**ABOUT THE AUTHORS**

*Harish Marepalli* had done bachelor’s in Electronics and Communication Engineering at Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, Telangana, India. He had 3 years of experience as a Systems Engineer in Tata Consultancy Services and his role was software developer. Currently, he is pursuing his master’s in Computer Engineering at San Jose State University. His native place is Hyderabad and mother tongue is Telugu. Additionally, he does speak English and Hindi and would love to connect with the people of any region and build network.

*Saiteja* is from Hyderabad, India. I completed my bachelors in Electronics and Communication Engineering from Gokaraju Rangaraju Institute of Engineering and Technology with a CGPA of 9.97/10 in the year 2019. I received a Gold Medal for the academic excellence from my university. Right after that, I got an opportunity to work for Tata Consultancy Services as a Systems Engineer and have worked for 3 years until July 2022. I speak English, Hindi, and Telugu where the last of these is my first language. My interests include but not limited to Cricket, Table-Tennis and learning new languages. Currently I am pursuing my master’s in Computer Engineering at San Jose State University.

Table of Contents

[LIST OF FIGURES AND TABLES 4](#_Toc120986325)

[INTRODUCTION: 5](#_Toc120986326)

[MY GROUP: 5](#_Toc120986327)

[GIVEN TASK: 5](#_Toc120986328)

[STEPS TAKEN TO COMPLETE THE TASK: 5](#_Toc120986329)

[DISCUSSION SECTION: 6](#_Toc120986330)

[1) System-on-chip (SoC) Design: 6](#_Toc120986331)

[2) Factorial Accelerator with Interface Wrapper: 8](#_Toc120986332)

[3) General Purpose I/O (GPIO) with Interface Wrapper: 13](#_Toc120986333)

[4) Pipelined MIPS Processor Design: 15](#_Toc120986334)

[5) Pipelined MIPS Processor with Data Forwarding: 19](#_Toc120986335)

[6) Performance Analysis: 20](#_Toc120986336)

[7) Simulation Waveforms (with Unit-Level simulations): 21](#_Toc120986337)

[a) Testbench for the Factorial Accelerator Interface Wrapper: 22](#_Toc120986338)

[b) Testbench for the GPIO Unit Interface Wrapper: 23](#_Toc120986339)

[c) Testbench for the System-on-Chip (SoC) Address Decoder: 24](#_Toc120986340)

[d) Testbench for the Pipelined System-on-Chip (SoC) Design: 25](#_Toc120986341)

[COLLABORATION SECTION: 27](#_Toc120986342)

[CONCLUSION: 27](#_Toc120986343)

[APPENDIX 28](#_Toc120986344)

[8) Schematic of the Data Memory module used in the pipelined processor: 28](#_Toc120986345)

[9) Schematic of the Instruction Memory module used in the pipelined processor: 28](#_Toc120986346)

[10) Schematic of the Control unit used in the pipelined MIPS processor: 28](#_Toc120986347)

[11) Datapath of the pipelined MIPS processor: 29](#_Toc120986348)

[12) Control unit Truth table for the pipelined processor (Decode Stage): 31](#_Toc120986349)

[13) Control unit Truth table for the pipelined processor (Execute Stage): 31](#_Toc120986350)

[14) Control unit Truth table for the pipelined processor (Memory Stage): 32](#_Toc120986351)

[15) Control unit Truth table for the pipelined processor (Writeback Stage): 32](#_Toc120986352)

[16) Source Code of tb\_system.v: 33](#_Toc120986353)

[17) Source Code of tb\_fact\_top.v: 34](#_Toc120986354)

[18) Source Code of tb\_gpio\_top.v: 36](#_Toc120986355)

[19) Source Code of tb\_system\_ad.v: 38](#_Toc120986356)

[20) Source Code of tb\_mips\_pipelined\_top.v: 38](#_Toc120986357)

[21) Source Code of system.v: 39](#_Toc120986358)

[22) Source Code of system\_AD.v: 42](#_Toc120986359)

[23) Source Code of fact\_top.v: 42](#_Toc120986360)

[24) Source Code of the gpio\_top.v: 45](#_Toc120986361)

[25) Source Code of mux4.v: 46](#_Toc120986362)

[26) Source Code of mips\_pipelined\_top.v: 47](#_Toc120986363)

[27) Source Code of mips\_pipelined.v: 48](#_Toc120986364)

[28) Source Code of imem.v: 60](#_Toc120986365)

[29) Source Code of dmem.v: 60](#_Toc120986366)

[30) Source Code of mux2.v: 61](#_Toc120986367)

[31) Source Code of dreg.v: 61](#_Toc120986368)

[32) Source Code of adder.v: 61](#_Toc120986369)

[33) Source Code of D\_Stage\_Reg.v: 61](#_Toc120986370)

[34) Source Code of controlunit.v: 62](#_Toc120986371)

[35) Source Code of regfile.v: 63](#_Toc120986372)

[36) Source Code of signext.v: 64](#_Toc120986373)

[37) Source Code of E\_Stage\_Reg.v: 64](#_Toc120986374)

[38) Source Code of alu.v: 68](#_Toc120986375)

[39) Source Code of mult\_inf.v: 69](#_Toc120986376)

[40) Source Code of M\_Stage\_Reg.v: 69](#_Toc120986377)

[41) Source Code of HiLo\_reg.v: 72](#_Toc120986378)

[42) Source Code of W\_Stage\_Reg.v: 73](#_Toc120986379)

[43) Source Code of maindec.v: 75](#_Toc120986380)

[44) Source Code of auxdec.v: 76](#_Toc120986381)

[45) Source Code of and\_fact.v: 77](#_Toc120986382)

[46) Source Code of fact\_ad.v: 77](#_Toc120986383)

[47) Source Code of fact\_reg.v: 78](#_Toc120986384)

[48) Source Code of FSMult.v: 79](#_Toc120986385)

[49) Source Code of fact\_res\_done\_reg.v: 80](#_Toc120986386)

[50) Source Code of fact\_res\_err\_reg.v: 81](#_Toc120986387)

[51) Source Code of DP.v: 82](#_Toc120986388)

[52) Source Code of CNT.v: 83](#_Toc120986389)

[53) Source Code of Multiplexer.v: 84](#_Toc120986390)

[54) Source Code of Register.v: 85](#_Toc120986391)

[55) Source Code of CMP.v: 86](#_Toc120986392)

[56) Source Code of CU.v: 86](#_Toc120986393)

[57) Source Code of gpio\_ad.v: 89](#_Toc120986394)

[58) Source Code of Multiplier.v: 90](#_Toc120986395)

[59) Assembly Code in pipelined MIPS to trigger the SoC: 91](#_Toc120986396)

# LIST OF FIGURES AND TABLES

*Figure1.1 – System-on-chip (SoC) schematic diagram*

*Table1.2 – Memory-mapped I/O design table*

*Table1.3 – Explanation of all SoC module*

*Figure1.4 – Factorial Accelerator CU-DP diagram*

*Figure1.5 – Factorial Accelerator Datapath diagram*

*Figure1.6 – Factorial Accelerator ASM chart*

*Figure1.7 – Interface Wrapper with Factorial Accelerator*

*Table1.8 – Memory map table for the Factorial Accelerator Wrapper*

*Table1.9 – Explanation of all modules in the Factorial Accelerator Wrapper*

*Figure1.10 – Schematic of the Interface Wrapper for the GPIO Unit*

*Table1.11 – Memory map table for the GPIO unit Interface Wrapper*

*Table1.12 – Explanation of all modules in the GPIO Interface Wrapper*

*Figure1.13 – Gantt Chart of the Pipelined MIPS processor*

*Figure1.14 – Data Forwarding using the Pipelined MIPS processor*

*Figure1.15 – Runtime cycles taken by each for the factorial computation*

*Figure1.16 – Runtime cycles taken by each for the factorial computation*

*Figure1.17 – Waveform of n=4 for the Factorial Accelerator Wrapper*

*Figure1.18 – Waveform of n=5 for the Factorial Accelerator Wrapper*

*Figure1.19 – Waveform of GPIO Unit Interface Wrapper after testing*

*Figure1.20 – Waveform of System Address Decoder after testing*

*Figure1.21 – Waveform of SoC when n=4 after testing*

*Figure1.22 – Waveform of SoC when n=5 after testing*

# INTRODUCTION:

This activity is to convert the fifteen-instruction 32-bit single-cycle MIPS processor into a five-stage, pipelined design. Apart from that, we interface the processor with a factorial accelerator and a general-purpose I/O unit using memory-mapped interface registers.

# MY GROUP:

**Name:** Student\_Team 6

**Members:** Harish Marepalli (016707314), Tirumala Saiteja Goruganthu (016707210)

# GIVEN TASK:

The task is to complete the design and functional verification of a pipelined MIPS processor. Also, we are further required to complete the design and functional verification of a small system-on-chip (SoC) system which includes the pipelined MIPS processor, a factorial accelerator (that we have completed in Assignment 1), and a simple general-purpose IO (GPIO) unit, all connected via direct interface to the MIPS local bus, using memory-mapped interface registers. This report must contain the following:

1. Draft of pipelined MIPS microarchitecture schematic.
2. Draft of SoC interface schematic.
3. Tables for MIPS control unit and memory maps for SoC interface.
4. Performance analysis of hardware accelerated n!
5. The unit-level including interface wrappers simulation.
6. Completed interface design for the SoC with the pipelined MIPS processor, the factorial unit (from Assignment 1) and the simple GPIO.

Note that apart from the above given tasks, we must do the following two activities.

1. Based on the enhanced single-cycle MIPS processor (which supports the additional instructions multu, mfhi, mflo, jr, jal, sll, slr) you successfully accomplished in Assignment 6, use the test program (factorial calculation via nested procedure calls) included in the assignment to test the pipelined MIPS processor.
2. For the SoC design , we must be able to use the factorial accelerator to handle 4-bit input data (n). In addition to that, we should use the simple GPIO for validation i.e., the MIPS should read input (n) via GPIO and return the result (n!) also via GPIO.

# STEPS TAKEN TO COMPLETE THE TASK:

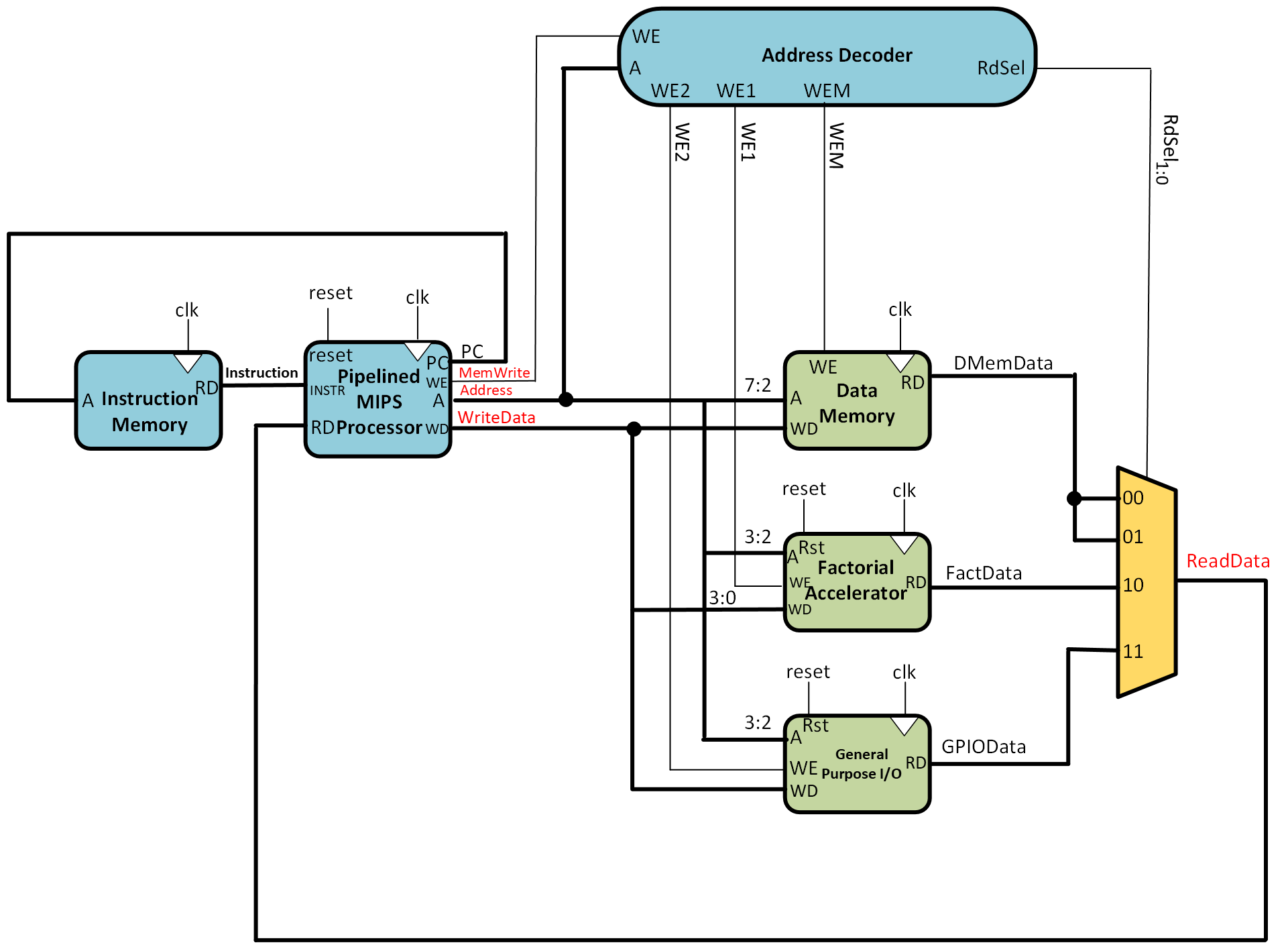
1. Spent time to understand the question carefully.
2. Discussed on how to proceed with the pipelined MIPS microarchitecture. This took us more than 1 week.
3. After coming up with a rough design, we started designing or drawing the architecture in Microsoft Visio tool. This too took us a week or so to complete the design in Visio.
4. Since, the control unit of the pipelined MIPS processor is pretty much like the single-cycle processor, we directly started preparing the tables for both auxiliary and main decoders by differentiating them with respect to their values in each pipeline stage.
5. Gone through the IO lecture notes to revisit some of the topics that are required to complete this assignment such as interface wrappers for both factorial accelerator and GPIO, memory-mapped IO design (using lw, sw instructions), memory map tables, and the verilog code to understand the core logic behind the interface wrappers.
6. Using the above knowledge, we started preparing the memory map tables for MIPS processor, Factorial Accelerator and the GPIO.
7. The above lecture notes also provide insight into the system-on-chip (SoC) design with a schematic already given in the notes. Spend some time to understand the logic behind the SoC schematic diagram.
8. Performance analysis is done for both single-cycle and MIPS pipelined processor with respect to the number of cycles factorial accelerator took to get output of the factorial computation.
9. After the above analysis, we started designing the verilog modules and writing corresponding logic for the modules.
10. To test the design, we created a new testbench for all major components such as Factorial Accelerator, GPIO unit, Pipelined MIPS and the SoC address decoder.
11. After that, we simulated the design and verified the output waveforms for the design correctness and concluded that each component is working fine on its own (had to make additional changes to the datapath design of the pipelined MIPS).
12. Now, it was time to test the whole SoC design, for which we had written a new testbench which accomplishes the above task of testing the whole system.
13. Finally, we simulated the final testbench and after getting the desired output, the whole pipelined datapath is updated using the Visio tool to match the logic written in verilog.

