PA project report

Erik Diers Nicolás Zhao

24. January, 2024

1 Processor description

Our pipelining processor has a total amount of five stages. Those stages are:

- Fetch: Fetches the instruction from the memory. This stage includes a *branch predictor* and *I-cache*.
- Decode: Decodes the fetched instruction and accesses the register file.
- ALU: This stages performs basic arithmetic operations. In this stage our pipeline splits. It either takes the *ALU-pipeline*, where it goes through *D-cache* and *write-back*. Or it takes the *MUL-pipeline*, where it accesses the five multiplication stages followed by the *write-back* stage.
- Cache: This stage includes the *D-cache* which is equipped with a store-buffer. Both *D-* and *I-cache* are accessing a uniform memory, where requests are organized by the arbiter.
- Write-back: The write-back stage gathers results from prior arithmetic calculations or loads and writes them into the register file, if needed.

Hazard and forwarding logic is included into our project.

2 ISA

The implemented ISA is subset of RISC-V I and M extensions. From both extensions we implemented the following instructions:

- Arithmetic: ADD, ADDI, SUB, AND, ANDI, OR, ORI, MUL
- Memory: LW, LB, SW, SB, AUIPC (needed for la pseudo-instruction)
- Control flow: JAL, BEQ

3 Features

Our processor design fulfills the following requirements from the project guideline:

- Unified memory which receives requests from a arbiter. The latter is driven by the two caches, which are able to request at the same time, but the memory has only one port.
- Full set of bypasses, making optimal forwarding possible.

- Memory request and response latency of 5 cycles (implemented artificially with 5 chained flip flops).
- 128 bits cache lines and 4 cache lines direct mapped caches.
- 5 multiplication stages (M1, M2, M3, M4, M5).
- Write-back data cache and 4 entries store buffer.
- No reorder buffer, neither virtual memory were implemented .
- Simple branch prediction scheme that stores the most recent taken branch result.

4 Performance

All of the performance tests executed successfully, with correct results. But due to long simulation time, we did not finish the `matmul` test because the computation time would exceed our limits. The following Table 1 shows our performance results.

Program	**Time in Cycles**
`Buffer Sum`	1.604
`Mem copy`	5.601
`Matrix multiply`	($\approx$) 57.344.000

Table 1: Performance test results

5 Write-back cache FSM

The implemented finite state machine is illustrated at Figure 1, which is composed of 3 states:

- `COMPARE_TAG`. This is our initial state of the cache which receives valid requests from the pipeline and moves to the other 2 stages based on the result of the lookup on the tag store of the cache. Since our cache is write-back based, writes can be first written into clean lines, so no transition is needed towards the other stages.
- `ALLOCATE`: The cache remains in this state until it receives the complete data from the main memory. Subsequently, it transitions to the `COMPARE_TAG` state.
- `WRITE_BACK`: The cache shifts to the write-back state upon receiving a request that results in a miss on a dirty block. It then initiates a write request to memory, incorporating the dirty block. The transition to the `ALLOCATE` state occurs after the write request has been successfully transmitted to memory.

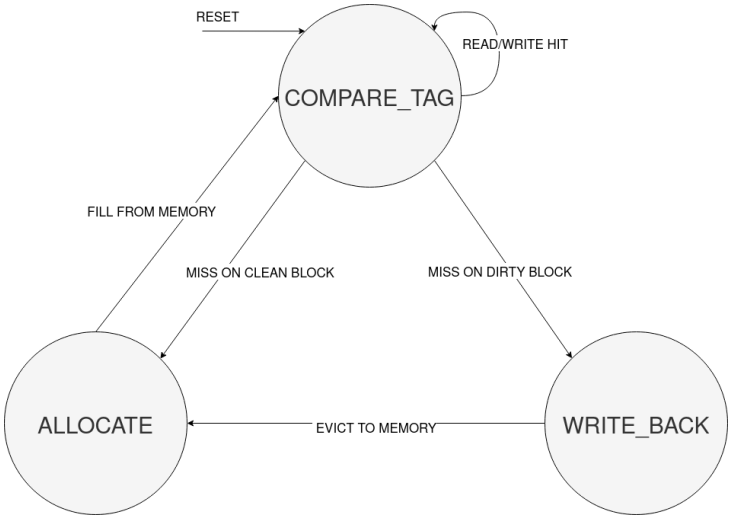

Figure 1: Write-back cache FSM

6 Control logic

Following are the most important control signals, which were used in our processor. We oriented our design after the proposed design in the book "Digital Design and Computer Architecture, RISC-V Edition" by *Sarah Harris* and *David Harris*.

