Final Project
Jonathan Gusler
04/28/2019

Overview
The purpose of the final project was to introduce the concept and implementation of cache memory, specifically for use in instruction memory. This instruction memory cache works alongside lower level memory to create a two-level system.

Background
This project built off the processor from the previous labs. The memory that had been there previously was kept but modified to behave as the lower level memory with a capacity of 256 32-bit blocks. The cache implemented has a capacity of only 8 blocks, each 36 bits wide.

Why is a cache necessary? Ideally, a machine would massive amount of memory capacity and be able to fetch information from that memory as fast as the computer can ask for it. In reality, the faster the memory, the more expensive that memory is. Memory that can give information as fast as the computer asks for it is very expensive, too expensive to store everything. Computer engineers needed to find a way to get as close to possible to this ideal state without breaking the bank. A cache is the ideal speed but is very expensive per unit of storage. Therein lies the constraint: memory capacity. Because the cache can't hold the whole program, the question is how to get the cache to work in tandem with lower levels of storage that hold the rest of the program.

Discussion
As stated before, the cache can only hold 8 instructions at a time and works with a slower memory unit that can hold 256 instructions. How do they work together with the processor? When the processor asks for an instruction, the cache checks if that instruction is currently in the cache. If it is, known as a hit, then that instruction is output into the processor; if not, known as a miss, the cache requests that instruction from memory and waits for it to get to the cache. Once the instruction is in cache, the next time the processor asks for that instruction, which it is constantly doing until it gets it, the cache can output that instruction. While the cache is getting an instruction from memory, the cache outputs NOP instructions. The time it takes to fetch an instruction is two clock cycles. So, on a miss, two NOPs are output before the requested instruction. Figure 1 shows a block diagram of the instruction memory and selection setup.

The cache is 36 bits wide. The lower 32 are the instruction, three for the tag, and one for the valid bit, which is the MSB. The valid bit and tag are concatenated to the instruction as the instruction comes from memory. The index is three bits but is not part of the cache's 36 bits.