# DISCUSSION SECTION:

## System-on-chip (SoC) Design:

The design explanation of the system-on-chip (SoC) will facilitate in explaining how the factorial accelerator and GPIO unit were designed and how they were interfaced with the CPU. The SoC contains the following elements: instruction memory, MIPS Processor (pipelined CPU), data memory, factorial accelerator, general purpose I/O (GPIO), an address decoder, and a 4-input multiplexer to control writeback data to the MIPS processor. The following design of this SoC is shown in *Figure 1.1*.

The MIPS processor fetches an instruction from instruction memory and processes the instruction accordingly. If the instruction is a save word (*sw*) or load word (*lw*) instruction, the processor will issue an address to the address decoder along with a memory write enable signal. The address bus is also connected to each I/O device. A write data bus is connected to the data memory, factorial accelerator, and GPIO unit. However, these I/O devices essentially do nothing until given a write enable signal from the address decoder. The address of the target I/O device is given in the *lw* or *sw* instruction. In this way, the MIPS processor uses memory-mapped I/O to communicate with connected I/O devices like the factorial accelerator, data memory, or GPIO unit. A table showing what addresses are mapped to each device is shown in *Table 1.2*. For example, a *lw $t1, 0x800($0)* instruction addresses the factorial accelerator.



*Figure1.1 – System-on-chip (SoC) schematic diagram*

|  |  |  |  |
| --- | --- | --- | --- |
| **MIPS Control** | | | |
| **Base Address** | **Address Range** | **R/W** | **Description** |
| 0x00000000 | 0x00 - 0xFC | R/W | Data Memory |
| 0x00000800 | 0x00 - 0x0C | R/W | Factorial Accelerator |
| 0x00000900 | 0x00 - 0x0C | R/W | General Purpose I/O |

*Table1.2 – Memory-mapped I/O design table*

The following *Table 1.3* explains each module used in the SoC design in detail.

|  |  |
| --- | --- |
| **Modules** | **Description** |
| Instruction Memory | Holds machine code from assembled source program. The MIPS processor retrieves an instruction from this memory module one at a time. This is a read-only module. |
| MIPS Processor | Either the single-cycle MIPS processor or pipelined MIPS processor (pipelined processor is used here in this assignment). This module processes all instructions and communicates with the connected I/O devices. |
| Data Memory | An I/O device that stores data when the MIPS processor issues a *lw* or *sw* instruction with the address range 0x000 - 0x7FF. This module is read or write module. |
| Factorial Accelerator | An I/O device that calculates n! given an input n and a Go signal. The device provides the result and status bits Done and Error via a write back data bus. This I/O device is only active when the MIPS processor issues a *lw* or *sw* instruction with the address range 0x800 - 0x80C. This module is read or write module. |
| General Purpose I/O | An I/O device that interfaces with connected input and output devices on external FPGA devices. The device takes two 32-bit inputs and outputs two 32 bit values and can deliver data to the MIPS processor via a writeback data data bus. This I/O device is only active when the MIPS processor issues a *lw* or *sw* instruction with the address range 0x900 - 0x90C. This module is read or write module. |
| Address Decoder | A module that contains simple logic to ensure that only a specific I/O device is active and writing data back to the MIPS processor. It receives an address from the MIPS processor and issues write enable signals to the respective I/O devices and controls the output of a connected 4-input multiplexer that is connected to the input read data port of the processor. |
| 4-Input Mux | A module that controls what data is written back to the MIPS processor. It takes data from the data memory, factorial accelerator, and GPIO unit as inputs. |

*Table1.3 – Explanation of all SoC modules*

## Factorial Accelerator with Interface Wrapper:

The factorial accelerator used in this assignment was based on the design of the factorial accelerator from Assignment 1. That is, the factorial module is designed from the bottom-up based on the following algorithm:

*INPUT n*

*product = 1*

*WHILE (n > 1) {*

*product = product \* n*

*n = n -1*

*}*

*OUTPUT product*

As shown in *Figure 1.4*, our Factorial module takes an input *n* and input *Go* signal to begin the factorial calculation. The control unit drives the calculation of n! inside the datapath. The *product* or result of the calculation is outputted along with a *Done* bit. If the input *n* is greater than 12, the factorial calculation does not take place. Instead, the datapath outputs an *Error* bit to the control unit which then outputs an *Error* bit. This was essential to Assignment 8 as our single-cycle CPU and pipelined CPU would rely on these status bits to find out when the factorial accelerator was done processing with the calculation. Therefore, as our factorial accelerator provided everything we needed, we opted to not change the design. *Figure 1.5 and 1.6* shows the design of the datapath and the ASM chart of the control unit for the factorial accelerator.

Shape

Description automatically generated with medium confidence

*Figure1.4 – Factorial Accelerator CU-DP diagram*

*Shape

Description automatically generated with medium confidence*

*Figure1.5 – Factorial Accelerator Datapath diagram*

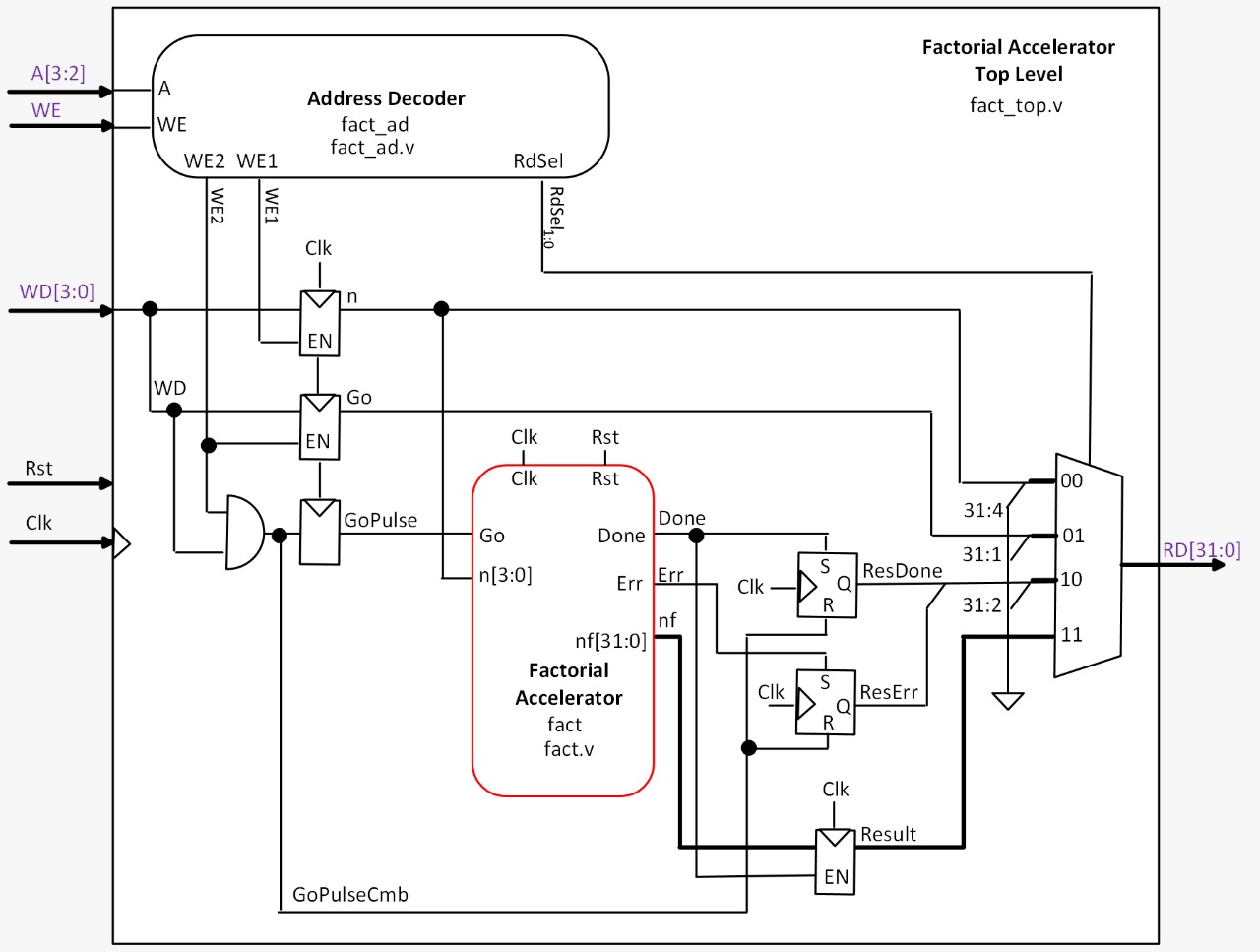
Shape

Description automatically generated with medium confidence

*Figure1.6 – Factorial Accelerator ASM chart*

We could not simply connect our factorial module to the pipelined CPU. The factorial accelerator would need a way to know when it was being addressed by a CPU. It would also need something to convert address values from the CPUs to something the factorial module could understand, i.e., an input ‘n’ or a Go signal to kick off the calculation. Therefore, we created an interface wrapper for the factorial unit. *Figure 1.7* shows the design of this interface wrapper.

The design of this interface wrapper was based on a design given to the class during the IO interface lecture. It works by accepting bits 3 and 2 from the connected address bus from the MIPS processor. Bits 3 and 2 of the input address were selected as these are the most important bits when the CPU addresses the factorial accelerator. This is because the program counter inside the MIPS processor always changes by a factor of 4. The binary value of 4 is 100, the binary value of 8 is 1000, and the binary value of 32 is 100000. As shown, the rightmost bits of all numbers of factor 4 end in 00 in binary. Therefore, it’s safe to disregard these bits and only worry about bits 3 and 2 as they are the only bits that change in numbers of factor 4. This design principle is applied in all the other I/O devices attached to the MIPS processor.



*Figure1.7 – Interface Wrapper with Factorial Accelerator*

The interface wrapper also accepts a write enable (WE) signal from the address decoder inside the SoC. Additionally, the factorial accelerator wrapper also takes an input ‘n’ given by the first four bits of the attached write data (WD) bus from the MIPS processor. In this way, whenever the MIPS processor processes a *lw* or *sw* instruction with an address between 0x800 and 0x80C, it will address the factorial accelerator interface. *Table 1.8* below shows the relevant address bits (A[3:2]) that determine what happens inside the interface. For example, if the MIPS processor processes a *sw $t1, 0x800(0)*, the processor will address the factorial accelerator and give it the data in $t1 as the input ‘n’. The address decoder in the SoC will also issue WE to enable the interface.

Inside the interface exists another address decoder, the factorial accelerator from Assignment 1, a few registers to hold input data and output data from the factorial accelerator, and a 4-input mux to control output data from the interface. The basic idea is that registers must hold input data from the MIPS processor (such as the Go signal or ‘n’) so that the MIPS processor can do other things while the factorial accelerator performs a calculation. Output registers must hold the result data from the factorial accelerator until the MIPS processor is ready to read the data with a *lw* instruction. The address decoder has simple logic to issue write enable signals to the relevant registers depending on the function the MIPS processor is telling the interface to perform. For example, in a *sw $t1, 0x804(0)*, the address decoder must enable a Go register and a GoPulse register to hold the input Go signal from the MIPS processor. Registers hold input data *n* and *Go* and output data *Done*, *Error*, and *Result*. The 4-input mux has a select input controlled by the address decoder. Output data from the interface wrapper is shown in *Table 1.8*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Factorial Accelerator Memory Map** | | | | | | | |
|  |  |  |  |  | **Decoder Output** | | |
| **Address** | **R/W** | **Register Name** | **Bits** | **Bit Definition** | **WE1** | **WE2** | **RdSel** |
| 00 | R/W | Data Input (n) | 31:4 | Unused |  |  |  |
| 3:0 | n[3:0] | 1 | 0 | 00 |
| 01 | R/W | Control Input (n) | 31:1 | Unused |  |  |  |
| 0 | Go Bit | 0 | 1 | 01 |
| 10 | R | Control Output (Done, Err) | 31:2 | Unused |  |  |  |
| 1 | Err Bit | 0 | 0 | 10 |
| 0 | Done Bit | 0 | 0 | 10 |
| 11 | R | Data Output (Result) | 31:0 | nf[31:0] | 0 | 0 | 11 |

*Table1.8 – Memory map table for the Factorial Accelerator Wrapper*

The following *Table 1.9* explains each module used in the Factorial Accelerator Interface Wrapper design in detail.

|  |  |
| --- | --- |
| **Modules** | **Description** |
| Address Decoder | This module controls which register is addressed by the MIPS processor. It issues write enable signals to the respective registers inside the interface wrapper. It also controls which register outputs its data to the read data bus. |
| Factorial Accelerator | This module performs the calculation of n! given an input ‘n’ and a Go signal. The result and status bits are stored in status registers until the MIPS processor is ready to read the data. |
| n Register | This module holds input ‘n’ data until the MIPS processor instructs the factorial accelerator to begin with the Go signal. |
| Go Register | This module holds the input Go bit if the MIPS processor ever wants to read the stored Go bit. |
| GoPulse Register | This module provides the Go pulse necessary to begin the calculation of n!. GoPulse = 1 when both the address decoder issues a write enable signal for this register and the MIPS processor issues the Go bit with a *sw $s, 0x804($0)* instruction. |
| ResDone Register | This module stores the Done bit output of the factorial accelerator until ready to be read by the MIPS processor. |
| ResError Register | This module stores the Error bit output of the factorial accelerator until ready to be read by the MIPS processor. |
| 4-Input Mux | This module controls which data is output from the interface back to the MIPS processor. Options include input ‘n’, the Go bit, the status bits Done and Error, and the result of n! |

*Table1.9 – Explanation of all modules in the Factorial Accelerator Wrapper*

## General Purpose I/O (GPIO) with Interface Wrapper:

The design of the GPIO unit mirrors that of the design of the factorial accelerator interface wrapper. The GPIO unit is responsible for interfacing with external input and output device. When addressed by the MIPS processor, the GPIO unit will either read an input from its input gpI1 or gpI2 input ports or write to its output ports gpO1 or gpO2. The design of this interface is shown in *Figure 1.10*.

Inside the interface wrapper exists another address decoder like the one in the SoC and the factorial accelerator interface wrapper. The address decoder takes bits 3 and 2 from the connected address bus from the MIPS processor. Like the factorial accelerator address decoder, only bits 3 and 2 are relevant to the operation of the GPIO unit. A write enable signal also ensures the GPIO unit is only active when enabled by the correct address range 0x900 - 0x90C. Additionally, two registers hold input data from the write data to data memory bus from the MIPS processor. These two registers only store new data when given a write enable signal by the address decoder within the GPIO interface wrapper. In this way, one register stores data input from the MIPS processor to be output to gpO1 port while the other register holds input data from the MIPS processor to output to the gpO2 port. Furthermore, this design principle enables each register to be accessed by a specific address from the MIPS processor. These two registers were designed to hold the output of the factorial accelerator’s calculation and the status bits from the factorial accelerator. For example, gpO2 holds the computation result, while gpO1 holds the status bits to output such as error bit. Finally, a 4-input mux like as in the SoC and factorial accelerator controls the output of the GPIO unit. It’s select port is controlled by the address decoder. *Table 1.11* shows what happens inside the GPIO unit when it is addressed by the MIPS processor. It also shows what data is output on the read data bus from the GPIO unit.

