Processor Design
Multi-threaded Core

Final Report

Joel Sanchez Moreno

January 17, 2021

1 Introduction

The project consists in adapting a single-threaded core to a multi-threaded core which number of threads can be adapted to the user requirements. This extension means implementing several changes on the core pipeline and stages as explained later in this document.

There are two main approaches that can be followed when implementing multithreading on a processor: coarse-grained and fine-grained.

1.1 Coarse-grained

This approach performs thread switching based on stall events, that is, a thread is executing until there is a cache miss or a data dependence that blocks its execution. The pipeline is flushed every time we switch from one thread to another, or when there is an exception.

1.2 Fine-grained

This approach performs thread switching every quantum cycles, usually quantum is set to 1. The threads follow a round-robin policy in which all threads are allowed to execute instructions. The pipeline of a given thread is flushed when an exception occurs during the thread execution.

1.3 Project goal

We have decided to apply fine-grained multithreading approach since it is more appropriate for real-time systems. The goal of this project is that the execution time of one thread must not have negative impact on the execution time of the rest of the threads, that is, the execution of a single thread should never be worse than if it was executed in isolation.

It is important to note that this fine-grained approach means that most of the pipeline registers are replicated and this causes an area and power increase in comparison with a coarse-grained approach, but allows fairness in thread execution.

2 Project design

This project is based on a single-threaded core [1] developed by the author of this report during Processor Architecture subject from the Master in Innovation and Research in Informatics (MIRI) at the Universitat Politecnica

de Catalunya (UPC).

The baseline core consisted in a 5-stage pipeline, with a fixed-width custom Instruction Set Architecture (ISA), data and instruction caches, data and instruction Translate Lookaside Buffers (TLB), a data cache store buffer and a reorder buffer.

During the next sections we explain the most important features of the core, how the core has been adapted to run with multiple threads and the additional features that have been added to increase performance.

2.1 Pipeline

This processor has 5 pipeline stages: Fetch, Decode, ALU/MUL, Cache and WriteBack. The MUL instructions have a 5 cycles fixed latency and due to this we have developed multiple MUL stages to compute the value and met the timing requirements. In the case of a MUL instruction, the minimum pipeline is F, D, M1, M2, M3, M4, M5 and WB.
There is a Reorder Buffer of 8 entries, which are shared among the threads, that ensures the core has a better performance. The ALU and MUL stages are connected to the Reorder Buffer (RoB) such that there is a full set of bypasses in order to speed up the execution.

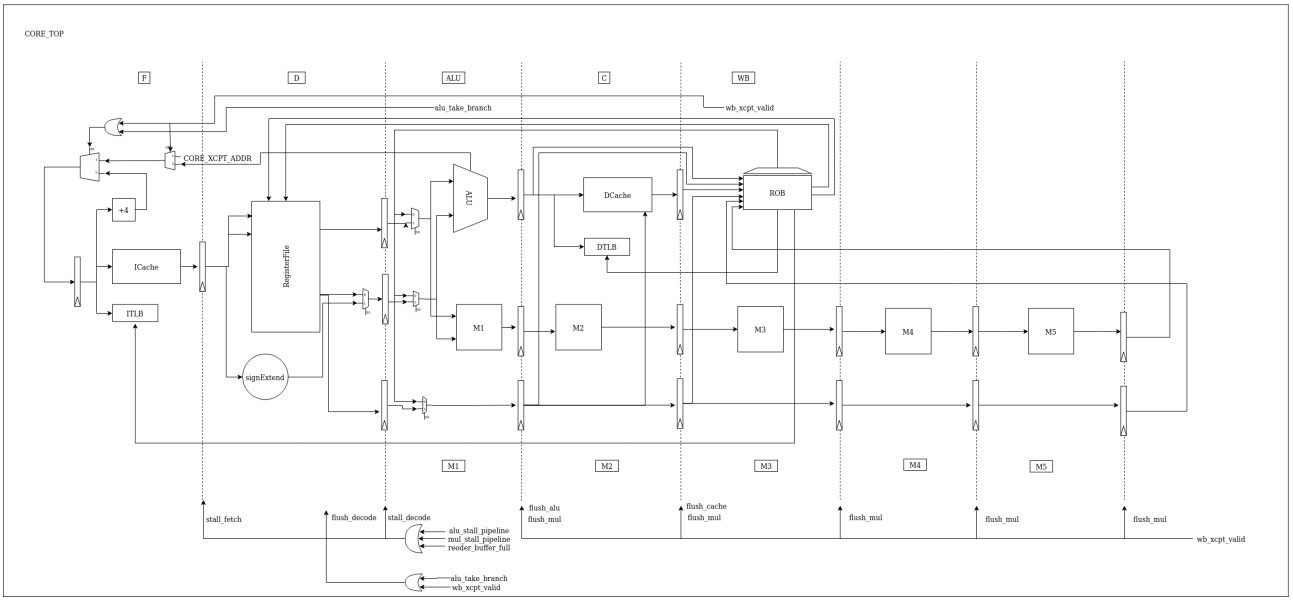

Figure 1: Core pipeline

On the other hand, the ALU can send M-type instructions to the RoB if Cache stage is busy, this allows the next thread non-dependant instructions to be computed without stalling the pipeline and boosting performance. The RoB sends the pendant M-type instructions to the Cache stage once it is ready making use of an arbiter that takes into account which is the active thread in order to handle priorities, but allowing other threads to send the request to Cache stage if possible.

It is important to note that instructions stall the thread pipeline if there is a dependency with a previous instruction. In addition, the exceptions raised during Cache stage can flush younger instructions stored on the RoB given that they have not been retired.

We can then conclude that the designed pipeline ensures that all instructions are retired in order even if they are not executed in order.

Detailed schematics of each pipeline stage and global simplified schematics that show which part of the processor is used when we execute each type of instruction can be found on annex section.

2.2 Instruction Set Architecture

The implemented Instruction Set Architecture (ISA) is based on MIPS. There are 3 types of instructions and we can see the encoding in the following figure:

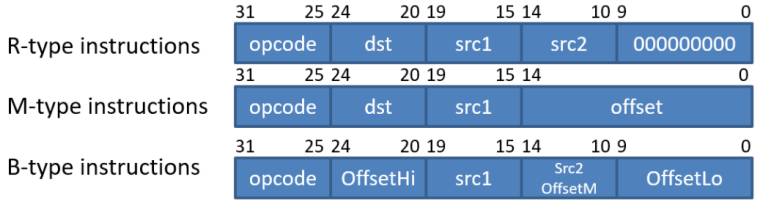

Figure 2: Core Instruction Set Architecture

The R-type supported instructions are ADD, SUB, MUL, ADDI, SUBI, NOP, SLL and SRL. The M-type instructions are memory access instructions on what we support LDB, LDW, STB, STW, LR, SCB and SCW, and finally the B-type instructions are the branches instructions for what we have BEQ, BLT, BGT, BLE, BGE, BNE and JUMP. In addition, we included more instructions in order to manage the exceptions and the threads, which are MOV, TLBWRITE, IRET, GETTHR and CHGMODE.

