Pipelined Multimedia Unit Design with VHDL Hardware Description Language

ESE 345

Fall 2023

Professor Mikhail Dorojevets

Written by:

Christopher Nielsen (Class of 2025)

Introduction

ESE 345, “Computer Architecture” is a course given at stony brook involves the learning and implementation surrounding processor design, memory structure, and further develops the concepts of systems design. As a final step in the class, it involves the creation of a Pipelined Multimedia Unit in our choice of hardware description language, as well as to design an assembler program specified for our hardware design. We were then required to test, validate, and present to a TA. The presentation required a showing of our assembler converting a mips style assembly program written by our TA, converting it to the binary machine code, execution of the code, and showing the TA the memory addresses where the results were stored. We also needed to explain the implementation of all 4 stages of our pipeline.

The project assignment (Found in the Repo titled: “Original Project Description”) lays out all the requirements for us to have completed, in summary the list of requirements were:
1. Multimedia ALU: design an alu to receive 3 different genres of opcode and complete a multitude of operations.
2. Register File: design a Register system of 128 bits of data containing 32 general use registers.
3. Instruction Buffer: design an instruction buffer to hold and queue a maximum of 64 instructions.
4. Forwarding Unit: Design hardware to detect when data needs to be forwarded for any operations.
5. Four-Stage Pipelined Multimedia Unit: Design an all in one processor with a 4 stage pipelined architecture.
6. Testbench: Implement a testbench to interface with the processor in order to load the program, and assert/compare expected values to the values stored in memory after a program runs on it.
7. Assembler: Code a program to convert MIPS style text file commands to a binary file to be ran on the processor and automatically loaded by the testbench.
8. Results File: Have the test bench write the state of the processor for each cycle it is run for any given program.

The final structure with all units required and the data path is shown below in Figure 1.

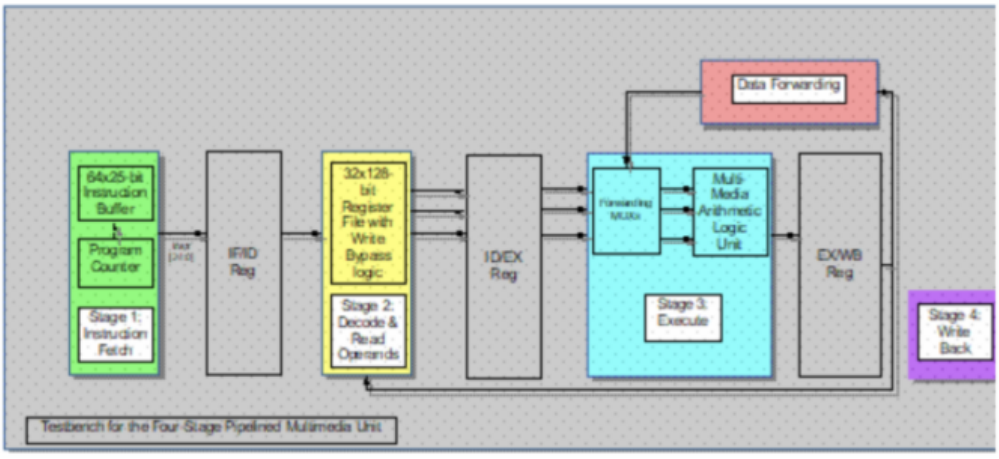

Figure 1

Our Pipelined Multimedia

Unit (PMU) was required to work with 3 different instruction types: LI, R3, and R4. These Instruction types are gone into further detail in the Original Project Description file, but let's review them.

Every instruction is 24 bits and there are 3 types of instructions.

1. I-type instructions are shown in ***Figure 2.1***. I-type signified by bit 23 being set to "1", then bit 22 to 21 signifies the byte of rd the 16-bit immediate value will be placed into. Bits 20 to 5 are the 16-bit immediate value that will be loaded into rd and then bits 4 to 0 specify the rd register.

