ECE 338 | Parallel Computer Architecture
TERM PROJECT

Spyridon Stamoulis 03775,
Konstantinos Andrikopoulos 03649,
Stavros Stathoudakis 03512

University of Thessaly
`{spstamoulis, sstathoudakis, kandrikop}@uth.gr`

CODE

You can find the code of this project in this github directory

FSM

Module Purpose

The `simpleFSM` module is a Finite State Machine designed to enhance processor performance by optimizing the execution of small, backward-branching loops. Its core function is to detect a compatible loop, cache its instructions in a dedicated Block RAM (BRAM), and feed the instructions to the CPU without using the Instruction Fetch stages of the RISC-V architecture. This process, bypasses the need to re-fetch instructions from main memory on each iteration and thus allows the cpu to "turn-off" the 2 Instruction Fetch stages. This results in significant performance gains, power savings, and reduced pipeline stalls for frequently executed loops.

FSM Operation and States

The controller is implemented as a four-state FSM that manages the instruction flow and controls the BRAM.

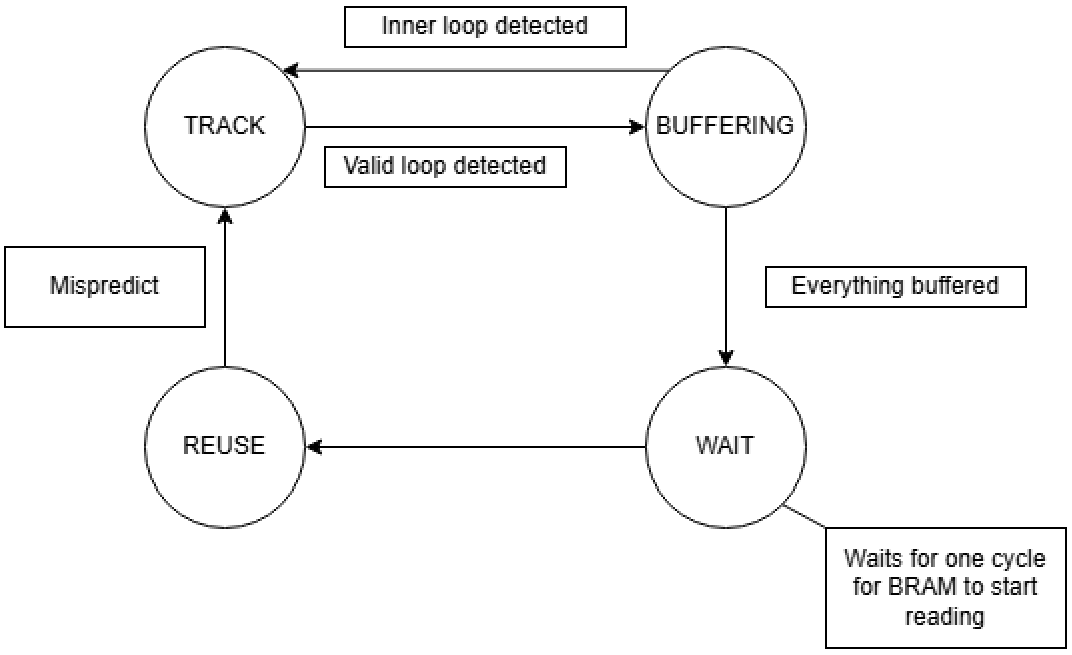
State Definitions

- `TRACK`: The initial state. It monitors the instruction stream for a valid backward branch that signifies a **small** loop. If a candidate loop is found, it transitions to `BUFFERING`.
- `BUFFERING`: In this state, the instructions comprising the loop body are sequentially written into the BRAM cache. The FSM remains in this state until the final branch instruction of the loop is stored. If we detect an inner loop we exit and go again to the track stage.
- `WAIT`: A single-cycle transitional state that allows the BRAM to write the last instruction of the loop (the branch) and then switch from writing to reading mode. It then asserts the `block_signal` which is responsible for disabling the IF stages.
- `REUSE`: The primary optimization state. The FSM supplies instructions directly from the BRAM to the processor pipeline. The FSM assumes that the all the paths from the branch are taken (which is ususally true when talking about for or while loops). Now despite not having a branch prediction-mechanism, the specific CPU now enjoys a stall-less flow of instructions when specific criteria are met. The REUSE state continues to assume that all the paths are Taken until the cpu signals that the last one was Not Taken. In that case the FSM transitions to the TRACK stage and awaits the next branch instruction.

