San Jose State University
Department of Computer Engineering

CMPE 140 Lab Report

Lab 8 Report

Title Pipelined MIPS Processor & I/O Interface

Semester Fall 2019 **Date** 11/19/19

by

Name Karine Worley *(typed)* **SID** 008876725 *(typed)*
Name Sidarth Shahri *(typed)* **SID** 009712248 *(typed)*
Name Manuel Chavez *(typed)* **SID** 012305930 *(typed)*
Name Akash Sindhu *(typed)* **SID** 010888384 *(typed)*

Lab Checkup Record

Week	Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
1	K.W M.C. S.S. A.S.		100%		
2	S.S. M.C. K.W. A.S.		100%		
3	S.S. M.C. K.W. A.S.		100%		
4	S.S. M.C. K.W. A.S.		100%		

*** Detailed descriptions must be given in the report.**

Table of Contents

Purpose

The purpose of this assignment was to convert the single-cycle MIPS processor created in Assignment 5 through 7 into a five-stage, pipelined processor using design principles learned in class. By pipelining the single-cycle processor, the performance when executing a MIPS could be greatly improved. This laboratory experience in creating a pipelined processor also offered insight into the design process required to create a processor in the real world.

Furthermore, we were tasked to interface the factorial accelerator designed in Assignment 1 along with a new general purpose I/O (GPIO) unit using memory-mapped interface registers. That is, blocks of memory addresses should be used to communicate with these new modules wrapped in an interface wrapper. This exercise provided us experience with creating memory-mapped I/O modules similar to exercises in CMPE 127.

Assignment 8 builds on topics covered in previous lab assignments. For example, the extended MIPS processor created in Assignment 7 that added support for *multu, mfhi, mflo, jr, jal, sll,* and *slr* was used as the basis for our pipelined MIPS processor design. The final result is a MIPS processor bundled with the factorial accelerator, GPIO unit, data memory, and instruction memory on a system-on-a-chip design.

Design Methodology

Due to the nature of this assignment, tasks were created to work towards the goal of creating a working pipelined MIPS processor. These tasks were as follows:

1) Create a draft of the pipelined MIPS microarchitecture schematic
2) Create a draft of the SoC interface schematic
3) Create tables for the MIPS control unit
4) Perform performance analysis of hardware accelerated n!
5) Create unit-level testbenches for the GPIO unit and factorial accelerator to examine simulation waveforms
6) Integrate the single-cycle MIPS processor with the factorial accelerator and GPIO unit to functionally verify the design on the Basys3 FGPA board
7) Integrate the pipelined MIPS processor with the factorial accelerator and GPIO unit to functionally verify the design on the Basys3 FPGA board

Because the design of the single-cycle MIPS processor was covered in Assignments 5 through 7, it is not covered in this report. The same is true for the design of the datapath created to support *multu, mfhi, mflo, jr, jal, sll,* and *slr*. Instead, the components necessary for these instructions are integrated into the pipelined MIPS microarchitecture schematic.

System-on-Chip (SoC) Design

The design of the system-on-chip (SoC) was the same across both the single-cycle CPU and the pipelined CPU. Explaining it first will facilitate 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 (either the single-cycle or 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 design of this SoC is shown in Figure 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. For example, a *lw $t1, 0x800($0)* instruction addresses the factorial accelerator.

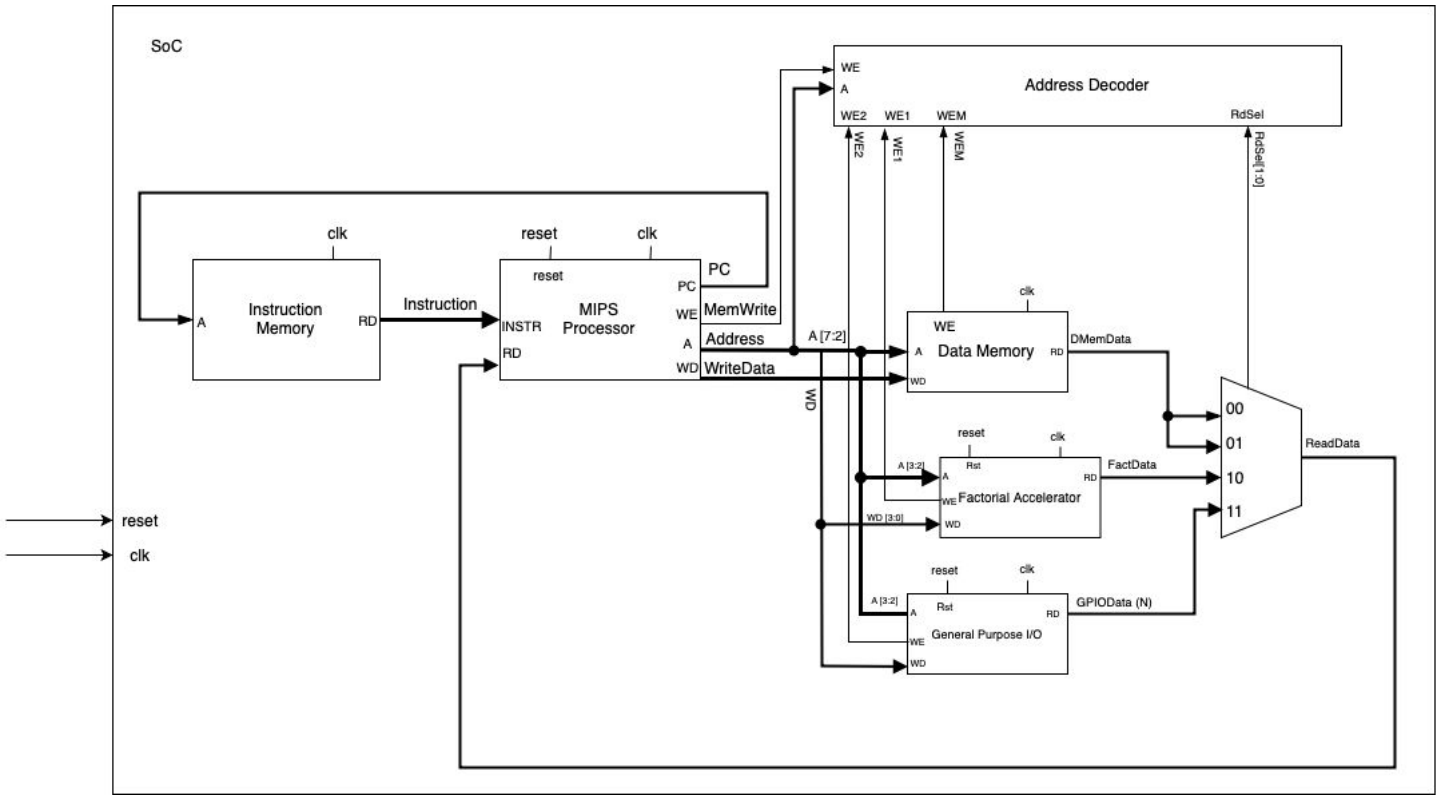


Figure 1: Design of the SoC that is shared by both CPU designs

Table 1: SoC Memory Maps

SoC Memory Maps			
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	GPIO

Table 2: SoC Modules

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. 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.

