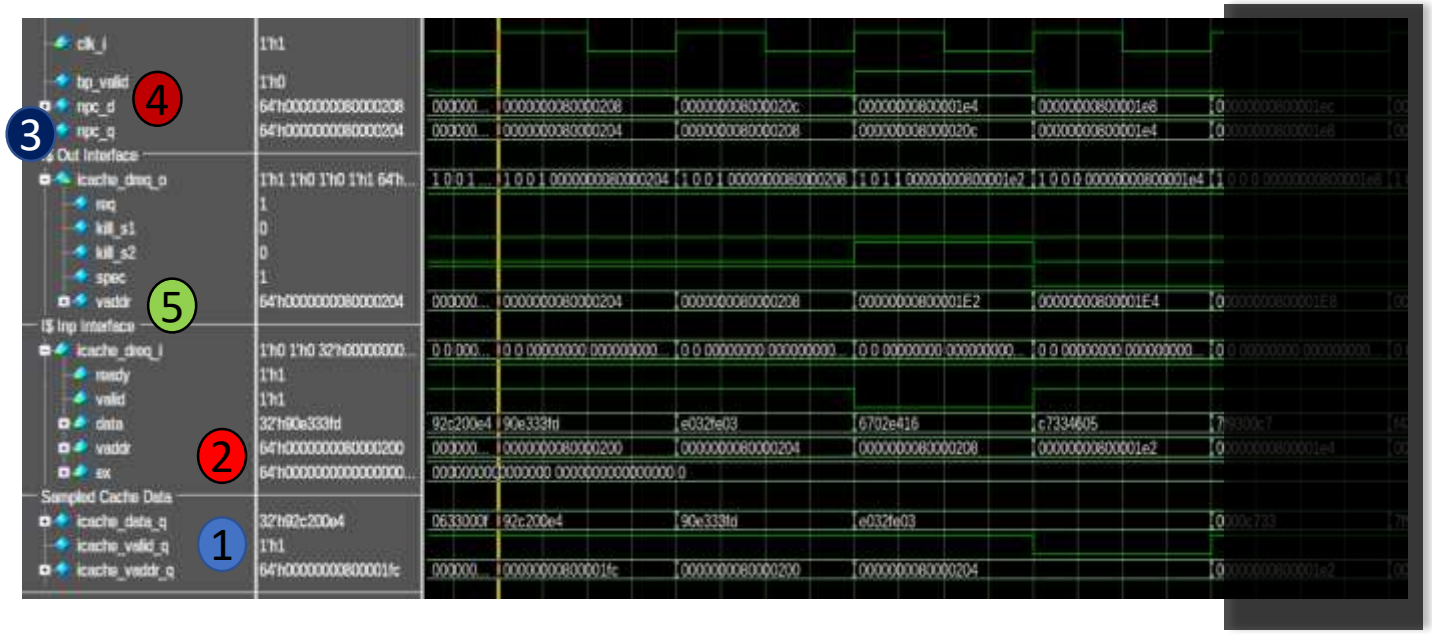


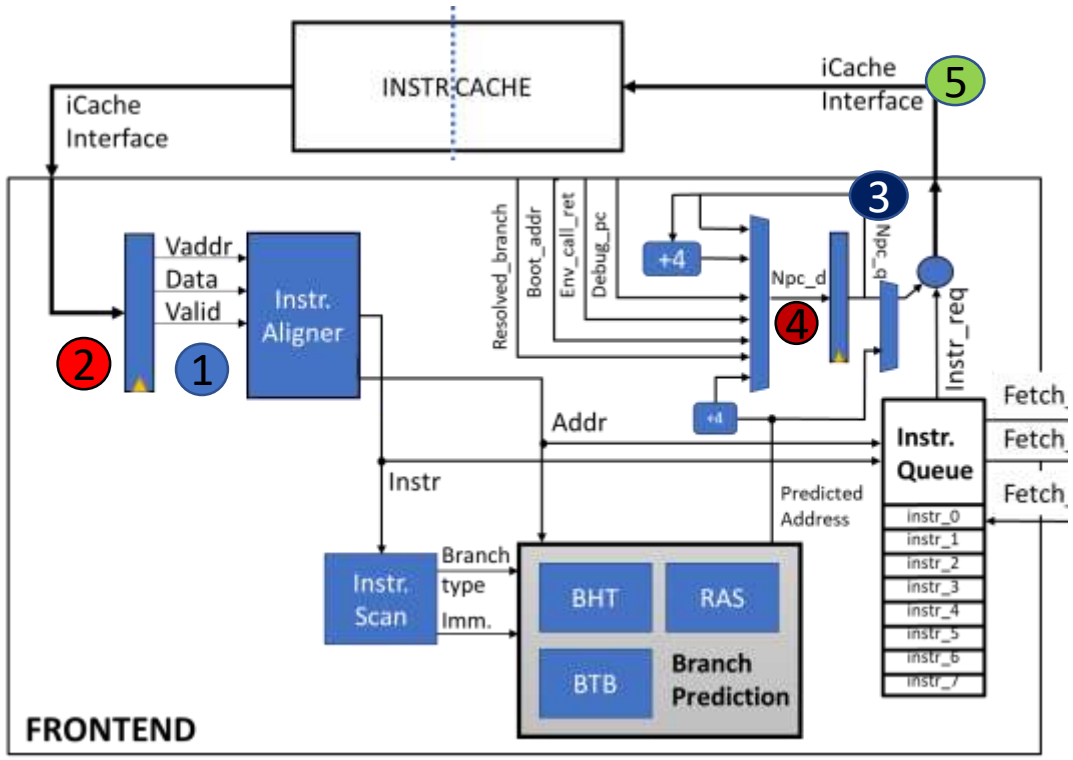


Invalidated

	SIGNAL	T0	T1	T2	T3	T4
1	ICACHE_VADDR_Q	0x01FC	0x0200	0x0204	0x0208	0x01E2
2	DREQ_I.VADDR	0x0200	0x0204	0x0208	0x01E2	0x01E4
3	NPC_Q	0x0204	0x0208	0x020C	0x01E4	0x01E8
4	NPC_D	0x0208	0x020C	0x01E4	0x01E8	0x01EC
5	DREQ_O.VADDR	0x0204	0x0208	0x01E2	0x01E4	0x01E8

Cache Requests Examples

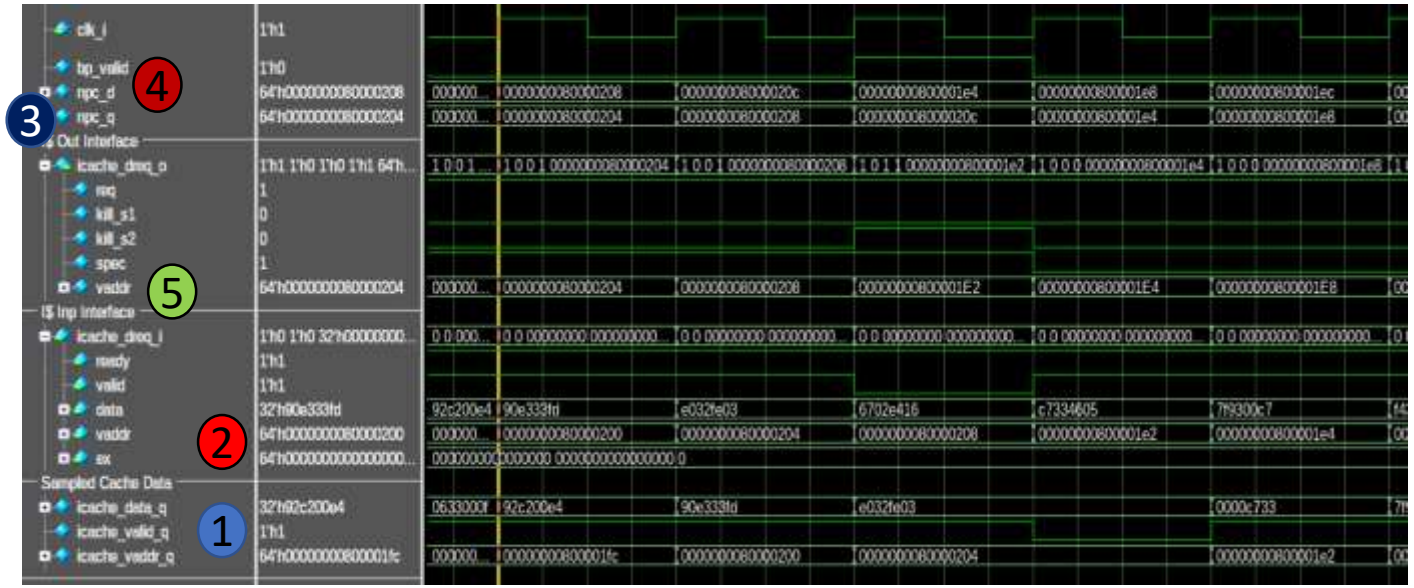


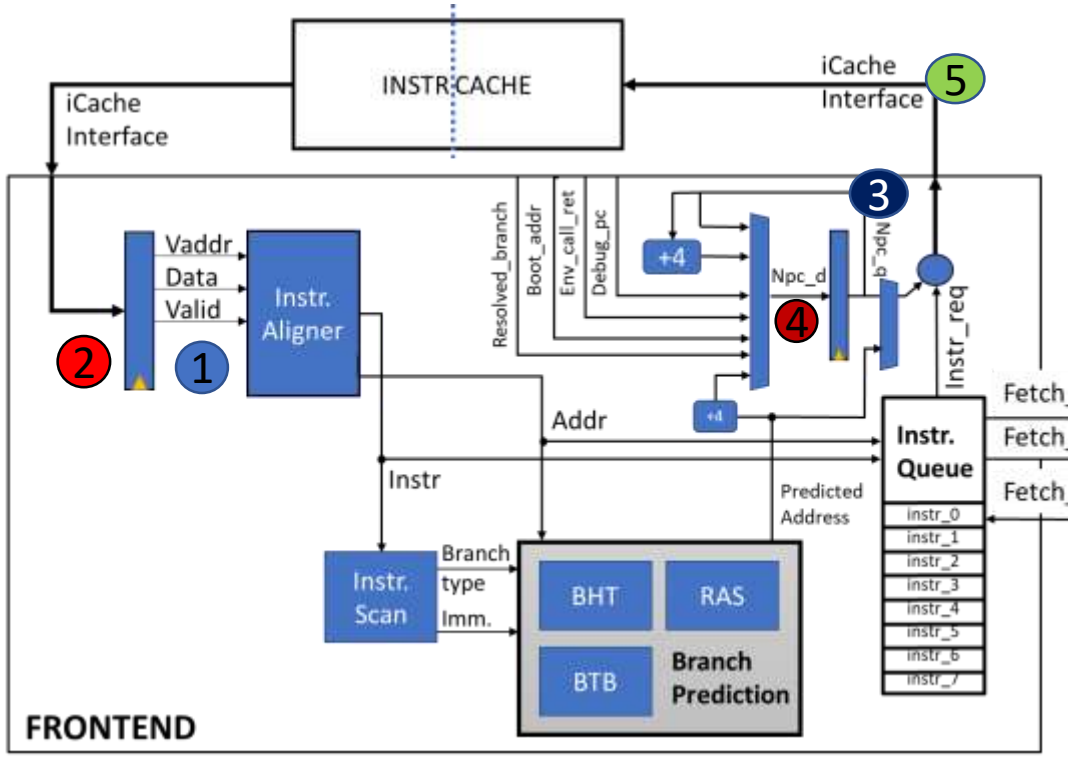


	SIGNAL	T0	T1	T2	T3	T4
1	ICACHE_VADDR_Q	0x01FC	0x0200	0x0204	0x0208	0x01E2
2	DREQ_I.VADDR	0x0200	0x0204	0x0208	0x01E2	0x01E4
3	NPC_Q	0x0204	0x0208	0x020C	0x01E4	0x01E8
4	NPC_D	0x0208	0x020C	0x01E4	0x01E8	0x01EC
5	DREQ_O.VADDR	0x0204	0x0208	0x01E2	0x01E4	0x01E8