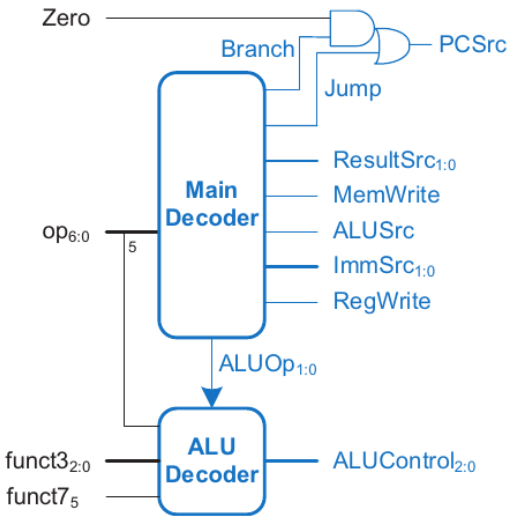

Figure 2: Control logic

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	**01**	0	00	0
`sw`	0100011	0	01	1	1	**xx**	0	00	0
R-type	0110011	1	xx	0	0	**00**	0	10	0
`beq`	1100011	0	10	0	0	**xx**	1	01	0
I-type ALU	0010011	1	00	1	0	**00**	0	10	0
`jal`	**1101111**	**1**	**11**	**x**	**0**	**10**	**0**	**xx**	**1**

Figure 3: Main decoder

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw`, `sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