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:

Factorial Algorithm

INPUT *n*
product = 1
WHILE (*n* > 1) {
  product = *product* * *n*
  n = *n* -1
}
OUTPUT *product*

As shown in Figure 2, 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 3 and 4 shows the design of the datapath and the ASM chart of the control unit for the factorial accelerator.

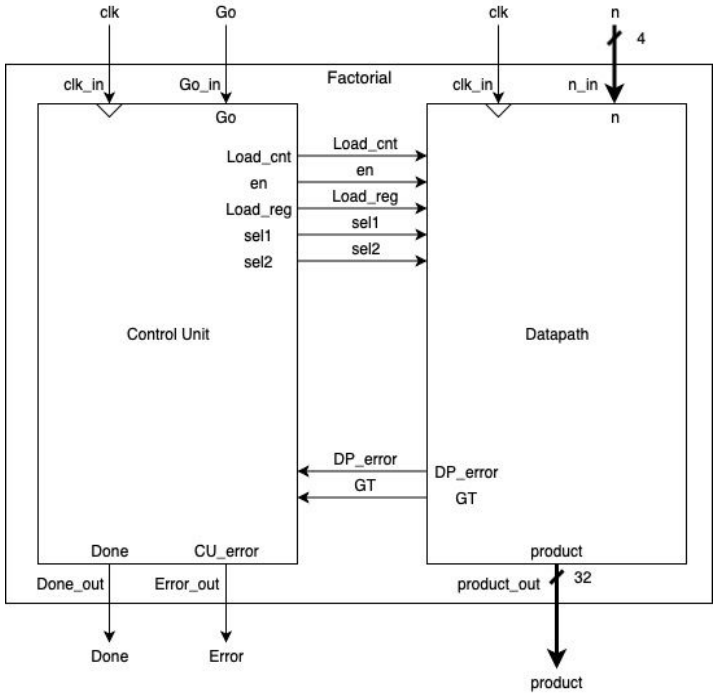


Figure 2: Factorial module containing control unit and datapath modules with their inputs and outputs.

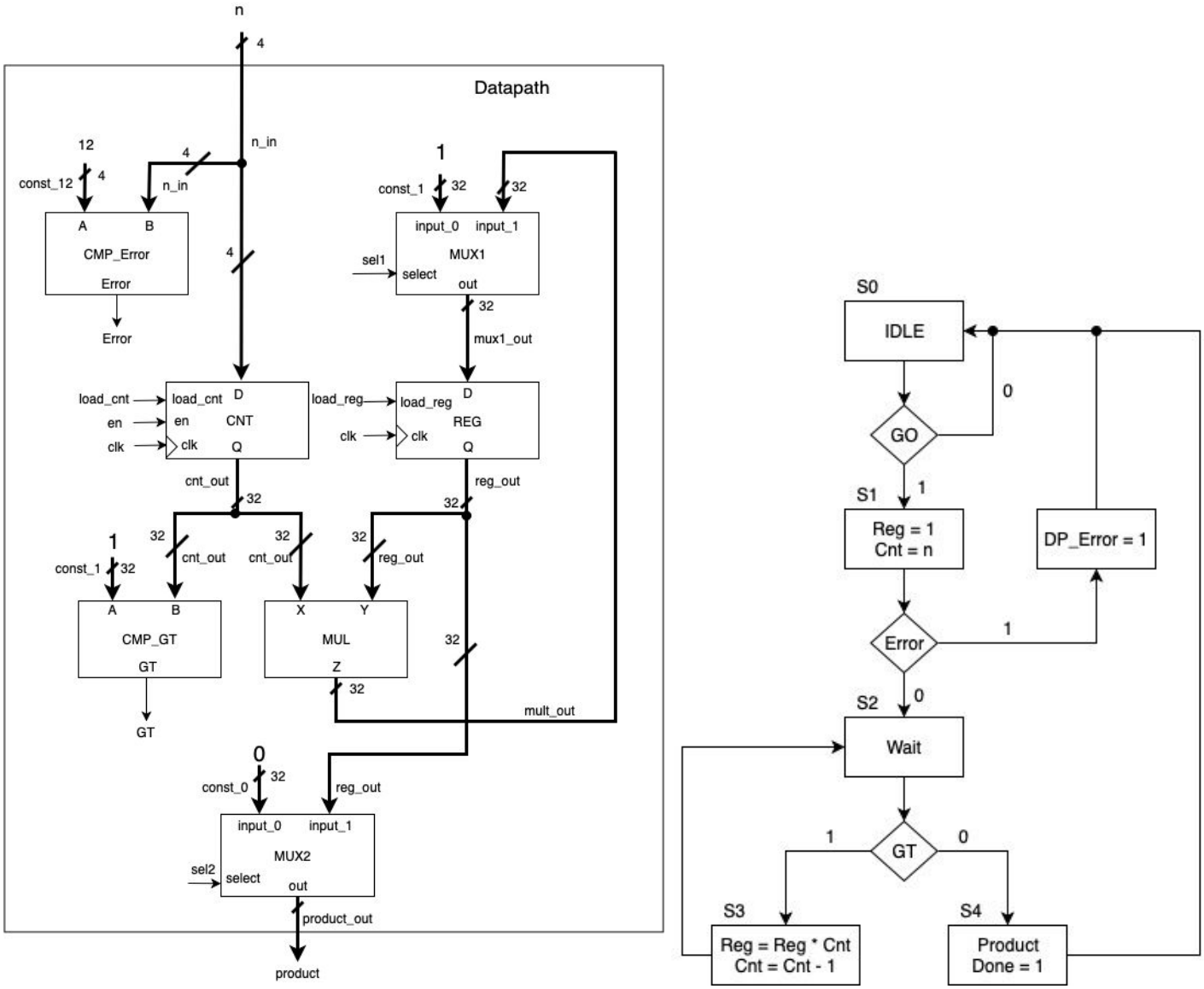


Figure 3 and 4. **Left**: Design of the datapath of the factorial accelerator. **Right**: Design of the control unit of the factorial accelerator.

We could not simply connect our factorial module to the single-cycle or 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 5 shows the design of this interface wrapper.

The design of this interface wrapper was based on a design given to the team in class. 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.

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 3 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 3.