Invalid

Cache Requests Examples

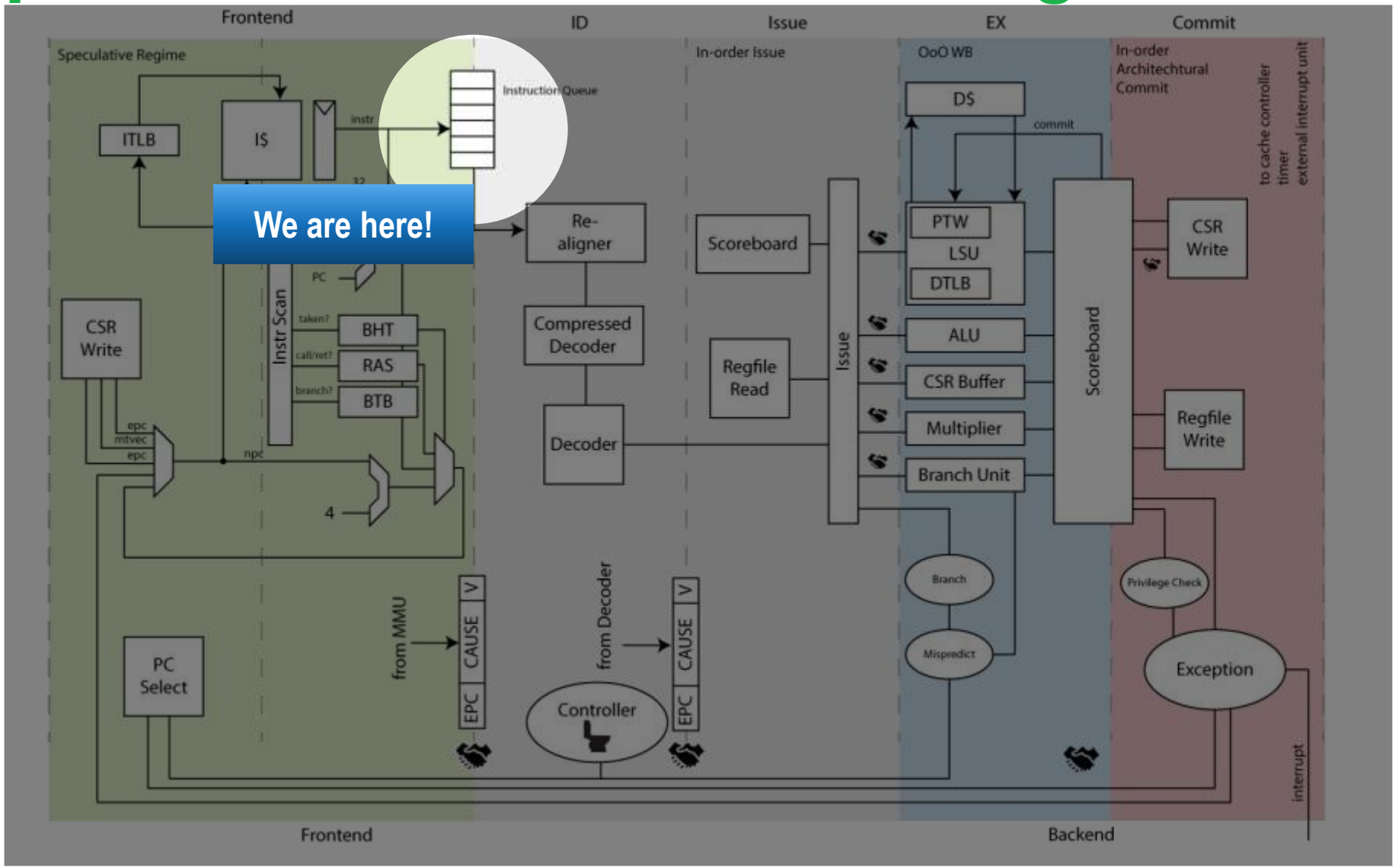




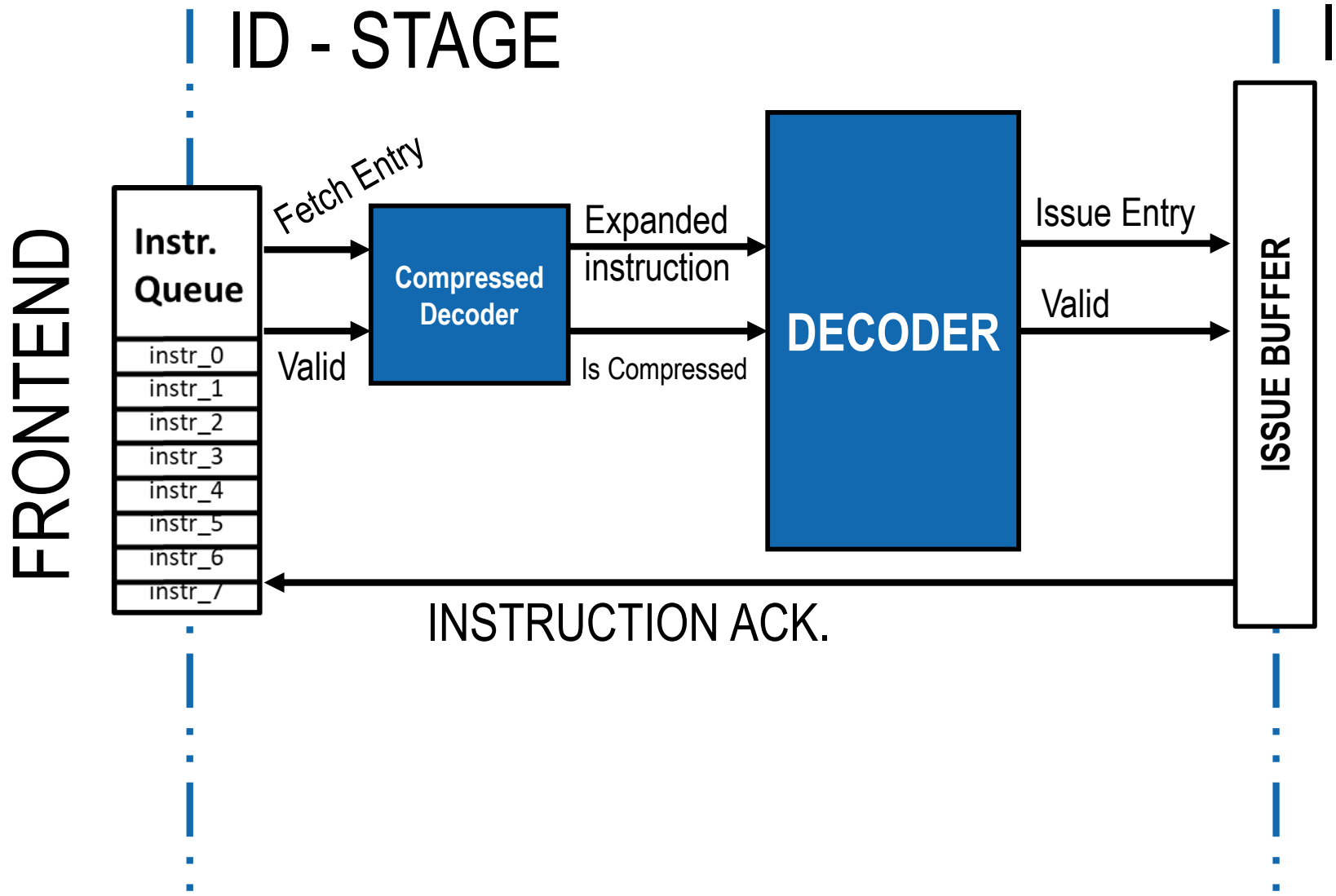
	SIGNAL	T0	T1	T2	T3	T4
1	ICACHE_VADDR_Q	0x01FC	0x0200	0x0204	0x0208	0x01E2
2	DREQ_I.VADDR	0x0200	0x0204	0x0208	0x01E2	0x01E4
3	NPC_Q	0x0204	0x0208	0x020C	0x01E4	0x01E8
4	NPC_D	0x0208	0x020C	0x01E4	0x01E8	0x01EC
5	DREQ_O.VADDR	0x0204	0x0208	0x01E2	0x01E4	0x01E8

Invalid

Deep Dive: Decode and Issue-Stage

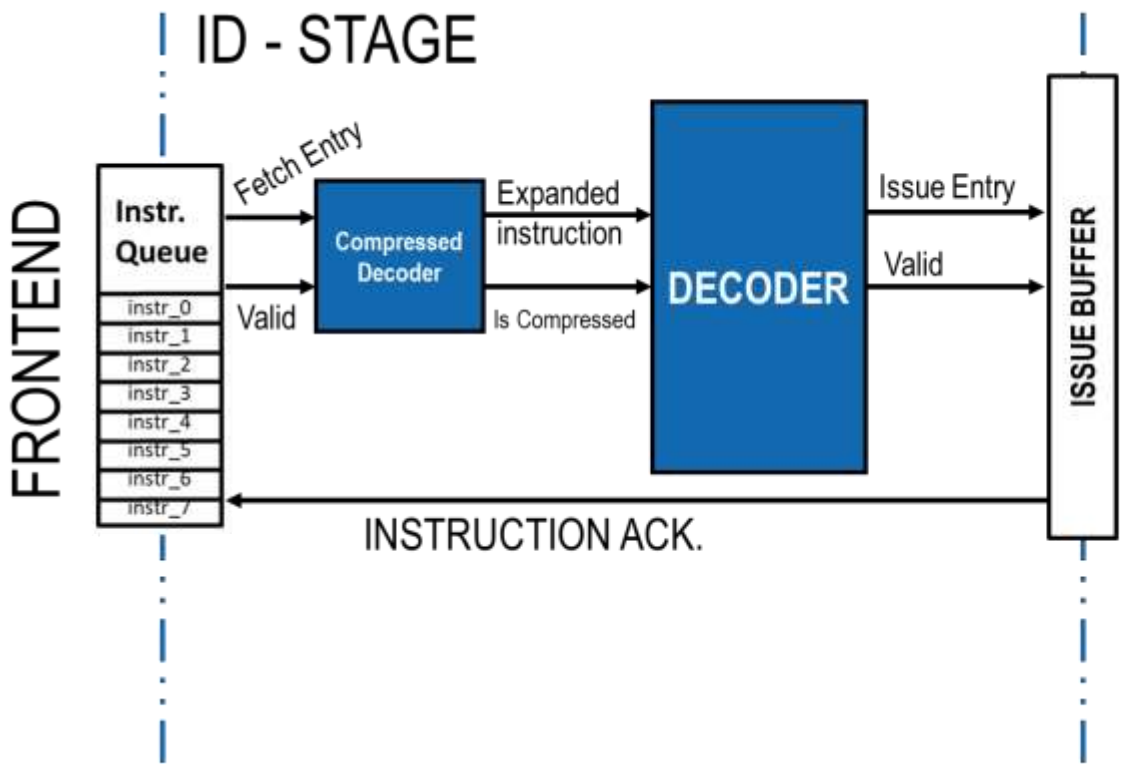


Decode Stage



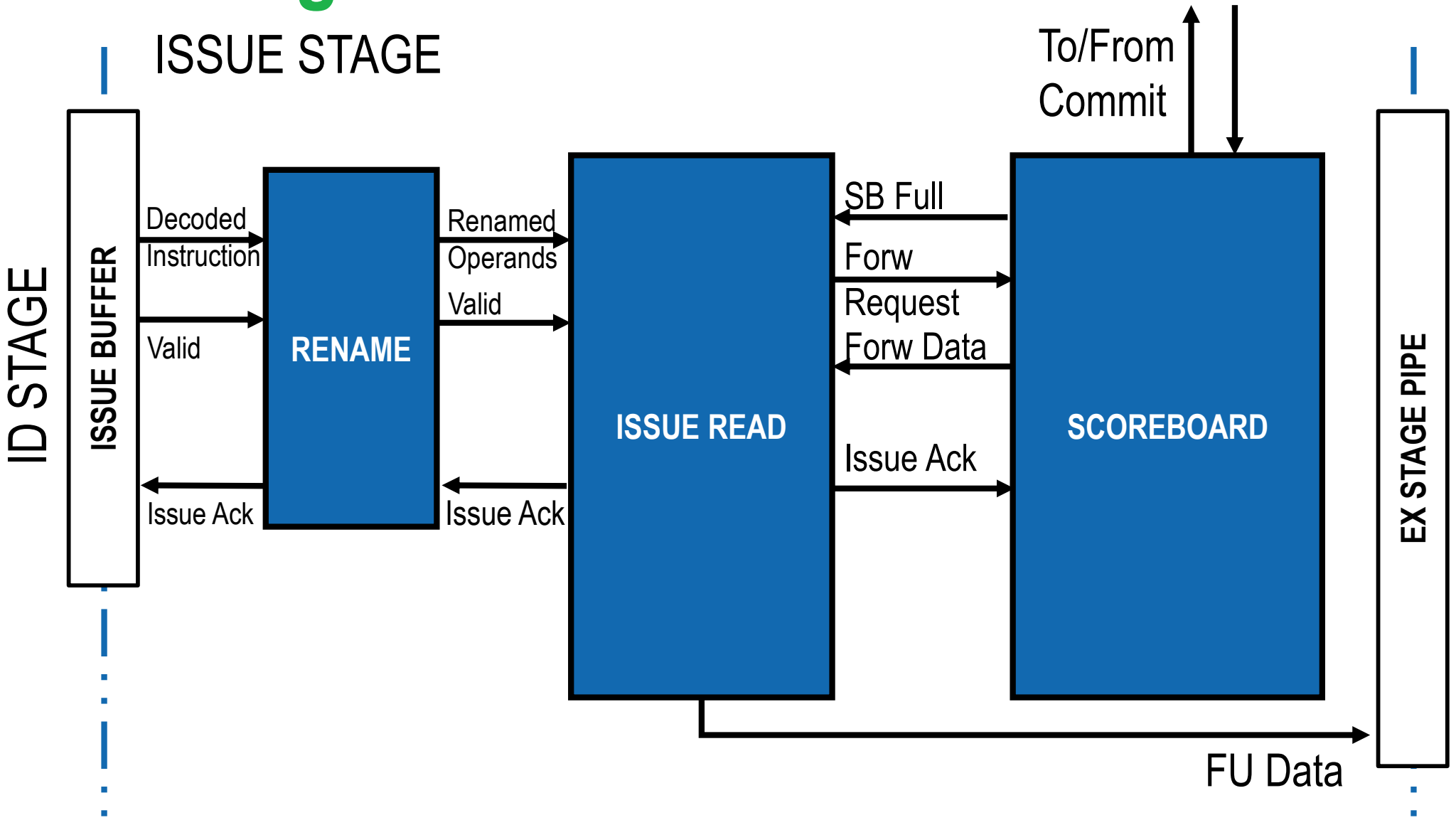
- Read Valid Entries From Instruction Queue
- Expand Compressed instruction into non-compress
- Write Decoded Information into the Issue Buffer

Decode Stage: Issue Interface

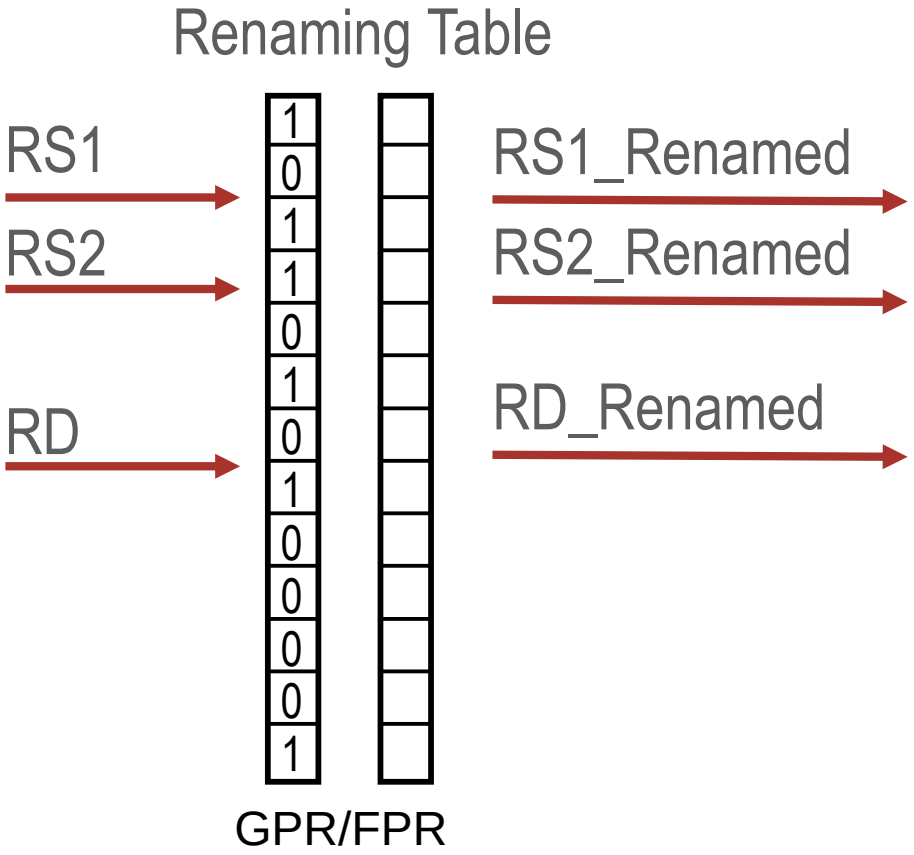


- Program Counter [63:0]
- Trans_ID: Refers to Scoreboard Entry
- FU: Indicate type of Functional Unit
- OP: Operation to be performed (e.g. ADD, MUL, etc)
- RS1: Address of Resource OP 1
- RS2: Address of Resource OP 2
- RD: Address of Destination Register
- RESULT: Initially Enocodes Other information
- VALID: Valid Entry
- USE IMMEDIATE
- USE PC
- Exception
- BP: Branch Prediction Information
- IS COMPRESSED

Issue Stage


ISSUE STAGE
ID STAGE
ISSUE BUFFER
Decoded Instruction
Valid
Issue Ack
RENAME
Renamed Operands
Valid
Issue Ack
ISSUE READ
SB Full
Forw Request
Forw Data
Issue Ack
SCOREBOARD
To/From Commit
FU Data
EX STAGE PIPE