Diagram, schematic

Description automatically generated

*Figure1.10 – Schematic of the Interface Wrapper for the GPIO Unit*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **GPIO Memory Map** | | | | | | |
|  |  |  |  | **Decoder Output** | | |
| **Address** | **R/W** | **Register Name** | **Bits** | **Bit Definition** | **WE2** | **Rdsel** |
| 00 | R | Input1 | 31:0 | General Input | 0 | 00 |
| 01 | R | Input2 | 31:0 | General Input | 0 | 01 |
| 10 | R/W | Output1 | 31:0 | General Output | 0 | 10 |
| 11 | R/W | Output2 | 31:0 | General Output | 1 | 11 |

*Table1.11 – Memory map table for the GPIO unit Interface Wrapper*

The following *Table 1.12* explains each module used in the GPIO Interface Wrapper design in detail.

|  |  |
| --- | --- |
| **Modules** | **Description** |
| Address Decoder | This module controls which register is addressed by the MIPS processor. It issues write enable signals to the respective registers inside the interface wrapper. It also controls which register outputs its data to the read data bus. |
| gpO1 Reg | This register holds input data from the write data bus from the MIPS processor. Data from this register is output via the gpO1 port to an external device outside of the SoC. |
| gpO2 Reg | This register holds input data from the write data bus from the MIPS processor. Data from this register is output via the gpO2 port to an external device outside of the SoC. |
| 4-Input Mux | This input controls what data is written back to the MIPS processor. Options include input data on gpI1, gpI2 or output data from the gpO1 or gpO2 register. The select signal for this mux is given by the address decoder inside this interface. |

*Table1.12 – Explanation of all modules in the GPIO Interface Wrapper*

## Pipelined MIPS Processor Design:

Designing the pipelined MIPS processor proved to be the most challenging task. We spent a lot of time grouping the respective modules by the following stages: fetch, decode, execute, memory, and writeback. Each stage processes part of one instruction per clock cycle, forwarding relevant data necessary for the next step of execution via a state register to the next stage. For example, in an *add* instruction, the instruction must be fetched in the fetch stage, the operands must be fetched in the decode stage, the addition must occur in the execute stage, the data must pass through the memory stage, and finally the result must be written back to the destination register in the writeback stage.

In the fetch stage, the MIPS processor tries to fetch an instruction from the instruction memory. Because instruction memory is external to the CPU, it must calculate the program counter value and output this address to the instruction memory to retrieve the next instruction. Therefore, the fetch stage contains several multiplexers and an adder to calculate program counter + 4. This adder is responsible for calculating the next instruction to retrieve. A series of multiplexers chooses which address is output to the instruction memory. They choose between the program counter + 4 address, branch address, or jump address, or jump return address. For example, a *j target* instruction will send the address of the instruction to jump to the instruction memory. Once the instruction memory releases an instruction back to the MIPS processor, the instruction is stored in a state register. Other data stored in the state register includes the program counter + 4 address. This data is stored in this fetch-to-decode state register until the next clock cycle to forward information to the decode stage.

In the decode stage, the MIPS processor retrieves the relevant operand data from the register addresses encoded in the machine code. The machine code comes from the instruction now stored in the fetch-to-decode state register. In this stage, the control unit also generates relevant control signals that are forwarded through the pipeline until they are necessary. Adhering to the principles of pipelining, data retrieved from the specified registers is clocked into the decode-to-execute state register to be processed on the next clock cycle. The control signals determined by the machine code inside the control unit are also stored in the same state register for the execute stage as well as the writeback address of the destination register. This address is also encoded in the machine code and must be forwarded along with the operands of this instruction (and eventually the result of the execution). The actual address given in the machine code changes depending on the type of instruction being executed. Therefore, a multiplexer chooses between bits 20:16 for R-Type instructions and bits 15:11 for all other instructions. A final multiplexer chooses between the previous multiplexer’s out register address or register 31 for *jal* instructions. This writeback address is stored along with all other data in the decode-to-execute state register. Finally, to compute branch control value and to reduce the branch overhead cycles, the logic for branch instruction is moved to the decode stage using a comparator between two read data outputs of the register file. This comparator value is then anded with the branch control signal to determine the next program counter value. Note that, during all this, branch target address will be calculated and ready.

In the execute stage, the actual processing of an execution takes place. For example, an *add* instruction’s operands are added inside the ALU and the operands of a *multu* instruction are multiplied here. Operand data for each instruction is output from the decode-to-execute state register to the ALU, and Multiplier modules. The ALU processes *and, or, add, sub,* and *slt* instructions. The Multiplier processes *multu* instructions. The *slr* and *sll* instructions will be taken care by the ALU itself. A multiplexer chooses between a second operand from the register file and a sign-extended immediate value to forward as the second operand for the ALU. The sign-extended immediate value comes from a SEXT module in the decode stage that extends a 16-bit immediate value given in the machine code (*addi* instructions) into a 32-bit value. This value also comes through the same state register. The ALU takes a “shamt” input given in the machine code. Therefore, the machine code is also forwarded through the pipeline for use in the execute stage. The output of all these modules is stored in another state register, execute-to-memory. Control signals needed for later in the pipeline are also forwarded from the previous decode-to-execute state register to the execute-to-memory state register. Refer to Appendix to see all control signals that are forwarded through each state register until needed. Data from previous stages that were not operated upon in this stage but are needed later are also forwarded through the execute stage and the state registers. This data includes the destination register address, data from source register 1 (for *jr* instructions), data from source register 2 to write to data memory or an I/O device, and program counter + 4.

In the memory stage, data is either output from the MIPS processor to memory or an I/O device or read from memory or an I/O device. This only occurs if a *sw* or *lw* instruction is being processed. Although the diagram in Appendix shows the data memory within the datapath, it’s important to note that memory is actually external to the CPU. The following signals are outputs of the processor: alu\_outM (Address) , wd\_dmM (WriteData), and we\_dmM (MemWrite). rd\_dmM (ReadData) is an input to the processor. The signals in parentheses correspond with the signals in *Figure 1.1* of the System-on-Chip Design section. Input data from the SoC (ReadData) is pipelined in the memory-to-writeback state register. Aside from memory, data from the *multu* instruction is also stored to a high and low register (HiLo\_reg) in this stage. In an earlier iteration of this design, we placed these registers in the decode stage. Recall from Assignment 6 that the *hi* register holds the upper 32 bits of the result of the multiplication while *low* register holds the lower 32 bits. Data from these registers are also stored in the memory-to-writeback state register. Control signals not needed in this stage are also stored in the memory-to-writeback state register for use later. Data from previous stages that were not operated upon in this stage but are needed later are also forwarded through the memory stage and the state registers. This data includes the output of the ALU to be stored in the register file, data from source register 1 (for *jr* instructions), program counter + 4, and the machine code from the fetch stage.

Finally, in the writeback stage, data is written back to the register file. R-Type instructions primarily use this stage. This stage is made up of several buses connected to previous portions in the pipeline and several multiplexers that control what data is written back to the register file. A 4-input multiplexer is choosen to transfer the data from the high register, low register, ALU output, and input from data memory or I/O devices to writeback depending on the instruction being processed. For example, *mflo $t1*, will forward data from the low register through this multiplexer. Additionally, several buses are directly connected from the memory-to-writeback state register back to various ports in the datapath. These include the register destination address that is given to the register file with the data to be written back in an R-Type or *jal* instruction and the jump address given by bits 25:0 of the machine code.

In each stage, certain modules require control signals. As mentioned before, control signals are created in the decode stage based on the input machine code. All control signals are pipelined and stored in state registers until needed. The control signals are unchanged from Assignment 6 and are only grouped by their respective stage in Appendix. These design choices were motivated by the following thought process: control signals should travel through the pipeline with the instruction until a module needs that control signal. In this way, processing of an instruction flows smoothly through the pipeline and does not have to wait for control signals as they are readily available.

As architecture diagram and control unit truth tables for the design of this pipelined CPU are too large to insert here, they have been added to Appendix. The explanation of the pipeline makes more sense as you follow along with the diagram of the datapath. Signals are named according to the state they originate in. For example, “instr” is a 32 bit bus that travels through all state registers. In each stage, a letter for the name of that stage is appended to the name of the signal: “instrF” for fetch, “instrD” for decode, “instrE” for execute, “instrM” for memory, and “instrW” for writeback. This naming convention applies to nearly all signals and buses in the datapath.

Because of the design of our datapath, few NOPs (no operation MIPS instruction) were added to the driver assembly code. NOPs were added whenever a data dependency was detected. That is, if an instruction saved data to a register that the following instruction required for its operation, we added NOPs to allow time for the data to flow through the entire pipeline and writeback to the register file. Because it takes a total of 3 clock cycles to travel from the decode stage to the writeback stage, we added 3 NOPs in cases of data dependencies. NOPs were also added for jump and branch instructions for the reasons described earlier in this paragraph. Time was needed for the address to jump or branch to be given back to the fetch stage. The modified code for this driver assembly code can be found in the Appendix. A Gantt chart showing the result of our pipelining efforts is present below in *Figure 1.13*. The diagram shows all instructions from the modified MIPS driver code with few NOPs inserted for instructions with data dependencies.

Chart, bar chart

Description automatically generated

*Figure1.13 – Gantt Chart of the Pipelined MIPS processor*

## Pipelined MIPS Processor with Data Forwarding:

To improve the performance of our pipelined MIPS processor, we attempted to implement a form of data forwarding. This would solve situations where instructions required data from registers that were not yet updated from the previous instruction. Essentially, we were trying to solve a data dependency hazard. Unfortunately, we could not get a working design complete before the demonstration deadline. The basic idea is that if an instruction requires data that has not been written back to the register file, but the data is present somewhere in the pipeline, then that data should be forwarded back to the execution stage to be used in execution. Therefore, there are two paths to retrieve data from the pipeline: a path from the memory stage and a path from the writeback stage. The logic equations are as follows:

*Forwarding from memory stage:*

*if (we\_regM and (instrE[25:21] != 0) and (rf\_waM == instrE[25:21])) then ForwardAE = 10*

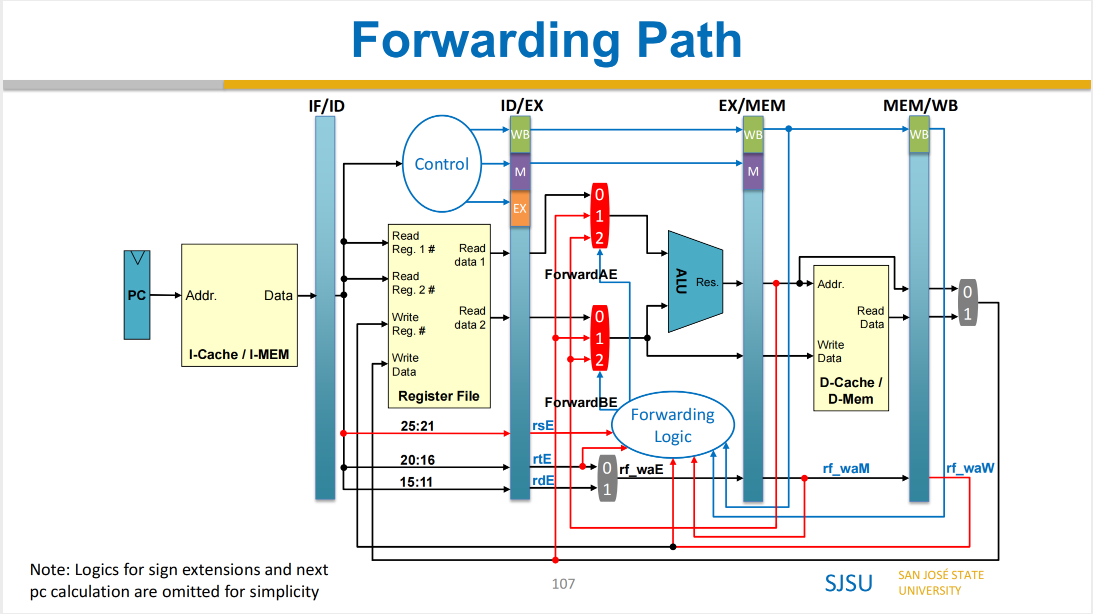
*if (we\_regM and (instrE[20:16] != 0) and rf\_waM == instrE[20:16])) then ForwardBE = 10*

*Forwarding from writeback stage*

*if (we\_regW and (instrE[25:21] != 0) and (rf\_waW == instrE[25:21]) then ForwardAE = 01*

*if (we\_regW and (instrE[20:16] != 0) and (rf\_waW == instrE[20:16])) then ForwardBE = 01*

ForwardAE and ForwardBE are control signals from a data forwarding logic unit. These control signals are fed to two 3-input multiplexer that controls which data is given to the ALU. One mux controls data for input port A of the ALU while the other controls data for port B of the ALU. The ForwardAE mux chooses between data from a register in the register file, the output of the ALU from the memory stage, or the writeback data from the writeback data. The ForwardBE mux chooses between the same input data as ForwardAE except with the second register output port of the register file. It’s important to note that no ports from the datapath modules are directly connected to these mux, only ports from the state registers. As data forwarding was not a core requirement of the assignment, it is not shown in the Waveforms section. To reiterate the above explanation, a figure from the lecture slides has been referenced below in *Figure 1.14*.



*Figure1.14 – Data Forwarding using the Pipelined MIPS processor*

## Performance Analysis:

The performance analysis involved the analysis of the pipelined MIPS processor design against the single-cycle (recursive) MIPS processor design for the factorial input of *n*. The input values of *n* included a range from 1 to 12 for computing the predicted results of *n!* of each design as depicted in *Figure 1.15*. With each transition to a new state, an additional cycle occurred. The runtime was calculated for the factorial accelerator by totaling the number of cycles for values of 1 to 12, equation (1) was used respectively. For the single-cycle MIPS design, the recursive version was assembled in the MIPS IDE. The instruction counter tool was relied upon to count the number of instructions executed for each input value. The total number of executed instructions totaled to the runtime of the single-cycle design. Overall, the total runtime for the single-cycle MIPS design was greater than the total runtime of the factorial accelerator for each factorial input of *N*. The total runtime for the factorial accelerator and the single-cycle MIPS design was computed for each factorial input of *N* and depicted in *Figure 1.15* using an excel sheet. The below equations are calculated by first setting up a few data points and then using interpolations techniques to get the approximated equations.

|  |  |
| --- | --- |
| Formula for Factorial Accelerator | y = 2n+2 |
| Formula for Single-Cycle MIPS | y = −3.5 + 14⋅x |

The *Figure 1.15* below shows the graphical representation of the runtime cycles taken by the single-cycle processor to perform the factorial computation and runtime cycles taken by the factorial accelerator to do the same.