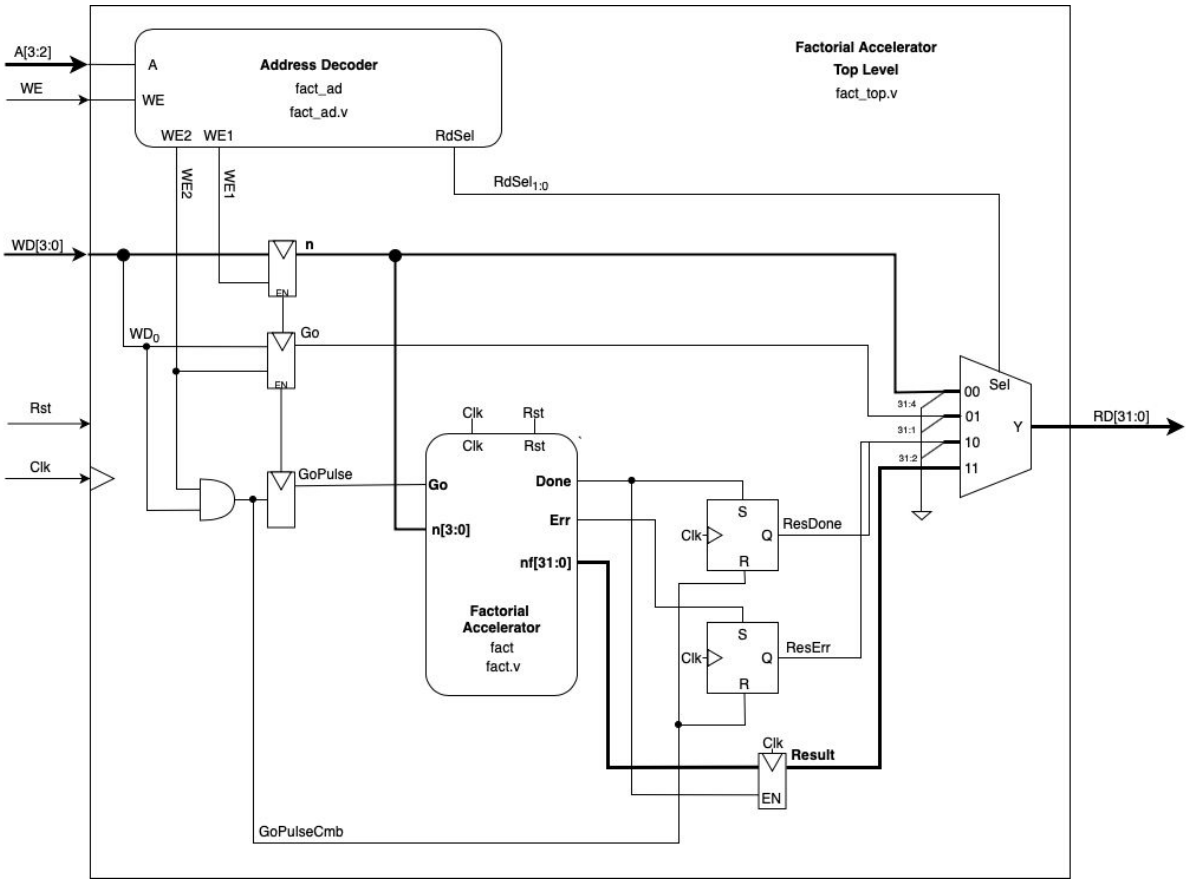


Figure 5: Diagram of the interface wrapper containing the factorial module from Assignment 1 with a few new components to control outputs of the factorial module and decode addresses given to the interface.

Table 3: Factorial Accelerator Memory Map. Active for address 0x800-0x80C.

Factorial Accelerator Memory Map					
Address	Address[3:2]	R/W	Register Name	Bits	Bit Definition
0x800	00	R/W	Data Input (n)	31:4	Unused
				3:0	Input Data, n[3:0]
0x804	01	R/W	Control Unit (Go)	31:1	Unused
				0	Go bit
0x808	10	R	Control Output (Done, Err)	31:2	Unused
				1	Err bit
				0	Done bit
0x80C	11	R	Data Output (Result)	31:0	Result Data, nf[31:0]

Table 4: Factorial Accelerator Interface Wrapper Modules

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!.

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 devices on an FPGA device. In this case, the GPIO unit acts as an intermediary between the Basys3’s 7-segment displays and switches. 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 6.

Inside the interface wrapper exists another address decoder similar to 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. Similar to 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 and outputs its data to the 7-segment display on the Basys3 board while gpO1 holds the status bits to output to the LEDs on the Basys3 board. This is further explained in the Hardware Validation section. Finally, a 4-input mux similar to 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 5 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.