2. The R3-type instructions are shown in ***Figure 2.2***, signified by bits 23 to 22 which have to be "00". Bits 21 to 15 signify the opcode which tells the Multimedia Unit the actual instruction to be executed. Bits 14 to 10, 9 to 5, and 4 to 0 represent the rs2, rs1, and rd registers respectively. Here are the defined instructions:

xxxx1101	ROTW: ***rotate bits in word*** : the contents of each 32-bit field in register ***rs1*** are rotated to the right according to the value of the 5 least significant bits of the corresponding 32-bit field in register ***rs2***. The results are placed in register ***rd***. Bits rotated out of the right end of each word are rotated in on the left end of the same 32-bit word field. (Comments: 4 separate 32-bit word values in each 128-bit register)
xxxx1110	SFWU: ***subtract from word unsigned***: packed 32-bit word unsigned subtract of the contentsof ***rs1*** from ***rs2*** (rd = rs2 - rs1). (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx1111	SFHS: ***subtract from halfword saturated***: packed 16-bit halfword signed subtraction **with saturation** of the contents of ***rs1*** from ***rs2*** (rd = rs2 - rs1). (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0011	CNTH: ***count 1s in halfword*** count 1s in each packed 16-bit halfword of the contents of register ***rs1***. The results are placed into corresponding slots in register ***rd***. (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0100	AHS: ***add halfword saturated***: packed 16-bit halfword signed addition **with saturation** of the contents of registers ***rs1*** and ***rs2*** . (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0101	OR: ***bitwise logical or*** of the contents of registers ***rs1*** and ***rs2***
xxxx0110	BCW: ***broadcast word***: broadcast the rightmost 32-bit word of register ***rs1*** to each of the four 32-bit words of register ***rd***
xxxx0111	MAXWS: ***max signed word***: for each of the four 32-bit word slots, place the maximum signed value between ***rs1*** and ***rs2*** in register ***rd***. (Comments: 4 separate 32-bit values in each128-bit register)
xxx01000	MINWS: ***min signed word***: for each of the four 32-bit word slots, place the minimum signed value between ***rs1*** and ***rs2*** in register ***rd*** . (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx1001	MLHU: ***multiply low unsigned***: the 16 rightmost bits of each of the four 32-bit slots in register ***rs1*** are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register ***rs2***, treating both operands as unsigned. The four 32-bit products are placed into the corresponding slots of register ***rd*** . (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx1010	MLHSS: ***multiply by sign saturated***: each of the eight signed 16-bit halfword values in register ***rs1*** is multiplied by the sign of the corresponding signed 16-bit halfword value in register ***rs2*** **with saturation**, and the result placed in register ***rd***. If a value in a 16-bit register ***rs2*** field is zero, the corresponding 16-bit field in ***rd*** will also be zero. (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx1011	AND: ***bitwise logical and*** of the contents of registers ***rs1*** and ***rs2***
xxxx1100	INVB: ***invert (flip) bits of the contents of register rs1.*** The result is placed in register ***rd***

3. The R4-type instructions are shown in ***Figure 2.3***,signified by bits 23 to 22, which have to be “01”. Bits 21 to 20 represent the MA/MS/I/h, which signifies which instruction to execute. Bits 19 to 15, 14 to 10, 9 to 5, and 4 to 0 represent the rs3, rs2, rs1, and rd registers respectively. Here are the defined instructions:

LI/SA/HL [22:20]	Description of Instruction Code
000	**Signed Integer Multiply-Add Low with Saturation**: Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then add 32-bit products to 32-bit fields of register **rs1**, and save result in register **rd**
001	**Signed Integer Multiply-Add High with Saturation**: Multiply high 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then add 32-bit products to 32-bit fields of register **rs1**, and save result in register **rd**
010	**Signed Integer Multiply-Subtract Low with Saturation**: Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then subtract 32-bit products from 32-bit fields of register **rs1**, and save result in register **rd**
011	**Signed Integer Multiply-Subtract High with Saturation**: Multiply high 16-bit- fields of each 32-bit field of registers **rs3** and **rs2**, then subtract 32-bit products from 32-bit fields of register **rs1**, and save result in register **rd**
100	**Signed Long Integer Multiply-Add Low with Saturation**: Multiply low 32-bit- fields of each 64-bit field of registers **rs3** and **rs2**, then add 64-bit products to 64-bit fields of register **rs1**, and save result in register **rd**
101	**Signed Long Integer Multiply-Add High with Saturation**: Multiply high 32-bit- fields of each 64-bit field of registers **rs3** and **rs2**, then add 64-bit products to 64-bit fields of register **rs1**, and save result in register **rd**
110	**Signed Long Integer Multiply-Subtract Low with Saturation**: Multiply low 32- bit-fields of each 64-bit field of registers **rs3** and **rs2**, then subtract 64-bit products from 64-bit fields of register **rs1**, and save result in register **rd**
111	**Signed Long Integer Multiply-Subtract High with Saturation**: Multiply high 32- bit-fields of each 64-bit field of registers **rs3** and **rs2**, then subtract 64-bit products from 64-bit fields of register **rs1**, and save result in register **rd**

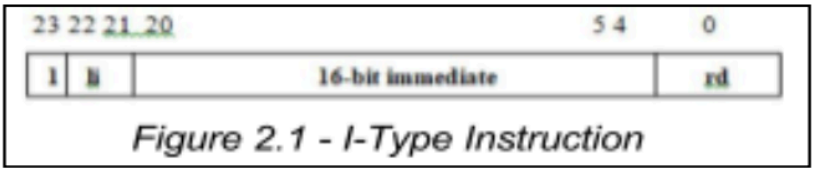

Figure 2.1 - I-Type Instruction

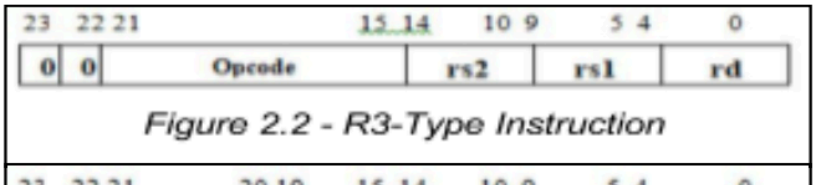

Figure 2.2 - R3-Type Instruction

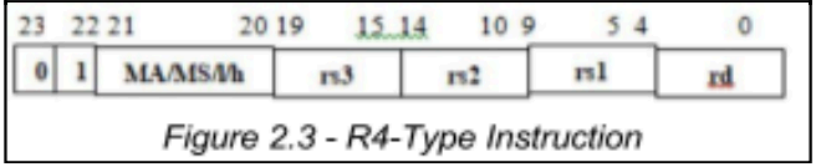

Figure 2.3 - R4-Type Instruction

Given all of the basics of the project let’s dive deeper into the implementation and strategies used to achieve the working system.

Implementation

Lets now dive into a piece by piece explanation of the implementation of the system as a whole.

Testbench: The testbench is responsible for multiple important functions. The testbench is responsible for programming and filling the instruction buffer, reading the current state of the processor and reporting it in the results file every positive edge cycle, providing appropriate clock and reset signals, and asserting the expected results file compared to the memory of the system.

Programming the instruction buffer: The test bench has 3 signals to send and control the IB. CLK, RESETBAR, and HOLD. Using these we can define 3 states of the PMU. Resetting, Programming, and Normal Operation. In programming mode the system is halted, but the instruction buffer is operating in a special state. The Test bench will feed in a number for the instruction and the instruction code itself every cycle, and the instruction buffer uses this data to place the instructions into the proper offsets into program memory.

Results File Writing: After every positive edge cycle the test bench will read the signals coming from the PMU at different stages of the pipelining process

Verification and Checking: The testbench is programmed to automatically compare the internal memory of the processor with an externally provided expected results file. This results file contains 32 lines of 128 bits of data. These values are directly compared to the data within the register file. Given you have the binary values for the results of your calculations and operations, you can easily check if the processor is operating properly. The command line will report a debug message for every register that has unexpected values : "Reg X unexpected value." and then after reporting each register which failed, will report either "Overall Test Failed" or "Overall Test Passed".