Figure 4: ALU decoder

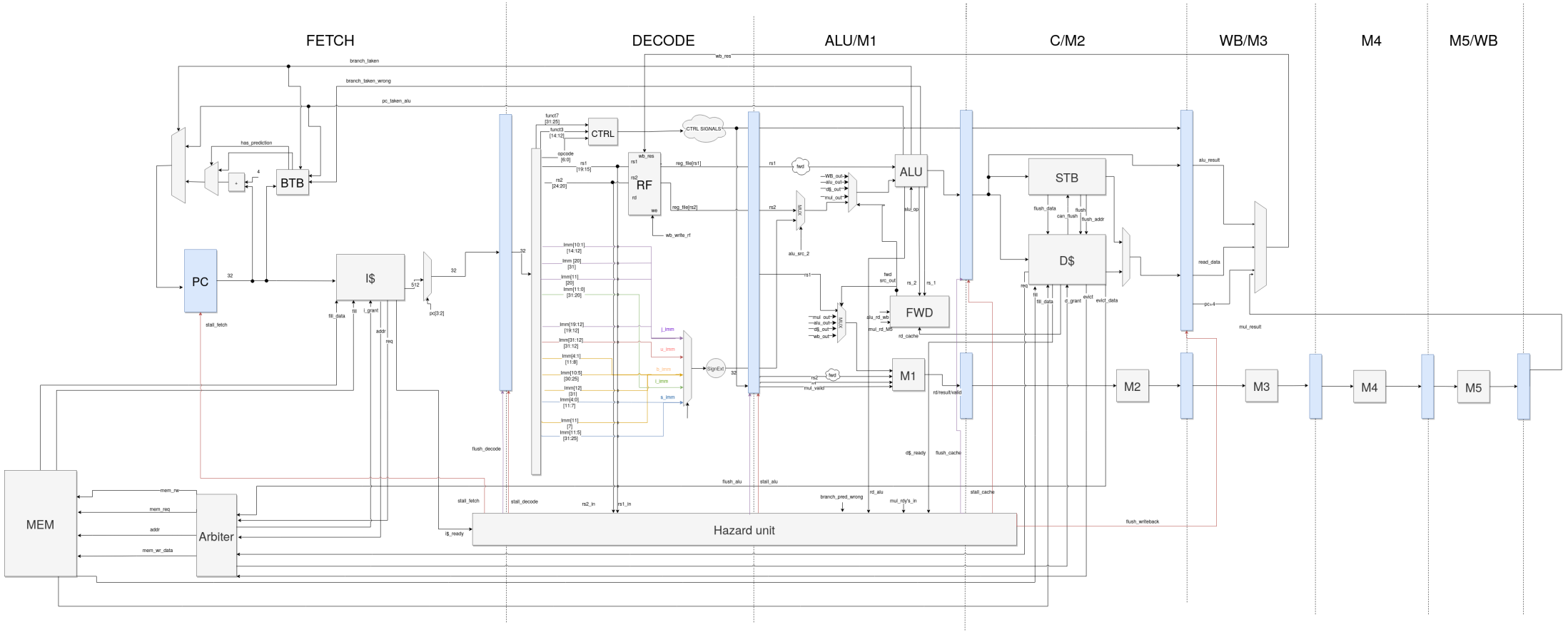

Figure 5: Block diagram. Fully detailed version.