We can see that there are two instructions that allow the user to manage different threads: GETTHR, which writes the thread id on the destination register, and CHGMODE, which changes between single-threaded mode and multi threaded mode.

In addition, memory instructions have also been added to allow the user to work with a non-relaxed consistency model. The new instructions are load reserved (LR) and store conditional (SCB, SCW) of different sizes. Load Reserved instruction allows the user to ensure that a cache line can only be modified by the same thread that reserved that instruction. The store to be performed must be conditional and has two different granularities: byte (SCB) or word (SCW).

When a thread performs a LR instruction it does not matter if a previous thread reserved that cache line, the cache line is always reserved for the latest thread that performed the LR instruction. In case a thread performs a store or a store conditional instruction o a reserved line by a different thread, an exception is generated. On the other hand, all threads can perform load

operations on reserved lines.

2.3 Registers

Each thread has its own register file which consists on 32 registers of 32 bits each and 4 special registers: rm0, rm1 and rm2 for exceptions; rm4 for the current privilege state of the machine (0 for user and 1 for supervisor).

Exception registers allow the user to know at which instruction the exception was triggered (rm0), the exception type (rm2) and the address value (rm1) in case of an address fault exception.

2.4 Core boot

Once reset is de-asserted the core boots in single-threaded mode in which thread 0 is the only active thread. All threads boot with Supervisor mode, which means that virtual memory is not enabled, and start fetching instructions from address 0x1000. In case of an exception, the thread is always forced to jump to address 0x2000 in which the user must place the exception handler routine.

2.5 Memory

The project has been verified with a memory hierarchy of two levels: the main memory, which has been placed on the core wrapper and has a fixed response latency of 10 cycles, and the core memory (L1), which consists on an instruction cache and a data cache.

The test bench contains the main memory, which has been implemented with Flip-Flops and two queues to handle Fetch and Cache misses. It is important to note that each thread can have up to two requests in flight at a time, one miss during Fetch stage and one miss/eviction during Cache stage.

Both the instruction and data caches have 4 cache lines of 128 bits per cache line, are two-way associative, implement an MSI protocol and have a LRU replacement policy. In addition, the data cache has a Store Buffer with 8 entries which holds stores that have not yet completed.

The store requests that have been stored on the Store Buffer are performed when the Cache stage is free. In other words, when the thread exe-

cuting on the ALU stage stalls or sends the instruction to the RoB. Then, the next cycle the Cache stage will be free so it will pop the store from the queue and perform the write operation on the corresponding cache line.

When the core has been configured in single-threading mode, then thread 0 can use all cache ways and there is split among threads.

The caches registers have not only been replicated for multiple threads, but we also implemented new features and a new control logic to handle misses in order to increase performance.

2.5.1 Thread communication with main memory

The instruction and data caches support receiving responses for a thread that is not active on a given cycle, in that case the response is stored such that the information is available to the thread that made the request once it is scheduled again. In addition, the data cache also supports evictions from multiple threads to be handled in parallel.

2.5.2 Snooping

A snooping feature has been added such that all threads have visibility of the content of all ways, which means that they can hit and read a line from another thread cache line. In case of a miss, the cache line is always allocated on the way(s) reserved for the thread that missed. This feature ensures consistency and a better performance.

2.5.3 Avoid repeating requests

When there is a miss, the cache looks if another thread already sent a request to memory to get the same line. If that is the case, then the active thread moves to a wait statement until the new line is received. Otherwise, the TAG is saved and a miss request is sent to memory.
For the data cache we also have to handle writes, so in case of a write request that hits on another thread cache line, then the line is read from the latter, modified and written in the same way. This avoids extra evicts and reads from Main Memory that would cause a loss on performance and ensures coherency since all threads have the same visibility over a given cache line.

2.6 Virtual Memory

The threads also support virtual memory, in order to do that we have an instruction TLB (iTlb) on the fetch stage and a data TLB (dTlb) on the cache stage. The Virtual Address (VA) size is 32 bits and the physical one is 20 bits, with pages of 4KB. The way the Physical Address (PA) is computed depends on the exception handler routine, which in our tests perform the next operation: PA = VA + 0x8000.

As has been noted in previous sections, threads boot in Supervisor mode in which virtual memory is disabled. When a TLB miss occurs, the thread state is saved on registers rm0, rm1 and rm2 before it jumps to the exception handler, which takes care of resolving the issue. The exception handler makes use of TLBWRITE, MOV and IRET instructions to resolve the exception and returns to the source code.

It is important to note that when a thread performs an IRET instruction to jump to the instruction that triggered the exception, it returns on User mode, which means that virtual memory is now enabled.

The TLBs do also have the snooping and avoid sending multiple requests features that have been explained in previous sections.

3 Verification

The software tools that have been used are Verilator, in order to compile the RTL and perform simulations, and Gtkwave to analyze the generated waveforms. The verification process not only includes the tests developed for the verification of the baseline PA core, but also an extension of those tests to verify multithreading mode. Since multithreading can increase severely the number of corner cases and failures, it is very important to test operations that require coherency with all the threads enabled.

For the different multithreading modes we have developed tests that verify the following features:

- Check that all R-type, B-type and M-type instructions work.
- Check that branches can be executed with MUL instruction latency.
- Check that load reserve and store conditionals work on multithreading.

- Check that new privileged instructions to get the thread ID and change the multithreading mode work.
- Check that all the data generated with the instructions is bypassed properly at the different stages of the execution.
- Check that non-finished load/store requests saved on the RoB are performed once data cache is ready.
- Check that all instructions are retired in order even if our implementation allows out-of-order execution.
- Check that in single-threading mode the thread 0 has access to all the ways of the instruction and data caches.
- Check that in multi-threading mode the threads have visibility of all ways, but in case of a miss they only modify the ways reserved to the thread.
- Check that the next exceptions work as expected on both single-threading and multi-threading mode: instruction TLB miss, fetch bus error on a main memory request, illegal instruction, overflow on ALU or MUL stages, data TLB miss, cache bus error on a main memory request, cache address fault on a main memory request and cache store conditional for a non-reserved line.
- Check that virtual memory is supported and multiple privilege modes work as expected.
- Check that the pipeline is flushed in case of an exception.
- Check that requests saved on the Store Buffer are performed while data cache is idle.
- Check that the core is able to perform a matrix multiply in both modes.

4 Performance

In order to ensure that we met the project goal we have compared the matrix multiplication performance of the proposed design with the base project design which implemented single-threaded.

The performance obtained is exactly the same, which means that we met our goal which was that the execution of a single thread should never be worse than if it was executed in isolation. The Matrix Multiplication test of 128x128 finished in 216.441.235ps, which means a performance of 989,4 MIPS.

5 Extension

During the course I have also participated on the Efabless Open MPW Shuttle Program [2], which allowed engineers to develop their designs, which had to be fully open-source projects, and submit them. The costs for fabrication, packaging, evaluation boards and shipping were covered by Google.