Chart, line chart

Description automatically generated

*Figure1.15 – Runtime cycles taken by each for the factorial computation*

Similar procedure has been done to get the runtime cycles taken by the pipelined MIPS processor. The *Figure 1.16* below shows the graphical representation of the runtime cycles taken by the single-cycle processor to perform the factorial computation and runtime cycles taken by the factorial accelerator to do the same. (Note the scale change in the below graph)

Chart, line chart

Description automatically generated

*Figure1.16 – Runtime cycles taken by each for the factorial computation*

## Simulation Waveforms (with Unit-Level simulations):

To functionally verify the designs of our factorial accelerator with interface wrapper, GPIO unit with interface wrapper, and pipelined SoC, me and my teammate wrote testbenches that stimulate the input of the respective top module and verified the outputs were as expected. In cases where the design did not produce the expected results, we debugged our design by probing more signals to find errors in our logic. This section is divided by the testbenches we created to test each major component of our SoC.

### Testbench for the Factorial Accelerator Interface Wrapper:

Testing the factorial accelerator with an interface wrapper involved writing a testbench that tested values for n from 0 to 13. This would ensure that our design was tested with edge cases where n = 0, 12, with normal values, and where n > 12. Our testbench instantiation of the factorial accelerator with interface wrapper as the design under test can be seen below. It then iterates from 0 to 13. In each loop iteration, the write data port (WD) is given an input based on the iteration number (i.e. n = i = 0, 1, 2…). Then the address input to the module is changed through all possible values as given in *Table 1.8* with a clock tick between each change. In this way, we examine all values of n. The following waveforms show verification of our design. This testbench is an eyeballing testbench where we must examine the waveforms to verify the design.

Graphical user interface

Description automatically generated

*Figure1.17 – Waveform of n=4 for the Factorial Accelerator Wrapper*

Graphical user interface, application

Description automatically generated

*Figure1.18 – Waveform of n=5 for the Factorial Accelerator Wrapper*

As shown in the *figures 1.17 and 1.18*, RD\_tb shows the output of the factorial accelerator interface wrapper. When A = 0, WD\_tb gives n = 4 to the wrapper, which is stored in a register. When A = 1, the Go bit is given to the wrapper, which is stored in a register. When A = 2, the calculation takes place until RD\_tb shows 1 indicating the Done bit is set. Finally, when A = 3, the result is shown on RD\_tb as 0x18H indicating the correct output of 4!=24d.

### Testbench for the GPIO Unit Interface Wrapper:

Testing of the GPIO unit with interface wrapper involved writing a testbench that tested the inputs of the GPIO interface module and viewing the outputs with an eyeball test. As the design of the GPIO interface wrapper was much simpler than the design of the factorial accelerator with interface wrapper, it follows that the testing of GPIO unit with interface wrapper was much simpler. To test, we gave the gpI1 and gpI2 inputs a value and examined the outputs of RD as we changed the input address (A) between all possible values as shown in *Table 1.11*. We also examined the gpO1 and gpO2 outputs. The waveform of this functional verification is shown in *Figure 1.19*.

Graphical user interface

Description automatically generated

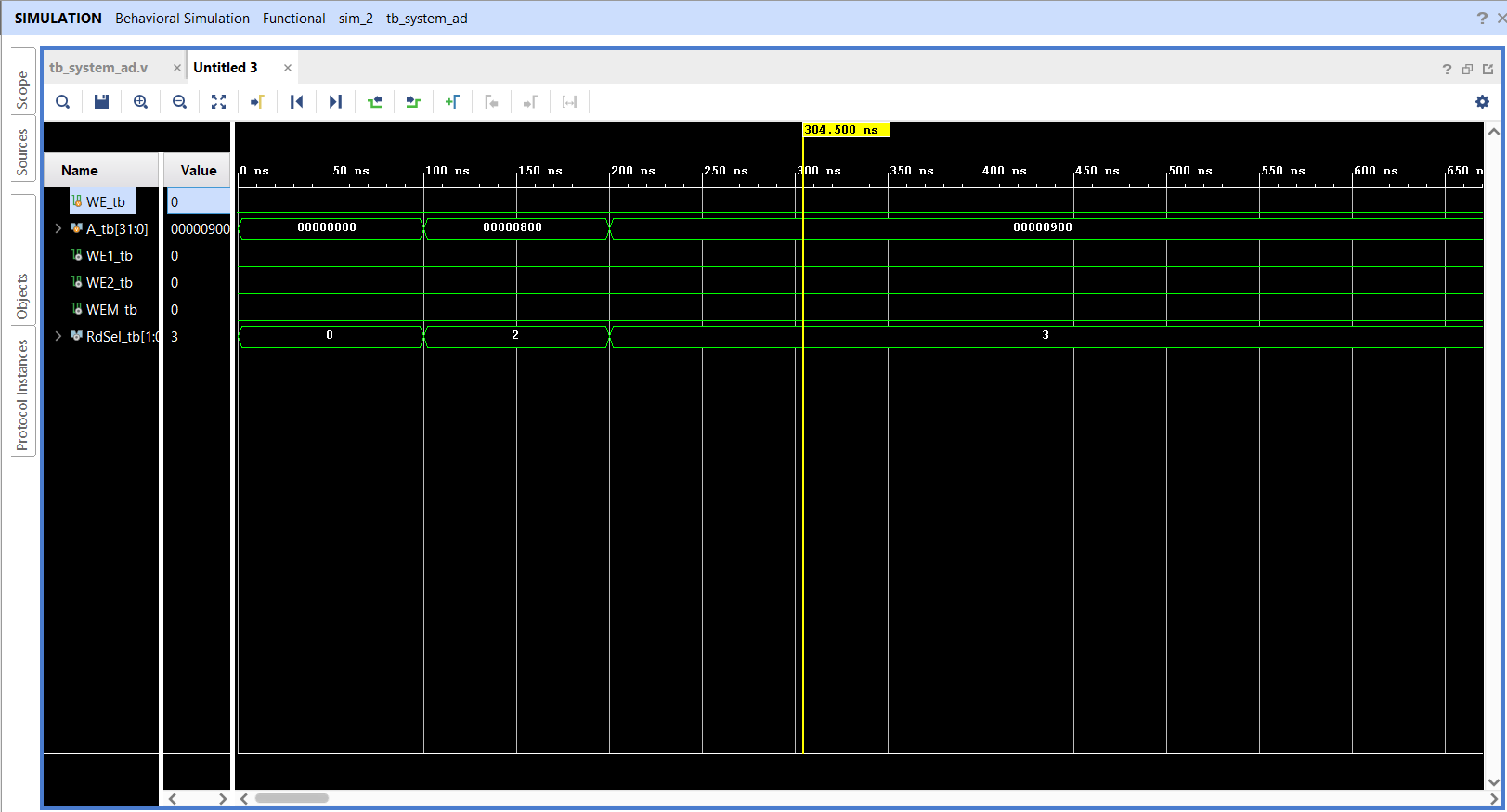
*Figure1.19 – Waveform of GPIO Unit Interface Wrapper after testing*

To safely confirm that our design works as intended, RD must output the following: the value of gpI1 when A = 0, the value of gpI2 when A = 1, the value of WD when A = 2, the value of WD when A = 3. Furthermore, gpO1 must output the value of WD after A = 2 and gpO2 must output the value of WD after A = 3. As seen in the waveform output, this holds true. Note that output data doesn’t appear right after A changes. This is because the registers internal to the interface wrapper do not store input values until given a write enable signal determined by the value of A and a high clock pulse.

### Testbench for the System-on-Chip (SoC) Address Decoder:

Testing of the SoC address decoder involved writing a testbench which takes address as input and gives out the outputs (RdSel) with an eyeball test. This decoder was also relatively simple to design and test. To test, we gave the address as input value and examined the outputs of RdSel as we changed the input address (A) between all possible values i.e., (0x0000, 0x0800, 0x0900).

To safely confirm that our design works as intended, RdSel must output the following: the value when A[3:2] = 00 must be decoded to RdSel = 0, the value when A[3:2] = 10 must be decoded to RdSel = 2, the value when A[3:2] = 11 must be decoded to RdSel = 3. The waveform of this functional verification is shown in *Figure 1.20*.



*Figure1.20 – Waveform of System Address Decoder after testing*

### Testbench for the Pipelined System-on-Chip (SoC) Design:

The functional verification of the pipelined system-on-chip proved to be the most challenging. There were so many modifications to the design of the CPU that it became extremely cumbersome to debug issues as they continued to pop up. Nevertheless, we created a working pipelined MIPS processor. We gave the pipelined SoC an input for *n* on gpI1 and Go signal on gpI2We also added and removed new probes to debug the processor along the way. The GPIO output ports (gpO1 and gpO2) were examined for the correct result of the factorial calculation and Error and Done bits, respectively. The memory file for this simulation was lightly modified with NOPs between instructions with data dependencies as described earlier in the Design Methodology section. Execution stops when the processor reaches the final instruction. The following waveforms shows the output of the whole SoC with different ‘n’ values.

The below waveform in *Figure 1.21* output shows the entire execution process of the pipelined MIPS processor. Of note is the input port gpI1 = 0x4 with n = 4 and the output port gpO2 with n! = 0x18 = 24. Apart from that, the Go signal is being given to the system by the user to start the factorial computation.

**Graphical user interface

Description automatically generated**

*Figure1.21 – Waveform of SoC when n=4 after testing*

The below waveform in *Figure 1.22* output shows the entire execution process of the pipelined MIPS processor. Of note is the input port gpI1 = 0x05 with n = 5 and the output port gpO2 with n! = 0x78 = 24. Apart from that, the Go signal is being given to the system by the user to start the factorial computation.

Graphical user interface

Description automatically generated

*Figure1.22 – Waveform of SoC when n=5 after testing*

# 

# COLLABORATION SECTION:

1. Divided the work of individual module design such as Harish worked on the design of Interface Wrappers, and I worked on the design of the Pipelined MIPS processor.
2. Worked together on the pipelined MIPS Datapath block diagram designs using Microsoft Visio tool.
3. Worked together to divide the control unit decoder truth tables into stages and inserted values into them using an excel sheet.
4. By collaborating with each other, started implementing the discussed logic in Vivado and observed the output waveforms after the simulation by writing a testbench for all major components used in the SoC design.
5. Worked together to devise the assembly code to let the pipelined processor call the corresponding interface wrappers and the factorial accelerator.
6. Collaborated in extracting the memfile.dat file from the MARS simulator using Dump to memory option.

# CONCLUSION:

In conclusion, we end the report by converting the fifteen-instruction 32-bit single-cycle MIPS processor into a five-stage, pipelined design. We also were able to interface the pipelined processor with a factorial accelerator and a GPIO unit using the memory-mapped interface registers. Apart from that, we also attempted to use data forwarding technique to overcome few data hazards but where unsuccessful in doing that. A performance analysis is also done between the runtime cycles of single-cycle, pipelined and factorial accelerator while performing the factorial computation using recursive assembly code. Finally, implemented few testbenches to test all the major components in the SoC and completed the overall design of the system. Even-though it was a tough assignment, we stuck to the tasks and the lecture notes with some help from the internet (YouTube) to successfully complete the assignment.

# APPENDIX

## Schematic of the Data Memory module used in the pipelined processor:

**Shape

Description automatically generated with medium confidence**

## Schematic of the Instruction Memory module used in the pipelined processor:

**Shape

Description automatically generated with medium confidence**

## Schematic of the Control unit used in the pipelined MIPS processor:

**A picture containing night sky

Description automatically generated**

## Datapath of the pipelined MIPS processor:

**Graphical user interface

Description automatically generated**

Final schematic of pipelined MIPS processor. Modules in red are modules added to support new instructions in Assignment 7. Vertical rectangle modules are pipeline registers which are shown in the above diagram with yellow fill. A larger, horizontal view is shown in the next page.

Graphical user interface

Description automatically generated

## Control unit Truth table for the pipelined processor (Decode Stage):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Instruction** | **Input** | | **Decode** |
| *opcode [31:26]* | *funct [5:0]* | *branchD* |
| R | MULTU | 00\_0000 | 01\_1000 | 0 |
| MFHI | 00\_0000 | 01\_0000 | 0 |
| MFLO | 00\_0000 | 01\_0010 | 0 |
| JR | 00\_0000 | 00\_1000 | 0 |
| SLL | 00\_0000 | 00\_0000 | 0 |
| SRL | 00\_0000 | 00\_0010 | 0 |
| J | JAL | 00\_0011 | X | 0 |
|  |  |  |  |  |
| R | ADD | 00\_0000 | 10\_0000 | 0 |
| SUB | 00\_0000 | 10\_0010 | 0 |
| AND | 00\_0000 | 10\_0100 | 0 |
| OR | 00\_0000 | 10\_0101 | 0 |
| SLT | 00\_0000 | 10\_1010 | 0 |
| I | LW | 10\_0011 | X | 0 |
| SW | 10\_1011 | X | 0 |
| BEQ | 00\_0100 | X | 1 |
| ADDI | 00\_1000 | X | 0 |
| J | J | 00\_0010 | X | 0 |

## Control unit Truth table for the pipelined processor (Execute Stage):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Type** | **Instruction** | **Input** | | **Execute** | | | | | |
| *opcode [31:26]* | *funct [5:0]* | *alu\_ctrlE* | *alu\_op* | *alu\_srcE* | *reg\_dstE* | *jal\_wa\_muxE* |
| R | MULTU | 00\_0000 | 01\_1000 | xxxx | 10 | 0 | 1 | 0 |
| MFHI | 00\_0000 | 01\_0000 | xxxx | 10 | 0 | 1 | 0 |
| MFLO | 00\_0000 | 01\_0010 | xxxx | 10 | 0 | 1 | 0 |
| JR | 00\_0000 | 00\_1000 | xxxx | 10 | 0 | 1 | 0 |
| SLL | 00\_0000 | 00\_0000 | 1001 | 10 | 0 | 1 | 0 |
| SRL | 00\_0000 | 00\_0010 | 1010 | 10 | 0 | 1 | 0 |
| J | JAL | 00\_0011 | X | X | 00 | 0 | 0 | 1 |
|  |  |  |  |  |  |  |  |  |
| R | ADD | 00\_0000 | 10\_0000 | 0010 | 10 | 0 | 1 | 0 |
| SUB | 00\_0000 | 10\_0010 | 0110 | 10 | 0 | 1 | 0 |
| AND | 00\_0000 | 10\_0100 | 0010 | 10 | 0 | 1 | 0 |
| OR | 00\_0000 | 10\_0101 | 0001 | 10 | 0 | 1 | 0 |
| SLT | 00\_0000 | 10\_1010 | 0111 | 10 | 0 | 1 | 0 |
| I | LW | 10\_0011 | X | X | 00 | 1 | 0 | 0 |
| SW | 10\_1011 | X | X | 00 | 1 | 0 | 0 |
| BEQ | 00\_0100 | X | X | 01 | 0 | 0 | 0 |
| ADDI | 00\_1000 | X | X | 00 | 1 | 0 | 0 |
| J | J | 00\_0010 | X | X | 00 | 0 | 0 | 0 |