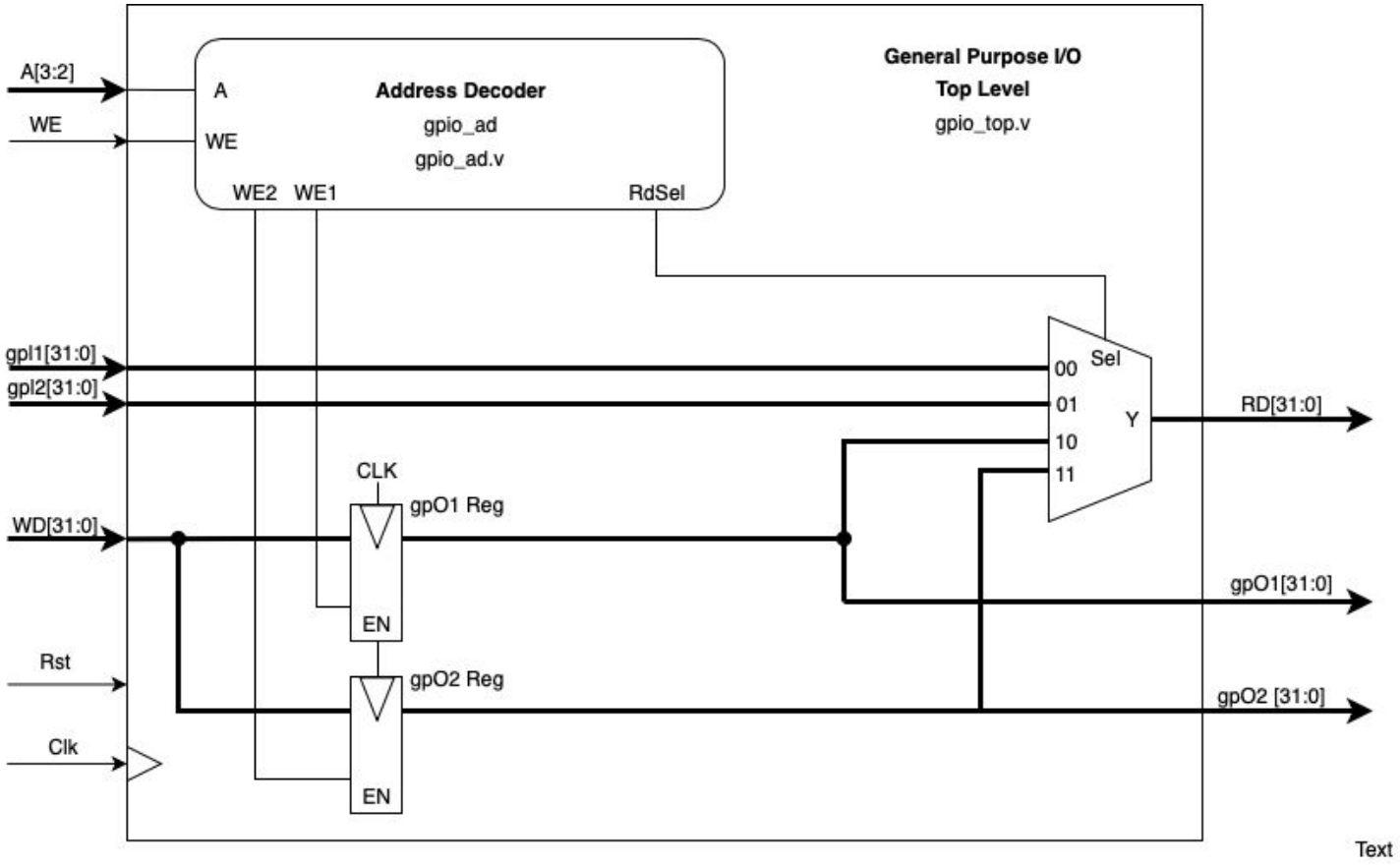


Figure 6: Diagram of the GPIO interface wrapper

Table 5: GPIO Memory Map. Active for addresses 0x900 - 0x90C

GPIO Memory-Mapped Registers					
Address	Address[3:2]	R/W	Register Name	Bits	Bit Definition
0x900	0	R	Input 1	31:0	General Input
0x904	1	R	Input 2	31:0	General Input
0x908	10	R/W	Output 1	31:0	General Output
0x90C	11	R/W	Output 2	31:0	General Output

Table 6: GPIO Unit Modules

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.

Single-cycle MIPS Processor

The single-cycle MIPS processor was designed to execute a single instruction in one clock cycle. Therefore, the length of the clock cycle was designed to be equal to the time it takes to execute the instruction that takes the most time. This severely impacted performance when executing an assembly program. Furthermore, performing a recursive factorial calculation required an increasing number of clock cycles as the input n increased. To solve this, we were tasked with designing a pipelined MIPS processor that could theoretically execute up to five instructions per clock cycle when the pipeline was fully loaded.

The task involving the single-cycle MIPS processor was mainly aimed at verifying the design and function of the GPIO unit and factorial accelerator. It was also an exercise in interfacing an existing CPU with new I/O devices. As the single-cycle MIPS processor was created in Assignment 5 through 7, its design is not covered here. Table 7 explains the instructions that the single-cycle processor is capable of processing. Appendix A and Appendix B contains relevant design diagrams of the single-cycle MIPS processor and its various components.

No changes were made to the core of the single-cycle MIPS processor design. The design was verified in Assignment 7. It was simply connected to the new I/O devices in a new top module title Single_Cycle_SoC.v as seen in Figure 1. This new top module also contains the new modules as described in the System-on-Chip Design section. Code for this implementation is shown in Appendix I.

Table 7: Supported instruction by the single-cycle MIPS processor

Instruction	Format	Description
add	add $d, $s, $t	Result of $s + $t stored in $d
sub	sub $d, $s, $t	Result of $s - $t stored in $d
and	and $d, $s, $t	Result of bitwise AND operation with operands $s and $t are stored in $d.
or	or $d, $s, $t	Result of bitwise OR operation with operands $s and $t are stored in $d.
slt	slt $d, $s, $t	If $s is less than $t, $d is set to 1. Otherwise, $d is set to 0.
lw	lw $t, offset($s)	The contents of the memory address in $s + offset is stored in $t.
sw	sw $t, offset($s)	The contents of $t is stored at the memory address in $s + offset.
beq	beq $s, $t, offset	Branch to address if $s and $t are equal
addi	addi $d, $s, imm	Result of $s +

		immediate value stored in $d
j	j target	Jump to calculated address
multu	multu $s, $t	Multiplies values in $s and $t and stores it result in $lo and $hi
mfhi	mfhi $d	Moves upper 32 bit result of multu into $d
mflo	mflo $d	Moves lower 32 bit result of multu into $d
jr	jr $s	Jump to the address stored in $s
sll	sll $d, $t, h	Shifts value in $t left by h and stores it in $d
slr	slr $d, $t, h	Shifts value in $t right by h and stores it in $d
jal	jal target	Jumps to target address and stores return address in $31
nop	nop	Machine code will be 0x00000000

Pipelined MIPS Processor

Designing the pipelined MIPS processor proved to be the most challenging task. The team was unsure of how to start the design process. At first, we felt we were handicapped because we made the mistake of using our Assignment 7 single-cycle MIPS processor datapath as a starting point. In that design diagram, nothing was grouped based on the stage the module operated in. For example, the program counter register was place all the way to the right of the datapath as shown in Appendix B, not in the fetch stage. Therefore, 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. 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.

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, MULTU, and SHIFTER modules. The ALU processes *and, or, add, sub,* and *slt* instructions. The MULTU processes *multu* instructions. The SHIFTER processes *slr* and *sll* instructions. 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. An adder also calculates a branch address to branch to in the event of a *beq* instruction. The SHIFTER 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 C 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 C 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) , rd2M (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 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 in this stage. In an earlier iteration of this design, we placed these registers in the decode stage. Therefore, data from *multu* and the MULT module would have to be carried through the pipeline through the writeback stage before being stored in the high and low register. This change was recommended to us by our TA as this would simplify our design and reduce how many clock cycles were necessary for a *multu, mflo,* and *mfhi* instruction. Recall from Assignment 7 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, the output of the SHIFTER module for storing in the register file, and the machine code from the fetch stage. Also worth to note is that the branch address for a *beq* instruction is returned back to the fetch stage if a branch is required.

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 total of 5 2-input muxes choose between data from the high register, low register, ALU output, input from data memory or I/O devices, program counter + 4, and the SHIFTER output to writeback depending on the instruction being processed. For example, *mflo $t1*, will forward data from the low register through these muxes. The relevant control signals that are forwarded from the decode stage through the pipeline serve as the select inputs for these muxes. 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 7 and are only grouped by their respective stage in Appendix C. 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 images and truth tables for the design of this pipelined CPU are too large to insert here, they have been added to Appendix C. 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.

Creating the pipelined CPU was an extremely involved process that required several iterations, feedback from our TA and professor, and internal input from the team. Certain design choices were taken to improve simplicity over performance such as carrying the jump and branch address through the entire pipeline instead of returning that address back to the fetch stage as soon as they were ready. Furthermore, there was room for improvement in removing the MULT module from the datapath and reworking the multiplication logic to work better with the pipeline as advised by our TA. Because of the design of our datapath, several 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 Appendix I. A time space diagram showing the result of our pipelining efforts is present below in Figure 7. The diagram shows all instructions from the modified MIPS driver code with NOPs inserted for instructions with data dependencies.

