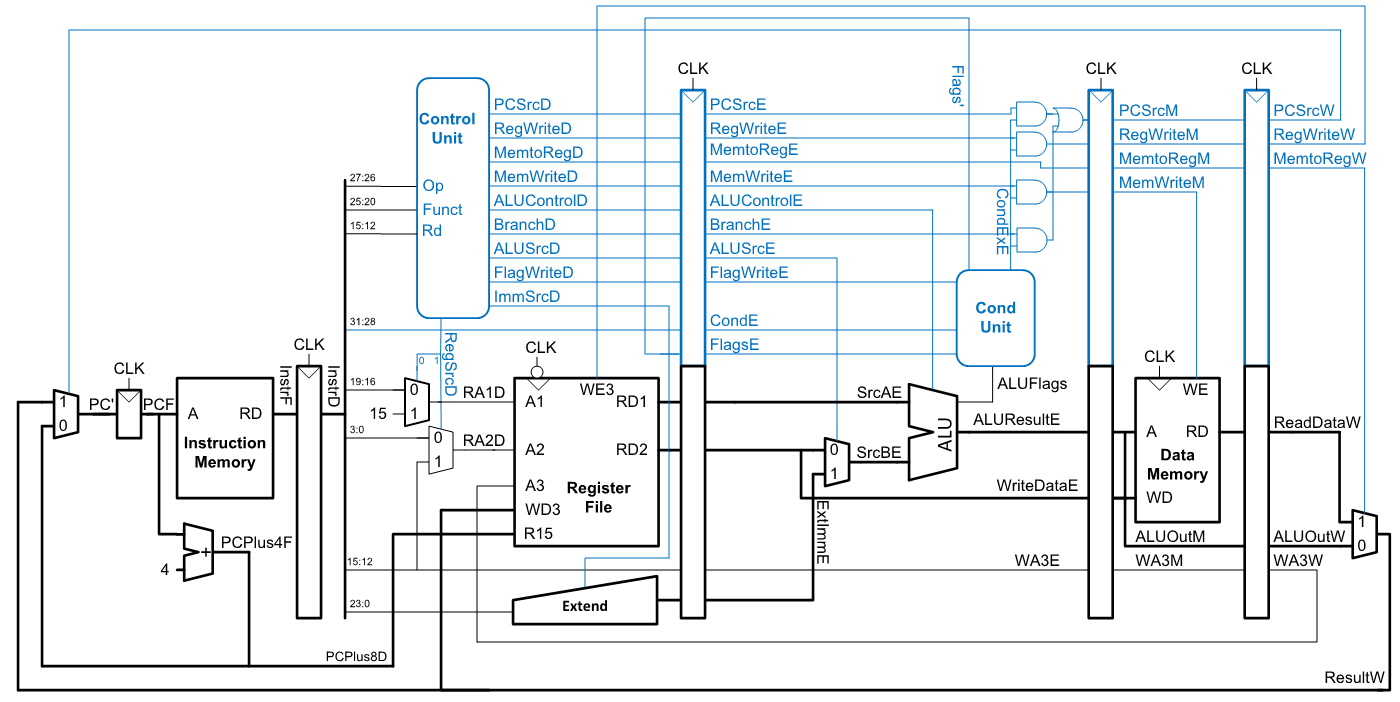
**Computer Architecture Lab 3**

Written by: Matthew Whitlock (11744571) and Nate Morton (11784201)

**Introduction**

In this lab, our goal was to write an instruction set architecture, using Verilog Hardware Descriptive Language (HDL) modeling language and implement it on the Elvis III board, but this time we were to implement it using a pipelined design instead of a single cycle layout. The purpose of this lab was for us to understand how the components of the hardware (such as the ALU, mux devices, and more) work together to make up a pipelined ARM Instruction Set Architecture. For this lab the following instructions needed to be encoded into our program: ADC, ~~ADD~~, ~~AND~~, ASR, ~~B~~, BIC, CMN, CMP, EOR, ~~LDR~~, LSL, LSR, MOV, MVN, ~~ORR~~, ROR, SBC, ~~STR~~, ~~SUB~~, TEQ, and TST (the ones crossed out were already encoded for us before hand). In order to make sure our code was correct we simulated it on ModelSim making sure each of our instructions worked with the given input files we gave it. Once we had written all our instruction, and confirmed they worked in simulation, we then started working and making debugging changes in the code to where we could get it to work on the Elvis boards. We then counted the cycles for each test on the board and compared them with our simulated tests that had perfect memory. Since this lab was very similar to the previous lab, with the main difference being the pipelined architecture vs the single-cycle, it ended up not being too difficult and was less of a hassle to get working.