There were three main requirements:

- Project had to be fully open source and documented.
- Project had to include a LICENSE file for an approved open-source license agreement.
- The target was 130nm process

On the other hand, they also forced the submitters to use the SkyWater Open source PDK set of tools [3] to develop the physical design placement and routing. Unfortunately, it was not possible to get to the last stage of submit process due to the fact that tools were not stable, there were multiple repositories with different development branches and the deadlines were not feasible.

6 References

[1]. GitHub website. (2020). MIRI Processor Architecture Core. Last time it was accessed was January 17, 2021.
Link: https://github.com/joelsanchezmoreno/MIRI_PA_core

[2]. Efabless website. (2020). Welcome to the Efabless Open MPW Shuttle Program. Last time it was accessed was January 17, 2021.
Link: https://efabless.com/open_shuttle_program

[3]. Github website. (2020). SkyWater Open Source PDK. Last time it was accessed was January 17, 2021.

Link:https://github.com/google/skywater-pdk

[4]. GitHub website. (2020). MIRI Processor Design Core. Last time it was accessed was January 17, 2021.
Link: https://github.com/joelsanchezmoreno/MIRI_PD_core

7 Annex

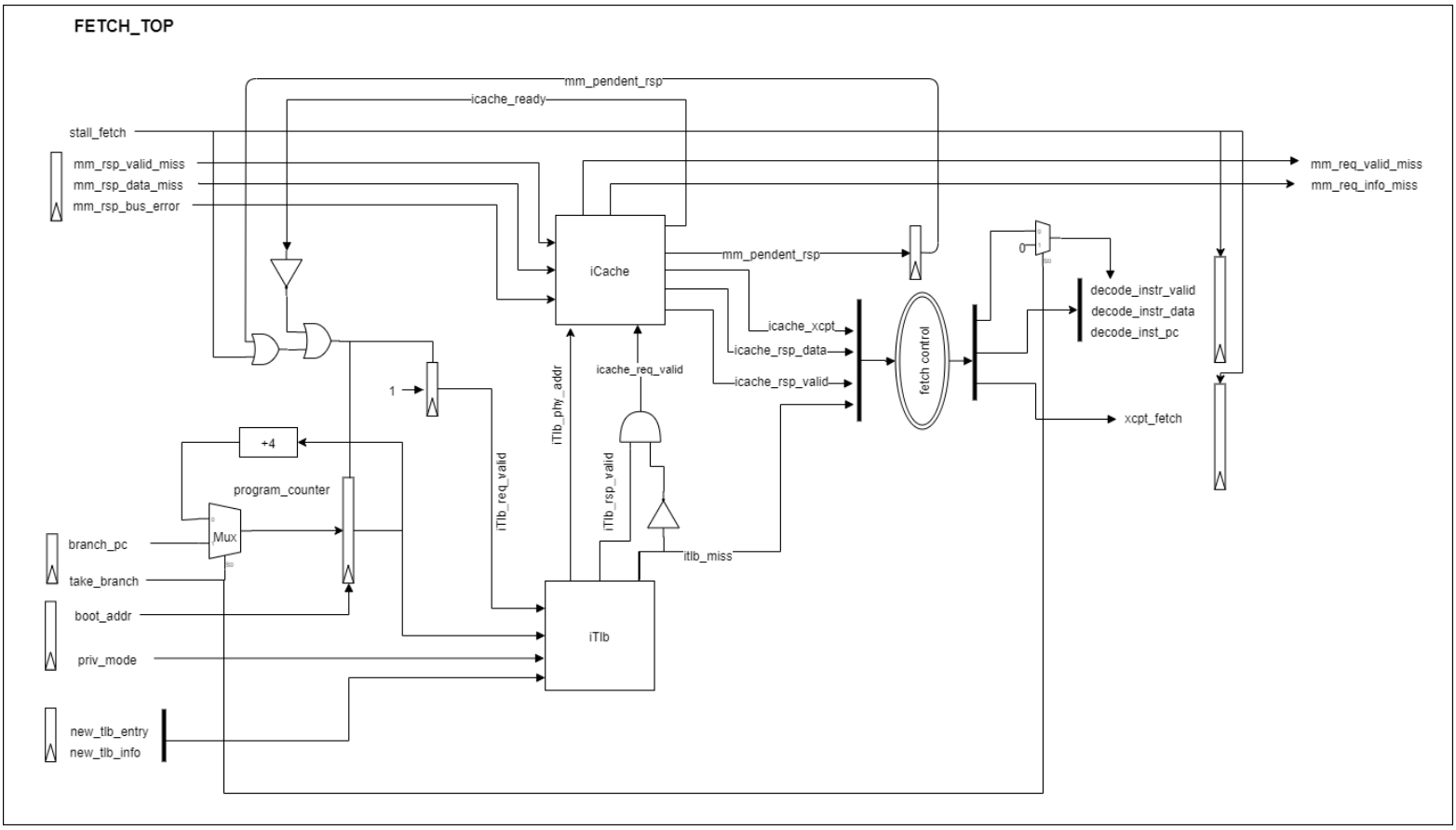

Figure 3: Fetch Stage

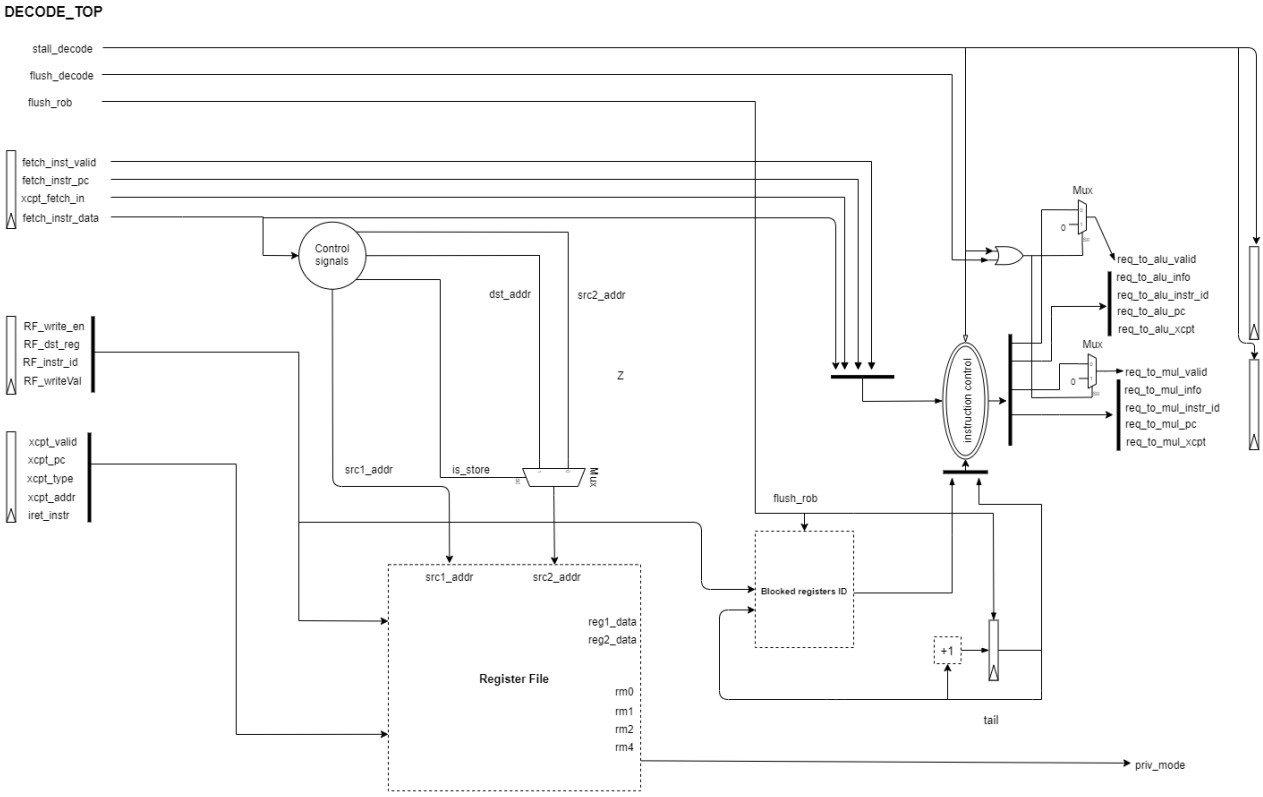