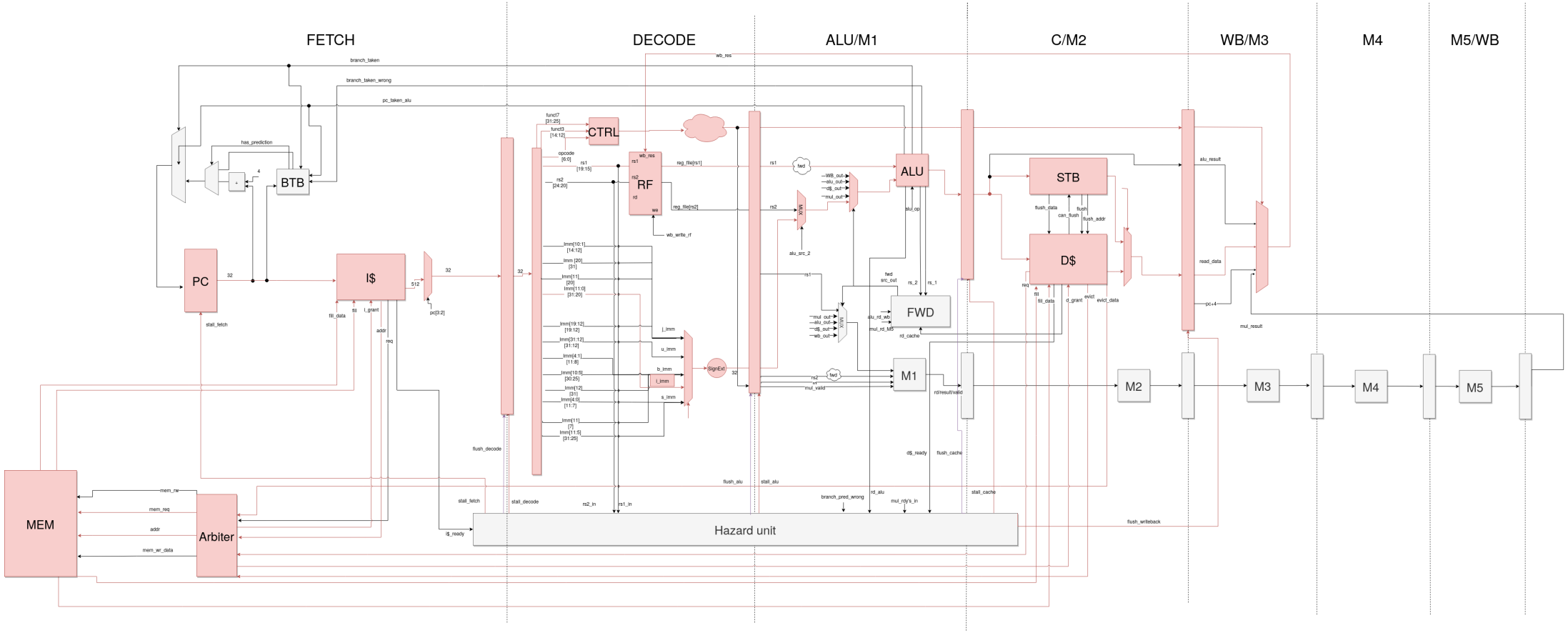

Figure 6: Load pipeline

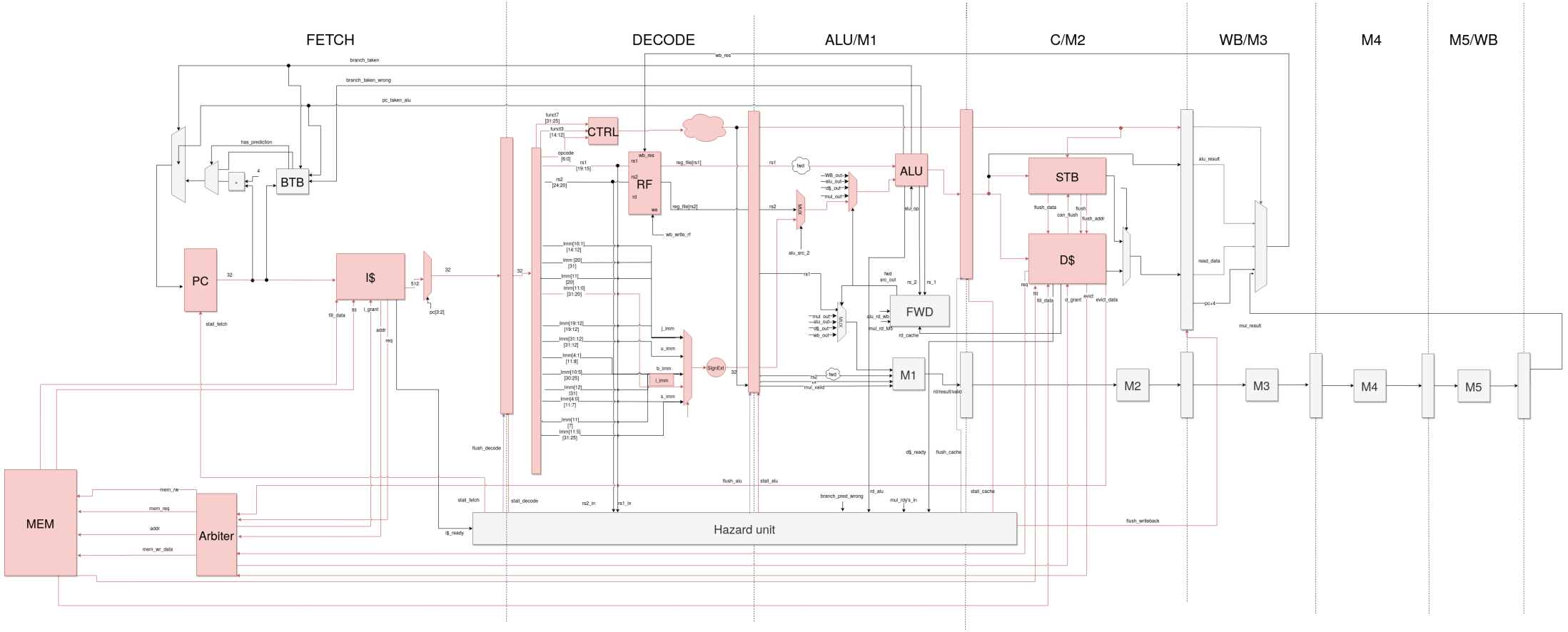