	Clock Cycles																																		
	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
addi																																				
addi																																				
sll	ID	EXE	MEM	WB																																
lw	IF	ID	EXE	MEM	WB																															
and		IF	NOP	NOP	NOP	ID	EXE	MEM	WB																											
sw						IF	ID	EXE	MEM	WB																											
sw							IF	ID	EXE	MEM	WB																										
lw								IF	ID	EXE	MEM	WB																									
beq									IF	NOP	NOP	NOP	ID	EXE	MEM	WB																				
srl													IF	NOP	NOP	NOP	ID	EXE	WB																	
and																	IF	ID	EXE	MEM	WB															
and																		IF	ID	EXE	MEM	WB														
or																			IF	NOP	NOP	NOP	ID	EXE	MEM	WB										
lw																							IF	ID	EXE	MEM	WB									
sw																								IF	NOP	NOP	ID	EXE	MEM	WB						
sw																											IF	NOP	NOP	ID	EXE	MEM	WB			
j																														IF	NOP	NOP	NOP	EXE	MEM	WB

Figure 7: Time Space Diagram for modified MIPS Driver Code (code in Appendix J)

Data Forwarding

In an effort 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. Our new diagram is present at the end of this report in Appendix K. 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 (in which the above equations are implemented in Verilog). 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 datapath modules are directly connected to these mux, only ports from the state registers. This is more clear when looking at the diagrams in Appendix L. Also, to reiterate, **this new design of the pipelined MIPS processor was not functional**. Functional verification did not produce the expected results and we did not have enough time to debug the design. As data forwarding was not a core requirement of the assignment, it is not shown in the Waveforms section or Hardware Validation section. However, code for the data forwarding logic and new multiplexers is given in Appendix M. We suspect the issue lies in the 3-input mux logic as data is not output by the mux at all. Once that issue is resolved, ideally, all that needs to be done is remove the NOPs from the assembly driver code and test everything.

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 8. For the pipelined MIPS design, the ASM chart of the factorial accelerator was analyzed to predict the result. 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, equation (2) could be used to arrive at the same result. 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 8.

$$y = 2n + 2 \qquad (1)$$

$$y = -3.394 + 13.958 \cdot x \qquad (2)$$

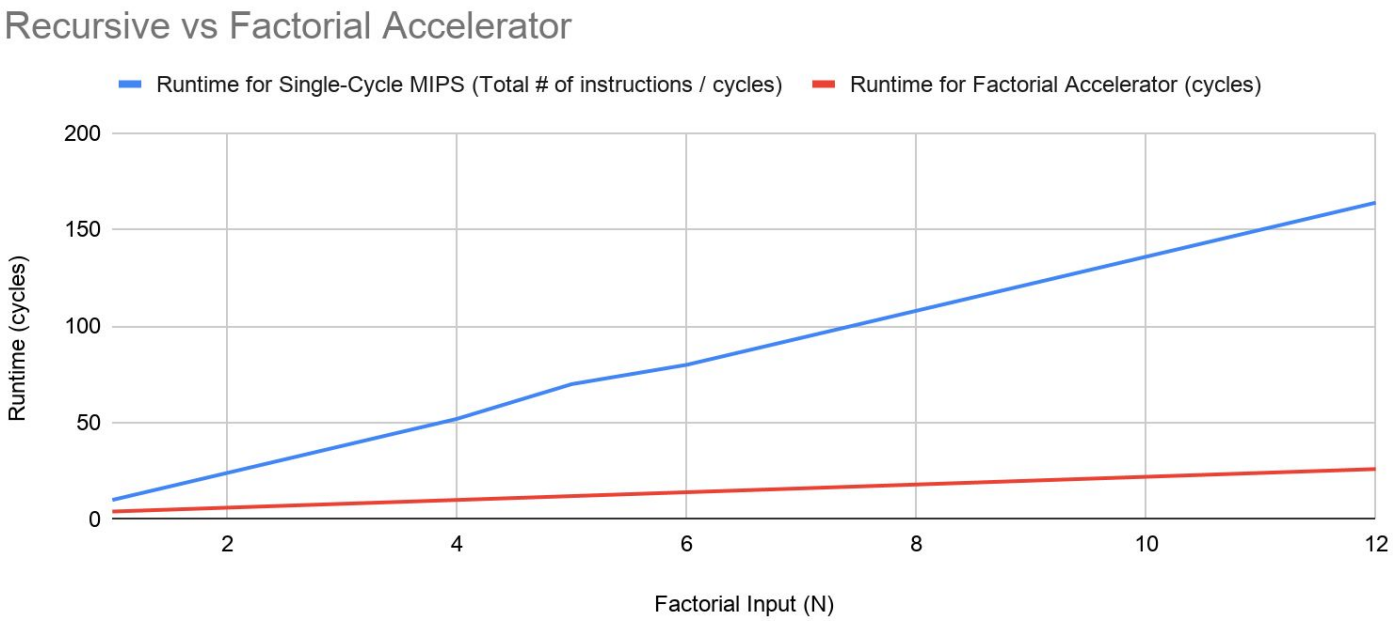


Figure 8. Performance Analysis line chart of *Recursive* vs. *Factorial accelerator*

Waveforms

To functionally verify the designs of our factorial accelerator with interface wrapper, GPIO unit with interface wrapper, single-cycle SoC, and pipelined SoC, our team 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. The testbench code for each module is placed at the end of the respective appendix containing the code for that module. See the below table for a reference.

Table 8: Testbench code location reference table

Module Under Test	**Testbench Code Location**
Factorial Accelerator with Interface Wrapper	End of Appendix F
GPIO Unit with Interface Wrapper	End of Appendix G
Single-Cycle SoC	End of Appendix I

Factorial Accelerator with Interface Wrapper Testbench

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 3 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.

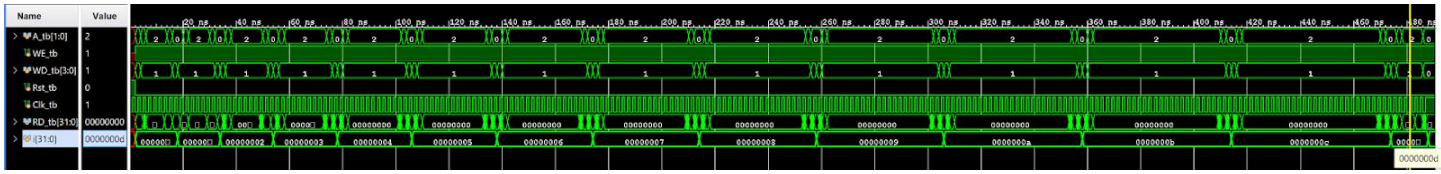


Figure 9: Waveform output for n = 0 through n = 13. Each iteration of n gets larger as more time is needed to calculate n! as n grows larger

```
Error bit triggered for n =          13
xsim: Time (s): cpu = 00:00:11 ; elapsed = 00:00:07 . Memory (MB): peak = 709.805 ; gain = 47.348
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_fact_top_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:16 ; elapsed = 00:00:16 . Memory (MB): peak = 709.805 ; gain = 48.035
```