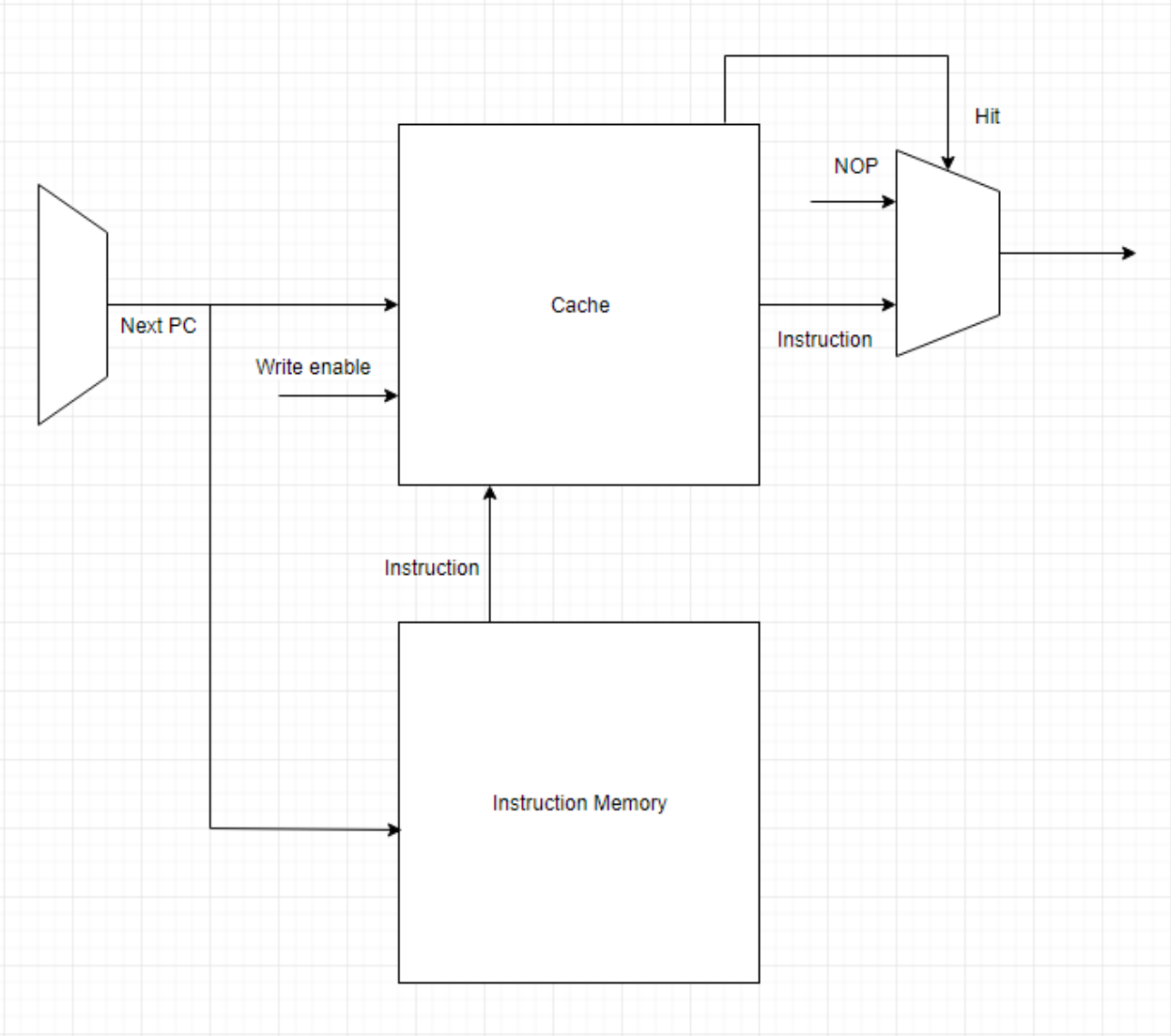

Figure 1: Block diagram of instruction memory and selection setup.

The cache/memory state machine has two states, B and C. Originally there was a state A with little content that was removed for simplification; the other two states weren't renamed because naming didn't affect the program. Behavior:

On reset, go to state B.
While in state B, accept instruction requests and check against cache contents. If the instruction is there and valid, output the instruction to the processor and set the hit signal high to select that instruction. If the instruction is not there, set hit to low to output NOP, don't increment the program counter, and move to state C.

When in state C, output a NOP, write the instruction to cache by raising the write enable signal, and then go back to state B to start the process over.

This logic is combined with previously created logic to stall the program, outputting NOPs and reverting the program counter as necessary.

Because the processor doesn't support branching or jumping, having a cache is merely an exercise in implementing one, as the compatible programs don't go back to previous instructions, thus never taking advantage of having something already in the cache. Because of this, every instruction must be fetched from memory. Given this, the hit rate is only 33%: two clock cycles to fetch, one miss, and one to output, one hit. Figure 2 shows a waveform presenting the results of a test program. This program contains a load word instruction at the beginning that creates a load-use stall scenario, stalling the program. The next instructions are R-type instructions that demonstrate the simple use case of the cache. As can be seen, not including load-use stalling, every instruction takes three clock cycles to be output, because each must be fetched from memory. Relevant signals shown in the waveform:

Mem_Addr_out	Cache index
PC	Program counter
Instruction Memory	Memory output
Instruction Cache	Cache input, memory output combined with tag and valid bit
Instruction out	Instruction given to processor
Hit	Hit, and conversely miss, signal
State_out	State machine for memory and cache system, low for state B, high for state C

Table 1: Relevant waveform signals.

Conclusion

The cache works perfectly with the capabilities and limitations of the processor, meaning it handles everything but branching and jumping because the processor is not capable of those. The Appendix contains the code for the memory, cache, and state machine.

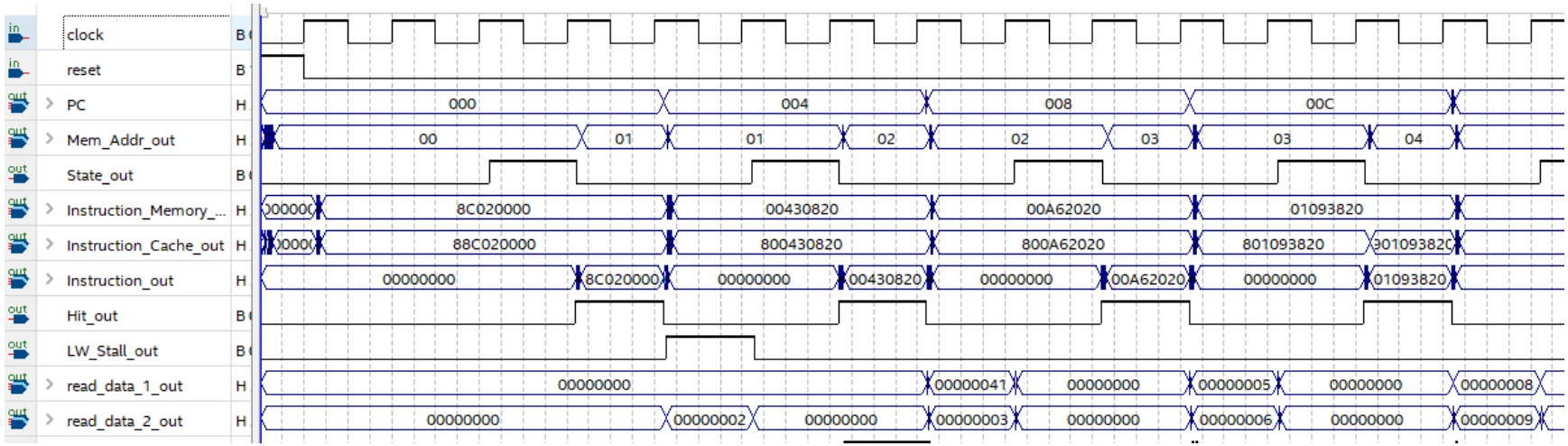

Figure 2: Waveform of test program.

