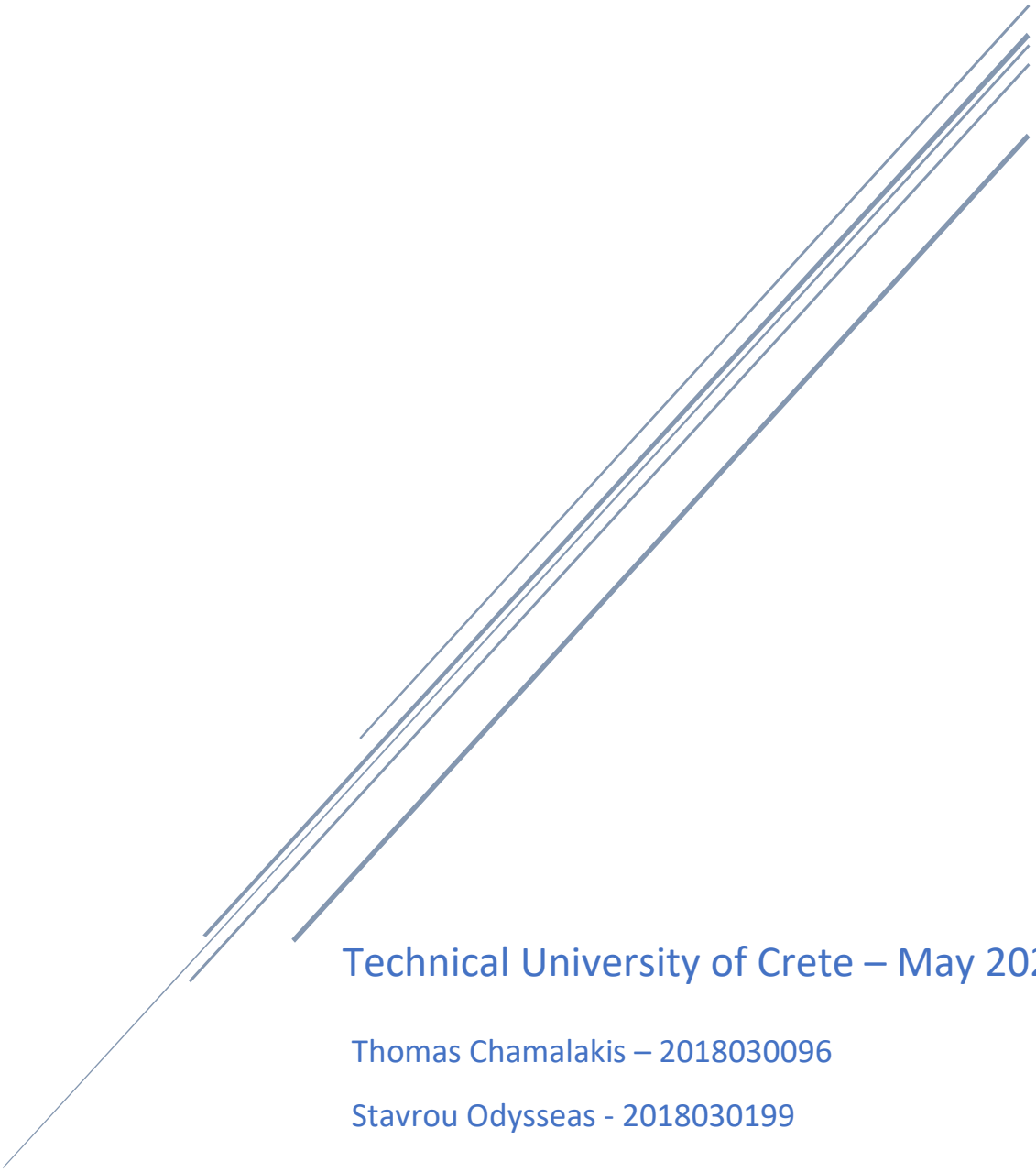
ACE312 PROJECT 2

Phase 4: Multi-Cycle Processor
Phase 5: Pipelining CPU



Technical University of Crete – May 2020

Thomas Chamalakis – 2018030096

Stavrou Odysseas - 2018030199

INTRODUCTION:

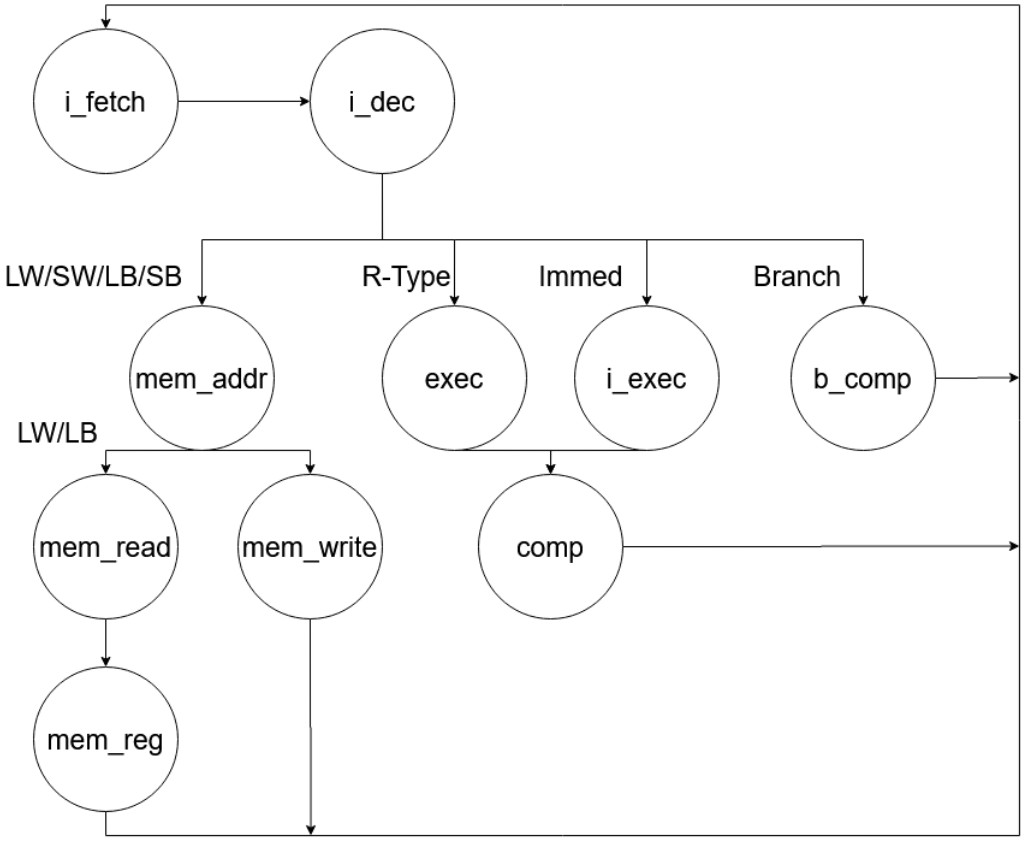
For phases 4 and 5 where were tasked to modify our Single Cycle Processor, done in Project 1 of this course, and transforming it to a multicycle processor (Phase 4) and then into a pipelining Processor (Phase 5).

PHASE 4:

The only fundamental difference was the modification of the conventional Control module into an FSM-based module responsible keeping in synch the operations of the Datapath. The control signals needed for controlling each stage are a function of the current FSM state. As soon as they are generated, they are handed to the proper stages accordingly. The other minor structural difference was to the Datapath itself, the addition of extra REGISTER modules in-between the pre-existing STAGES, holding the information for the next stage. Also, additions have been made in the execution stage, where now we use the ALU to compute the branch values of the PC counter rather that relying on a different stage altogether.

The advantage of this method is the reduction of the clock period, which is no longer decided by the critical path of the Datapath but rather the longest critical path of one particular stage.

With the modification of the Datapath in place we created the fsm-based control unit based on this diagram below.



