ESE 345 2018 Project

Pipelined multimedia unit design with the VHDL/Verilog hardware description language

Presentation Slot: 9:30 a.m 12/05/2018

Students: Ayman Azad and Manuel Castillo Arenas

Teacher Assistant: Ryan Thielke

Instructor: Mikhail Dorojevets

The goal for this project is to build a triple-stage pipelined multimedia unit with a reduced set of multimedia instructions.

Our design is a union of several modules that are operating simultaneously. This design is composed by the instruction buffer that loads up instructions from a text file and writes them into to the register file.

The start of the design was the alu since it was the most important and difficult step of the design process. Initially we designed it as a group of ALUs which had multiple stages, the first, second, and third stage would have 8, 4, 2, and 1 alus respectively. Each stage in the ALU would have inputs in its ALU set so that 2 128bit values could be loaded up to each. The design we came up with would take the output of each stage and input it into the next stage. Each stage would take inputs from the register values based on a multiplexor, each stage would have its outputs taken and input into yet another multiplexor with a control signal based on the instruction. The result would be written back to the register file. This was designed for the load function and the r3 type instructions but was scrapped later due to the complexity needed to use this r4 type needing shifting and comparing. Overall this would be a much more efficient way of designing the ALU which I wish we had the chance to implement. Since just doing it in a behavioral style would be much simpler and require less control bits, it was done in this fashion instead.

The first in this design was to create a compiler in java which would take assembly type instructions from a text file in the VHDL folder and write the machine code in the text file also in the same folder. This made it very user friendly and I will explain later how it used in the processor. This code is in the Index presented in this document.

The VHDL project starts with the Load\_imm block which takes the li halfword select and the 16 bit immediate value to be loaded. It also takes the rd register content in order to write to it. The new output to be written to the register file is output. The next part of the ALU is the r4\_alu which takes the appropriate 3 register datas and a performs the required function based on the ALUop control input. The same is done with the R3 type instructions in the r3\_alu file. These the mentioned blocks will always perform execution based on the values of the pipeline register. The output of this execution stage is chosen through multiplexors which function based on the control signals. The main ALU will consist of the R3 and R4 blocks since they are both functions which require a similar input field formats. The overall execution stage was designed as a structural design and only needed inputs from the id\_ex pipeline register to perform its task.

The next major component is the register file, this block was designed to asynchronously read the registers which were dictated by the instruction fields. These values were output always based on the contents of the registers. If we were to read a register which was just written to then usually vhdl would output the values which were just written. This unit performs its task on every clock cycle. It also has a reset input which synchronously clears all registers. This is done at the beginning of the testbench. The register based on the RW input is only written to the register on a positive edge and if the regWrite signal is asserted.

The following control unit is designed to be in the ID stage since that is when it is needed. This generates the control signals regSrc (1bit), rType (1bit), regWrite (1bit), rdImm (1bit) and aluOp (4bit). regSrc is a signal to determine if we were to write the load\_imm value or the ALU output to the register file. The rType signal is the signal which determines if the instruction is an r3 or r4 type instruction. This signal is only input to the ALU block so it is a don’t care in the load\_imm block. RegWrite is a signal that is asserted when ever we wish to write to a register. This is always asserted except on one occasion being a the nop instruction. This means that we would not write to the register file. This Is also used in the forwarding process later explained. The alu\_op is the function to be performed in the alu stage. It only requires 4 bits since the r3 instructions take 4 bits and the r4 take 2. The rdImm control bit is used for a particular r3 instruction only. It is the shlhi instruction which takes the r4 instruction field as a immediate value. This is always used in the EX block to choose what will be input into the rd input of the ALU, it the input of a multiplexor which chooses between the two. This can be seen in the block diagram.

The next step in the design process is the use of the pipelines to make it a multicycle processor. This is done by using each main functional units only once in each stage. The IF stage will have a PC register which increments every clock cycle. This is done in the Instruction buffer block which reads the instructions based off a text file at the beginning of any test and then outputs the instruction based on the value of the pc. This is the main component of the IF stage and only requires a clock and reset input. The resulting instruction is placed in the IF\_ID pipeline register for the next stages to use. The next stage is the ID stage of the processor. This has 3 major components: the control unit, the register file, and the forwarding unit (shown later). This stage actually has three values determined the EX stage but they are only used to write to the register file. The instruction is read from the if\_id register and used to determine the control signals, read the required registers, and is placed in the id\_ex register for the next stage to use along with the 4 registers data values and the control signals. The forwarding unit is used in this stage to determine what values to write to the id\_ex register since the execution stage may be determining the value of any of the 4 registers to be read in this stage. Thus, a forwarding unit is placed in in between the register file and the pipeline register. This takes the output of the next stage and replaces any of the registers data values based of the following criteria:

* The regWrite signal in the execution stage is asserted, basically is not a nop.
* The register numbers being read are not the same as the register number being written to in the EX stage. This in done by taking the instruction fields of the pipeline resisters if\_if and id\_ex, the regWrite in the ex stage, the result of the ex stage, the register values read from the id stage.

If the conditions are met then the register values written to the next pipeline registers are not the register values written from the register file but the result of the next stage. This is done in parallel so even if multiple register values need to be replaced like where 2 of the values come from the same register, they output values will both be the result input.

Now that that we have the design of the processor complete with pipelines and forwarding. We can put all of it together in a structural design. I wanted to implement this in a way so the results are easily tracked and running the processor in a test will be very simple. The way I designed the testing to be done is that who ever wishes to run an assembly style instruction code can use the provided code without any knowledge of how the code works or how to code at all. Here are the following steps on how to run any program a user creates. The assembly.txt file is where the user will input instructions in a easy to read format, for example the instruction (and 4 5 0) will logical and the values in register 4 and 5 and place the result in register 0. Then the user can run the java program to compile the machine code for the processor to use. The instructions are written to machine.txt for the vhdl code to use. The next step is to open the vhdl program and use the top level as cpu\_tb which is a test bench for the cpu. Notice how simple and minimal this testbench is, it only requires a clock and reset since those are the inputs of the cpu block. If you were to run a simulation you can easily see the results in two places. All steps the processor takes and all register contents are shown in a wave form. What if the user doesn’t know how to read from the wave form or it is simply too much work? The solution is to write the simplified results in a text file called log.txt. This is done by creating a package that contains signals to be written which are the pc count, register values, and the instructions in each stage of the processor. These values are read on every positive clock trigger and written to the text file in the cpu test bench. Thus, to see the instruction as it goes through the multiple stages of the processor, the user has to perform the three simple steps:

* Write assembly language instuctions to assembly.txt, you don’t even have to write 32 instructions since the rest are filled with nop instructions.
* Compile the program with the provided java program.
* Simulate the processor using these instructions on vhdl, and view the results in the log.txt file.

To conclude and show this ease of use of the processor. We came up with a program in assembly language which uses all required instructions and simulated them. These files and the results are provided. As you can see, each are tested and you can even follow the pipeline as it flows through the steps in the processor.