RENAMING



Avoids WAW Hazard by renaming Register

- Rare on In-order Single-Issue configuration but some Application benefits significantly

Example:

- **Check renaming table**
- **Feed renamed operands to scoreboard**

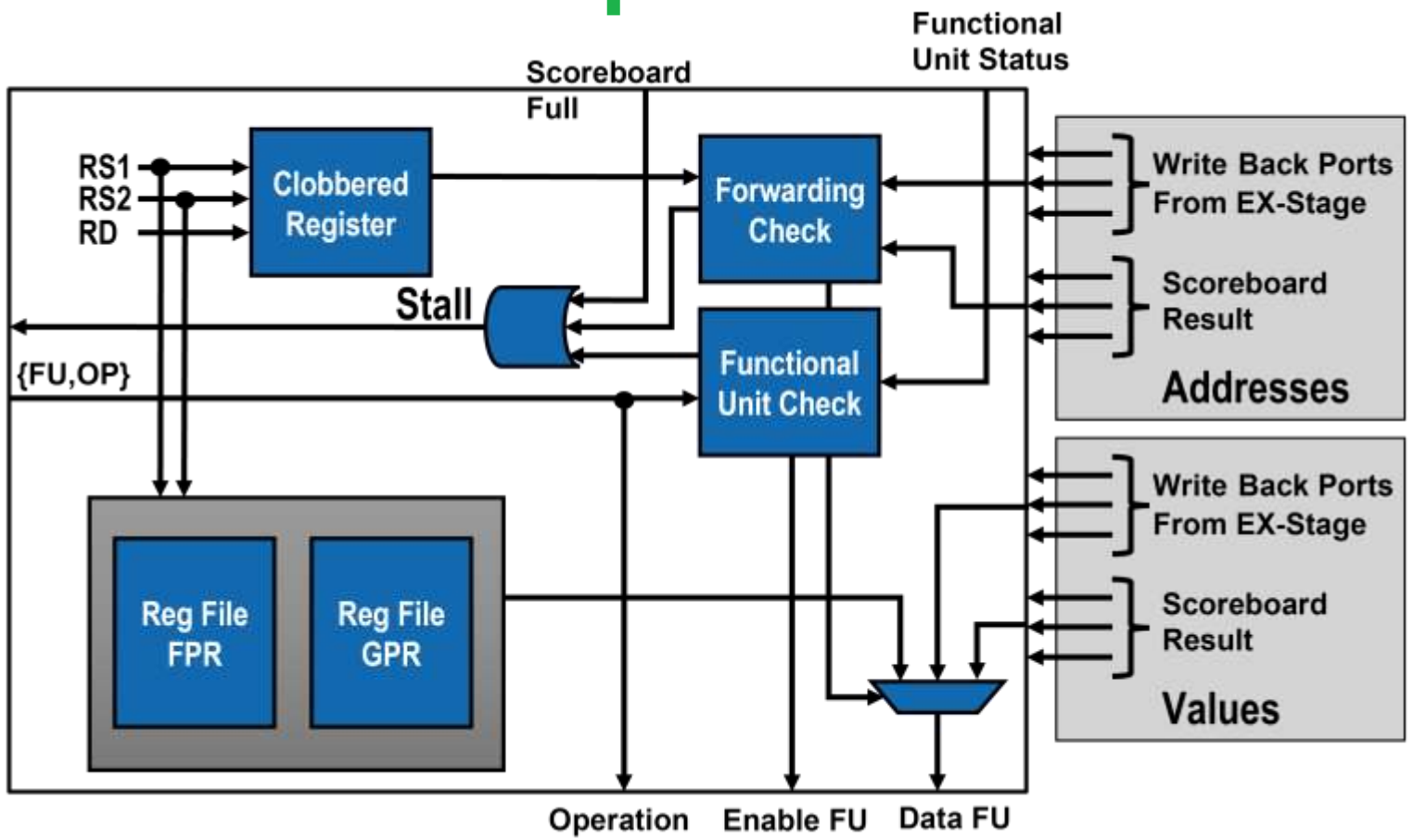
RS1	RS2	RNT[17]	RNT[10]	RS1 REN.	RS2 REN.
17	10	0	1	17	42

Update Renaming table:

- **The renaming Table is updated only by the RD**
- **Tick/Untick Corresponding RD bit**

RD	RNT[15]	RD REN.	RNT_N[15]
15	0	15	1

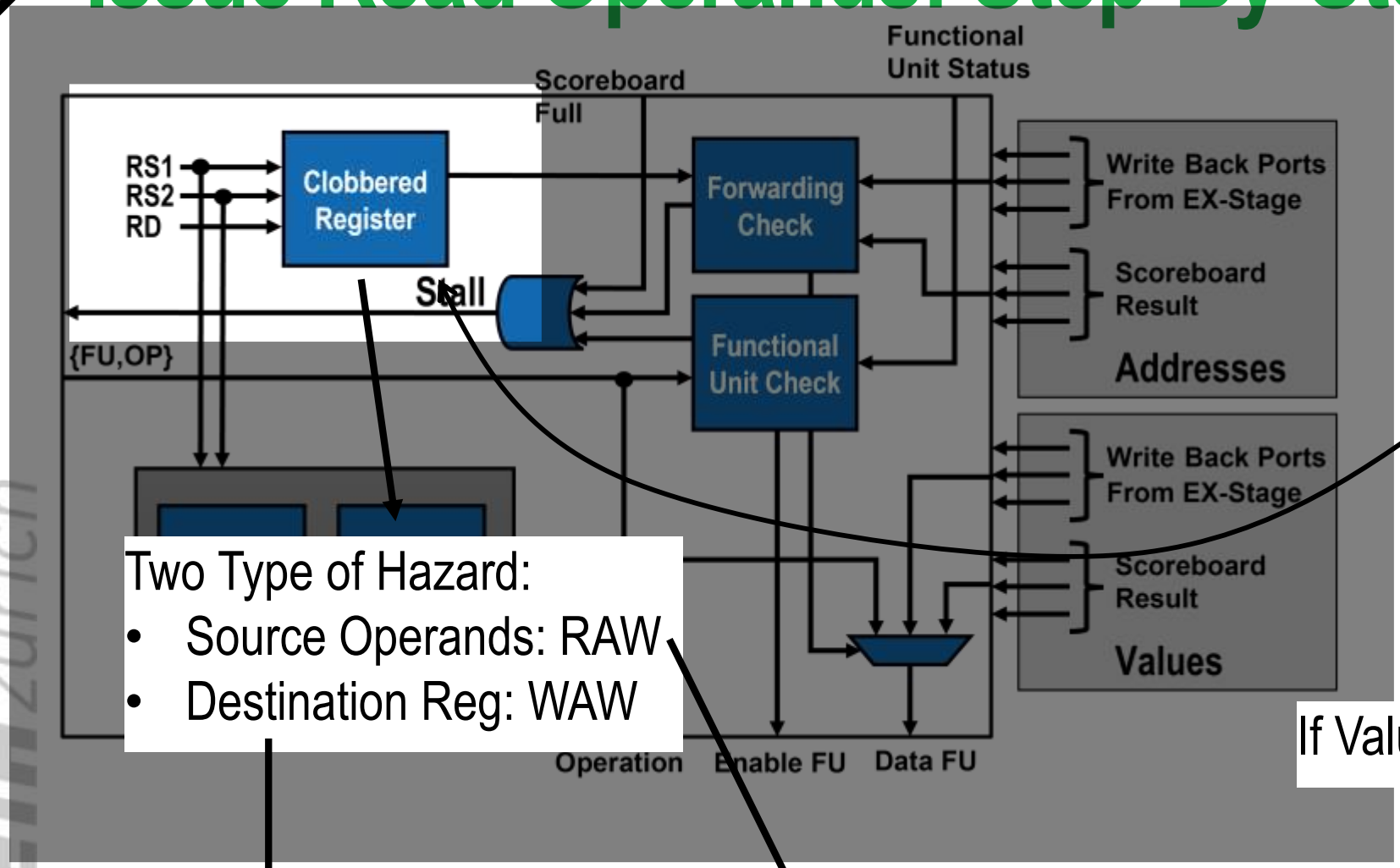
Issue Read Operands



Main Tasks:

- Check for Data Hazard
- Check if Functional Units are Ready
- Forwarding Operands
- Interfacing to the Ex-Stage

Issue Read Operands: Step-By-Step



1. Check any inflight instruction is writing to source/destination operands.

Clobbered Register contains 64 entries (one per each register and renamed)

(a) If Value = NONE → Read From Register File

(b) If Value = FU → Check For Forwarding

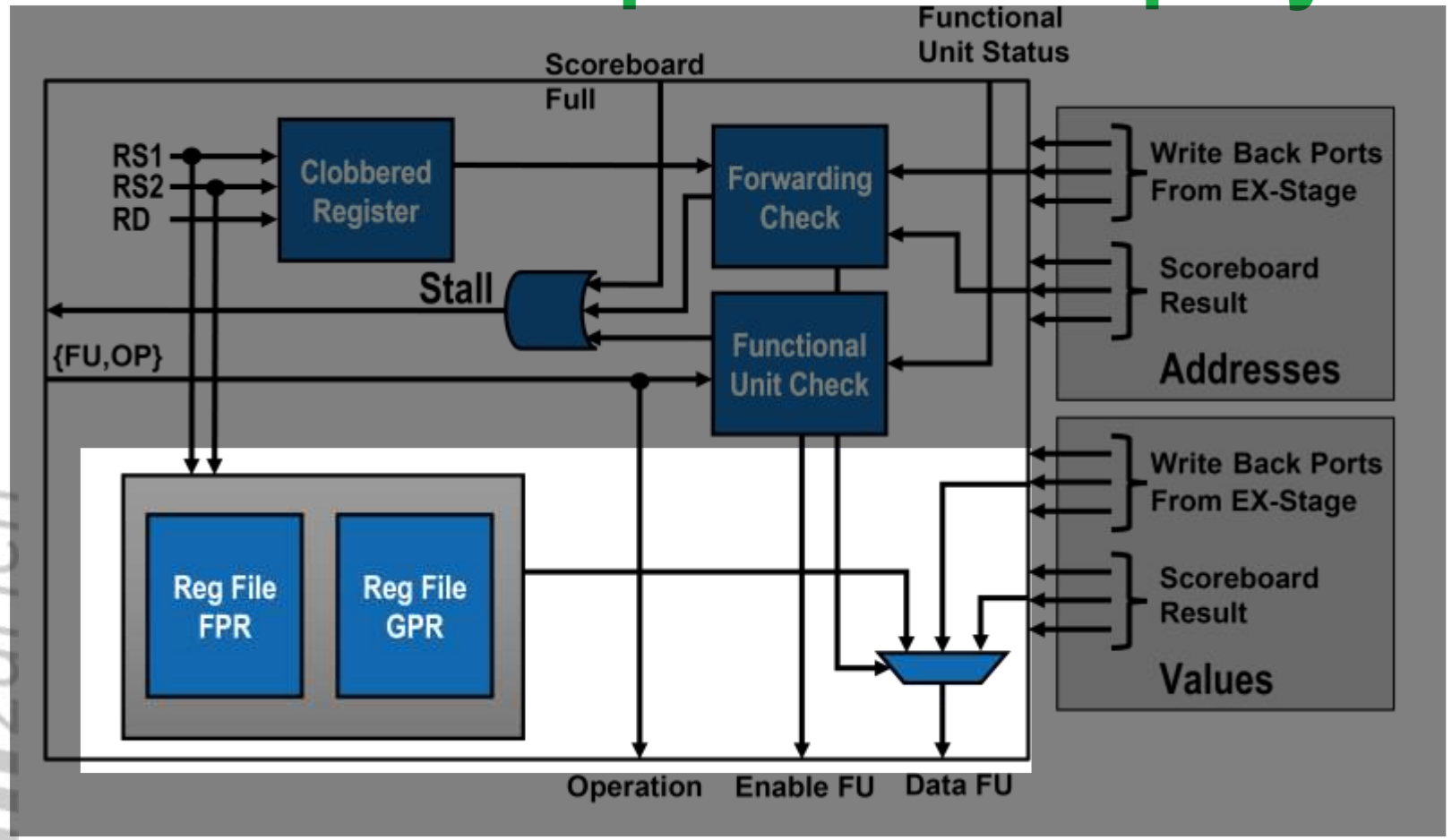
Two Type of Hazard:

- Source Operands: RAW
- Destination Reg: WAW

Mitigated With Renaming

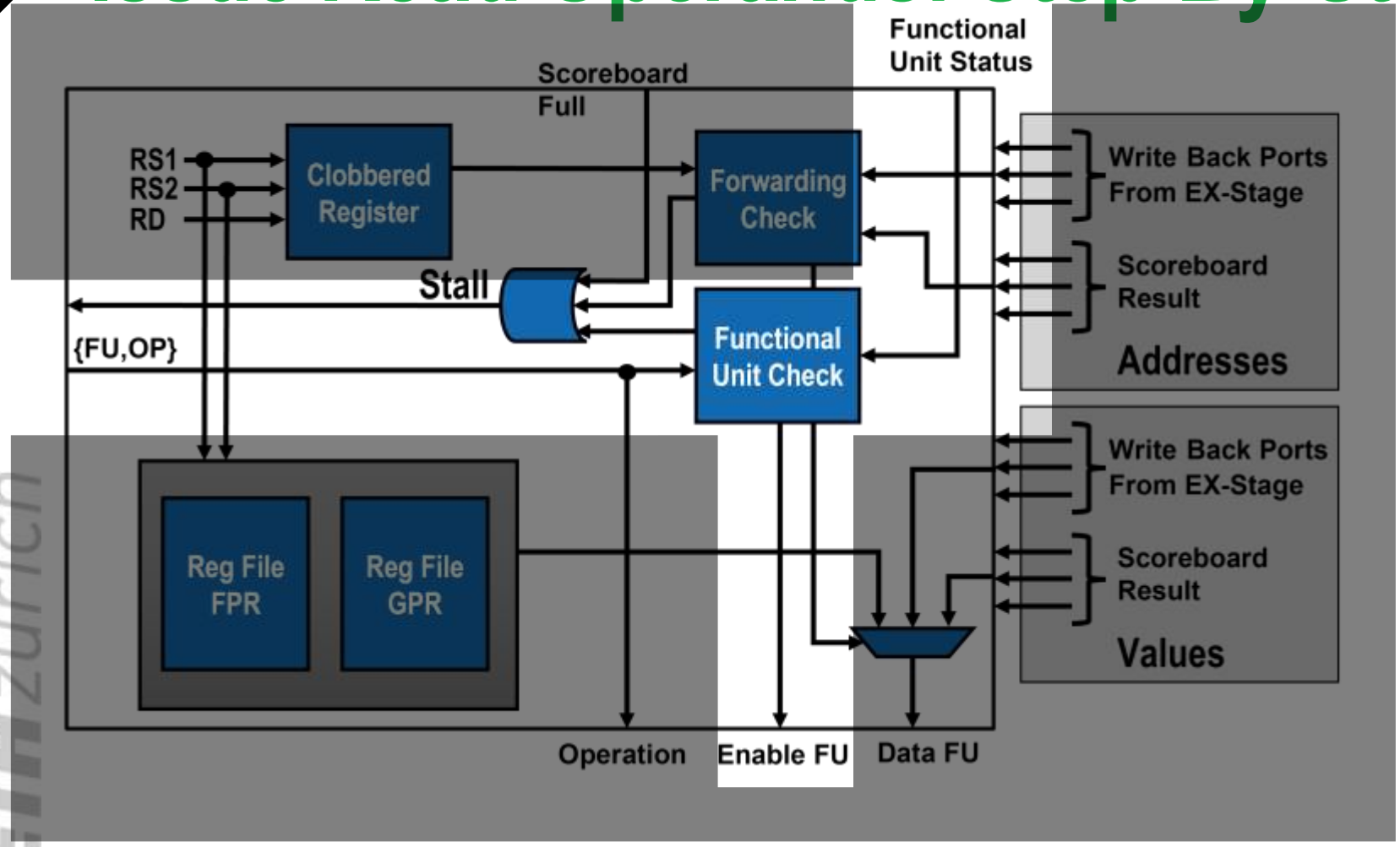
Mitigated With Forwarding

Issue Read Operands: Step-By-Step



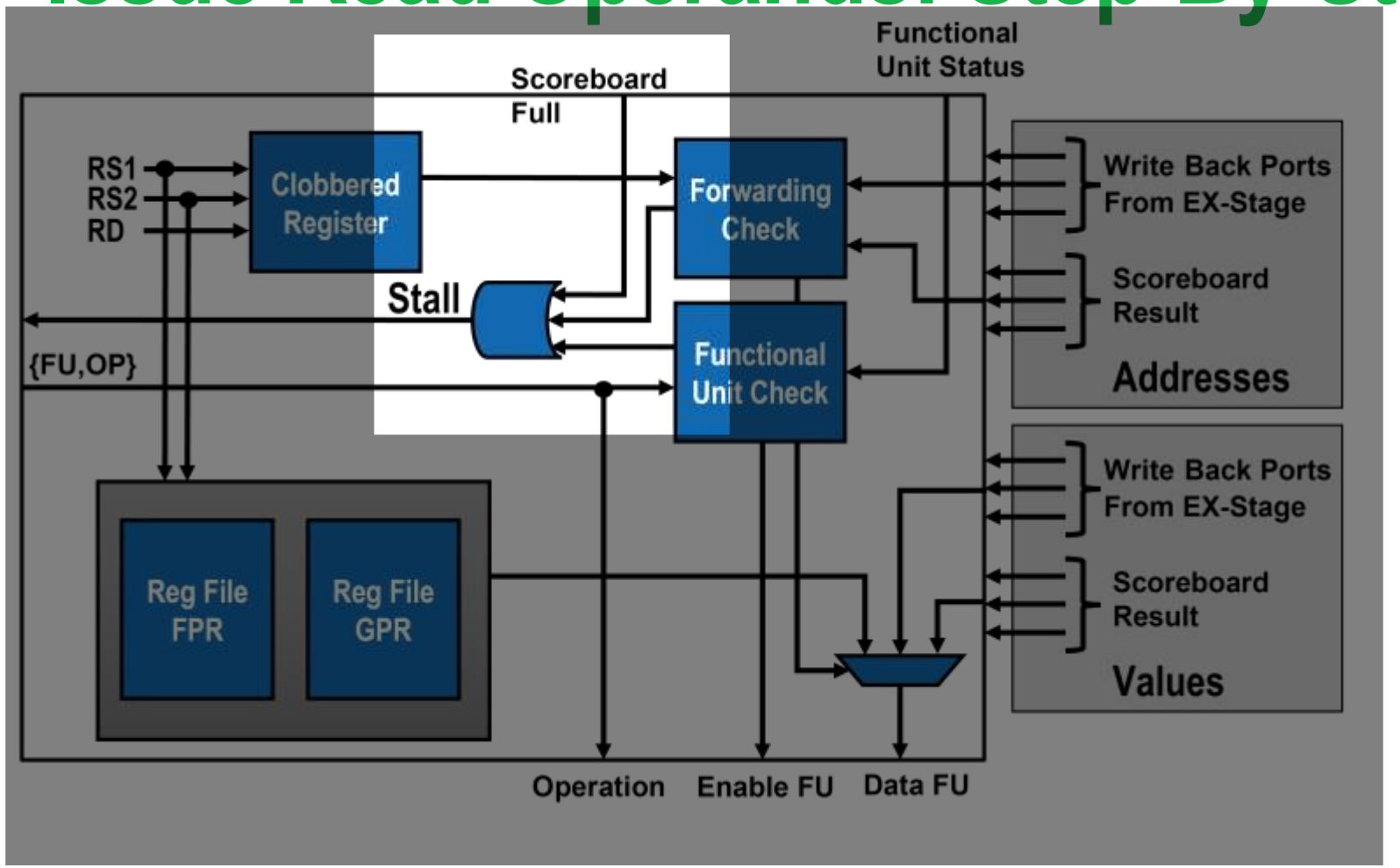
1. Check any inflight instruction is writing to source/destination operands.
2. (a) Read Values From Register File

Issue Read Operands: Step-By-Step



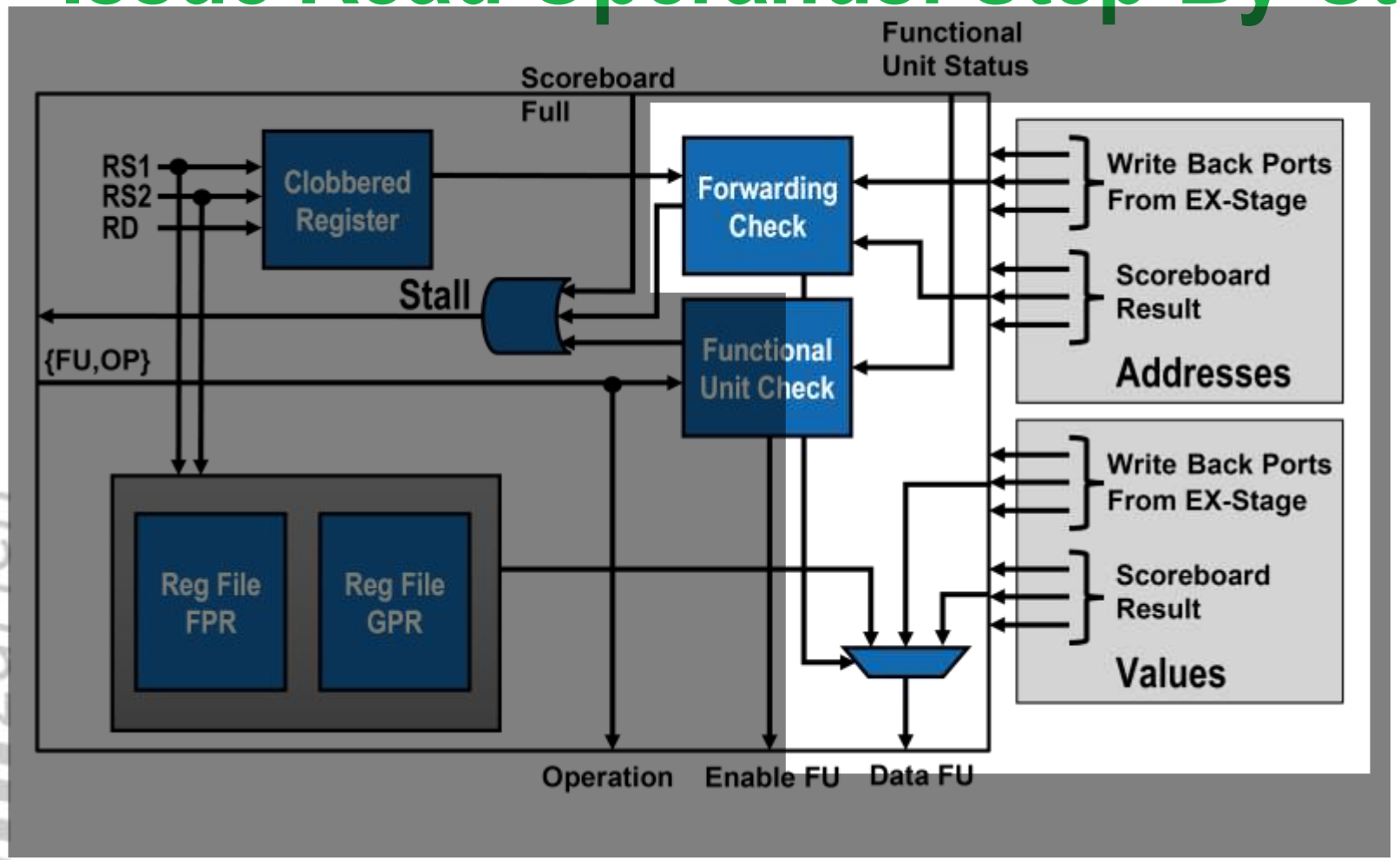
1. Check any inflight instruction is writing to source/destination operands.
2. (a) Read Values From Register File
3. Check If FU is Ready
   - Stall If not

Issue Read Operands: Step-By-Step



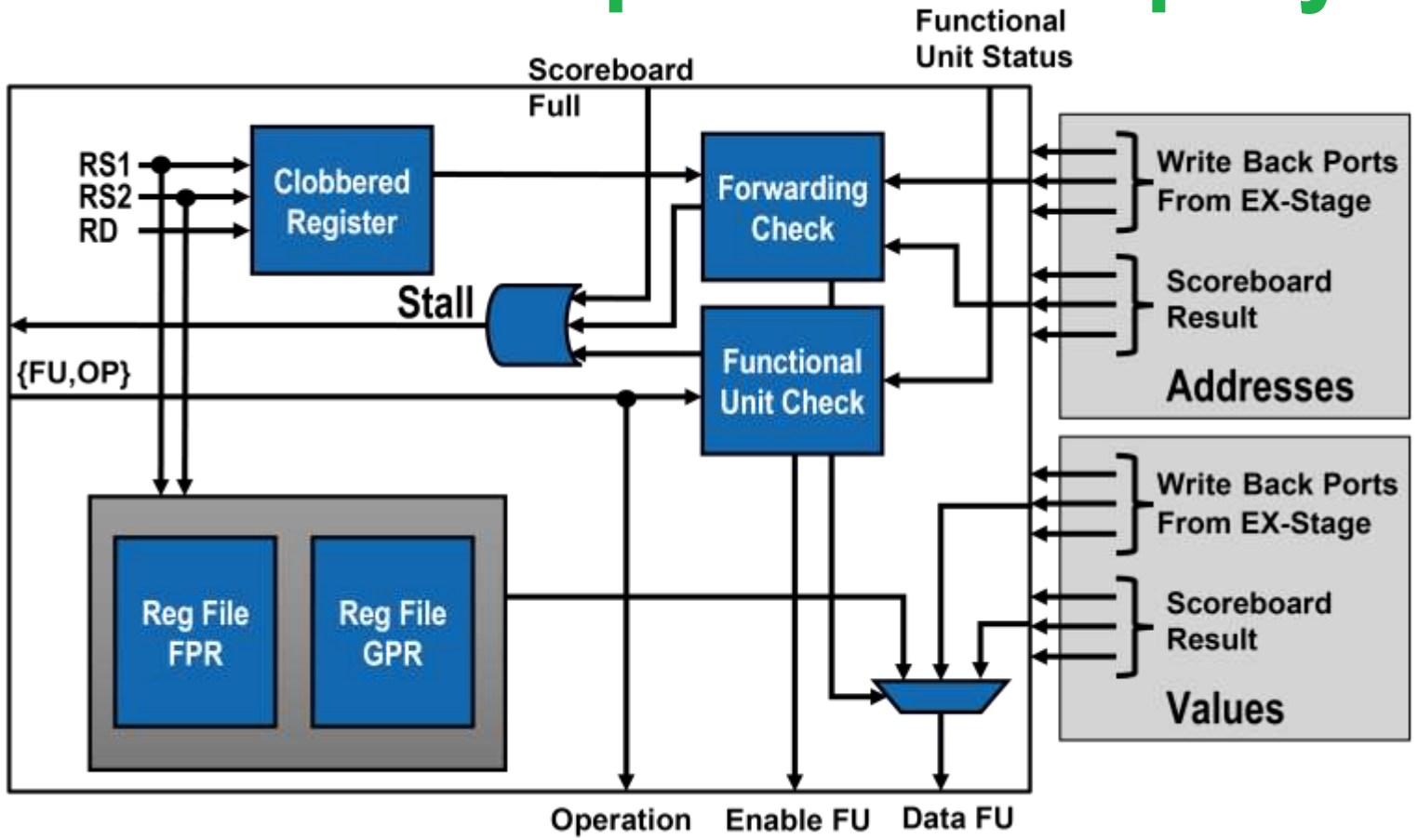
1. Check any inflight instruction is writing to source/destination operands.
2. (a) Read Values From Register File
3. Check If FU is Ready
   - Stall If not
4. Check if Scoreboard is full
   - Stall if it is
5. ISSUE