In each stage we are responsible for creating the proper signals that control the flow of data. In Example: At **mem_reg** stage we need to set the proper register enable signal to write the value we read from memory. Some signals like the one responsible for defining the Immediate extension method is not a function of the current state but rather of only the instruction. Same goes for the signal selecting the 2nd register (**RF_B_Sel**)

Data flow is pretty much the same, the only difference is that the result in each register placed in-between the stages are appearing 1 cycle later because they write their input on the rising edge of the clock signal. Note that, since the altered PC is computed using the ALU at this point, it's up to the FSM to switch the selection from the normal PC to the branched one (the +4 PC is always calculated, at the i_dec stage and stored in ALU_OUT).

Please see "PROC_MC_SCHEM.pdf" for the block diagram.

PHASE 5:

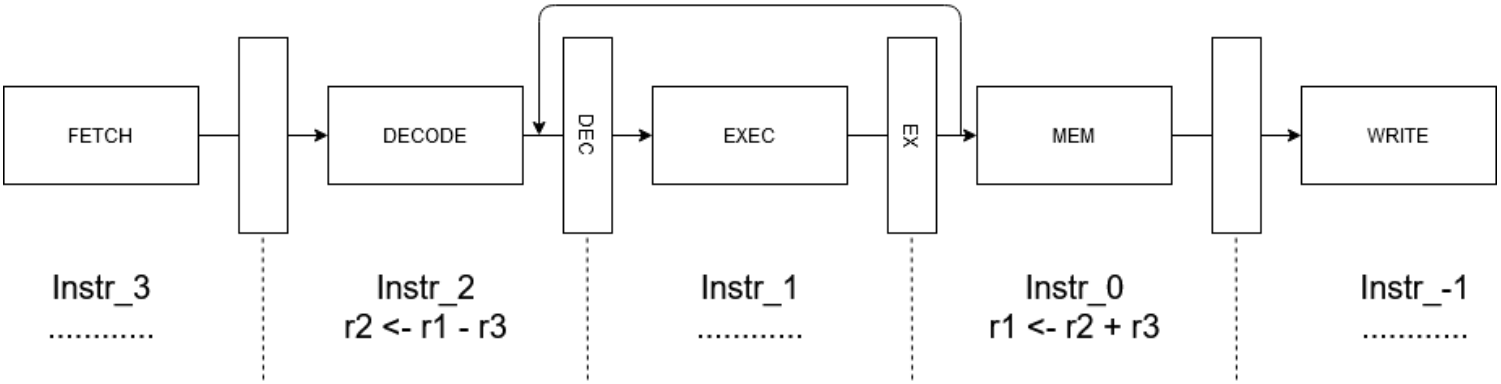
For Phase 5 we firstly had to understand how the pipeline principle work so we could apply it in the approaching design.

The changes in the Datapath were vast, almost redesigned from the ground up. For the pipeline model to work we did not have plain registers that hold a single 32-bit value, but entire modules that completely isolate the entire stage from the rest of the system, that means anything the next stage needs to perform an operation, flags, operands, decoded registers etc. So, the next stage in line, can function independently, regardless of the actions of the rest of the modules. Approaching this was quite the feat, we had to redesign the whole Datapath twice before getting things straight, due to the many modules and even more signals connecting them together. See "PROC_PIPELINE_SCHEM.pdf" for the complete block diagram of the design.

The control system used was a modified version from the Single Cycle Phase. It no longer outputs 1 signal for each flag, but combines signals into vectors depending on their usage throughout the Datapath. And now also, it writes, conditionally, 0 on these signals to achieve stalling and flushing the pipe when necessary (more on that below). These flags are stored in their proper modules and are forwarded along with their corresponding data so the stage executes the proper function on the proper time, with the correct data of course.