Figure 4: Decode Stage

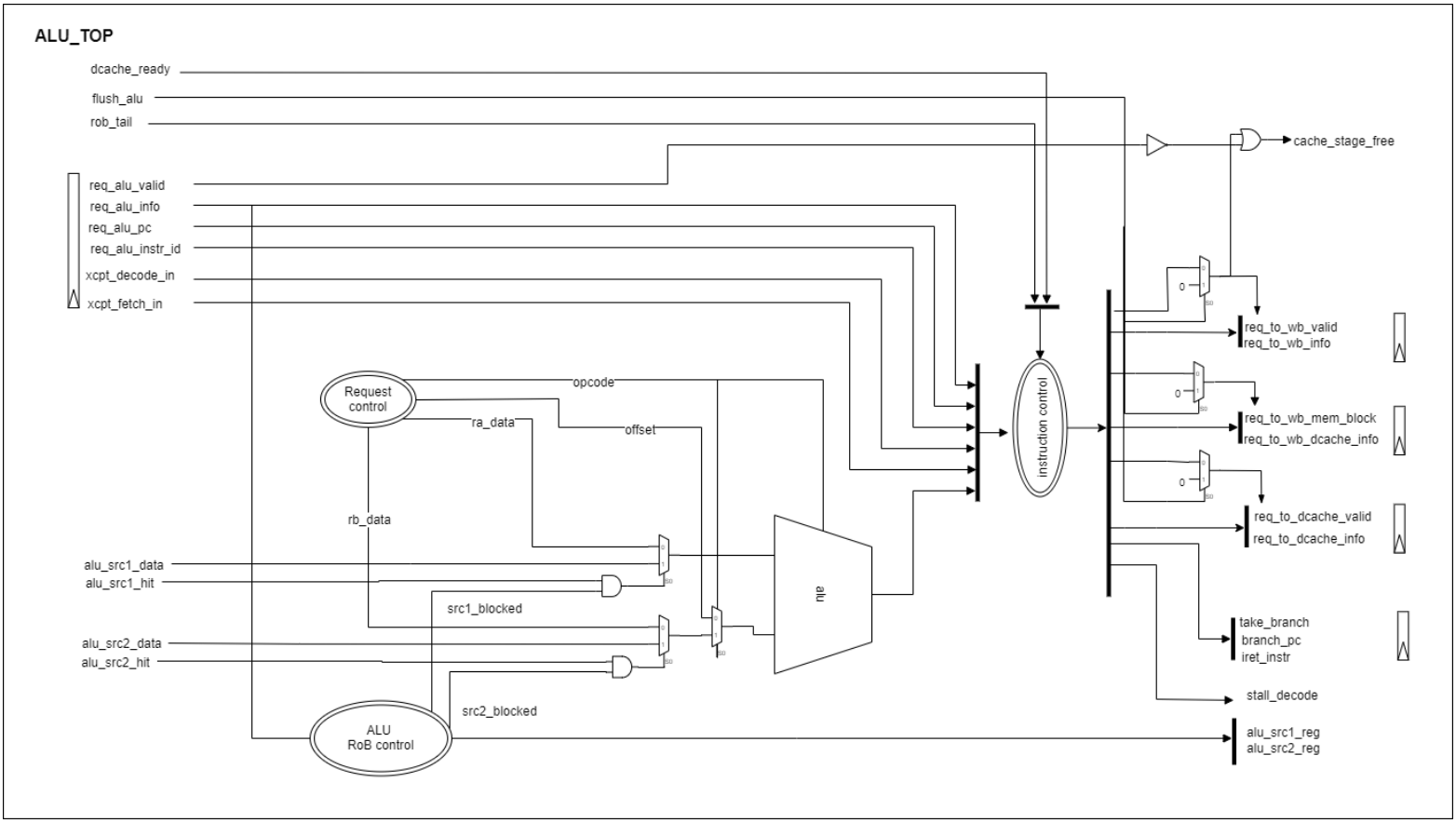

Figure 5: ALU Stage

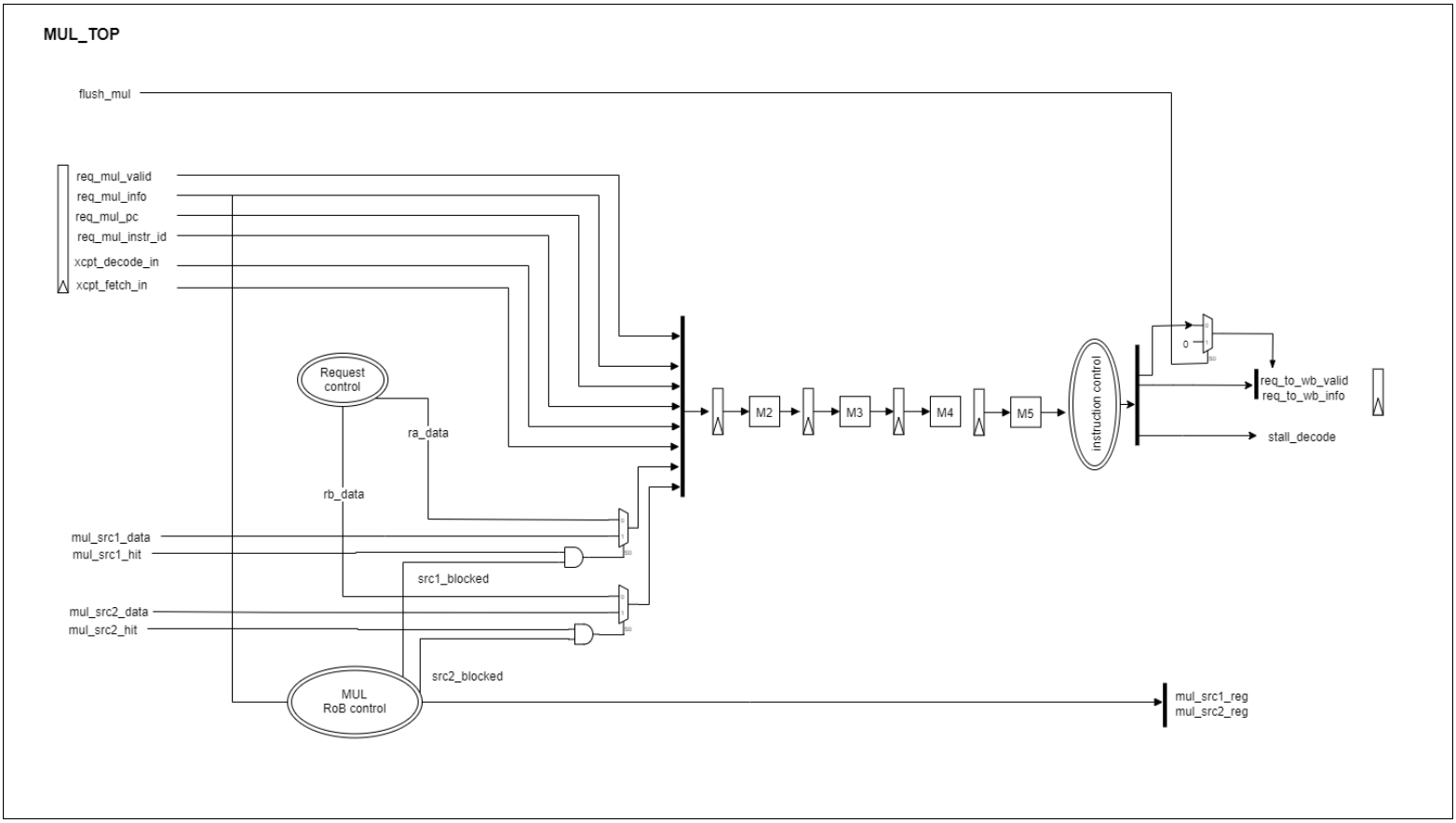

Figure 6: MUL Stage

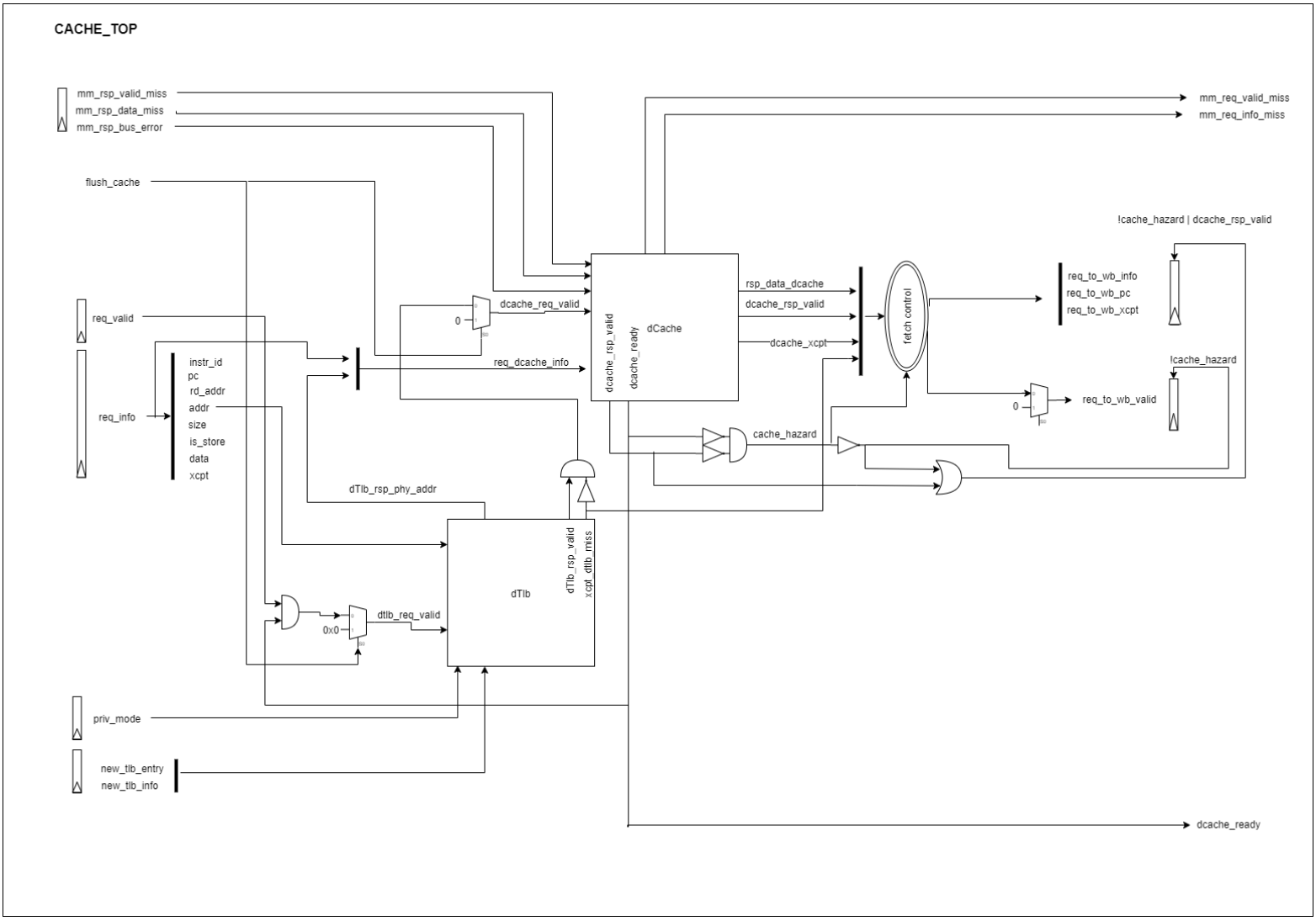

Figure 7: Cache Stage

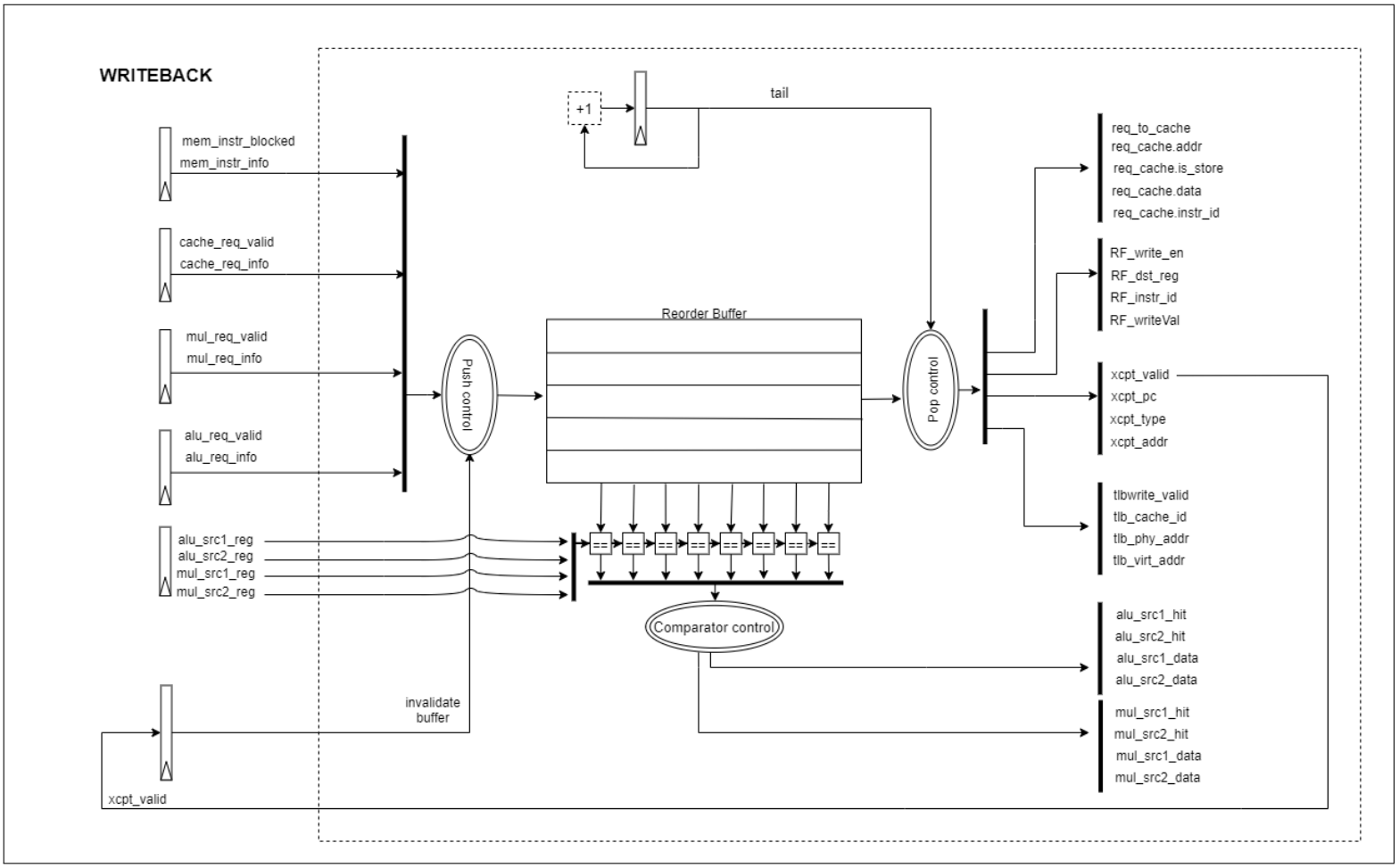

Figure 8: WriteBack Stage

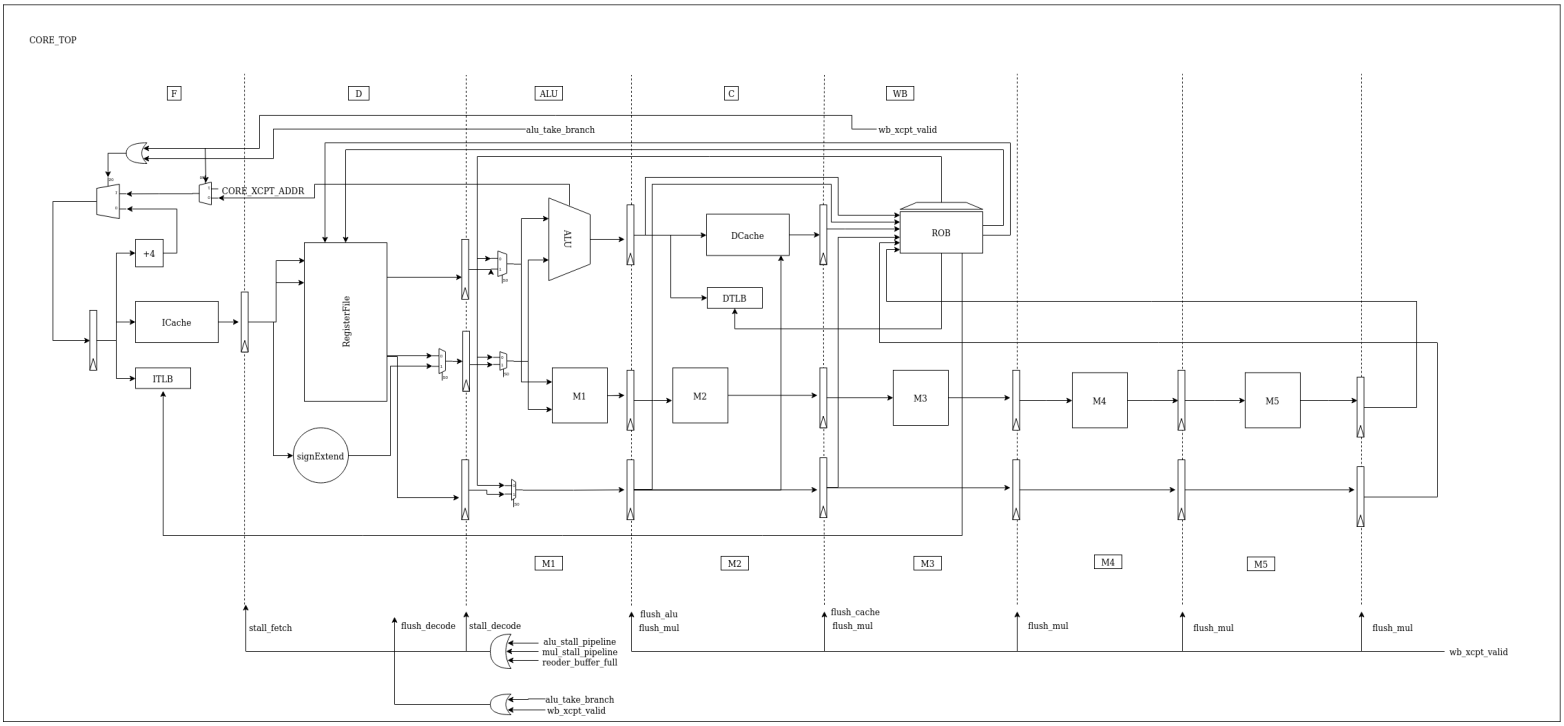

Figure 9: Full Pipeline