Issue Read Operands: Step-By-Step



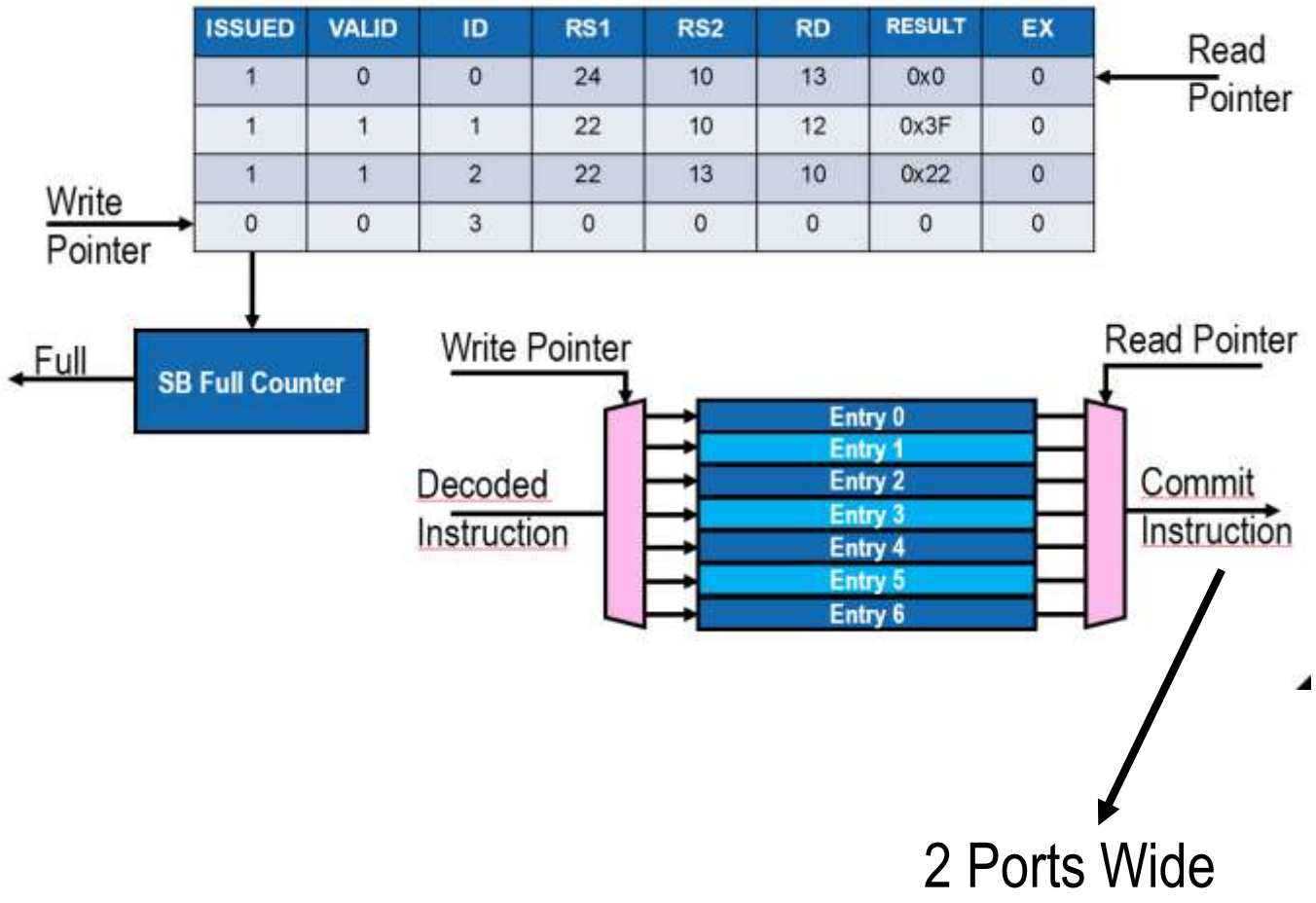
1. Check any inflight instruction is writing to source/destination operands.
2. (b) Check if Forwardable
   - Write Back Ports of Functional Units
   - Scoreboard Entries

Issue Read Operands: Step-By-Step



1. Check any inflight instruction is writing to source/destination operands.
2. (b) Check if Forwardable
   - Write Back Ports of Functional Units
   - Scoreboard Entries
3. Check If FU is Ready
   - Stall If not
4. Check if Scoreboard is full
   - Stall if it is
5. ISSUE

ScoreBoard



Track Each Issued and not Committed Instruction

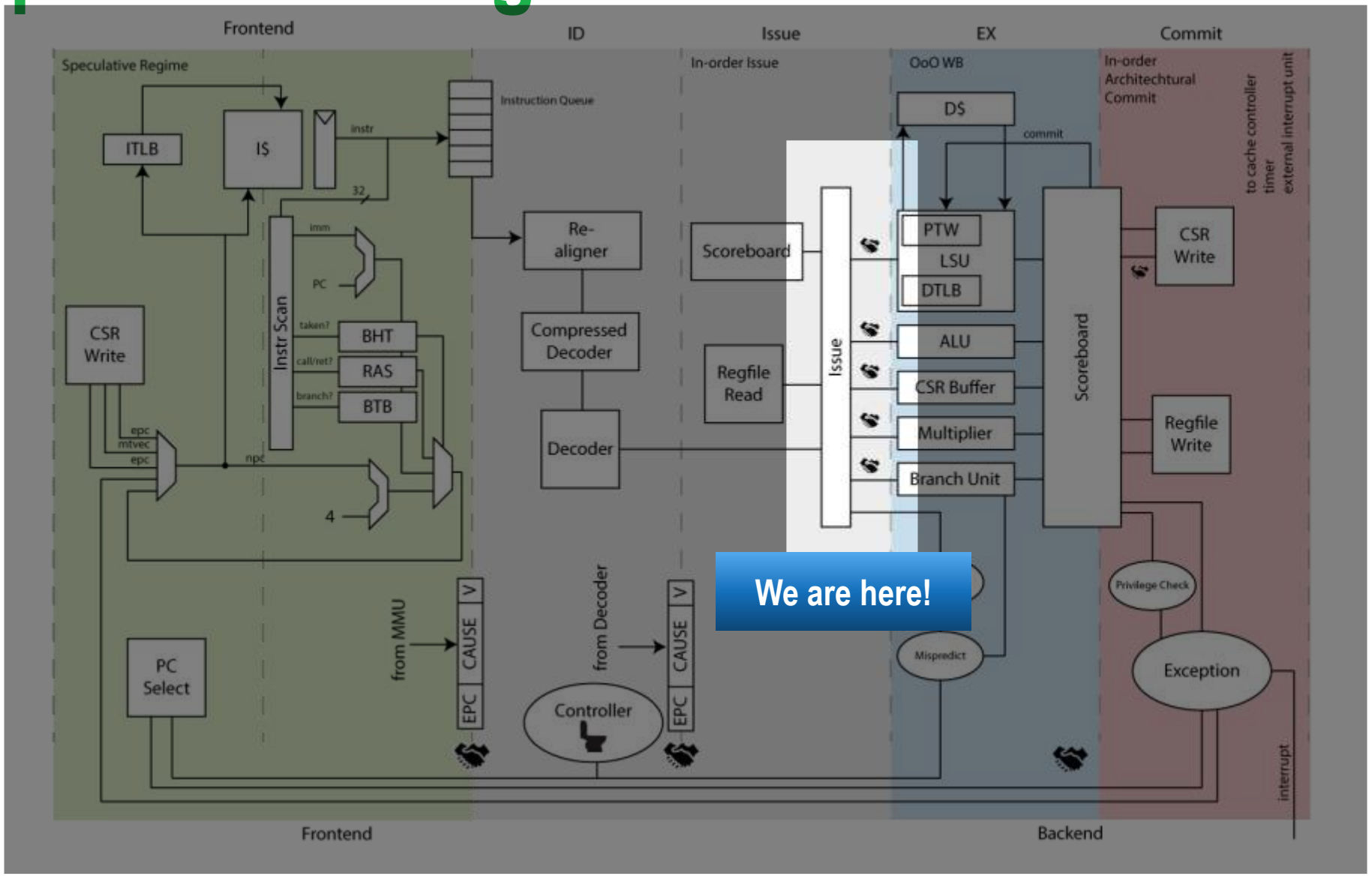
Used for hiding load latency

Write pointer Advance on issued Instructions

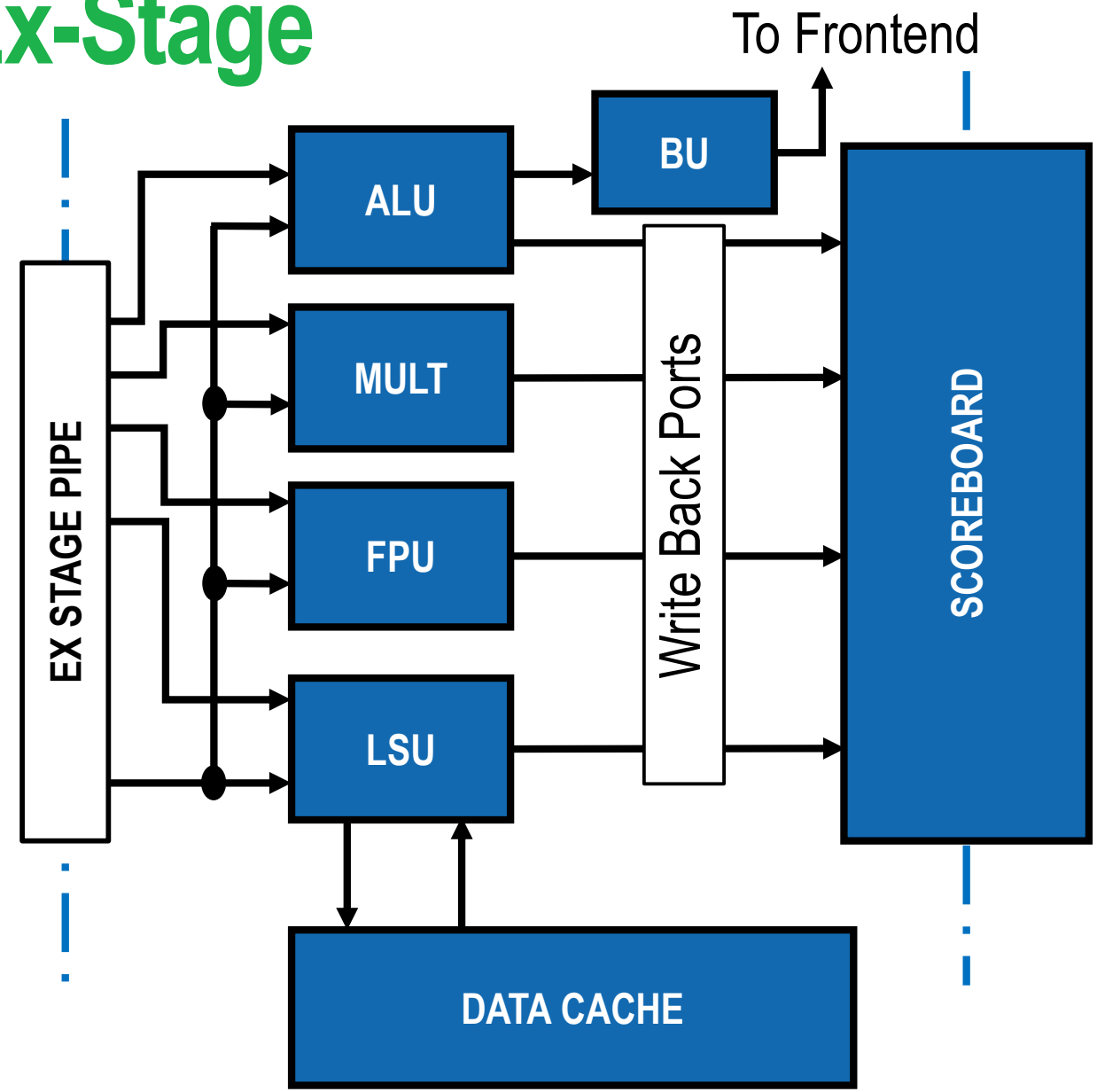
Read pointer Advance on Valid Instructions

Issued Instructions are counted to check if SB is Full

Deep Dive: Ex Stage



Ex-Stage



Functional Unit Are selected Via enable from issue stage

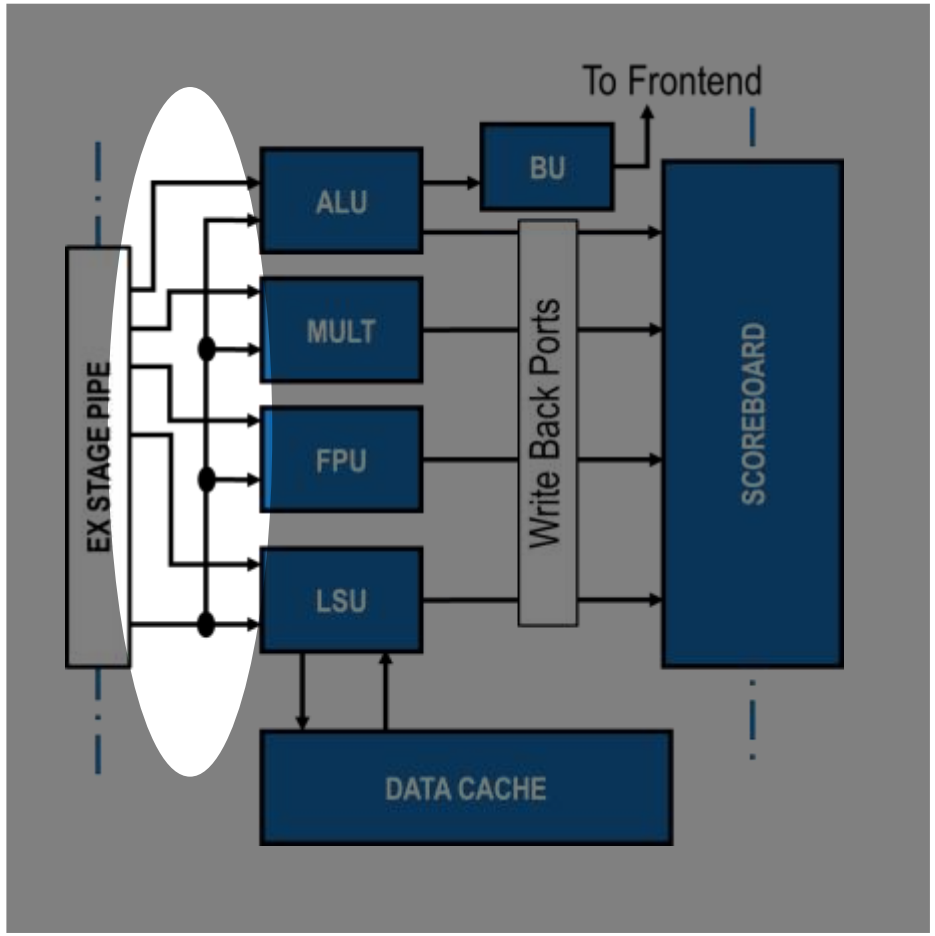
Write Back Ports are Connected to the Scoreboard

Branch Unit resolution is Connected to Frontend

Multiplier and FPU are internally Pipelined

Data Cache has 2 pipeline Stages, support hit under miss

Ex Stage Interface



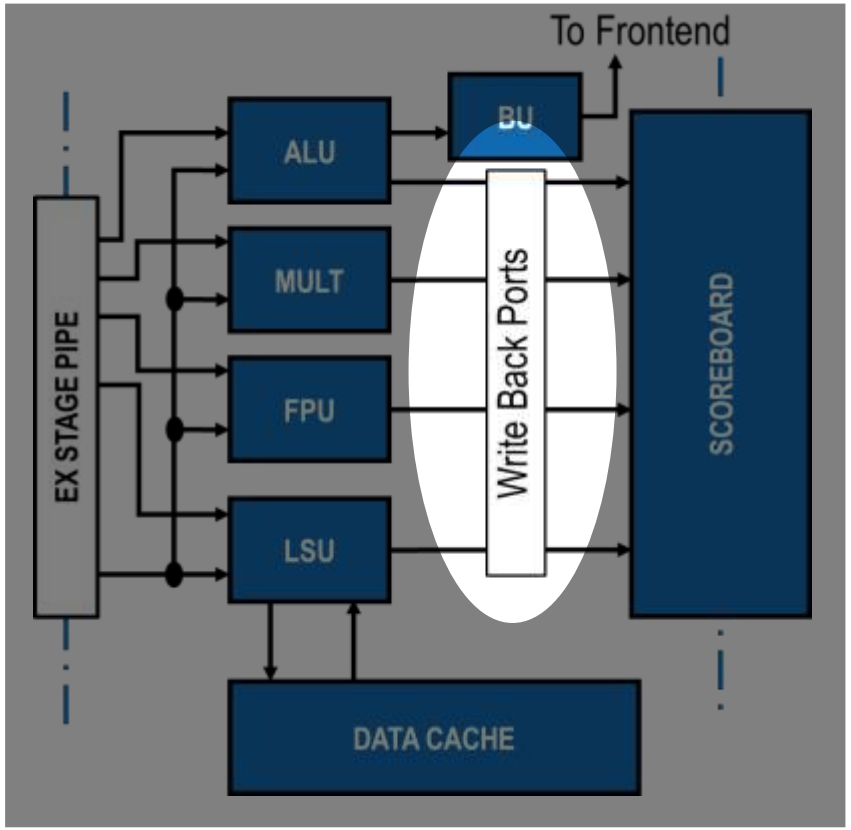
Enable

FU Data

- **Operator:** Type Of Operation
- **Operand_a[63:0]:** Value of OP_A
- **Operand_b[63:0]:** Value of OP_B
- **Immediate[63:0]:** Value of Immediate/OP_C
- **Trans_ID:** ID in the Scoreboard