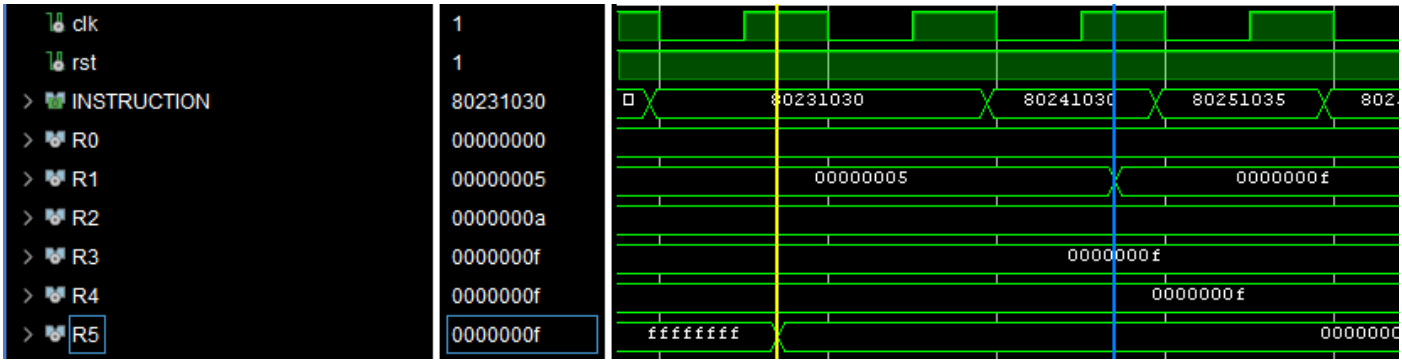
Forwarding and Hazards:

Since we are using the pipeline methodology, that means many instructions are present in different stages of the Datapath. But what happens when an instruction needs data from the instruction currently executing, or does a reference to a register in its way to be overwritten? That is called a "Data Hazard" and we can solve this by injecting specific information at strategic spots inside the Datapath.



In Example: At any given time Instr_0 is finished executing and Instr_2 is currently being decoded. We can see the r1 register presents as an operant and has not yet been produced. A simple solution would be to inject the result back into the corresponding operand INPUT slot of the DEC stage “overwriting” the “false” value of the register and producing a correct result. We can also inject the information just after the DEC stage if Instr_2 was in the EXEC stage. The spot of injection and operand of injection are decided based on how far apart are the two Instructions from each other and which operands match each other.

Data Hazards can also appear when we try to access the memory and on the next instruction reference that register as an operand. In that case, because we don’t have the information anywhere within our Datapath we have to stall the pipe and wait for the instruction to move far enough ahead and be at a point where we can inject the data read elsewhere. When the pipe is stalled, we forbit the IFSTAGE to fetch new instructions and write them out to the module (overwriting the stalled Instruction). The CONTROL also won’t produce any useful flags so we are left with a junk Instruction similar to the **“nop”** instruction, that will have no effect until the stall is finished and can resume its path with the proper flags.



Observing the changes of R5 & R1 we can see that the yellow write and the blue write to the registers are 2 clock cycles apart where they should be back to back, there is a blank cycle which means the pipe has stalled to align the data so the data has enough time to be read from the memory and then injected to the proper spot.

This injection and stalling the pipe is handled by two modules: FWD_UNIT and HAZARD_DETECT respectively. The forwarding unit is responsible of detecting which registers (and operand position) match when instructions (that conflict with each

other) reach the specific stages and then inject the proper data into the correct locations. The hazard detection unit stalls the pipe when it detects two instructions that would need a delay between them to execute correctly.

Note that this processor does not support handling Control Hazards, ergo, no support for Branch Instructions.

Results and Conclusion:

- Using different stages to process the instruction minimize the clock cycle. The more optimized each stage is, then the faster the processor.
- Pipelining the instructions fully utilize, hardware-wise (no idle components), the processor, maximising speed and efficiency.
- We understood in-depth how the processor executes instructions and how it manages the memory.
- A processor is a really simple component with very specific functionality, however optimizing it to increase its throughput and performance can be a daunting task.

Used Software:

- Xilinx Vivado
- CHARIS Assembler - source

Full Source Available: GitHub