Key State Transitions The logic for transitioning between states is as follows:

- `TRACK` → `BUFFERING`: Occurs upon detection of a valid backward branch, which jumps less or equal instructions as the PARAMETER `LOOP_SIZE` is set to (parameter is in bytes). The branch instruction may be a B-type or JAL with a negative immediate.
- `BUFFERING` → `WAIT`: Occurs once the branch instruction that closes the loop is buffered, and the loop is validated as having no internal branches.

- `BUFFERING` → `TRACK`: Occurs on two occasions. One is if we detect a branch or jump instruction inside our loop that is not the one that we detected in the TRACK stage. And the other one is if the main branch that we detected in the TRACK stage turns out that is NT. This needs to be checked because the moment we detect a compatible branch in the TRACK stage we continue to buffer the next instructions without considering the possibility that the branch may not be taken.
- `WAIT` → `REUSE`: An unconditional, single-cycle transition to begin reusing instructions.
- `REUSE` → `TRACK`: Occurs when a branch misprediction is signaled, indicating the loop has terminated. This transition also asserts a `flush` signal to clear the pipeline after the IF stages.



Implementation Details

- When we read from the bram we check for stalls (because our fsm stalls the ifid pipeline we check the idex pipeline). This ensures that if the pipeline needs to be stalled while we read from bram the instruction out of the BRAM must not be the next in order. That is why when we need to stall we point to a read address that always contains zero's and thus the instructiond fed into the CPU is a no-op.
- By checking for branch and jump instructions inside the loop and exiting if we find such an instruction, we ensure that the loop won't be terminated in any point other that the initial branch by misprediction.
- When we detect a compatible branch in the TRACK stage, we store the PC of the next instruction of the branch in a separate variable, so that we can feed it back to the CPU once a misprediction signal has been issued. This allows the CPU to continue its operation after the loop.

UOP Cache - BRAM

To implement the uop cache, we used one of Xilinx's BRAM (Block RAM) primitives. This BRAM stores instructions when the write enable signal is asserted.

Writing to the BRAM

Writing to the BRAM is more involved than reading. After consulting Xilinx documentation, we chose to operate the BRAM in Simple Dual-Port (SDP) mode. This mode provides separate ports for reading and writing and is the only mode that supports the 32-bit data width required for RISC-V instructions.

However, the BRAM ports do not natively support 32-bit transfers. Instead, data must be split across two 16-bit inputs. When writing through Port B, the lower 16 bits of the instruction are provided on the

DIADI port, and the upper 16 bits on the DIBDI port.

In addition to asserting the write enable (ENBWREN) signal, care must be taken with the byte-wise write enable signal WEBWE. This is a 4-bit signal that determines which of the four bytes in the 32-bit word are actually written. For example, if WEBWE = 4'b0011, only the lower 16 bits are written. To write the full 32-bit instruction, we use WEBWE = 4'b1111.

Reading from the BRAM

Reading is more straightforward. As with writing, the 32-bit instruction is split across two 16-bit output ports: DOADO and DOBDO. To initiate a read, only the read enable signal ENARDEN (1-bit) needs to be asserted.

BRAM Addressing

The BRAM uses a 14-bit address bus. However, in SDP mode with 32-bit wide access, only the upper 9 bits of the address are used to index memory. The lower 5 bits are ignored and effectively zero. That means each memory location is aligned to 32-bit word boundaries, and the effective address is formed by concatenating the upper 9 address bits with five zeros at the bottom. Since we increment the address by 8 and we have 9 bits of address, the maximum number of instructions that we are able to store in the BRAM is **64**.

Testing

Link to create RISC-V code in assembly click_here

Tests

We created 3 general tests.

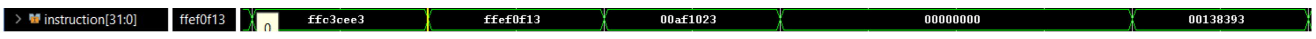
1. In the first one, we had 3 small loops (3-4) instructions
2. In the second one we had a big loop with the max amount of instructions (27) to check the upper edge case.
3. In the third one we had 1 instruction per loop to check the lower edge case.

Considerations

In our implementation we took in account some considerations to make sure the Stream Loop Detector works in the RISC-V code.