- The FU Data is shared Among Functional Units
- Enable Decide which FU to Activate

Ex Stage Interface



Enable

FU Data

- **Operator:** Type Of Operation
- **Operand_a[63:0]:** Value of OP_A
- **Operand_b[63:0]:** Value of OP_B
- **Immediate[63:0]:** Value of Immediate/OP_C
- **Trans_ID:** ID in the Scoreboard

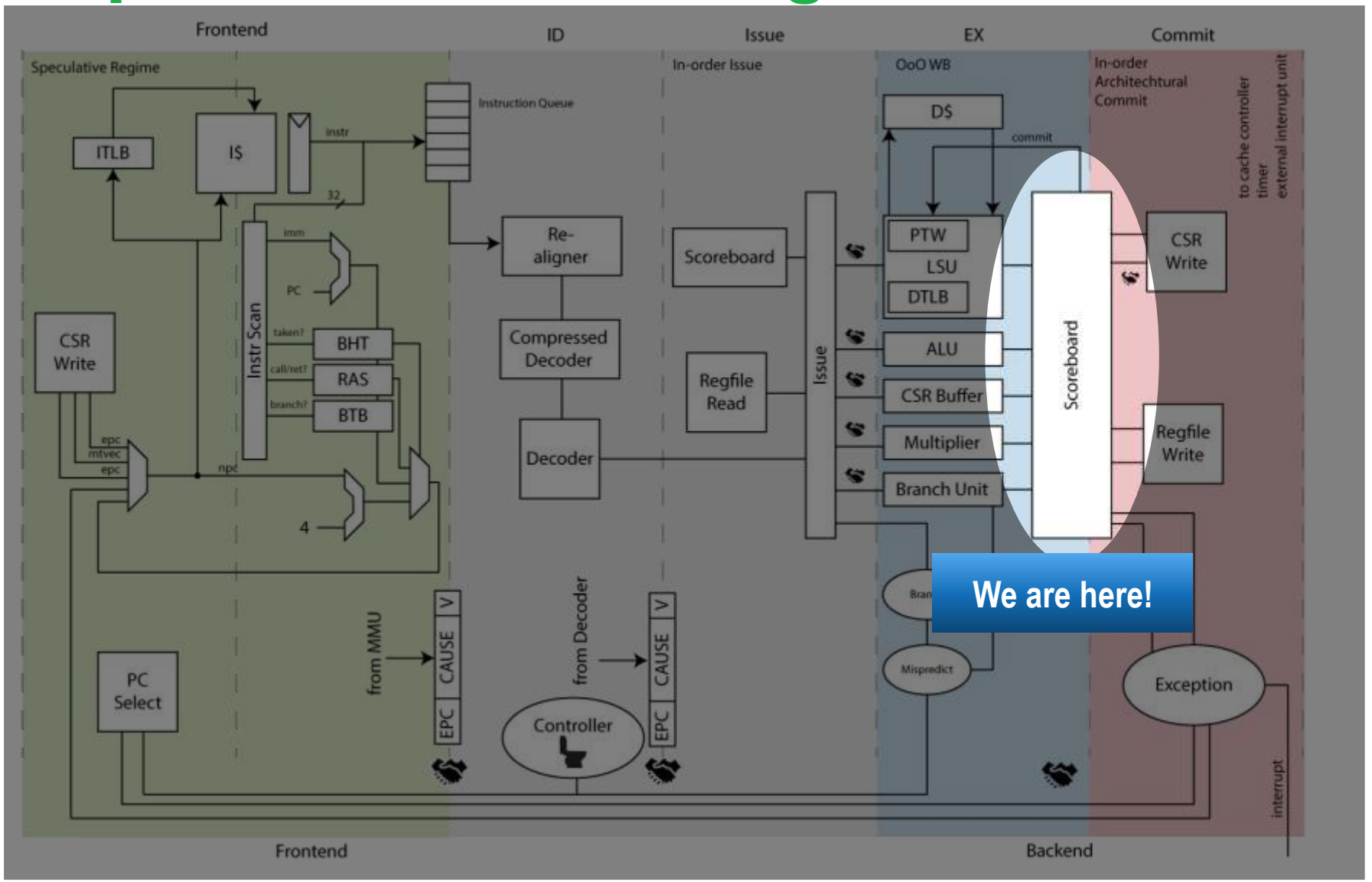
Valid

Trans_ID

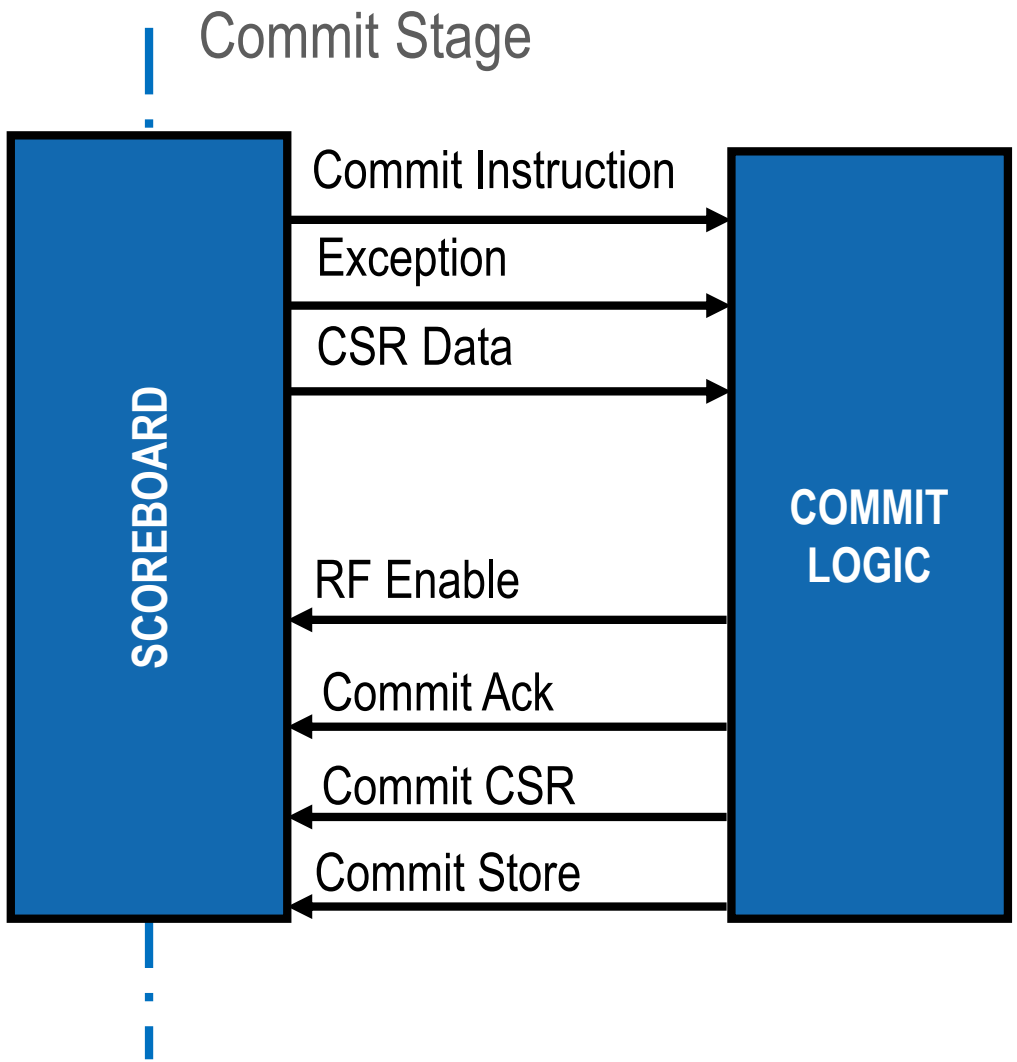
Result[63:0]

- **Results Are Redirected to the scoreboard**

Deep Dive: Commit Stage

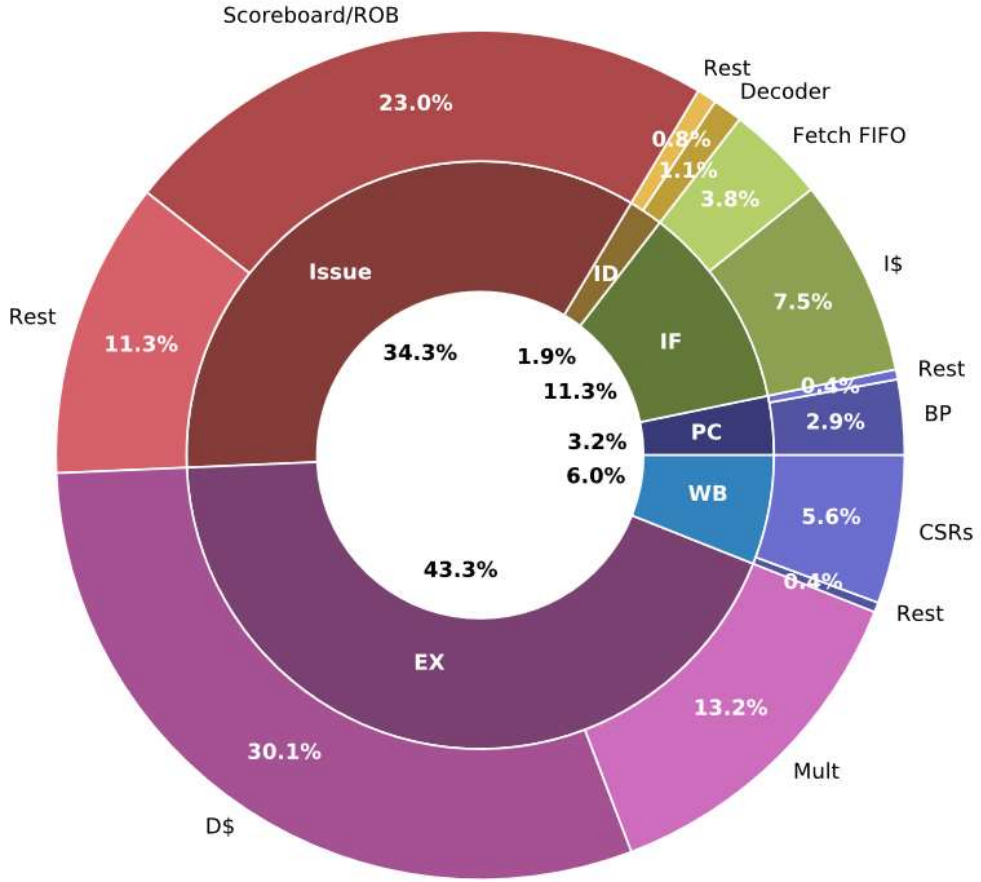


Commit Stage



- Commit Stage Read the Scoreboard for Valid Instructions
- Committing an instruction has the following effect:
  - Write result to register File
  - Write to CSR
  - Allow Store to write main Memory
- Commit Stage retires Instruction in Order

IMPLEMENTATION RESULTS: Area



- Area Results at 22nm FDSOI from GF
- Biggest Contribution are Cache Macros
- In general Sequential logic dominates Area

ETH zürich

IMPLEMENTATION RESULTS: Energy

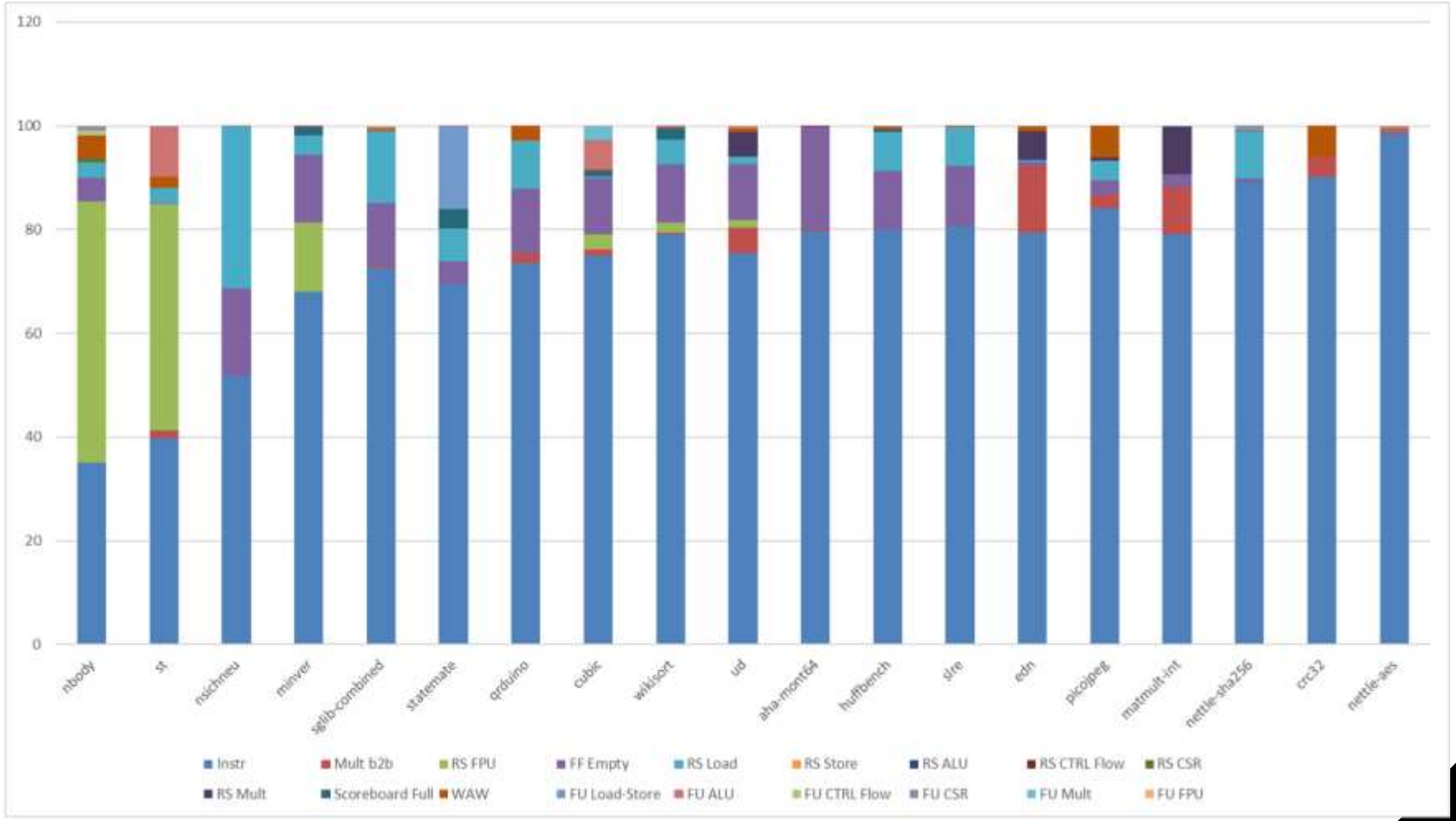
TABLE IV

ENERGY PER OPERATION CLASS [pJ], LEAKAGE [mW]