Figure 7: Store pipeline

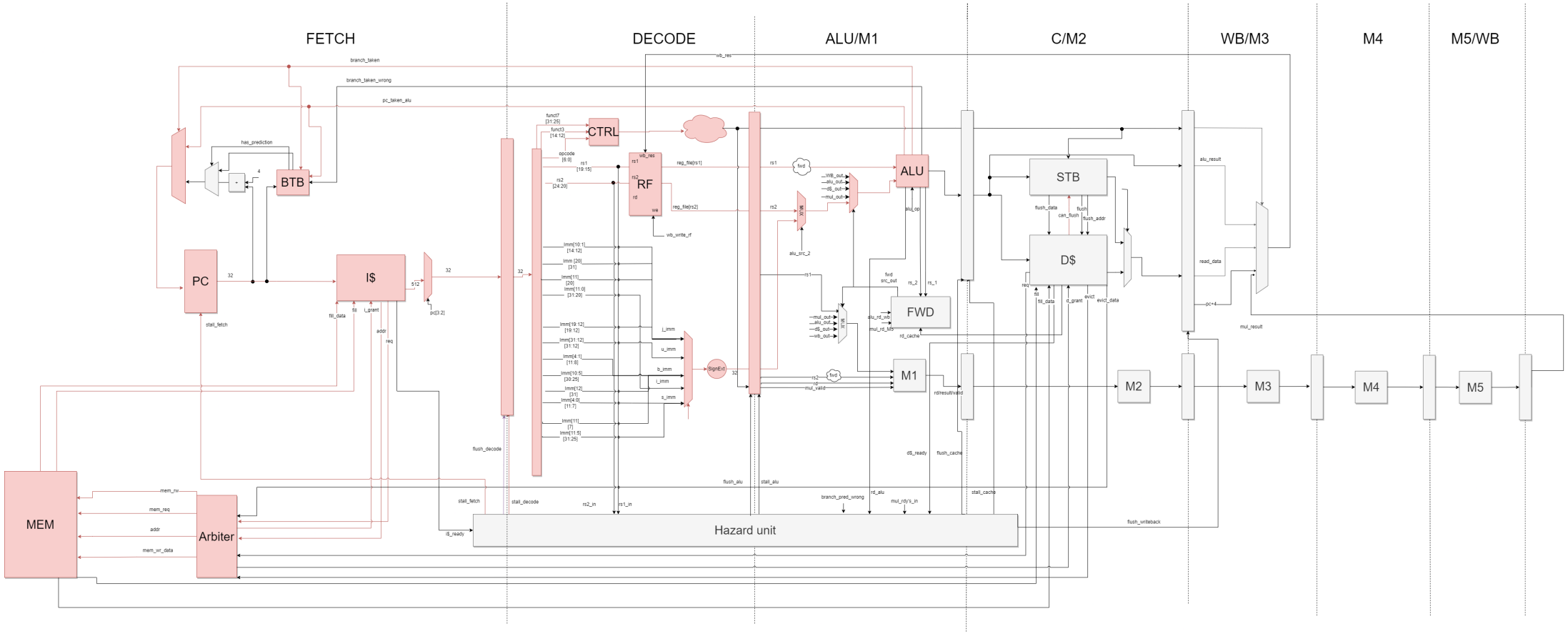

Figure 8: Branch pipeline

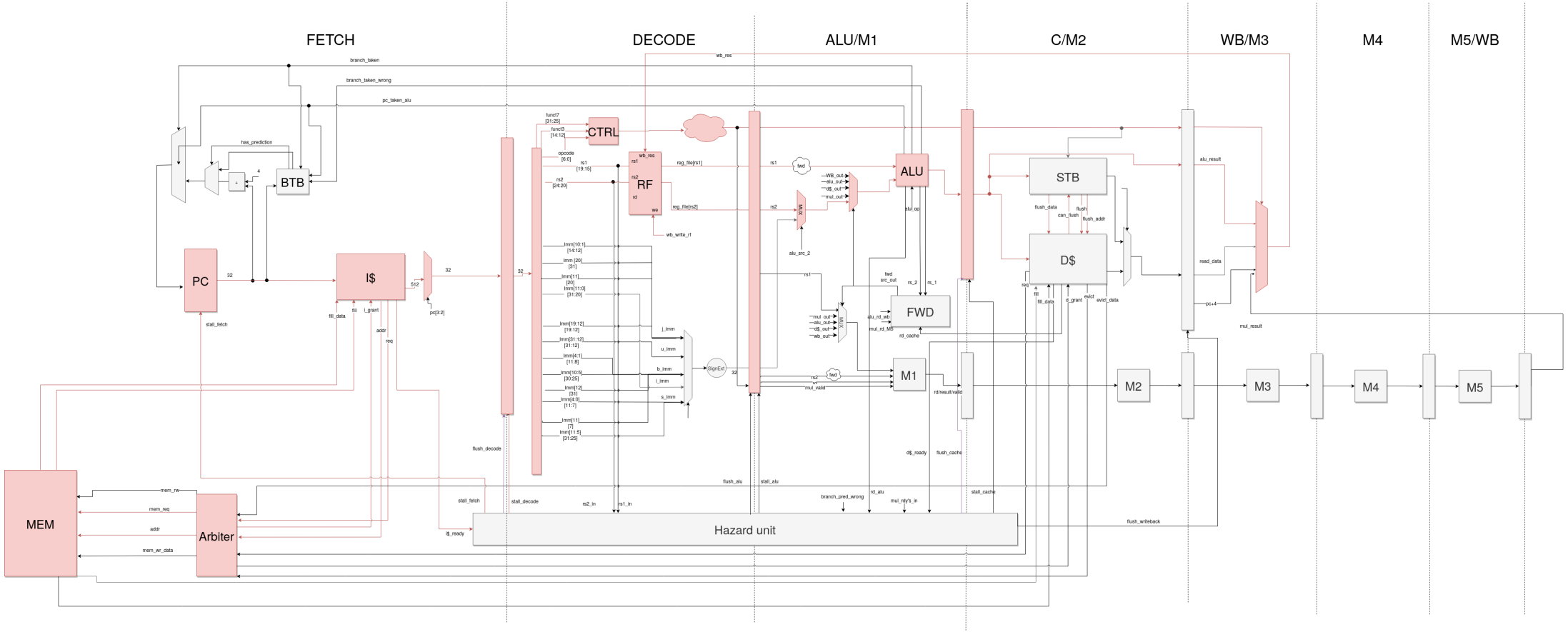