Figure 10: Console output showing the error bit triggered for n > 12

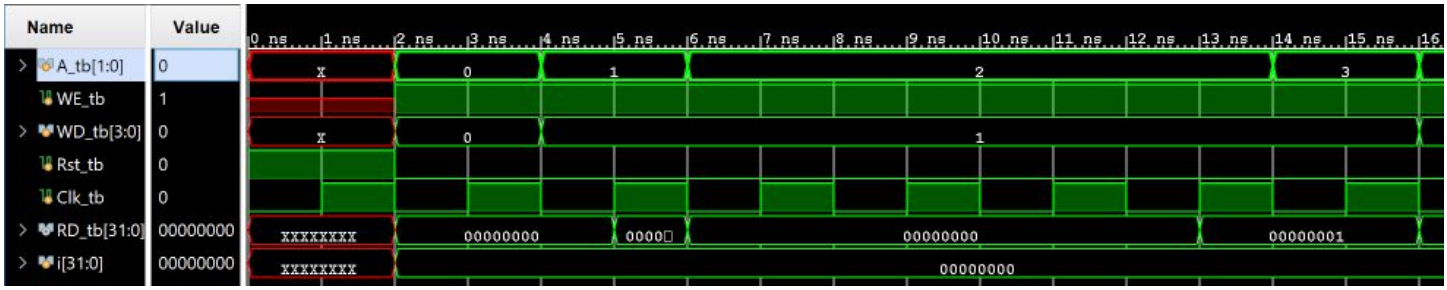


Figure 11: A close up view of n = 0. RD_tb shows the output of the factorial interface wrapper. It changes as input A (address) changes according to Table 3. When A = 0, input n = 0 is stored internally in the wrapper from WD_tb. When A = 1, the Go bit is stored in the wrapper from WD_tb. When A = 2, the calculation of n! occurs and the testbench waits until RD_tb != 0 indicated the Done or Error bit is triggered. In this case, RD_tb = 1 since the Done bit is triggered. When A = 3, the result of n! is shown. In this case, n! = 1 for n = 0, which is the expected output.

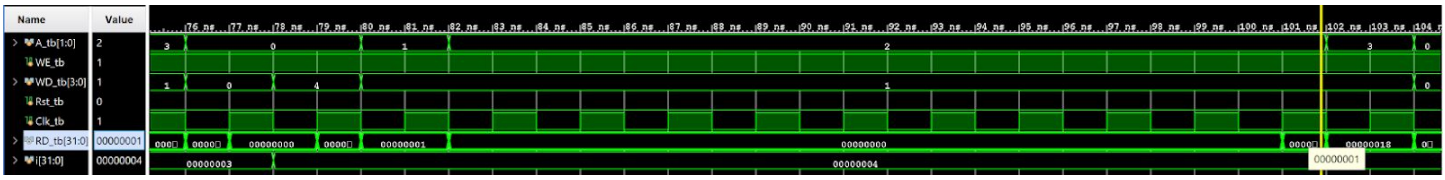


Figure 12: A close up view of n = 4. Same as in Figure 11, RD_tb shows the output of the factorial interface wrapper. When A = 0, WD_tb gives n = 4 to the wrapper, which is stored in a register. When A = 1, the Go bit is given to the wrapper, which is stored in a register. When A = 2, the calculation 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 0x18 indicating the correct output of 4!

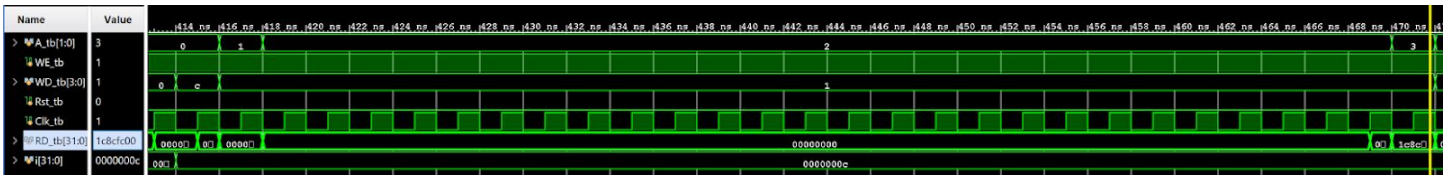


Figure 13: A close up view of n = 12. RD_tb shows the output of the factorial interface, which changes depending on input A. When A = 0, WD_tb gives 12 to the wrapper, which is stored in a register. When A = 1, WD_tb gives the Go bit. When A = 2, the calculation takes place until RD_tb shows 1 indicating the Done bit is set. When A = 3, RD_tb shows 0x1c8cf00, which is the correct result for 12!

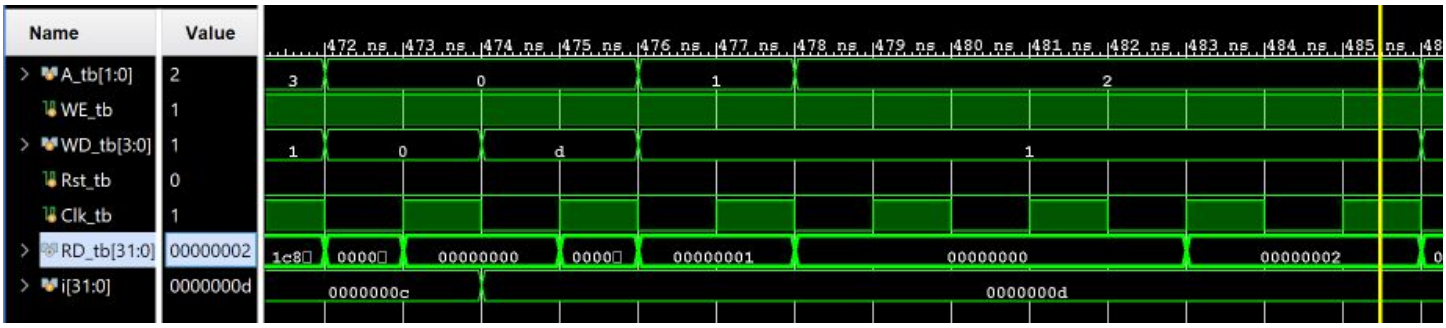


Figure 14: A close up view of n = 13. Note than no value is calculated here since n! is only calculated for n < 13. n = 13 is still loaded and stored in the wrapper when A = 0, and the Go bit is given when A =1. Of note is when A = 2. The calculation kicks off but stops when the error bit is triggered. This is shown in RD_tb = 2. Since the error bit is the second bit of the output of the wrapper when A = 2, this is correct. Figure 10 shows the message that is displayed on the console for this case.

GPIO Unit with Interface Wrapper Testbench

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 5. We also examined the gpO1 and gpO2 outputs. The waveform of this functional verification is shown in Figure 15.

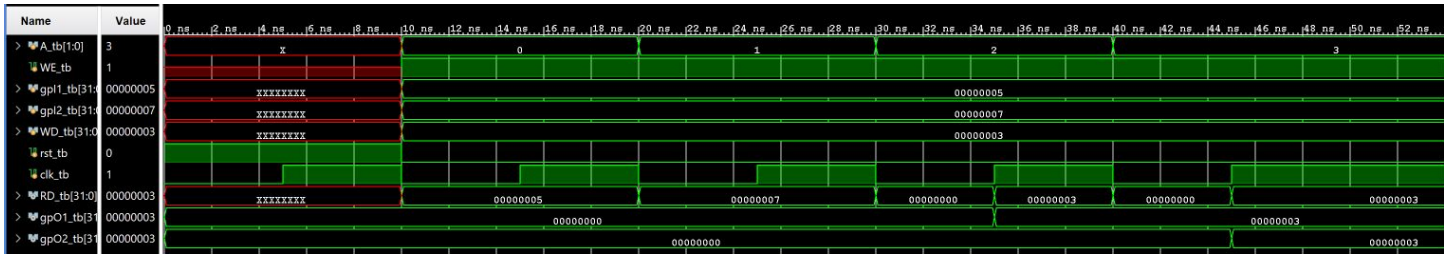


Figure 15: This waveform output shows the result of a testbench for the GPIO unit with interface wrapper. To safely confirm that our design works as intended, RD must output the following: the value of gpI1 when A = 0, the value of gpI2 when A = 1, the value of WD when A = 2, the value of WD when A = 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.