Instr. Class	PC	IF Stage		ID Stage		Issue	EX Stage						WB	CSR	CTS	Rest	Tot
		I$	Rest	Dec	Rest		L/S	VM	Mult	ALU	D$	Rest					
Mul	0.30	4.72	0.51	0.01	0.09	1.42	0.22	3.46	0.97	0.02	5.53	0.07	0.05	0.22	4.25	0.76	22.60
%	1.33	20.88	2.26	0.04	0.40	6.28	0.97	15.31	4.29	0.09	24.47	0.31	0.22	0.97	18.81	3.36	100.00
Div	0.25	3.19	0.35	0.00	0.02	1.11	0.22	3.43	0.68	0.00	5.54	0.05	0.02	0.20	4.07	0.81	19.94
%	1.25	16.00	1.76	0.00	0.10	5.57	1.10	17.20	3.41	0.00	27.78	0.25	0.10	1.00	20.41	4.06	100.00
LS w/ VM	0.32	4.63	0.54	0.01	0.09	1.38	0.30	3.50	0.09	0.03	9.18	0.18	0.06	0.22	4.06	0.62	25.21
%	1.27	18.37	2.14	0.04	0.36	5.47	1.19	13.88	0.36	0.12	36.41	0.71	0.24	0.87	16.10	2.46	100.00
LS w/o VM	0.30	4.39	0.51	0.00	0.07	1.36	0.30	3.48	0.07	0.02	9.12	0.17	0.06	0.22	4.04	0.64	24.75
%	1.21	17.74	2.06	0.00	0.28	5.49	1.21	14.06	0.28	0.08	36.85	0.69	0.24	0.89	16.32	2.59	100.00
ALU	0.30	4.36	0.50	0.05	0.13	1.69	0.24	3.47	0.11	0.03	5.53	0.08	0.08	0.24	4.05	0.72	21.58
%	1.39	20.20	2.32	0.23	0.60	7.83	1.11	16.08	0.51	0.14	25.63	0.37	0.37	1.11	18.77	3.34	100.00
IGEMM	0.61	10.17	1.59	0.19	0.65	5.88	0.61	3.84	4.41	0.71	13.75	1.00	0.31	1.12	4.68	2.28	51.80
%	1.18	19.63	3.07	0.37	1.25	11.35	1.18	7.41	8.51	1.37	26.54	1.93	0.60	2.16	9.03	4.40	100.00
Leakage	0.02	0.11	0.02	0.00	0.00	0.12	0.02	0.07	0.08	0.01	0.33	0.04	0.01	0.05	0.00	0.20	1.08

PERFORMANCE: Embench-iot Benchmark Suite

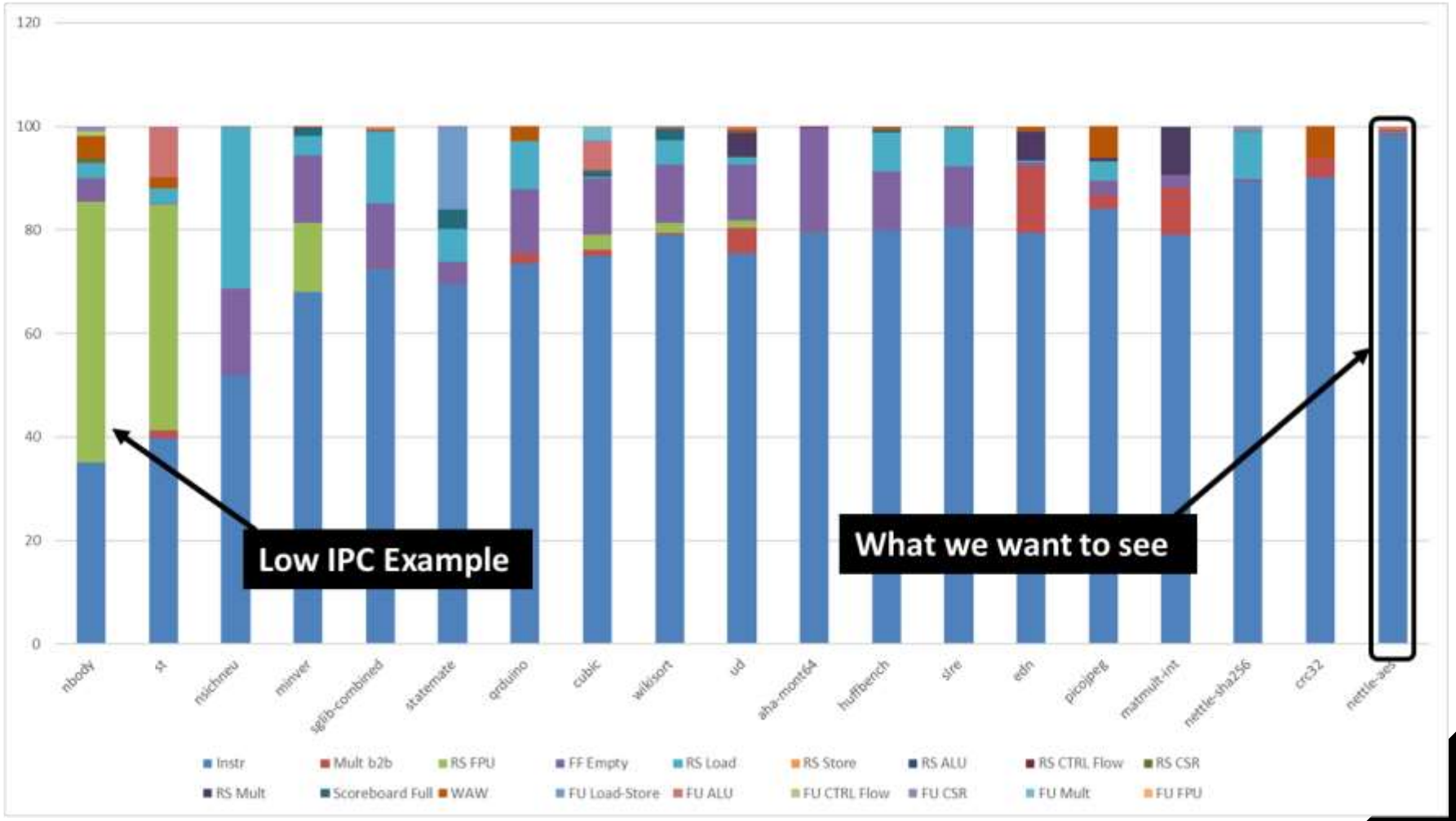
Test Name	IPC
aha-mont64	0,751773
crc32	0,901578
cubic	0,730032
edn	0,837711
huffbench	0,736758
matmult-int	0,889643
minver	0,512024
nbody	0,327144
nettle-aes	0,986432
nettle-sha256	0,862099
nsichneu	0,429473
picojpeg	0,831389
primecount	0,57092
qrduino	0,694356
sglib-combined	0,656466
slre	0,75285
st	0,396254
statemate	0,701807
ud	0,647145
wikisort	0,813702



ETH zürich

PERFORMANCE: Embench-iot Benchmark Suite

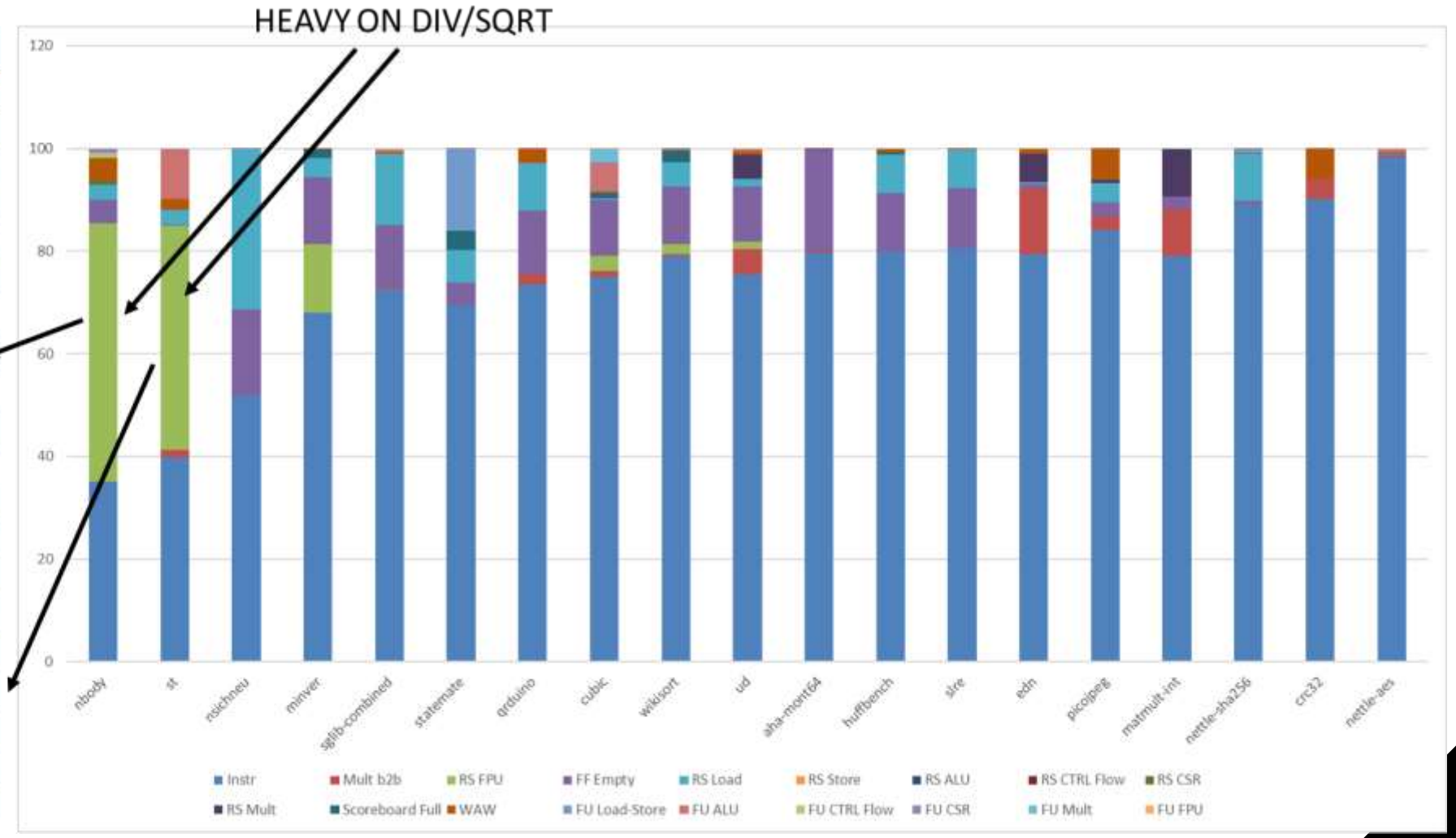
Test Name	IPC
aha-mont64	0,751773
crc32	0,901578
cubic	0,730032
edn	0,837711
huffbench	0,736758
matmult-int	0,889643
minver	0,512024
nbody	0,327144
nettle-aes	0,986432
nettle-sha256	0,862099
nsichneu	0,429473
picojpeg	0,831389
primecount	0,57092
qrduino	0,694356
sglib-combined	0,656466
slre	0,75285
st	0,396254
statemate	0,701807
ud	0,647145
wikisort	0,813702



ETH zürich

PERFORMANCE: Embench-iot Benchmark Suite

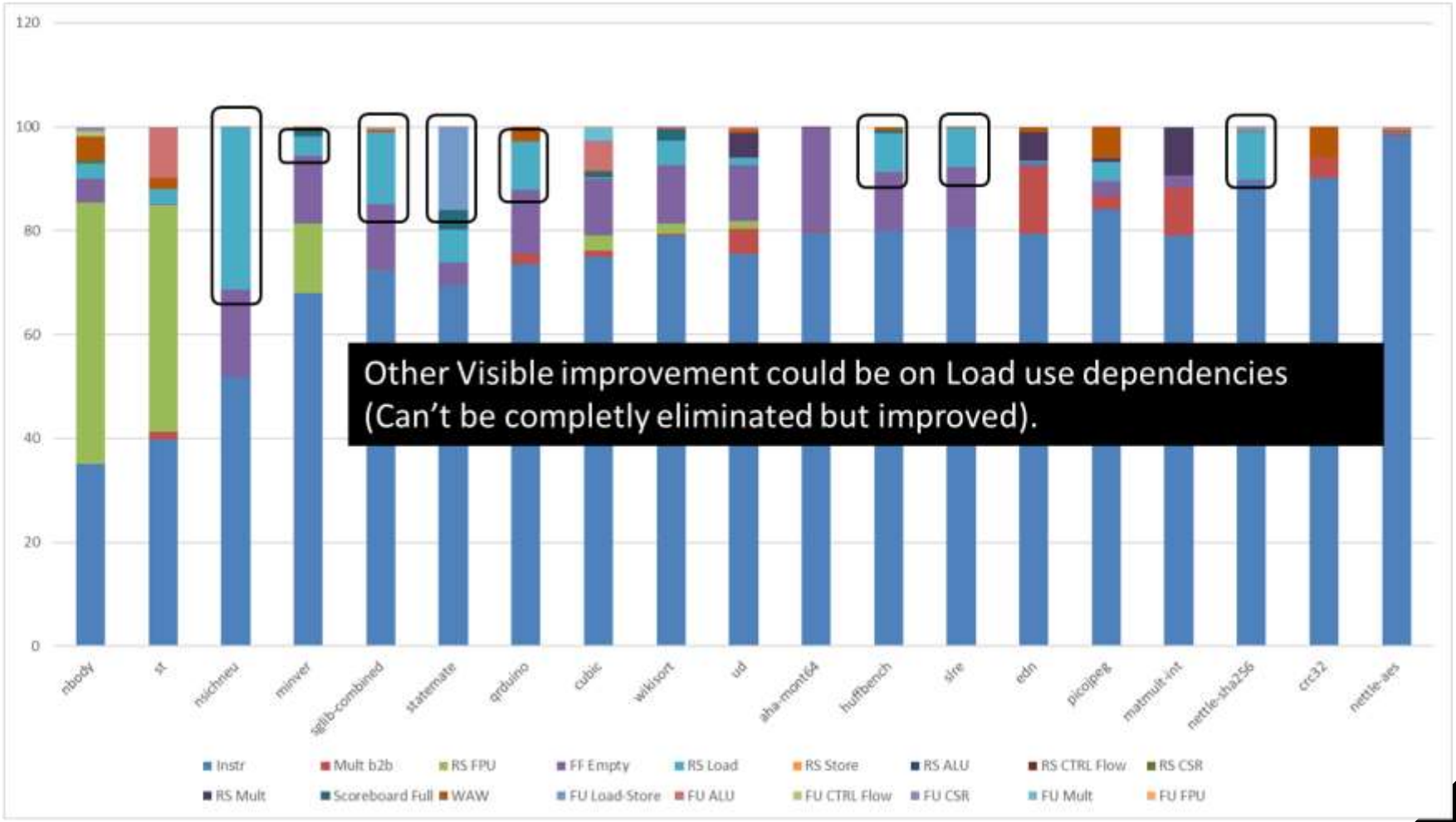
Test Name	IPC
aha-mont64	0,751773
crc32	0,901578
cubic	0,730032
edn	0,837711
huffbench	0,736758
matmult-int	0,889643
minver	0,512024
nbody	0,327144
nettle-aes	0,986432
nettle-sha256	0,862099
nsichneu	0,429473
picojpeg	0,831389
primecount	0,57092
qrduino	0,694356
sglib-combined	0,656466
slre	0,75285
st	0,396254
statemate	0,701807
ud	0,647145
wikisort	0,813702