Figure 9: Arithmetic pipeline

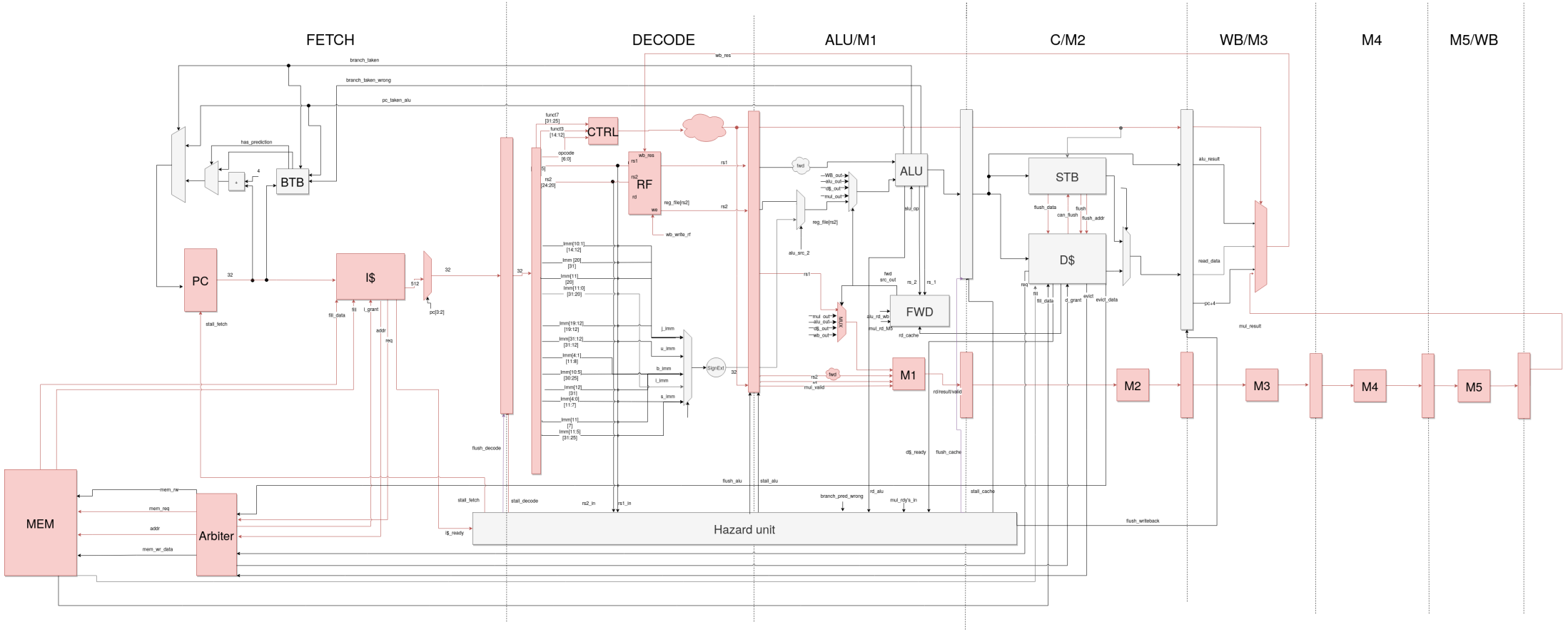

Figure 10: Multiplication pipeline