Single-Cycle SoC Testbench

To functionally verify the design of the single-cycle CPU and the system-on-chip (SoC) design, a testbench was created to stimulate inputs to the SoC. If the CPU design worked properly and was connected to all memory modules and I/O devices correctly, the SoC's outputs would follow our expectations. The testbench for the single-cycle SoC only gives an input to the GPIO input ports (gpI1 and gpI2) and drives the clock. Everything else happens internally. To help verify the design, the following signals were probed from the CPU: instr, pc_current, alu_out, wd_dm, rd_dm, and rd3. GPIO outputs were also monitored to ensure the correct result of the factorial calculation. A "memfile.dat" memory file containing machine code for the given driver assembly code was used as the input for the instruction memory module. This memory file and the corresponding assembly code are given at the beginning of Appendix H.

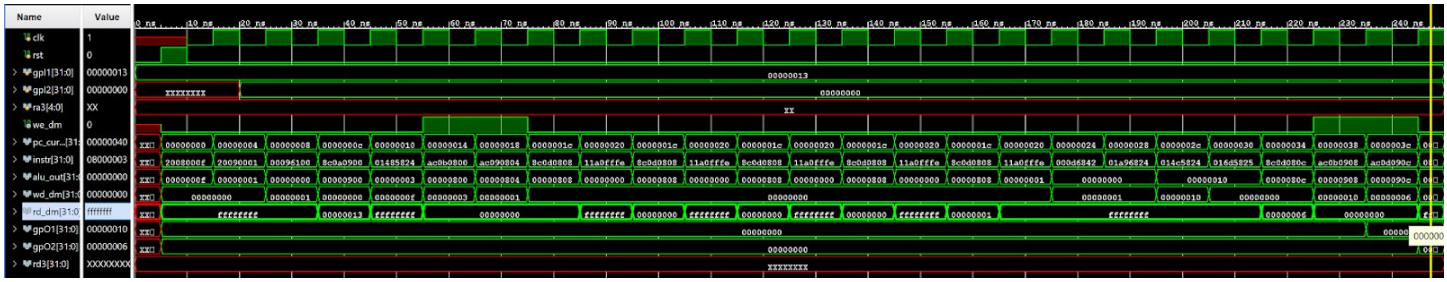


Figure 16: The full screenshot of the single-cycle SoC testbench. Execution continues until the final jump instruction is reached. For this run, input n is given on gpI1. gpI1 = 0x13 = 10011. The rightmost 4 bits encode n while the leftmost bit encodes the display select bit used later in hardware validation. Display select does not do anything in this simulation. Of note is the output shown on gpO2 at the very right of the waveform. The marker is placed on the last clock cycle to help show what these values are. Notice that gpO2 = 0x6. This is the

result of 3!. The SoC correctly calculates n! where n = 3 and places the output on a GPIO output port. Three clock cycles before the end, 0x6 is shown on rd_dm. This is the result of a *lw* instruction in which the CPU reads the result/output of the factorial accelerator. Each instruction being processed can also be seen on the entire length of the waveform. This can be compared with the MARS tool executing the same assembly program to verify proper execution. We also examined alu_out during R-Type instructions to ensure things were operating smoothly. gpO1 also shows 0x10 as the display select input but was set.

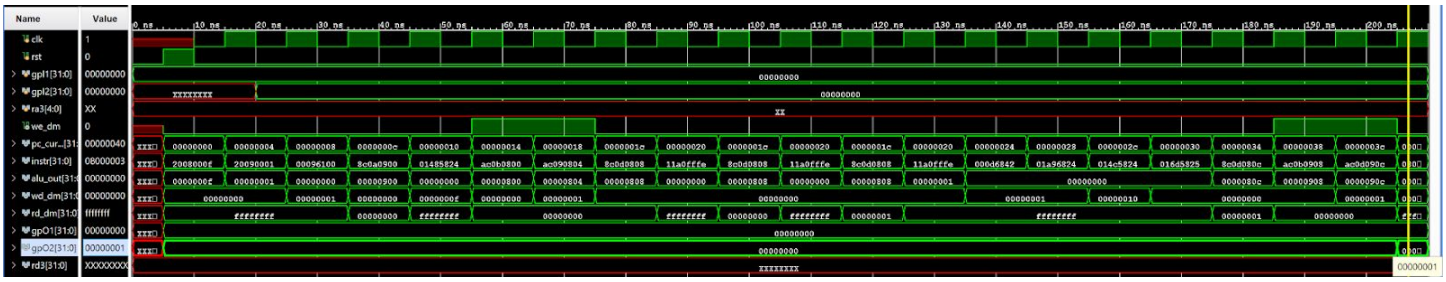


Figure 17: Another example of execution of the driver assembly program with n = 0. The result 0! = 0x1 is shown on gpO2 correctly.

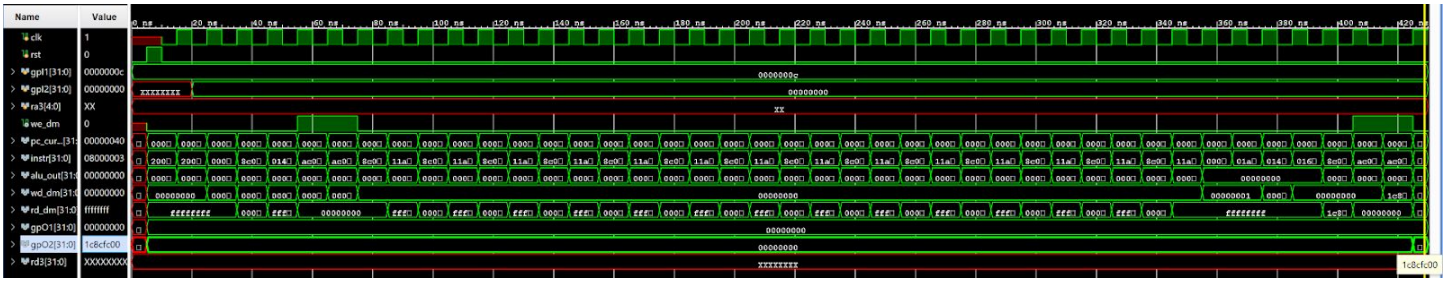


Figure 18: Another example of execution of the driver assembly program with n = 12. The result 12! = 0x1c8cfc00 is shown correctly. Note that program execution is much longer than before because the CPU waits for the factorial accelerator to finish its calculation before continuing program execution flow.