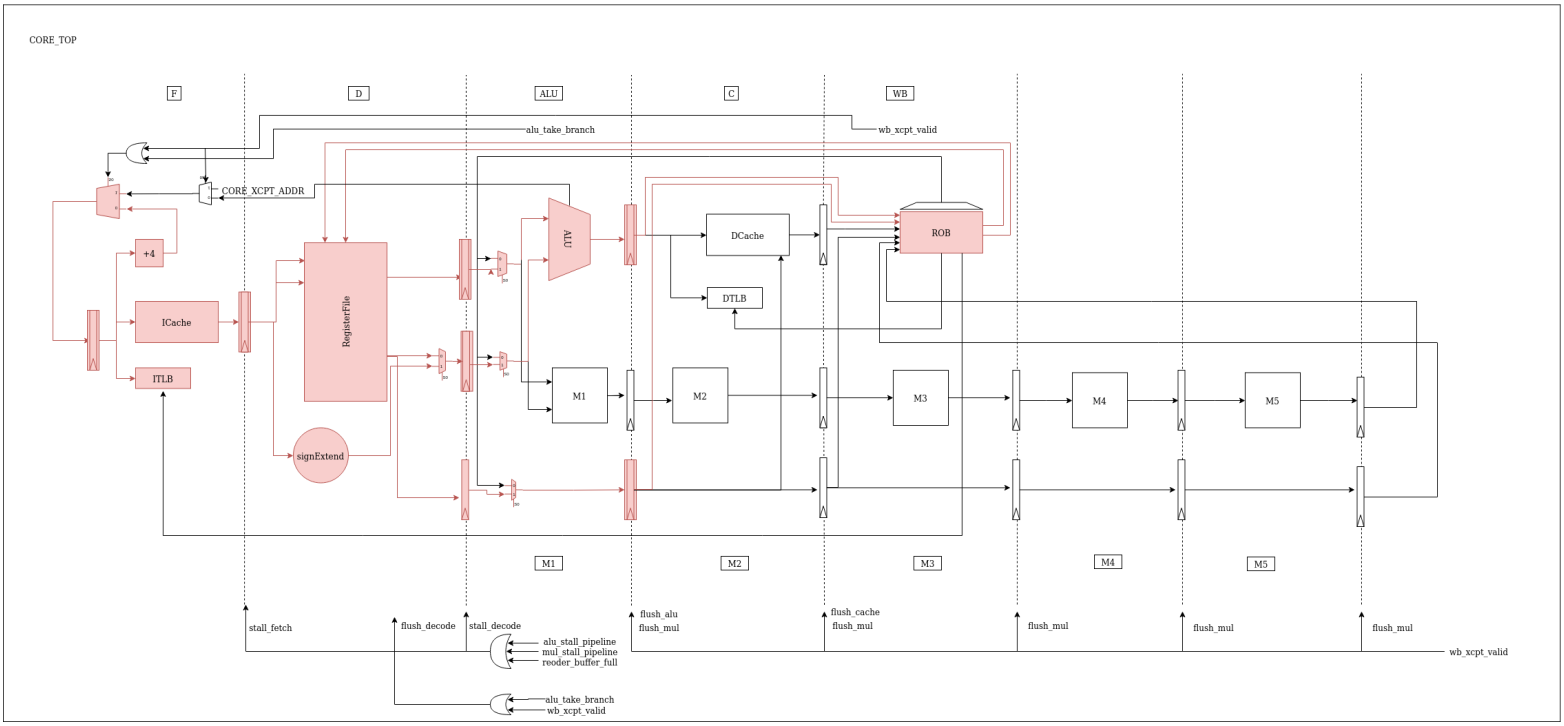

Figure 10: Full Pipeline - R-type Instructions

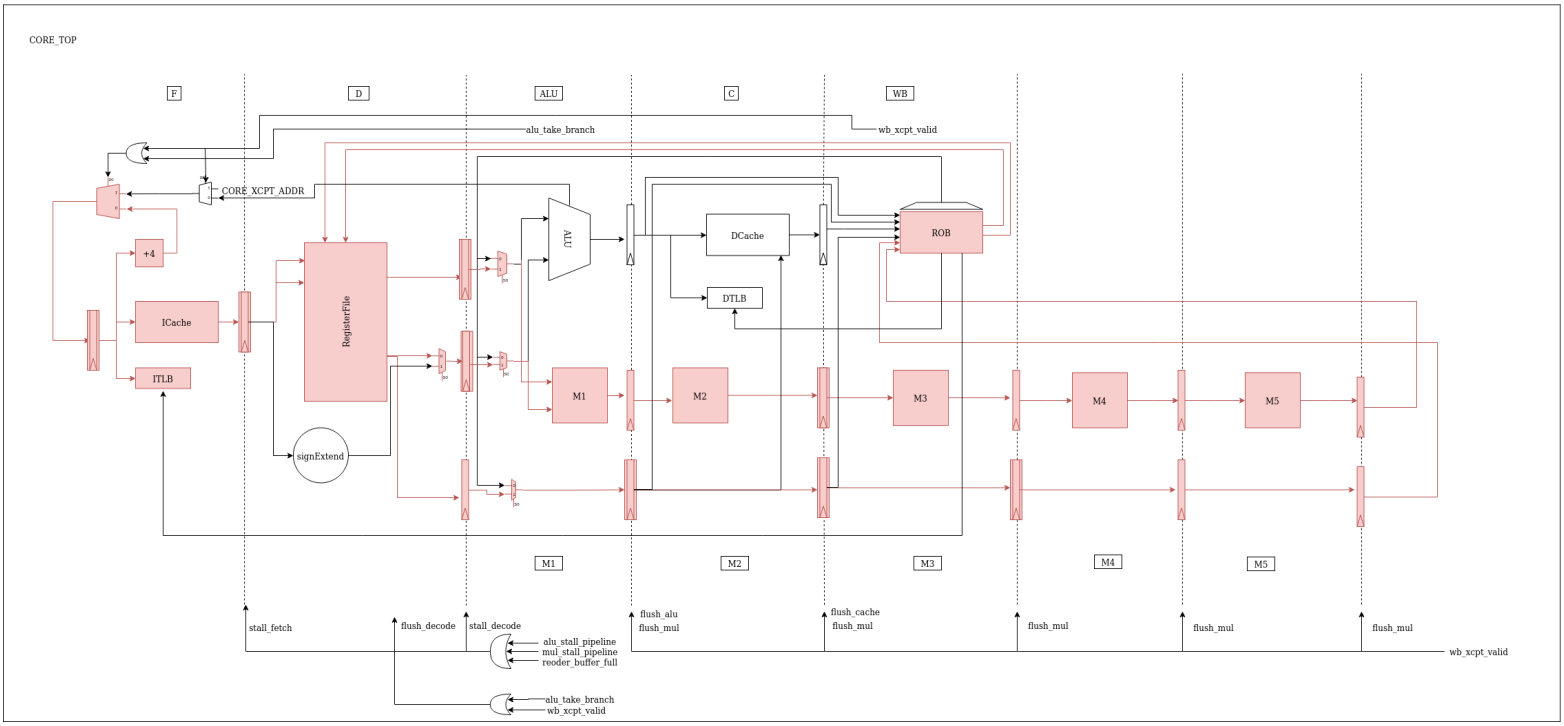

Figure 11: Full Pipeline - MUL Instruction

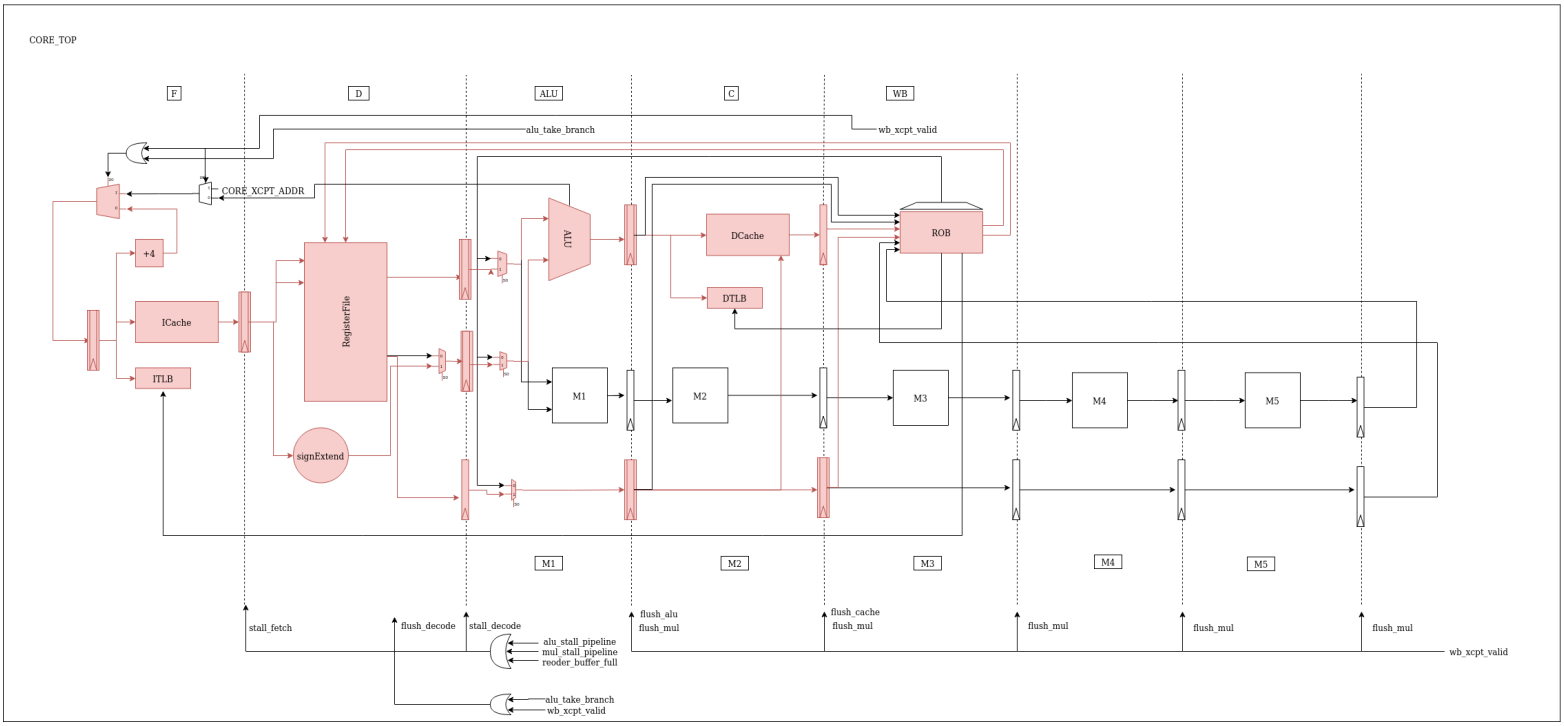

Figure 12: Full Pipeline - M-type Instructions

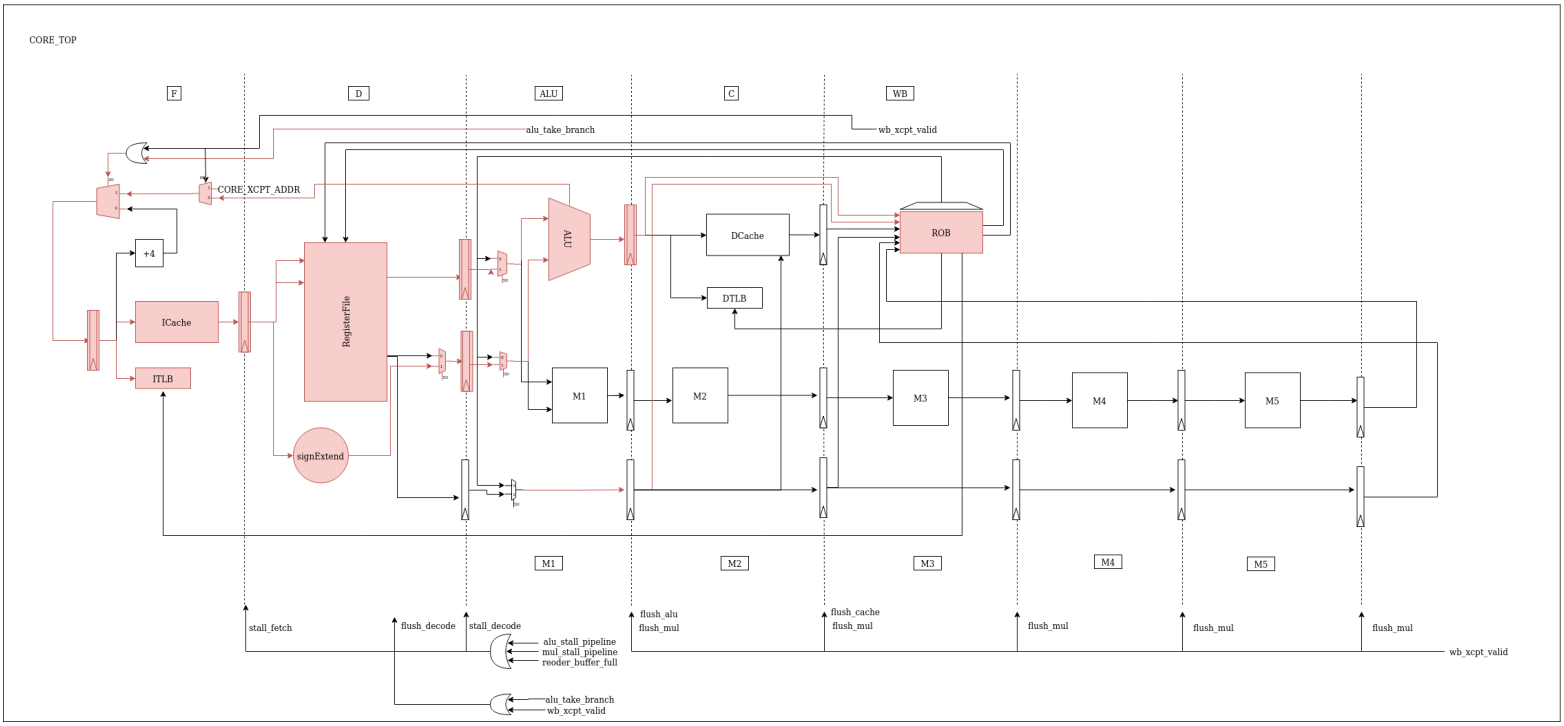

Figure 13: Full Pipeline - B-type Instructions