Appendix

```
                    --ROM for Instruction Memory, this will feed into the cache
inst_memory: altsyncram

    GENERIC MAP (
        operation_mode => "ROM",
        width_a => 32,                              --data input width
        widthad_a => 8,                            --address input width, 256 blocks
        lpm_type => "altsyncram",
        outdata_reg_a => "UNREGISTERED",
        init_file => "Lab09program.MIF",
        intended_device_family => "Cyclone"
    )
    PORT MAP (
        clock0     => clock,                           --clock input
        address_a   => Mem_Addr,                   --address input
        q_a             => Instruction_Memory); --data output, goes to cache


                    -- Cache memory unit, implemeted using

cache: altsyncram
    GENERIC MAP  (
        operation_mode => "SINGLE_PORT",
        width_a => 36,
        widthad_a => 3,
        lpm_type => "altsyncram",
        outdata_reg_a => "UNREGISTERED",
        intended_device_family => "Cyclone"
    )
    PORT MAP (
        wren_a => cache_write,
        clock0 => write_clock,
        address_a => Index,
        data_a => Instruction_Cache, --Input data
        q_a => Instruction_IFID );    --Output data
-- Load memory address register with write clock
        write_clock <= NOT clock;

        Instruction_Cache <= '1' & Mem_Addr(4 DOWNTO 2) & Instruction_Memory(31 DOWNTO 0); -- To make the data 36 bits wide
        Instruction_Cache_out <= Instruction_Cache;
        Instruction_Memory_out <= Instruction_Memory;

                    -- Calculates Index for each memory location
    PROCESS(Mem_Addr, Index) IS
        BEGIN
            IF (Mem_Addr = X"00" OR Mem_Addr = X"08" OR Mem_Addr = X"10" OR Mem_Addr = X"18" OR Mem_Addr = X"20") THEN
                Index <= "000";
            ELSIF (Mem_Addr = X"01" OR Mem_Addr = X"09" OR Mem_Addr = X"11" OR Mem_Addr = X"19" OR Mem_Addr = X"21") THEN
                Index <= "001";
            ELSIF (Mem_Addr = X"02" OR Mem_Addr = X"0A" OR Mem_Addr = X"12" OR Mem_Addr = X"1A" OR Mem_Addr = X"22") THEN
                Index <= "010";
            ELSIF (Mem_Addr = X"03" OR Mem_Addr = X"0B" OR Mem_Addr = X"13" OR Mem_Addr = X"1B" OR Mem_Addr = X"23") THEN
                Index <= "011";
            ELSIF (Mem_Addr = X"04" OR Mem_Addr = X"0C" OR Mem_Addr = X"14" OR Mem_Addr = X"1C" OR Mem_Addr = X"24") THEN
                Index <= "100";
            ELSIF (Mem_Addr = X"05" OR Mem_Addr = X"0D" OR Mem_Addr = X"15" OR Mem_Addr = X"1D" OR Mem_Addr = X"25") THEN
                Index <= "101";
            ELSIF (Mem_Addr = X"06" OR Mem_Addr = X"0E" OR Mem_Addr = X"16" OR Mem_Addr = X"1E" OR Mem_Addr = X"26") THEN
                Index <= "110";
            ELSE --(Mem_Addr = X"07" OR Mem_Addr = X"0F" OR Mem_Addr = X"17" OR Mem_Addr = X"1F" OR Mem_Addr = X"27") THEN
                Index <= "111";
            END IF;
    END PROCESS;
```

```
Hit_out <= Hit;
Mem_Addr_out <= Mem_Addr;
                -- Cache state machine
PROCESS (clock, reset) IS
  BEGIN
     IF (reset = '1') THEN
        State <= B;

     ELSIF ( clock'EVENT ) AND ( clock = '1' ) THEN

            CASE State IS

                WHEN B =>
                        State_out <= '0';
                        cache_write <= '0';
                -- if tag in cache = tag for requested address and valid = true
                    IF ( Instruction_IFID(34 DOWNTO 32) = Mem_Addr(7 DOWNTO 5) AND Instruction_IFID(35) = '1') THEN
                        Hit <=  '1';

                        -- Last edit
                        IF (LW_Stall = '1' OR LW_Branch = '1' OR LW_Branch_1 = '1') THEN
                            Instruction <= X"00000000";
                        ELSE
                            Instruction <= Instruction_IFID(31 DOWNTO 0);
                        END IF;

                    ELSE
                        Hit <= '0';
                        Instruction <= X"00000000";
                        State <= C;
                    END IF;


                WHEN C =>
                        State_out <= '1';
                        -- Progess to B since memory is constantly spitting stuff out anyways
                        Hit <= '0'; -- Continue to stall
                        cache_write <= '1';
                        State <= B;
                        --Keep here for safety
                        Instruction <= X"00000000";


                WHEN others =>
                        State <= B;
            END CASE;
    END IF;
END PROCESS;
```