Instruction Buffer: The Instruction Buffer is responsible for a number of critical functions for the processor as a whole. It holds all the instructions to be executed for any run of the system (64 max), generates valid instruction flags to propagate throughout the system, must output the instructions in proper order on every new positive clock cycle, and must tell the system when all program code has been run.

Instruction Memory: The system is able to take in instructions when the test bench prepares it for program loading. The instruction memory can hold 64 instructions 25 bits long.

Valid Instruction Flags: The Instruction buffer provides a signal to be sent down the pipeline for every instruction in the program. When the instruction list ends the instruction buffer leaves the valid instruction signal as a “0” to indicate every cycle afterwards there are no more instructions to be processed. This is to ensure the system does not read any more instruction inputs internally every cycle and processes bad data.

Instruction Output: Every positive clock cycle the Instruction Buffer will output the next instruction in the list. The instruction memory will be iterated through until the very end at which point it will seize outputting and instruct the system there are no more instructions to handle afterwards.

Instruction Buffer Forwarding Register:

The Instruction Buffer Forwarding Register is responsible for the critical function of providing pipeline spacing for the processor's "Instruction Fetch" stage. It accomplishes this with simple Input to Output logic dependent on the positive edges of the clock. The Forwarding register also checks for the valid instruction flag from the Instruction buffer, which itself will then provide to the next stage. The Buffer also provides the pipelined Instruction data to the next stage.

Register File:

The Register File is responsible for the function of storing written back data and providing the registers to the pipeline for arithmetic operations.

Outputting Register Data: Depending on the opcode type, and instruction type, the register file will properly output the data requested by RD, RS1, RS2, RS3.

Writing Back Data: The Register File before any data is output will check for any data hazards between write back and data providing. The register file will then replace the data

Instruction Decode Forwarding Register:

The Instruction Buffer Forwarding Register is responsible for the critical function of providing pipeline spacing for the processor's "Instruction Decode" stage. It accomplishes this with simple Input to Output logic dependent on the positive edges of the clock. The Forwarding register also checks for the valid instruction flag from the Register File, which itself will then provide to the next stage. The Buffer also provides the pipelined Instruction data and associated requested register data to the next stage.

Verification Results / Strategies

For our verification results/strategies it was as hardware and structure related as it was manual and intimate work with the PMU to prove everything was working as scheduled and in order. The PMU was designed at a base level with data reporting and transparency in mind. As such, verification was handled as follows:

1. Given the instruction compiling/encoding structure we generate a series of instructions to demonstrate and prove functionality across the instructions feature set.
2. Write down and manually calculate the state of memory after every single clock, including final state.
3. Load the program using the project/testbench.
4. Step through the program ensuring that each and every clock has matching memory values after handling and writing back.
5. For any case of discrepancy, immediately begin work to track back the issue.
6. Continue stepping through the program, if issue found go back to step 5.
7. Once the final stage of memory is matched to the calculated memory, you have proved your test program.

How to Navigate the Repo File Structure and Use the PMU

The Github repository for the project contains all that is needed to run, manipulate, and even iterate on the PMU. Lets go over each part.

The assembler is a folder to open a project in Visual Studio Code to compile and create (BASED ON THE ORIGINAL INSTRUCTION SET AND OUTLINES BASED OFF THE ORIGINAL PROJECT DESCRIPTION FOUND IN THE REPO) programs and executable code for the PMU to execute.

There are numerous folders covering tests of different instruction set types.

The RTL and Block diagrams folder cover some visualizations and guides to better understand the interal structure of the project.

The Source Files folder is the whole project folder, openable by Synplify (tested in V14). In “Source Files/ALU_BETA/src” you will find the VHD code for the whole processor, along with the input_data.txt file which serves as the input for the instruction buffer (the code and text you can run from the compiled

code from the assembler. Also in “Source Files/ALU_BETA” you will find “results.txt”. This is the result and final state of memory for the PMU.

Conclusion

Through many tests of the system, our pipelined multimedia unit works. The registers at the end had all of the expected results after 64 instructions. Additionally, the instructions were forwarded when necessary and we were able to see the updates happening in each cycle for each instruction as expected. On this assignment me and my partner received a 100 on this assignment.