ETH zürich

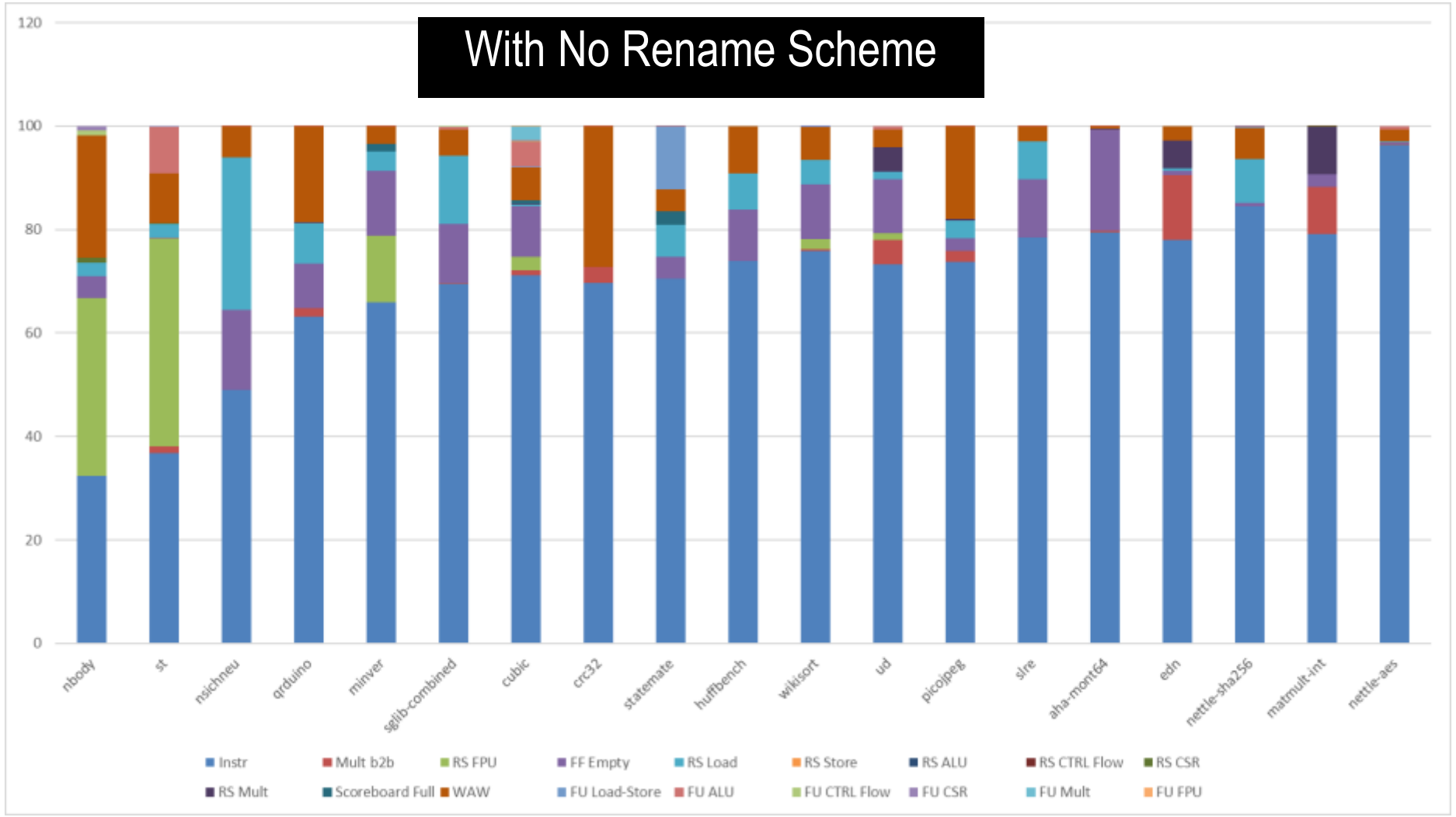
PERFORMANCE: Embench-iot Benchmark Suite

Test Name	IPC
aha-mont64	0,751773
crc32	0,901578
cubic	0,730032
edn	0,837711
huffbench	0,736758
matmult-int	0,889643
minver	0,512024
nbody	0,327144
nettle-aes	0,986432
nettle-sha256	0,862099
nsichneu	0,429473
picojpeg	0,831389
primecount	0,57092
qrduino	0,694356
sglib-combined	0,656466
slre	0,75285
st	0,396254
statemate	0,701807
ud	0,647145
wikisort	0,813702

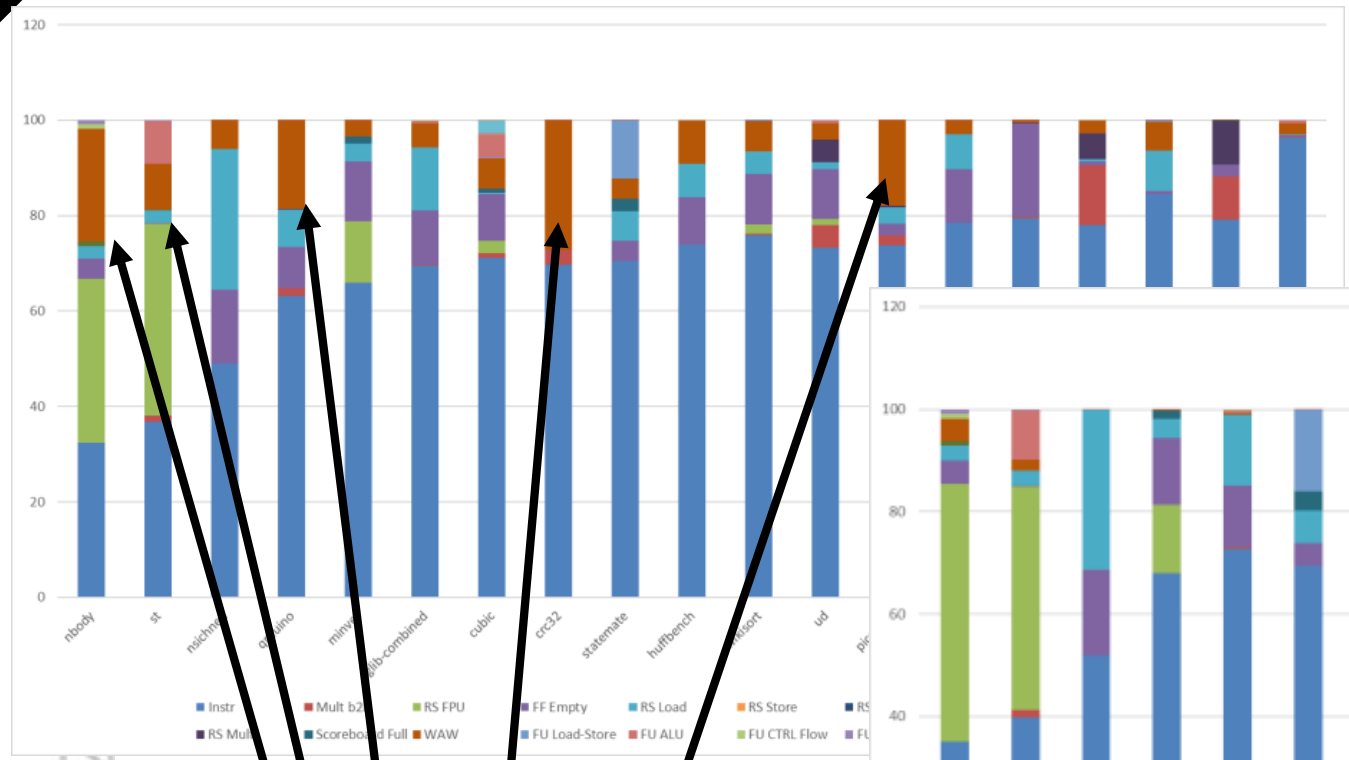


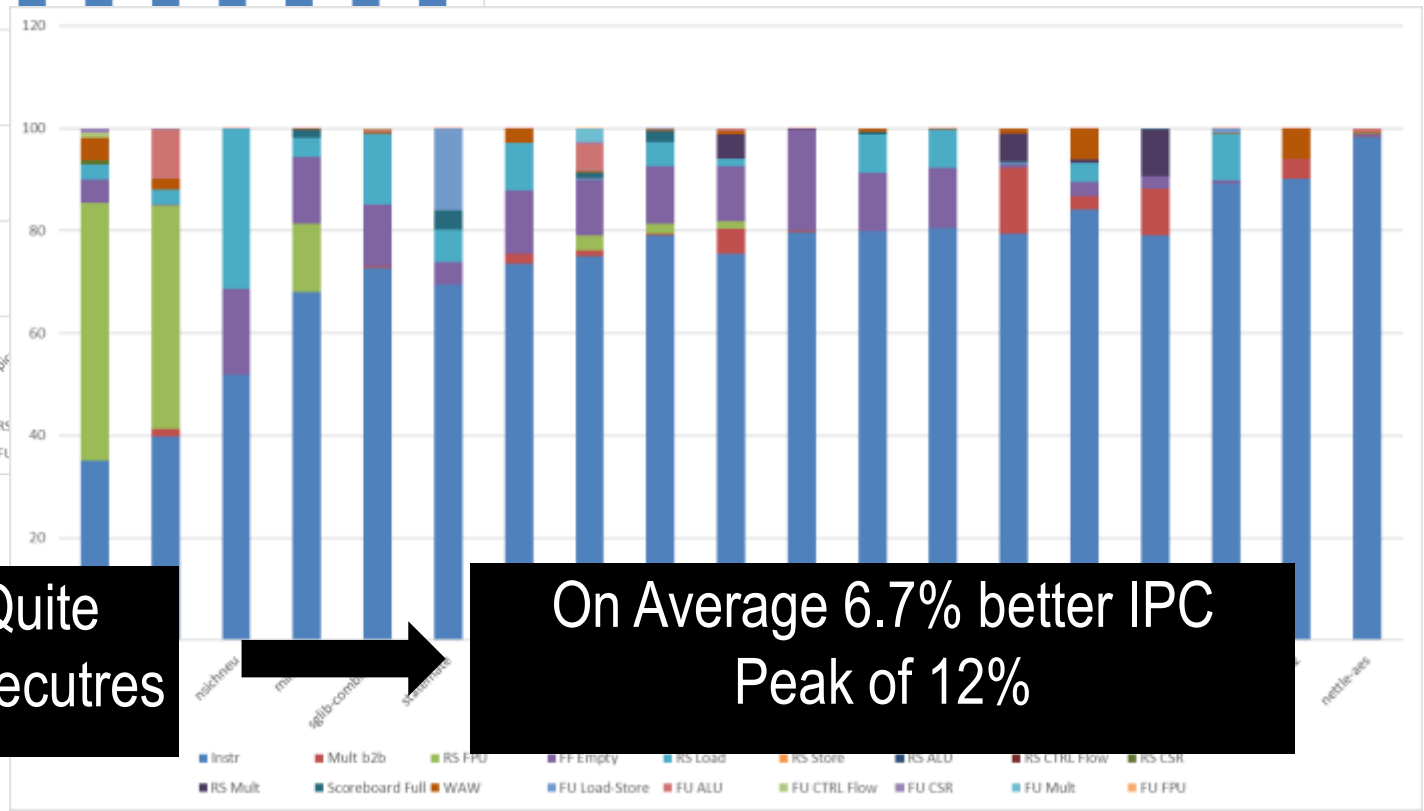
ETH zürich

PERFORMANCE: Embench-iot Benchmark Suite



PERFORMANCE: Embench-iot Benchmark Suite



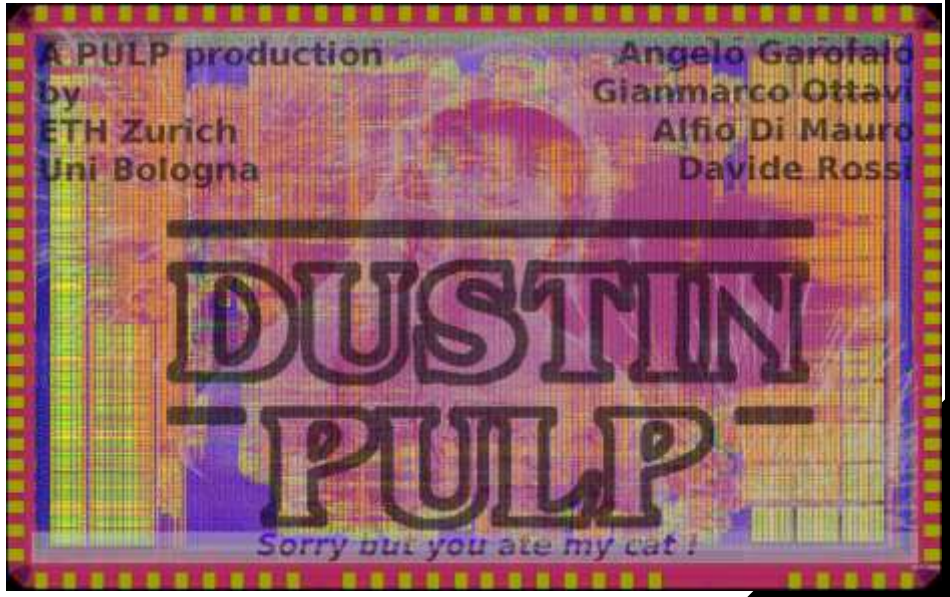


1 Bit Rename Scheme Quite effective on In-Order Architecutres

On Average 6.7% better IPC
Peak of 12%

CVA6 is open-source

- You can find all the RTL of the Core at:
  - https://github.com/openhwgroup/cva6
- CVA6 is just one of the core born from the PULP Project
  - https://pulp-platform.org/
  - https://github.com/pulp-platform
- From RTL to Silicon



ETH zürich