## Control unit Truth table for the pipelined processor (Memory Stage):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Instruction** | **Input** | | **Memory** | |
| *opcode [31:26]* | *funct [5:0]* | *hilo\_weM* | *we\_dmM* |
| R | MULTU | 00\_0000 | 01\_1000 | 1 | 0 |
| MFHI | 00\_0000 | 01\_0000 | 0 | 0 |
| MFLO | 00\_0000 | 01\_0010 | 0 | 0 |
| JR | 00\_0000 | 00\_1000 | 0 | 0 |
| SLL | 00\_0000 | 00\_0000 | 0 | 0 |
| SRL | 00\_0000 | 00\_0010 | 0 | 0 |
| J | JAL | 00\_0011 | X | 0 | 0 |
|  |  |  |  |  |  |
| R | ADD | 00\_0000 | 10\_0000 | 0 | 0 |
| SUB | 00\_0000 | 10\_0010 | 0 | 0 |
| AND | 00\_0000 | 10\_0100 | 0 | 0 |
| OR | 00\_0000 | 10\_0101 | 0 | 0 |
| SLT | 00\_0000 | 10\_1010 | 0 | 0 |
| I | LW | 10\_0011 | X | 0 | 0 |
| SW | 10\_1011 | X | 0 | 1 |
| BEQ | 00\_0100 | X | 0 | 0 |
| ADDI | 00\_1000 | X | 0 | 0 |
| J | J | 00\_0010 | X | 0 | 0 |

## Control unit Truth table for the pipelined processor (Writeback Stage):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Type** | **Instr** | **Input** | | **Writeback** | | | | | | |
| *opcode [31:26]* | *funct [5:0]* | | *jr\_mux\_ctrlW* | *jumpW* | *hilo\_mux\_ctrlW* | *dm2regW* | *we\_regW* | *jal\_wd\_muxW* |
| R | MULTU | 00\_0000 | 01\_1000 | | 0 | 0 | 00 | 0 | 1 | 0 |
| MFHI | 00\_0000 | 01\_0000 | | 0 | 0 | 11 | 0 | 1 | 0 |
| MFLO | 00\_0000 | 01\_0010 | | 0 | 0 | 01 | 0 | 1 | 0 |
| JR | 00\_0000 | 00\_1000 | | 1 | 0 | 00 | 0 | 0 | 0 |
| SLL | 00\_0000 | 00\_0000 | | 0 | 0 | 00 | 0 | 1 | 0 |
| SRL | 00\_0000 | 00\_0010 | | 0 | 0 | 00 | 0 | 1 | 0 |
| J | JAL | 00\_0011 | X | | 0 | 1 | 00 | 0 | 1 | 1 |
|  |  |  |  | |  |  |  |  |  |  |
| R | ADD | 00\_0000 | 10\_0000 | | 0 | 0 | 00 | 0 | 1 | 0 |
| SUB | 00\_0000 | 10\_0010 | | 0 | 0 | 00 | 0 | 1 | 0 |
| AND | 00\_0000 | 10\_0100 | | 0 | 0 | 00 | 0 | 1 | 0 |
| OR | 00\_0000 | 10\_0101 | | 0 | 0 | 00 | 0 | 1 | 0 |
| SLT | 00\_0000 | 10\_1010 | | 0 | 0 | 00 | 0 | 1 | 0 |
| I | LW | 10\_0011 | X | | 0 | 0 | 00 | 1 | 1 | 0 |
| SW | 10\_1011 | X | | 0 | 0 | 00 | 0 | 0 | 0 |
| BEQ | 00\_0100 | X | | 0 | 0 | 00 | 0 | 0 | 0 |
| ADDI | 00\_1000 | X | | 0 | 0 | 00 | 0 | 1 | 0 |
| J | J | 00\_0010 | X | | 0 | 1 | 00 | 0 | 0 | 0 |

## Source Code of tb\_system.v:

module tb\_system;

reg clk;

reg rst;

reg [31:0] gpi1\_var;

reg [31:0] gpi2\_var;

wire [31:0] gpO1\_var;

wire [31:0] gpO2\_var;

reg [3:0] n\_tb;

integer i;

system DUT(

.clk(clk),

.reset(rst),

.gpi1(gpi1\_var),

.gpi2(gpi2\_var),

.gpO1(gpO1\_var),

.gpO2(gpO2\_var)

);

task tick;

begin

clk = 1'b0; #5;

clk = 1'b1; #5;

end

endtask

task reset;

begin

rst = 1'b0; #5;

rst = 1'b1; #5;

rst = 1'b0;

end

endtask

initial begin

reset;

n\_tb = 4'd1;

gpi1\_var = 32'h4;

gpi2\_var = 32'h1;

for(i=0;i<30;i=i+1)

begin

tick;

end

$finish;

end

endmodule

## Source Code of tb\_fact\_top.v:

module tb\_fact\_top;

reg [1:0] A\_tb;

reg WE\_tb;

reg [3:0] WD\_tb;

reg Rst\_tb;

reg Clk\_tb;

wire [31:0] RD\_tb;

integer i;

task tick;

begin

Clk\_tb = 0; #1;

Clk\_tb = 1; #1;

end

endtask

fact\_top DUT1 (

.A(A\_tb),

.WE(WE\_tb),

.WD(WD\_tb),

.Rst(Rst\_tb),

.clk(Clk\_tb),

.RD(RD\_tb)

);

initial begin

// Reset fact\_top module

Rst\_tb = 1;

tick;

Rst\_tb = 0;

for (i=0; i<14; i=i+1) begin

// Load n and assert WE

WD\_tb = i;

WE\_tb = 1;

A\_tb = 2'b00;

tick;

// Give go signal

WD\_tb = 1;

A\_tb = 2'b01;

tick;

// Monitor Control Output (Done, Err)

A\_tb = 2'b10;

tick;

while (RD\_tb == 0) begin

tick;

end

// Examine result if Done = 1

if(RD\_tb == 1) begin

// Examine Result

A\_tb = 2'b11;

tick;

end else if(RD\_tb == 2) begin

$display("Error bit triggered for n = %d", i);

tick;

end

// Reset factorial module

Rst\_tb = 0;

WD\_tb = 0;

A\_tb = 0;

tick;

end

end

endmodule

## Source Code of tb\_gpio\_top.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 12/01/2022 12:10:16 AM

// Design Name:

// Module Name: tb\_gpio\_top

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module tb\_gpio\_top;

//inputs

reg [1:0] A\_tb;

reg WE\_tb;

reg [31:0] gpI1\_tb;

reg [31:0] gpI2\_tb;

reg [31:0] WD\_tb;

reg rst\_tb;

reg clk\_tb;

//outputs

wire [31:0] RD\_tb;

wire [31:0] gpO1\_tb;

wire [31:0] gpO2\_tb;

gpio\_top DUT(

.A(A\_tb),

.WE(WE\_tb),

.gpi1(gpI1\_tb),

.gpi2(gpI2\_tb),

.WD(WD\_tb),

.RST(rst\_tb),

.CLK(clk\_tb),

.RD(RD\_tb),

.gpo1(gpO1\_tb),

.gpo2(gpO2\_tb)

);

task tick;

begin

clk\_tb = 0; #5; // Set bit 1 --> 0 after 5 time units

clk\_tb = 1; #5;

end

endtask

initial begin

begin

// Reset GPIO module

rst\_tb = 1;

tick;

rst\_tb = 0;

// Set up GPIO module

WE\_tb = 1'b1;

// Give test values to input

WD\_tb = 3;

gpI1\_tb = 5;

gpI2\_tb = 7;

// Test all values for A

A\_tb = 2'b00;

tick;

A\_tb = 2'b01;

tick;

A\_tb = 2'b10;

tick;

A\_tb = 2'b11;

tick;

end

end

endmodule

## Source Code of tb\_system\_ad.v:

module tb\_system\_ad;

reg WE\_tb;

reg [31:0] A\_tb;

wire WE1\_tb, WE2\_tb, WEM\_tb;

wire [1:0] RdSel\_tb;

system\_AD DUT1 (

.WE(WE\_tb),

.A(A\_tb),

.WE1(WE1\_tb),

.WE2(WE2\_tb),

.WEM(WEM\_tb),

.RdSel(RdSel\_tb)

);

initial begin

WE\_tb = 0;

A\_tb = 32'h000; #100;

A\_tb = 32'h800; #100;

A\_tb = 32'h900; #100;

end

endmodule

## Source Code of tb\_mips\_pipelined\_top.v:

module tb\_mips\_pipelined\_top;

reg clk;

reg rst;

wire we\_dm;

wire [31:0] pc\_current;

wire [31:0] instr;

wire [31:0] alu\_out;

wire [31:0] wd\_dm;

wire [31:0] rd\_dm;

wire [31:0] DONT\_USE;

integer i;

mips\_pipelined\_top DUT(

.clk(clk),

.rst(rst),

.ra3(5'h0),

.we\_dm(we\_dm),

.pc\_current(pc\_current),

.instr(instr),

.alu\_out(alu\_out),

.wd\_dm(wd\_dm),

.rd\_dm(rd\_dm),

.rd3(DONT\_USE)

);

task tick;

begin

clk = 1'b0; #5;

clk = 1'b1; #5;

end

endtask

task reset;

begin

rst = 1'b0; #5;

rst = 1'b1; #5;

rst = 1'b0;

end

endtask

initial begin

reset;

for(i=0;i<300;i=i+1)

begin

tick;

end

reset;

while(pc\_current != 32'hb4) tick;

$finish;

end

endmodule

## Source Code of system.v:

module system(

input clk,

input reset,

input [31:0] gpi1,

input [31:0] gpi2,

output [31:0] gpO1,

output [31:0] gpO2

);

wire [31:0] PC;

wire [31:0] Instruction;

wire MemWrite;

//wire [31:0] address;

wire [31:0] WriteData;

wire [31:0] DMemData;

wire [31:0] FactData;

wire [31:0] GPIOData;

wire [31:0] ReadData;

wire WE1;

wire WE2;

wire WEM;

wire [1:0] RdSel;

imem iMemory(

.a(PC[7:2]),

.y(Instruction)

);

wire [4:0] dummy5;

wire [31:0] alu\_out;

wire [31:0] dummy;

//wire [31:0] address;

mips\_pipelined MIPS(

.clk(clk),

.rst(reset),

.ra3(dummy5),

.we\_dm(MemWrite),

.pc\_current(PC),

.instr(Instruction),

.alu\_out(alu\_out),

.wd\_dm(WriteData),

.rd\_dm(ReadData),

.rd3(dummy)

//.address(address)

);

dmem dMemory(

.clk(clk),

.we(WEM),

.a(alu\_out[7:2]),

.d(WriteData),

.q(DMemData)

);

fact\_top fact(

.A(alu\_out[3:2]),

.WE(WE1),

.WD(WriteData[3:0]),

.Rst(reset),

.clk(clk),

.RD(FactData)

);

gpio\_top gpio(

.A(alu\_out[3:2]),

.WE(WE2),

.gpi1(gpi1),

.gpi2(gpi2),

.WD(WriteData),

.RST(reset),

.CLK(clk),

.RD(GPIOData),

.gpo1(gpO1),

.gpo2(gpO2)

);

system\_AD AD(

.WE(MemWrite),

.A(alu\_out),

.WEM(WEM),

.WE1(WE1),

.WE2(WE2),

.RdSel(RdSel)

);

mux4 #(32) rdDM\_mux(

.sel(RdSel),

.a(DMemData),

.b(DMemData),

.c(FactData),

.d(GPIOData),

.y(ReadData)

);

endmodule

## Source Code of system\_AD.v:

module system\_AD(

input WE,

input [31:0] A,

output reg WEM,

output reg WE1,

output reg WE2,

output reg [1:0] RdSel

);

always @ (\*) begin

// Address range in if statements based on memory map table

if (A >= 32'h0 && A <= 32'hFC) begin

WEM = WE;

WE1 = 0;

WE2 = 0;

RdSel = 2'b00;

end

else if (A >= 32'h800 && A <= 32'h80C) begin

WEM = 0;

WE1 = WE;

WE2 = 0;

RdSel = 2'b10;

end

else if (A >= 32'h900 && A <= 32'h90C) begin

WEM = 0;

WE1 = 0;

WE2 = WE;

RdSel = 2'b11;

end

else begin

WEM = 0;

WE1 = 0;

WE2 = 0;

RdSel = 2'b00;

end

end

endmodule

## Source Code of fact\_top.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:49:51 PM

// Design Name:

// Module Name: fact\_top

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module fact\_top(

input [1:0] A,

input WE,

input [3:0] WD,

input Rst,

input clk,

output [31:0] RD

);

wire [1:0] RdSel;

wire WE1, WE2;

wire [3:0] n;

wire GoPulseCmb, GoPulse;

wire Go;

wire [31:0] nf;

wire fact\_done, fact\_err;

wire [31:0] result;

wire ResDone, ResErr;

and\_fact #(1) and\_fact (

.a (WE2),

.b (WD[0]),

.c (GoPulseCmb)

);

fact\_ad address\_decoder (

.A (A),

.WE (WE),

.WE2 (WE2),

.WE1 (WE1),

.RdSel (RdSel)

);

fact\_reg #(4) n\_reg (

.Rst (Rst),

.Clk (clk),

.Load\_Reg (WE1),

.D (WD),

.Q (n)

);

fact\_reg #(1) go\_reg (

.Rst (Rst),

.Clk (clk),

.Load\_Reg (WE2),

.D (WD[0]),

.Q (Go)

);

fact\_reg #(1) go\_pulse\_reg (

.Rst (Rst),

.Clk (clk),

.Load\_Reg (1'b1),

.D (GoPulseCmb),

.Q (GoPulse)

);

//fact fact (); // TODO

FSMmult #(32) fact (

.D ({28'b0,n}),

.GO (GoPulse),

.RST (Rst),

.CLK (clk),

.doneF (fact\_done),

.ERROR (fact\_err),

.out (nf)

);

fact\_reg #(32) result\_reg (

.Rst (Rst),

.Clk (clk),

.Load\_Reg (fact\_done),

.D (nf),

.Q (result)

);

fact\_res\_done\_reg fact\_res\_done\_reg(

.Clk (clk),

.Rst (Rst),

.GoPulseCmb (GoPulseCmb),

.Done (fact\_done),

.ResDone (ResDone)

);

fact\_res\_err\_reg fact\_res\_err\_reg (

.Clk (clk),

.Rst (Rst),

.GoPulseCmb (GoPulseCmb),

.Err (fact\_err),

.ResErr (ResErr)

);

mux4 #(32) fact\_mux (

.sel (RdSel),

.a ({28'b0,n}),

.b ({31'b0,Go}),

.c ({30'b0,ResErr,ResDone}),

.d (result),

.y (RD)

);

endmodule

## Source Code of the gpio\_top.v:

module gpio\_top #(parameter WIDTH=32)(

input wire [1:0] A,

input wire WE,

input wire [WIDTH -1:0] gpi1, //gpi1 not L or I

input wire [WIDTH -1:0] gpi2,

input wire [WIDTH -1:0] WD,

input wire RST,

input wire CLK,

output wire [WIDTH -1:0] RD,

output wire [WIDTH -1:0] gpo1,

output wire [WIDTH -1:0] gpo2

);

wire WE1;

wire WE2;

wire [1:0] RdSel;

gpio\_ad gpio\_ad(

.A (A),

.WE (WE),

.WE1 (WE1),

.WE2 (WE2),

.RdSel (RdSel)