**Baseline Design**



The Instruction Set Architecture is ran as a pipelined microarchitecture where each cycle can be doing multiple instructions and each instruction can have more than one cycle. For the most part we keep the baseline design of the original proposed diagram from the lecture slides with a few changes. But just like the diagram, the architectural state of the machine (except the memory) is stored in registers, there is a global wire called the clock (clk) that synchronizes the events within each register, and registers capture the snapshot of the values on their inputs and the registers hold the capture values and feed them to their outputs. The outputs of each register are fed into a combinational circuit consisting of logic gates for control, afterwards the result of the logic gates are fed back as input to the register(s) that only change on the positive edge of the clock. Following the rising edge on the clock the registers capture the values of their inputs, and at each rising edge the instructions are initiated. Following the previous rising edge, the next one should be where the values within the registers should be updated, such that, the instruction can be considered to have correctly executed. Finally all of these aspects of are design are then implemented within the overall design of a Princeton Architecture.

**Design**

The design we have chosen is only slightly modified from the initial design, with most of the modifications being small changes to the control and ALU to allow for more instructions. More notable changes include adding an enable input to the PC, to pause execution during memory operations, adding a barrel shifter, and including a mux to allow for setting register values to the output of the barrel shifter without feeding through the ALU.

The barrel shifter is a vital part of the redesign, allowing for move and shifting operations as both standalone operations and as instruction-level-parallelism. The barrel shifter is inserted before the ALU, to allow for the instruction-level-parallelism that ARMv8 tilizes. Control input is sent to the shifter to specify left shifts, logical and arithmetic right shifts, and rotate rights. While the shifts are accomplished using built-in instructions for verilog (namely <<, >>, and >>>), the rotation required a little more thought. We chose a design which is far from space-friendly, but is quick and simple to implement and conceptualize. Rather than actually rotating the bits, we create a logic which has two copies of the input side-by-side. Then, we do a standard right shift on the new 64-bit value - by pulling from the least-significant portion of this, we get the equivalent of a rotated value. After performing the rotation, flags (c, v, and z) are set based the result - it is then up to the control logic to choose between the ALU flags and the shifter flags.

**Testing strategy**

As a first step to testing, we ran the test-code from our ARMv8 simulator in ModelSim. This gave us a baseline of if simple instructions functioned as expected in the general-use case. However, it is nigh impossible to test every single possible input, so we also used the benchmark applications as tests. Even though the test application from our previous simulator test file succeeded, we ran into issues during the benchmark execution.

Namely, we discovered that the barrel shifter was receiving bits from the middle of the immediate value during a branch operation. Our initial testing only did small, forward branches, and it required either large branches or negative branches for the shift amount to be non-zero. Further, we discovered that we had not implemented flags for shifting operations, and were only updating flags for ALU operations. After updating these two issues, we re-tested each test file and ran successfully. In general, our mindset with testing is that it will be impossible to truly verify that the CPU works perfectly for all instructions, so we did our best during the design phase and ensured that each instruction at least works in a general test case.

After ensuring that the CPU worked in ModelSim, we then used Vivado to export the CPU to an FPGA to test for correctness in the implementation.

**Evaluation**

We evaluated the code using the provided in the benchmarks section of the zipped files. Below are our results in ModelSim with a perfect memory simulation, and in the FPGA with a DDR3 memory controller.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Benchmark | Sim (single) | DDR3 (single) | DDR3 (single, reference) | Sim (pipeline) | DDR3 (pipeline) |
| Memfile | 21 cycles | 133 cycles | 129 cycles | 39 cycles | 152 cycles |
| Fib | 142 cycles | 214 cycles | 215 cycles | 286 cycles | 362 cycles |
| Mult | 75 cycles | 191 cycles | 178 cycles | 105 cycles | 208 cycles |

You will notice that the pipelined CPU takes more cycles, generally. This makes sense, as the pipelined CPU has to handle data hazards by stalling occasionally. The catch to this is that (in theory) the pipelined CPU can run with a faster frequency. Looking at the simulated runs, the pipelined CPU would need to have 1.4-2.01 times the frequency of the single-cycle implementation in order to take the same amount of time. The variance in what frequency speedup is needed to break even is a good indicator of the pitfalls of comparing two CPUs by frequency or number of cycles alone.