– Since there is no branch predictor, it is by default not taken. So each branch when taken has four instructions that are "wasted" (The 2 instructions after branch and 2 instructions for the flush of the pipeline). To avoid this we started reading from BRAM 4 instructions after the initial writing point. This ensures that we only take the instructions from within the loop. This may seem like a big downside of the whole mechanism, but 4 cycles compared to hundreds or thousands cycles that are going to be saved is nothing.

Fig. 1: Example for Branch NT



For example in the Fig. 1, we can see that after the branch instruction (ffc3cee3) we have 4 instructions before the first one of the loop (00138393). By skipping these 4 we ensure the correct functionality in our program.

– In BRAM 1 cycle is needed to change from write to read. So, we had to add a new state in the fsm to wait 1 cycle for the BRAM to be read to start reading.
– The state reset that used to exist lasted only one cycle and was similar to the WAIT state. This happened because we do not wipe the BRAM clean every time; we just rewrite it and so, we do not need to reset anything.
– To ensure we do not exit from the loop at an unpredictable moment through a random branch inside the loop, our LSD in the BUFFERING state if it detects a branch or jump opcode it goes back to the TRACK state. This also ensures no inner loops.

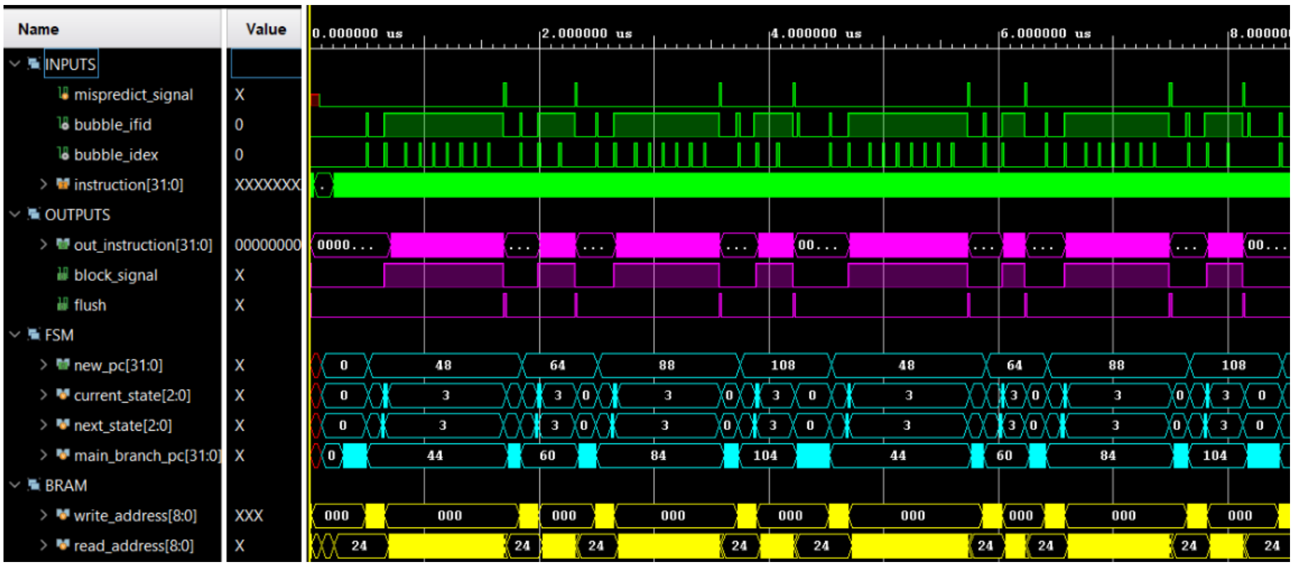
Challenges

– We had to study about the input and output ports of BRAM and find how they work to connect them correctly.
– In order to make the post-implementation simulations run, we had to add some extra sequential logic in the code because of some timing errors that were occuring.
– Because the BRAM takes one cycle to write, it was hard to synchronize it with the FSM. That is the reason why we added one extra stage in the FSM (WAIT stage).

Results

Here is a behavioral simulation of the Loop Stream Detector working in Vivado when integrated in the RISC-V architecture:

Fig. 2: Example for Working Loop Stream Detector


Name
Value
INPUTS
mispredict_signal
bubble_ifid
bubble_idex
instruction[31:0]
OUTPUTS
out_instruction[31:0]
block_signal
flush
FSM
new_pc[31:0]
current_state[2:0]
next_state[2:0]
main_branch_pc[31:0]
BRAM
write_address[8:0]
read_address[8:0]
0.000000 us
2.000000 us
4.000000 us
6.000000 us
8.00000