);

//gpo1

fact\_reg #(32) gpo1\_reg(

.Clk(CLK),

.Rst(RST),

.D(WD),

.Load\_Reg(WE1),

.Q(gpo1)

);

//gpo2

fact\_reg #(32) gpo2\_reg(

.Clk(CLK),

.Rst(RST),

.D(WD),

.Load\_Reg(WE2),

.Q(gpo2)

);

mux4 #(32) mux\_out(

.sel(RdSel),

.a (gpi1),

.b (gpi2),

.c (gpo1),

.d (gpo2),

.y (RD)

);

endmodule

## Source Code of mux4.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer: Tirumala Saiteja Goruganthu and Harish Marepalli

//

// Create Date: 11/30/2022 10:53:29 PM

// Design Name:

// Module Name: mux4

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module mux4 #(parameter WIDTH = 8) (

input wire [1:0] sel,

input wire [WIDTH-1:0] a,

input wire [WIDTH-1:0] b,

input wire [WIDTH-1:0] c,

input wire [WIDTH-1:0] d,

output reg [WIDTH-1:0] y

);

always @ (\*)

begin

case (sel)

2'b00: y <= a;

2'b01: y <= b;

2'b10: y <= c;

2'b11: y <= d;

default: y <= {WIDTH{1'bx}};

endcase

// assign y = (sel) ? b : a;

end

endmodule

## Source Code of mips\_pipelined\_top.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 12/01/2022 04:11:53 PM

// Design Name:

// Module Name: mips\_pipelined\_top

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module mips\_pipelined\_top(

input wire clk,

input wire rst,

input wire [4:0] ra3,

input wire [31:0] instr,

input wire [31:0] rd\_dm,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3

);

wire [31:0] DONT\_USE;

mips\_pipelined mips\_pipelined(

.clk(clk),

.rst(rst),

.ra3(ra3),

.we\_dm(we\_dm),

.pc\_current(pc\_current),

.instr(instr),

.alu\_out(alu\_out),

.wd\_dm(wd\_dm),

.rd\_dm(rd\_dm),

.rd3(rd3)

);

imem iMemory(

.a(pc\_current[7:2]),

.y(instr)

);

dmem dMemory(

.clk(clk),

.we(we\_dm),

.a(alu\_out[7:2]),

.d(wd\_dm),

.q(rd\_dm)

);

endmodule

## Source Code of mips\_pipelined.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer: Tirumala Saiteja Goruganthu and Harish Marepalli

//

// Create Date: 12/01/2022 04:26:16 PM

// Design Name:

// Module Name: mips\_pipelined

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module mips\_pipelined(

input wire clk,

input wire rst,

input wire [4:0] ra3,

input wire [31:0] instr,

input wire [31:0] rd\_dm,

output wire we\_dm,

output wire [31:0] pc\_current,

output wire [31:0] alu\_out,

output wire [31:0] wd\_dm,

output wire [31:0] rd3,

output wire [31:0] address

);

/\* Wires \*/

/\* FETCH STAGE \*/

wire [31:0] pc\_currentF;

wire [31:0] pc\_plus4F;

/\* DECODE STAGE \*/

wire pc\_srcD;

wire [31:0] instrD;

wire [31:0] pc\_plus4D;

wire [31:0] btaD;

wire [31:0] rd1D;

wire [31:0] rd2D;

wire [31:0] wdD;

wire [31:0] jtaD;

/\* CU \*/

wire branchD;

wire jumpD;

wire reg\_dstD;

wire we\_regD;

wire alu\_srcD;

wire we\_dmD;

wire dm2regD;

wire [3:0] alu\_ctrlD;

wire [1:0] hilo\_mux\_ctrlD;

wire hilo\_weD;

wire jr\_mux\_ctrlD;

wire jal\_wd\_muxD;

wire jal\_wa\_muxD;

wire [31:0] sext\_immD;

wire [4:0] instrD\_20\_16;

wire [4:0] instrD\_15\_11;

wire [4:0] shamtD;

wire [31:0] addressD;

/\* EXECUTE STAGE \*/

wire [31:0] wdE;

wire jr\_mux\_ctrlE;

wire jumpE;

wire hilo\_weE;

wire [1:0] hilo\_mux\_ctrlE;

wire dm2regE;

wire we\_dmE;

wire branchE; //doubt

wire [3:0] alu\_ctrlE;

wire alu\_srcE;

wire reg\_dstE;

wire we\_regE;

wire jal\_wa\_muxE;

wire jal\_wd\_muxE;

wire [31:0] rd1\_outE;

wire [31:0] rd2\_outE;

wire [4:0] instrE\_20\_16;

wire [4:0] instrE\_15\_11;

wire [31:0] pc\_plus4E; //doubt

wire [4:0] shamtE;

wire [31:0] sext\_immE;

wire [31:0] alu\_outE;

wire [31:0] addressE;

wire [31:0] jtaE;

wire [31:0] jrE;

wire [4:0] waE;

wire [31:0] mult\_hiE;

wire [31:0] mult\_loE;

wire [31:0] wd\_dmE;

assign wd\_dmE = rd2\_outE;

//assign wd\_dm = wd\_dmE;

assign alu\_out = alu\_outM;

/\* MEMORY STAGE \*/

//wire pc\_srcM;

// wire [31:0] btaM;

wire jr\_mux\_ctrlM;

wire jumpM;

wire hilo\_weM;

wire [1:0] hilo\_mux\_ctrlM;

wire dm2regM;

wire we\_dmM;

wire we\_regM;

wire [31:0] alu\_outM;

wire [31:0] wd\_dmM;

wire [31:0] mult\_hiM;

wire [31:0] mult\_loM;

wire [31:0] jrM;

wire [4:0] waM;

wire jal\_wa\_muxM;

wire jal\_wd\_muxM;

wire [31:0] rd\_dmM;

wire [31:0] hi\_outM;

wire [31:0] lo\_outM;

wire [31:0] jtaM;

/\* WRITEBACK STAGE \*/

wire [31:0] pc\_nextW;

wire jr\_mux\_ctrlW;

wire jumpW;

wire [31:0] jrW;

wire we\_regW;

wire [4:0] waW;

wire [31:0] jtaW;

wire jal\_wd\_muxW;

wire [31:0] hilo\_mux\_outW;

wire [1:0] hilo\_mux\_ctrlW;

wire dm2regW;

wire [31:0] alu\_outW;

wire [31:0] hi\_outW;

wire [31:0] lo\_outW;

wire [31:0] wd\_rfW;

wire [31:0] rd\_dmW;

assign pc\_current = pc\_currentF;

/\* FETCH STAGE \*/

wire [31:0] pc\_pc\_src\_mux;

wire [31:0] pc\_jump\_mux;

//for branches

mux2 #(32) pc\_src\_mux (

.sel (pc\_srcD),

.a (pc\_plus4D), //TODO: double check that this shouldn't be 27:0

.b (btaD),

.y (pc\_pc\_src\_mux)

);

mux2 #(32) jump\_mux (

.sel (jumpW),

.a (pc\_pc\_src\_mux),

.b (jtaW),

.y (pc\_jump\_mux)

);

mux2 #(32) jr\_mux (

.sel (jr\_mux\_ctrlW),

.a (pc\_jump\_mux),

.b (jrW),

.y (pc\_nextW)

);

dreg #(32) pc\_reg (

.clk (clk),

.rst (rst),

.d (pc\_nextW),

.q (pc\_currentF)

);

//imem

adder pc\_plus4 (

.a (pc\_currentF),

.b (32'd4),

.y (pc\_plus4F)

);

/\* D Stage Reg Interface \*/

D\_Stage\_Reg D\_Stage\_Reg (

.clk (clk),

.rst (rst),

.instrF (instr),

.pc\_plus4F (pc\_plus4F),

.instrD (instrD),

.pc\_plus4D (pc\_plus4D)

);

/\* DECODE STAGE \*/

controlunit cu (

.opcode (instrD[31:26]),

.funct (instrD[5:0]),

.branch (branchD),

.jump (jumpD),

.reg\_dst (reg\_dstD),

.we\_reg (we\_regD),

.alu\_src (alu\_srcD),

.we\_dm (we\_dmD),

.dm2reg (dm2regD),

.alu\_ctrl (alu\_ctrlD),

.hilo\_mux\_ctrl (hilo\_mux\_ctrlD),

.hilo\_we (hilo\_weD),

.jr\_mux\_ctrl (jr\_mux\_ctrlD),

.jal\_wd\_mux\_sel (jal\_wd\_muxD),

.jal\_wa\_mux\_sel (jal\_wa\_muxD)

);

regfile rf (

.clk (clk),

.we (we\_regW),

.ra1 (instrD[25:21]),

.ra2 (instrD[20:16]),

.ra3 (ra3),

.wa (waW),

.wd (wdD),

.rd1 (rd1D),

.rd2 (rd2D),

.rd3 () //TODO not sure if we're using this

);

signext se (

.a (instrD[15:0]),

.y (sext\_immD)

);

mux2 #(32) jal\_wd\_mux (

.sel (jal\_wd\_muxW),

.a (hilo\_mux\_outW),

.b (pc\_plus4D),

.y (wdD)

);

assign shamtD = instrD[10:6];

assign instrD\_20\_16 = instrD[20:16];

assign instrD\_15\_11 = instrD[15:11];

assign jtaD = {pc\_plus4D[31:28], instrD[25:0], 2'b00}; //TODO not sure about this one.

adder pc\_plus\_br (

.a ({sext\_immD[29:0], 2'b00}),

.b (pc\_plus4D),

.y (btaD)

);

assign pc\_srcD = ((rd1D == rd2D) && branchD) ? 1 : 0;

assign addressD = instrD;

/\* E Stage Reg Interface \*/

E\_Stage\_Reg E\_Stage\_Reg (

.clk (clk),

.rst (rst),

.jr\_mux\_ctrlD (jr\_mux\_ctrlD),

.jumpD (jumpD),

.hilo\_weD (hilo\_weD),

.hilo\_mux\_ctrlD (hilo\_mux\_ctrlD),

.dm2regD (dm2regD),

.we\_dmD (we\_dmD),

//.branchD (branchD),

.alu\_ctrlD (alu\_ctrlD),

.alu\_srcD (alu\_srcD),

.reg\_dstD (reg\_dstD),

.we\_regD (we\_regD),

.jal\_wa\_muxD (jal\_wa\_muxD),

.jal\_wd\_muxD (jal\_wd\_muxD),

.rd1D (rd1D),

.rd2D (rd2D),

.instrD\_20\_16 (instrD\_20\_16),

.instrD\_15\_11 (instrD\_15\_11),

.pc\_plus4D (pc\_plus4D),

.shamtD (shamtD),

.sext\_immD (sext\_immD),

.addressD (addressD),

.jtaD (jtaD),

.jr\_mux\_ctrlE (jr\_mux\_ctrlE),

.jumpE (jumpE),

.hilo\_weE (hilo\_weE),

.hilo\_mux\_ctrlE (hilo\_mux\_ctrlE),

.dm2regE (dm2regE),

.we\_dmE (we\_dmE),

//.branchE (branchE),

.alu\_ctrlE (alu\_ctrlE),

.alu\_srcE (alu\_srcE),

.reg\_dstE (reg\_dstE),

.we\_regE (we\_regE),

.jal\_wa\_muxE (jal\_wa\_muxE),

.jal\_wd\_muxE (jal\_wd\_muxE),

.rd1\_outE (rd1\_outE),

.rd2\_outE (rd2\_outE),

.instrE\_20\_16 (instrE\_20\_16),

.instrE\_15\_11 (instrE\_15\_11),

.pc\_plus4E (pc\_plus4E),

.shamtE (shamtE),

.sext\_immE (sext\_immE),

.addressE (addressE),

.jtaE (jtaE)

);

/\* EXECUTE STAGE \*/

wire [31:0] alu\_paE;

wire [31:0] alu\_pbE;

wire zeroE;

wire [4:0] rf\_wa;

assign alu\_paE = rd1\_outE;

mux2 #(32) alu\_pb\_mux (

.sel (alu\_srcE),

.a (rd2\_outE),

.b (sext\_immE),

.y (alu\_pbE)

);

alu alu (

.op (alu\_ctrlE),

.a (alu\_paE),

.b (alu\_pbE),

//.zero (zeroE),

.y (alu\_outE),

.shamt (shamtE)

);

mux2 #(5) rf\_wa\_mux ( //TODO move to D

.sel (reg\_dstE),

.a (instrE\_20\_16),

.b (instrE\_15\_11),

.y (rf\_wa)

);

mux2 #(5) jal\_wa\_mux (

.sel (jal\_wa\_muxE),

.a (rf\_wa),

.b (5'd31),

.y (waE)

);

mult\_inf #(32) mult (

.a (rd1\_outE),

.b (rd2\_outE),

.out ({mult\_hiE, mult\_loE})

);

/\* M Stage Interface \*/

M\_Stage\_Reg M\_Stage\_Reg (

.clk (clk),

.rst (rst),

.jr\_mux\_ctrlE (jr\_mux\_ctrlE),

.jumpE (jumpE),

.hilo\_weE (hilo\_weE),

.hilo\_mux\_ctrlE (hilo\_mux\_ctrlE),

.dm2regE (dm2regE),

.we\_dmE (we\_dmE),

.we\_regE (we\_regE),

.alu\_outE (alu\_outE),

.wd\_dmE (wd\_dmE),

.mult\_hiE (mult\_hiE),

.mult\_loE (mult\_loE),

.jrE (jrE),

.waE (waE),

.jal\_wa\_muxE (jal\_wa\_muxE),

.jal\_wd\_muxE (jal\_wd\_muxE),

.addressE (addressE),

.jtaE (jtaE),

.jr\_mux\_ctrlM (jr\_mux\_ctrlM),

.jumpM (jumpM),

.hilo\_weM (hilo\_weM),

.hilo\_mux\_ctrlM (hilo\_mux\_ctrlM),

.dm2regM (dm2regM),

.we\_dmM (we\_dmM),

.we\_regM (we\_regM),

.alu\_outM (alu\_outM),

.wd\_dmM (wd\_dm),

.mult\_hiM (mult\_hiM),

.mult\_loM (mult\_loM),

.jrM (jrM),

.waM (waM),

.jal\_wa\_muxM (jal\_wa\_muxM),

.jal\_wd\_muxM (jal\_wd\_muxM),

.addressM (address),

.jtaM (jtaM)

);

/\* MEMORY STAGE \*/

assign alu\_out = alu\_outM;

assign wd\_dm = wd\_dmM;

assign we\_dm = we\_dmM;

assign rd\_dmM = rd\_dm;

//inferred and gate

//connections to data mem

HiLo\_reg #(32) hi\_lo\_reg (

.clk (clk),

.rst (rst),

.we (hilo\_weM),

.hi (mult\_hiM),

.lo (mult\_loM),

.hi\_out (hi\_outM),

.lo\_out (lo\_outM)

);

/\* W Stage Reg Interface \*/

W\_Stage\_Reg W\_Stage\_Reg (

.clk (clk),

.rst (rst),

.jr\_mux\_ctrlM (jr\_mux\_ctrlM),

.jumpM (jumpM),

.hilo\_mux\_ctrlM (hilo\_mux\_ctrlM),

.dm2regM (dm2regM),

.we\_regM (we\_regM),

.alu\_outM (alu\_outM),

.rd\_dmM (rd\_dmM),

.hi\_outM (hi\_outM),

.lo\_outM (lo\_outM),

.jrM (jrM),

.waM (waM),

.jtaM (jtaM),

.jal\_wd\_muxM (jal\_wd\_muxM),

.jr\_mux\_ctrlW (jr\_mux\_ctrlW),

.jumpW (jumpW),

.hilo\_mux\_ctrlW (hilo\_mux\_ctrlW),

.dm2regW (dm2regW),

.we\_regW (we\_regW),

.alu\_outW (alu\_outW),

.rd\_dmW (rd\_dmW),

.hi\_outW (hi\_outW),

.lo\_outW (lo\_outW),

.jrW (jrW),

.waW (waW),

.jtaW (jtaW),

.jal\_wd\_muxW (jal\_wd\_muxW)

);

/\* WRITEBACK STAGE \*/

mux2 #(32) rf\_wd\_mux (

.sel (dm2regW),

.a (alu\_outW),

.b (rd\_dmW),

.y (wd\_rfW)

);

mux4 #(32) hilo\_mux (

.sel (hilo\_mux\_ctrlW),

.a (wd\_rfW),

.b (lo\_outW),

.c (hi\_outW),

.d (32'd0),

.y (hilo\_mux\_outW)

);

endmodule

## Source Code of imem.v:

module imem (

input wire [5:0] a,

output wire [31:0] y

);

reg [31:0] rom [0:63];

initial begin

$readmemh ("H:\\Masters\\Semester1\\CMPE200\\Assignment7\\memfile\_withnops\_updated2.dat", rom);

end

assign y = rom[a];

endmodule

## Source Code of dmem.v:

module dmem (

input wire clk,

input wire we,

//input wire rst,

input wire [5:0] a,

input wire [31:0] d,

output wire [31:0] q

);

reg [31:0] ram [0:63];

integer n;

initial begin

for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;

end

always @ (posedge clk) begin

if (we) ram[a] <= d;

end

assign q = ram[a];

endmodule

## Source Code of mux2.v:

module mux2 #(parameter WIDTH = 8) (

input wire sel,

input wire [WIDTH-1:0] a,

input wire [WIDTH-1:0] b,

output wire [WIDTH-1:0] y

);

assign y = (sel) ? b : a;

endmodule

## Source Code of dreg.v:

module dreg # (parameter WIDTH = 32) (

input wire clk,

input wire rst,

input wire [WIDTH-1:0] d,

output reg [WIDTH-1:0] q

);

always @ (posedge clk, posedge rst) begin

if (rst) q <= 0;

else q <= d;

end

endmodule

## Source Code of adder.v:

module adder (

input wire [31:0] a,

input wire [31:0] b,

output wire [31:0] y

);

assign y = a + b;

endmodule

## Source Code of D\_Stage\_Reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:58:47 PM

// Design Name:

// Module Name: D\_Stage\_Reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module D\_Stage\_Reg(

input clk, rst,

input [31:0] instrF,

input [31:0] pc\_plus4F,

output reg [31:0] instrD,

output reg [31:0] pc\_plus4D

);

always @ (negedge clk, posedge rst) begin

if (rst) begin

instrD <= 0;

pc\_plus4D <= 0;

end

else begin

instrD <= instrF;

pc\_plus4D <= pc\_plus4F;

end

end

endmodule

## Source Code of controlunit.v:

module controlunit (

input wire [5:0] opcode,

input wire [5:0] funct,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire [3:0] alu\_ctrl,

output wire [1:0] hilo\_mux\_ctrl,

output wire hilo\_we,

output wire jr\_mux\_ctrl,

output wire jal\_wd\_mux\_sel,

output wire jal\_wa\_mux\_sel

);

wire [1:0] alu\_op;

wire [1:0] hilo\_mux\_internal;

maindec md (

.opcode (opcode),

.branch (branch),

.jump (jump),

.reg\_dst (reg\_dst),

.we\_reg (we\_reg),

.alu\_src (alu\_src),

.we\_dm (we\_dm),

.dm2reg (dm2reg),

.alu\_op (alu\_op),

.jal\_wa\_mux\_sel (jal\_wa\_mux\_sel),

.jal\_wd\_mux\_sel (jal\_wd\_mux\_sel)

);

auxdec ad (

.alu\_op (alu\_op),

.funct (funct),

.alu\_ctrl (alu\_ctrl),

.hilo\_mux\_ctrl (hilo\_mux\_internal),

.hilo\_we (hilo\_we),

.jr\_mux\_ctrl (jr\_mux\_ctrl)

);

assign hilo\_mux\_ctrl = (hilo\_mux\_internal) ? hilo\_mux\_internal : 2'b0;

endmodule

## Source Code of regfile.v:

module regfile (

input wire clk,

input wire we,

input wire [4:0] ra1,

input wire [4:0] ra2,

input wire [4:0] ra3,

input wire [4:0] wa,

input wire [31:0] wd,

output wire [31:0] rd1,

output wire [31:0] rd2,

output wire [31:0] rd3

);

reg [31:0] rf [0:31];

integer n;

initial begin

for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;

rf[29] = 32'h100; // Initialize $sp

end

always @ (posedge clk) begin

if (we) rf[wa] <= wd;

end

assign rd1 = (ra1 == 0) ? 0 : rf[ra1];

assign rd2 = (ra2 == 0) ? 0 : rf[ra2];

assign rd3 = (ra3 == 0) ? 0 : rf[ra3];

endmodule

## Source Code of signext.v:

module signext (

input wire [15:0] a,

output wire [31:0] y

);

assign y = {{16{a[15]}}, a};

endmodule

## Source Code of E\_Stage\_Reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:58:47 PM

// Design Name:

// Module Name: E\_Stage\_Reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module E\_Stage\_Reg(

input clk, rst,

input jr\_mux\_ctrlD,

input jumpD,

input hilo\_weD,

input [1:0] hilo\_mux\_ctrlD,

input dm2regD,

input we\_dmD,

//input branchD,

input [3:0] alu\_ctrlD,

input alu\_srcD,

input reg\_dstD,

input we\_regD,

input jal\_wa\_muxD,

input jal\_wd\_muxD,

input [31:0] rd1D,

input [31:0] rd2D,

input [4:0] instrD\_20\_16,

input [4:0] instrD\_15\_11,

input [31:0] pc\_plus4D,

input [4:0] shamtD,

input [31:0] sext\_immD,

input [31:0] addressD,

input [31:0] jtaD,

output reg jr\_mux\_ctrlE,

output reg jumpE,

output reg hilo\_weE,

output reg [1:0] hilo\_mux\_ctrlE,

output reg dm2regE,

output reg we\_dmE,

//output reg branchE,

output reg [3:0] alu\_ctrlE,

output reg alu\_srcE,

output reg reg\_dstE,

output reg we\_regE,

output reg jal\_wa\_muxE,

output reg jal\_wd\_muxE,

output reg [31:0] rd1\_outE,

output reg [31:0] rd2\_outE,

output reg [4:0] instrE\_20\_16,

output reg [4:0] instrE\_15\_11,

output reg [31:0] pc\_plus4E,

output reg [4:0] shamtE,

output reg [31:0] sext\_immE,

output reg [31:0] addressE,

output reg [31:0] jtaE

);

always @ (negedge clk, posedge rst) begin

if (rst) begin

jr\_mux\_ctrlE <= 0;

jumpE <= 0;

hilo\_weE <= 0;

hilo\_mux\_ctrlE <= 0;

dm2regE <= 0;

we\_dmE <= 0;

//branchE <= 0;

alu\_ctrlE <= 0;

alu\_srcE <= 0;

reg\_dstE <= 0;

we\_regE <= 0;

jal\_wa\_muxE <= 0;

jal\_wd\_muxE <= 0;

rd1\_outE <= 0;

rd2\_outE <= 0;

instrE\_20\_16 <= 0;

instrE\_15\_11 <= 0;

pc\_plus4E <= 0;

shamtE <= 0;

sext\_immE <= 0;

addressE <= 0;

jtaE <= 0;

end

else begin

jr\_mux\_ctrlE <= jr\_mux\_ctrlD;

jumpE <= jumpD;

hilo\_weE <= hilo\_weD;

hilo\_mux\_ctrlE <= hilo\_mux\_ctrlD;

dm2regE <= dm2regD;

we\_dmE <= we\_dmD;

//branchE <= branchD;

alu\_ctrlE <= alu\_ctrlD;

alu\_srcE <= alu\_srcD;

reg\_dstE <= reg\_dstD;

we\_regE <= we\_regD;

jal\_wa\_muxE <= jal\_wa\_muxD;

jal\_wd\_muxE <= jal\_wd\_muxD;

rd1\_outE <= rd1D;

rd2\_outE <= rd2D;

instrE\_20\_16 <= instrD\_20\_16;

instrE\_15\_11 <= instrD\_15\_11;

pc\_plus4E <= pc\_plus4D;

shamtE <= shamtD;

sext\_immE <= sext\_immD;

addressE <= addressD;

jtaE <= jtaD;

end

end

endmodule

## Source Code of alu.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:45:02 PM

// Design Name:

// Module Name: alu

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module alu (

input wire [3:0] op,

input wire [31:0] a,

input wire [31:0] b,

input wire [4:0] shamt,

output wire zero,

output reg [31:0] y, hi, lo

);

assign zero = (y == 0);

always @ (op, a, b) begin

case (op)

4'b0000: y <= a & b;

4'b0001: y <= a | b;

4'b0010: y <= a + b;

4'b0110: y <= a - b;

4'b0111: y <= (a < b) ? 1 : 0;

4'b1000: {hi,lo} <= a \* b;

4'b1001: y <= b << shamt;

4'b1010: y <= b >> shamt;

endcase

end

endmodule

## Source Code of mult\_inf.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:01:17 PM

// Design Name:

// Module Name: mult\_inf

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module mult\_inf #(parameter SIZE = 32)(

input wire [SIZE - 1:0] a, b,

output reg [2\*SIZE - 1:0] out

);

always @ (a, b) begin

out <= a \* b;

end

endmodule

## Source Code of M\_Stage\_Reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:58:47 PM

// Design Name:

// Module Name: M\_Stage\_Reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module M\_Stage\_Reg(

input clk, rst,

input jr\_mux\_ctrlE,

input jumpE,

input hilo\_weE,

input [1:0] hilo\_mux\_ctrlE,

input dm2regE,

input we\_dmE,

//input branchE,

input we\_regE,

//input zeroE,

input [31:0] alu\_outE,

input [31:0] wd\_dmE,

input [31:0] mult\_hiE,

input [31:0] mult\_loE,

//input [31:0] btaE,

input [31:0] jrE,

input [4:0] waE,

input jal\_wa\_muxE,

input jal\_wd\_muxE,

input [31:0] addressE,

input [31:0] jtaE,

output reg jr\_mux\_ctrlM,

output reg jumpM,

output reg hilo\_weM,

output reg [1:0] hilo\_mux\_ctrlM,

output reg dm2regM,

output reg we\_dmM,

//output reg branchM,

output reg we\_regM,

//output reg zeroM,

output reg [31:0] alu\_outM,

output reg [31:0] wd\_dmM,

output reg [31:0] mult\_hiM,

output reg [31:0] mult\_loM,

//output reg [31:0] btaM,

output reg [31:0] jrM,

output reg [4:0] waM,

output reg jal\_wa\_muxM,

output reg jal\_wd\_muxM,

output reg [31:0] addressM,

output reg [31:0] jtaM

);

always @ (negedge clk, posedge rst) begin

if (rst) begin

jr\_mux\_ctrlM <= 0;

jumpM <= 0;

hilo\_weM <= 0;

hilo\_mux\_ctrlM <= 0;

dm2regM <= 0;

we\_dmM <= 0;

//branchM <= 0;

we\_regM <= 0;

//zeroM <= 0;

alu\_outM <= 0;

wd\_dmM <= 0;

mult\_hiM <= 0;

mult\_loM <= 0;

//btaM <= 0;

jrM <= 0;

waM <= 0;

jal\_wa\_muxM <= 0;

jal\_wd\_muxM <= 0;

addressM <= 0;

jtaM <= 0;

end

else begin

jr\_mux\_ctrlM <= jr\_mux\_ctrlE;

jumpM <= jumpE;

hilo\_weM <= hilo\_weE;

hilo\_mux\_ctrlM <= hilo\_mux\_ctrlE;

dm2regM <= dm2regE;

we\_dmM <= we\_dmE;

//branchM <= branchE;

we\_regM <= we\_regE;

//zeroM <= zeroE;

alu\_outM <= alu\_outE;

wd\_dmM <= wd\_dmE;

mult\_hiM <= mult\_hiE;

mult\_loM <= mult\_loE;

//btaM <= btaE;

jrM <= jrE;

waM <= waE;

jal\_wa\_muxM <= jal\_wa\_muxE;

jal\_wd\_muxM <= jal\_wd\_muxE;

addressM <= addressE;

jtaM <= jtaE;

end

end

endmodule

## Source Code of HiLo\_reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:01:17 PM

// Design Name:

// Module Name: HiLo\_reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module HiLo\_reg #(parameter WIDTH = 32) (

input wire clk, rst, we,

input wire [WIDTH - 1:0] hi, lo,

output reg [WIDTH - 1:0] hi\_out, lo\_out

);

always @ (posedge clk, posedge rst)

begin

if (rst) {hi\_out, lo\_out} <= 0;

else if (we) {hi\_out, lo\_out} <= {hi,lo};

else {hi\_out, lo\_out} <= {hi\_out, lo\_out};

end

endmodule

## Source Code of W\_Stage\_Reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 10:58:47 PM

// Design Name:

// Module Name: W\_Stage\_Reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module W\_Stage\_Reg(

input clk, rst,

input jr\_mux\_ctrlM,

input jumpM,

input [1:0] hilo\_mux\_ctrlM,

input dm2regM,

input we\_regM,

input [31:0] alu\_outM,

input [31:0] rd\_dmM,

input [31:0] hi\_outM,

input [31:0] lo\_outM,

input [31:0] jrM,

input [4:0] waM,

input [31:0] jtaM,

input jal\_wd\_muxM,

output reg jr\_mux\_ctrlW,

output reg jumpW,

output reg [1:0] hilo\_mux\_ctrlW,

output reg dm2regW,

output reg we\_regW,

output reg [31:0] alu\_outW,

output reg [31:0] rd\_dmW,

output reg [31:0] hi\_outW,

output reg [31:0] lo\_outW,

output reg jal\_wd\_muxW,

output reg [31:0] jrW,

output reg [4:0] waW,

output reg [31:0] jtaW

);

always @ (negedge clk, posedge rst) begin

if (rst) begin

jr\_mux\_ctrlW <= 0;

jumpW <= 0;

hilo\_mux\_ctrlW <= 0;

dm2regW <= 0;

we\_regW <= 0;

alu\_outW <= 0;

jal\_wd\_muxW <= 0;

rd\_dmW <= 0;

hi\_outW <= 0;

lo\_outW <= 0;

jrW <= 0;

waW <= 0;

jtaW <= 0;

end

else begin

jr\_mux\_ctrlW <= jr\_mux\_ctrlM;

jumpW <= jumpM;

hilo\_mux\_ctrlW <= hilo\_mux\_ctrlM;

dm2regW <= dm2regM;

we\_regW <= we\_regM;

alu\_outW <= alu\_outM;

rd\_dmW <= rd\_dmM;

hi\_outW <= hi\_outM;

lo\_outW <= lo\_outM;

jrW <= jrM;

waW <= waM;

jtaW <= jtaM;

jal\_wd\_muxW <= jal\_wd\_muxM;

end

end

endmodule

## Source Code of maindec.v:

module maindec (

input wire [5:0] opcode,

output wire branch,

output wire jump,

output wire reg\_dst,

output wire we\_reg,

output wire alu\_src,

output wire we\_dm,

output wire dm2reg,

output wire [1:0] alu\_op,

output wire jal\_wa\_mux\_sel,

output wire jal\_wd\_mux\_sel

);

reg [10:0] ctrl;

assign {branch, jump, reg\_dst, we\_reg, alu\_src, we\_dm, dm2reg, alu\_op, jal\_wa\_mux\_sel, jal\_wd\_mux\_sel} = ctrl;

always @ (opcode) begin

case (opcode)

6'b00\_0000: ctrl = 11'b0\_0\_1\_1\_0\_0\_0\_10\_0\_0; // R-type

6'b00\_1000: ctrl = 11'b0\_0\_0\_1\_1\_0\_0\_00\_0\_0; // ADDI

6'b00\_0100: ctrl = 11'b1\_0\_0\_0\_0\_0\_0\_01\_0\_0; // BEQ

6'b00\_0010: ctrl = 11'b0\_1\_0\_0\_0\_0\_0\_00\_0\_0; // J

6'b00\_0011: ctrl = 11'b0\_1\_0\_1\_0\_0\_0\_00\_1\_1; // JAL //TODO

6'b10\_1011: ctrl = 11'b0\_0\_0\_0\_1\_1\_0\_00\_0\_0; // SW

//6'b10\_0011: ctrl = 11'b0\_0\_0\_1\_1\_0\_1\_00\_0\_0; // LW

6'b10\_0011: ctrl = 11'b0\_0\_0\_1\_1\_0\_1\_00\_0\_0; // LW

default: ctrl = 11'bx\_x\_x\_0\_x\_0\_x\_xx\_x\_x;

//default: ctrl = 11'b0\_0\_0\_0\_0\_0\_0\_00\_0\_0;

endcase

end

endmodule

## Source Code of auxdec.v:

module auxdec (

input wire [1:0] alu\_op,

input wire [5:0] funct,

output wire [3:0] alu\_ctrl,

output wire [1:0] hilo\_mux\_ctrl,

output wire hilo\_we,

output wire jr\_mux\_ctrl

);

reg [7:0] ctrl;

assign {alu\_ctrl, hilo\_mux\_ctrl, hilo\_we, jr\_mux\_ctrl} = ctrl;

always @ (alu\_op, funct) begin

case (alu\_op)

2'b00: ctrl = 8'b0010\_00\_0\_0; // ADD

2'b01: ctrl = 8'b0110\_00\_0\_0; // SUB

default: case (funct)

6'b10\_0100: ctrl = 8'b0000\_00\_0\_0; // AND

6'b10\_0101: ctrl = 8'b0001\_00\_0\_0; // OR

6'b10\_0000: ctrl = 8'b0010\_00\_0\_0; // ADD

6'b10\_0010: ctrl = 8'b0110\_00\_0\_0; // SUB

6'b10\_1010: ctrl = 8'b0111\_00\_0\_0; // SLT

6'b01\_1001: ctrl = 8'bxxxx\_00\_1\_0; // MULTU

6'b01\_0000: ctrl = 8'b0000\_11\_0\_0; // MFHI

6'b01\_0010: ctrl = 8'b0000\_01\_0\_0; // MFLO

6'b00\_0000: ctrl = 8'b1001\_00\_0\_0; // SLL

6'b00\_0010: ctrl = 8'b1010\_00\_0\_0; // SRL

6'b00\_1000: ctrl = 8'bxxxx\_00\_0\_1; // JR

default: ctrl = 8'bxxxx\_xx\_x\_x;

endcase

endcase

end

endmodule

## Source Code of and\_fact.v:

module and\_fact #(parameter WIDTH = 32)(

input [WIDTH - 1:0] a, b,

output wire c

);

assign c = a & b;

endmodule

## Source Code of fact\_ad.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:02:37 PM

// Design Name:

// Module Name: fact\_ad

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module fact\_ad(

input wire [1:0] A,

input wire WE,

output reg WE1, WE2,

output wire [1:0] RdSel

);

always @ (\*) begin

case(A)

2'b00: begin

WE1 <= WE;

WE2 <= 1'b0;

end

2'b01: begin

WE1 <= 1'b0;

WE2 <= WE;

end

2'b10: begin

WE1 <= 1'b0;

WE2 <= 1'b0;

end

2'b11: begin

WE1 <= 1'b0;

WE2 <= 1'b0;

end

default: begin

WE1 <= 1'bx;

WE1 <= 1'bx;

end

endcase

end

assign RdSel = A;

endmodule

## Source Code of fact\_reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:11:45 PM

// Design Name:

// Module Name: fact\_reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module fact\_reg #(parameter w = 32)(

input wire Clk, Rst,

input wire [w-1:0] D,

input wire Load\_Reg,

output reg [w-1:0] Q

);

always @ (posedge Clk, posedge Rst) begin

if (Rst)

Q <= 0;

else if (Load\_Reg)

Q <= D;

else

Q <= Q;

end

endmodule

## Source Code of FSMult.v:

module FSMmult #(parameter DATA\_WIDTH = 32)(

input [DATA\_WIDTH-1:0] D,

input GO,

input RST,

input CLK,

output wire doneF,

output wire ERROR,

output wire [DATA\_WIDTH-1:0] out

);

wire load\_cnt, cnt\_en, mux\_sel, load\_reg, buf\_oe, gt, ec;

CU control(

.GO(GO),

.GT(gt),

.CLK(CLK),

.RST(RST),

.EC(ec),

.load\_cnt(load\_cnt),

.cnt\_en(cnt\_en),

.mux\_sel(mux\_sel),

.load\_reg(load\_reg),

.buf\_oe(buf\_oe),

.done(doneF),

.error(ERROR)

);

DP dataP(

.clk(CLK),

.load\_cnt(load\_cnt),

.en\_cnt(cnt\_en),

.sel\_mux(mux\_sel),

.load\_reg(load\_reg),

.oe\_buf(buf\_oe),

.gt(gt),

.ec(ec),

.in(D),

.out(out)

);

endmodule

## Source Code of fact\_res\_done\_reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:19:40 PM

// Design Name:

// Module Name: fact\_res\_done\_reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module fact\_res\_done\_reg(

input Clk, Rst, GoPulseCmb, Done,

output reg ResDone

);

always @ (posedge Clk, posedge Rst) begin

if (Rst)

ResDone <= 1'b0;

else

ResDone <= (~GoPulseCmb) & (Done | ResDone);

end

endmodule

## Source Code of fact\_res\_err\_reg.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:34:29 PM

// Design Name:

// Module Name: fact\_res\_err\_reg

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module fact\_res\_err\_reg(

input Clk, Rst, GoPulseCmb, Err,

output reg ResErr

);

always @ (posedge Clk, posedge Rst) begin

if (Rst)

ResErr <= 1'b0;

else

ResErr <= (~GoPulseCmb) & (Err | ResErr);

end

endmodule

## Source Code of DP.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: DP

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module DP #(parameter DATA\_WIDTH = 32)(

input wire clk,

input wire load\_cnt,

input wire en\_cnt,

input wire sel\_mux,

input wire load\_reg,

input wire oe\_buf,

output wire gt,

output wire ec,

input wire [DATA\_WIDTH-1:0] in,

output wire [DATA\_WIDTH-1:0] out);

wire [DATA\_WIDTH-1:0] mux\_to\_reg;

wire [DATA\_WIDTH-1:0] mul\_to\_mux;

wire [DATA\_WIDTH-1:0] cnt\_to\_cmp\_and\_mul;

wire [DATA\_WIDTH-1:0] reg\_to\_mul\_and\_buf;

CNT #(DATA\_WIDTH) down\_counter(

.in(in),

.load\_cnt(load\_cnt),

.en(en\_cnt),

.clk(clk),

.out(cnt\_to\_cmp\_and\_mul)

);

Multiplexer #(DATA\_WIDTH) product\_multi(sel\_mux, mul\_to\_mux, 1, mux\_to\_reg);

Register #(DATA\_WIDTH) product\_reg(clk, load\_reg, mux\_to\_reg, reg\_to\_mul\_and\_buf);

Multiplexer #(DATA\_WIDTH) product\_buf(

oe\_buf,

reg\_to\_mul\_and\_buf,

0,

out);

CMP #(DATA\_WIDTH) cmp\_gt(

cnt\_to\_cmp\_and\_mul, 1, gt);

Multiplier #(DATA\_WIDTH) multiply(

reg\_to\_mul\_and\_buf, cnt\_to\_cmp\_and\_mul, mul\_to\_mux);

CMP #(DATA\_WIDTH) cmp\_ec(

in, 12, ec);

endmodule

## Source Code of CNT.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: CNT

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module CNT #(parameter DATA\_WIDTH = 32)(

input wire [DATA\_WIDTH - 1:0] in,

input wire load\_cnt,

input wire en,

input wire clk,

output reg [DATA\_WIDTH -1:0] out);

always@(posedge clk) begin

if(load\_cnt) begin

out = in;

end

if(en) begin

out = out - 1'b1;

end

end

endmodule

## Source Code of Multiplexer.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: Multiplexer

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module Multiplexer #(parameter DATA\_WIDTH = 32)(

input wire sel,

input wire [DATA\_WIDTH-1:0] a,

input wire [DATA\_WIDTH -1:0] b,

output wire [DATA\_WIDTH-1:0] out);

assign out = (sel == 0) ? a : b;

endmodule

## Source Code of Register.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: Register

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module Register #(parameter DATA\_WIDTH = 32)(

input wire clk,

input wire load\_reg,

input wire [DATA\_WIDTH-1:0] in,

output reg [DATA\_WIDTH-1:0] out);

always@(posedge clk) begin

if(load\_reg) begin

out = in;

end

// else out = out

end

endmodule

## Source Code of CMP.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: CMP

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module CMP #(parameter DATA\_WIDTH = 32)(

input wire [DATA\_WIDTH-1:0] a,

input wire [DATA\_WIDTH-1:0] b,

output wire gt);

assign gt = (a > b) ? 1 : 0;

endmodule

## Source Code of CU.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: CU

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module CU #(parameter DATA\_WIDTH = 32)(

input GO,

input GT,

input RST,

input CLK,

input EC,

output reg load\_cnt,

output reg cnt\_en,

output reg mux\_sel,

output reg load\_reg,

output reg buf\_oe,

output reg done,

output reg error

);

parameter S0 = 3'b000,

S1 = 3'b001,

S2 = 3'b010,

S3 = 3'b011,

S4 = 3'b100;

reg [2:0] CS, NS;

//NS driver

always @ (posedge CLK, posedge RST)

begin

if(RST == 1) CS <= S0;

else CS <= NS;

end

//Output logic

always @ (CS, GT, EC)

begin

case(CS)

S0: begin

load\_cnt =0;

cnt\_en = 0;

mux\_sel = 0;

load\_reg = 0;

buf\_oe = 0;

done = 0;

error = 0;

end

//Load

S1: begin

load\_cnt =1;

cnt\_en = 0;

mux\_sel = 1;

load\_reg = 1;

buf\_oe = 0;

done = 0;

error = 0;

end

//Finished

S2: begin

if(EC)begin

load\_cnt <= 0;

cnt\_en <= 0;

mux\_sel <= 0;

load\_reg <= 0;

buf\_oe <= 0;

done <= 1;

error <= 1;

end

else if(!EC && !GT)begin

load\_cnt = 0;

cnt\_en = 0;

mux\_sel = 0;

load\_reg = 0;

buf\_oe = 0;

done = 1;

error = 0;

end

else begin

load\_cnt = 0;

cnt\_en = 1;

mux\_sel = 0;

load\_reg = 1;

buf\_oe = 0;

done = 0;

error = 0;

end

end

endcase

end

always @ (CS, GO, GT, EC)

begin

case(CS)

S0: begin

if(GO == 1) NS <= S1; // Go = 1 and D <= 12, go state 1

else NS <= S0; // Go = 0

end

S1: begin

NS <= S2;

end

//Finished

S2:

if(GT == 0 || EC == 1) NS <= S0;

else NS <= S2;

endcase

end

endmodule

## Source Code of gpio\_ad.v:

module gpio\_ad(

input wire [1:0] A,

input wire WE,

output reg WE1,

output reg WE2,

output wire [1:0] RdSel

);

always @ (\*) begin

case(A)

2'b00:begin

WE1 = 1'b0;

WE2 = 1'b0;

end

2'b01: begin

WE1 = 1'b0;

WE2 = 1'b0;

end

2'b10: begin

WE1 = WE;

WE2 = 1'b0;

end

2'b11: begin

WE1 = 1'b0;

WE2 = WE;

end

default: begin

WE1 = 1'bx;

WE2 = 1'bx;

end

endcase

end

assign RdSel = A;

endmodule

## Source Code of Multiplier.v:

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 11/30/2022 11:08:41 PM

// Design Name:

// Module Name: Multiplier

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module Multiplier #(parameter DATA\_WIDTH = 32)(

input wire [DATA\_WIDTH-1:0] a,

input wire [DATA\_WIDTH-1:0] b,

output wire [DATA\_WIDTH-1:0] out);

assign out = a \* b;

endmodule

## Assembly Code in pipelined MIPS to trigger the SoC:

main: addi $t0, $0, 0x0F

lw $t1, 0x0900($0) #Take n value from GPIO

lw $t2, 0x0904($0) #Take Go value from GPIO

sll $0, $0, 0

sll $0, $0, 0

sw $t1, 0x0800($0) #Give this n value to Factorial Accelerator

sw $t2, 0x0804($0) #Give the Go signal to Factorial Accelerator

poll: lw $t3, 0x0808($0) #Continuosuly poll the Done/Err bit

sll $0, $0, 0

sll $0, $0, 0

sll $0, $0, 0

beq $t3, $0, poll #Continuosuly poll the Done/Err bit

srl $t4, $t3, 1 #Shift right to access the Err bit

lw $t5, 0x080C($0) #Get the n! value into processor

sll $0, $0, 0

sll $0, $0, 0

sw $t4, 0x0908($0) #Processor gives Err bit to GPIO1

sw $t5, 0x090C($0) #Processor gives n! value to GPIO2