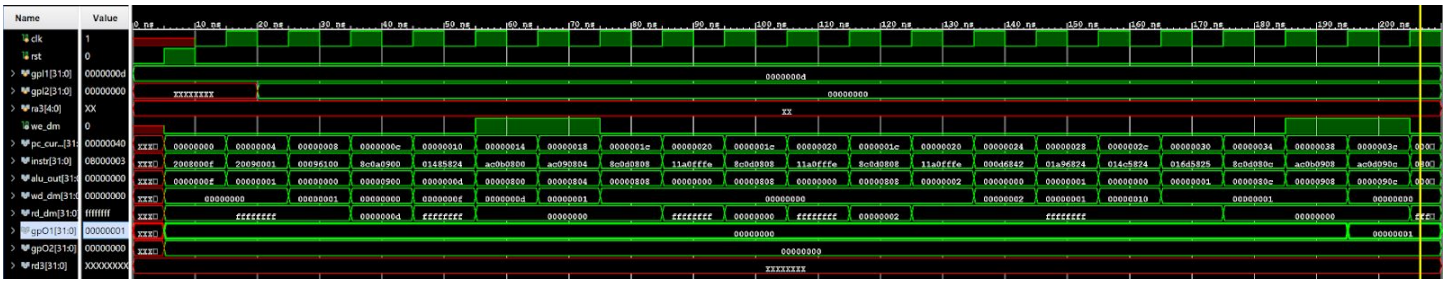


Figure 19: In this example, n = 13. n! should not calculate for this. Instead, gpO1 should show an error bit at the first bit of its output. This is correctly shown in the waveform at the right of the waveform.

Pipelined SoC Testbench

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. Since the GPIO unit and factorial accelerator were verified in the single-cycle SoC testing, we assumed that they worked correctly. Therefore, only the CPU was modified in this test. Similar to the testing of the single-cycle SoC, we gave the pipelined SoC an input for *n* on gpI1 and gpI2. We examined the same outputs from the single-cycle CPU along with some new ones to prove the pipeline was working properly. We also added and removed new probes to debug the processor along the way. Note the machine code outputs from each stage in the pipeline: instrF, instrD, instrE, instrM, and instrW. These instructions seem to cascade in the waveform output indicating that each clock cycle moves the instruction through the pipeline where it is then processed by that respective stage. 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 heavily modified with NOPs between instructions with data dependencies as described earlier in the Design Methodology section. Execution stops when the processor reaches the final jump instruction.

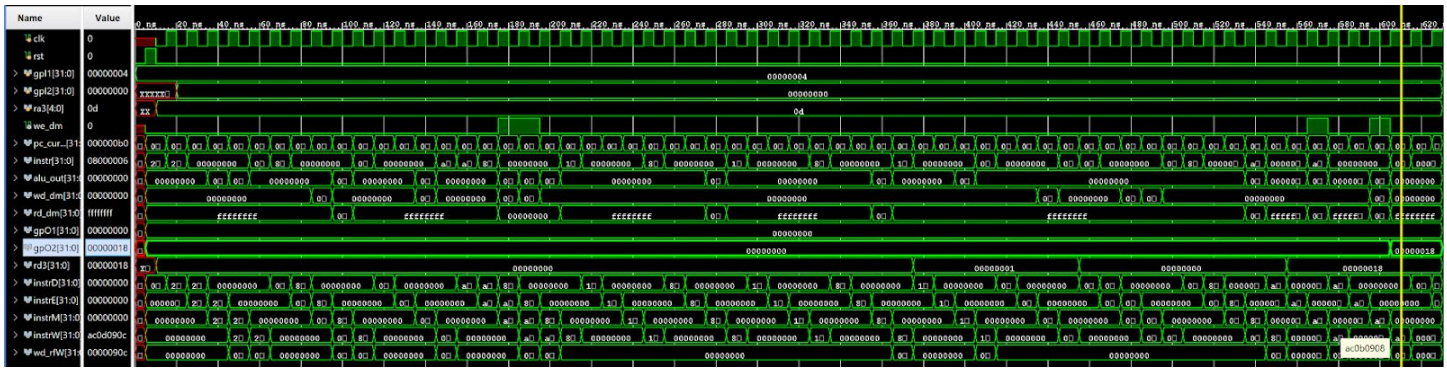


Figure 20: This massive waveform output shows the entire execution process of the pipelined MIPS processor. Of note is the input port gpI1 = 0x4 with n = 4 and the output port gpO2 with n! = 0x18 = 24. This is the correct result of the factorial calculation. The setup for this waveform is similar to the waveforms in the single-cycle SoC simulations. Since not much is visible in this waveform, more will be explained later.

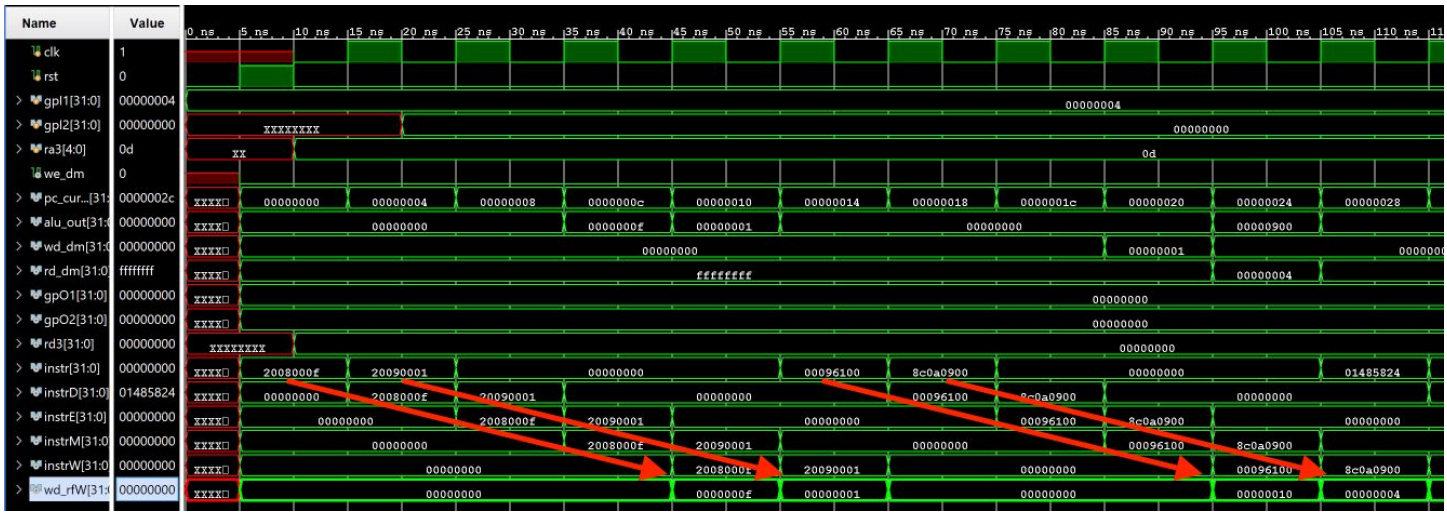


Figure 21: To verify that the pipeline is actually working, look at how the machine code cascade through the waveform. Instructions are loaded into the CPU at the beginning of a sequence of 5 clock cycles. That

instruction then moves through each stage as shown in instrD, instrE, instrM, and instrW every clock cycle. The instruction is processed in stages as evidenced by alu_out when an instruction is shown in instrE. For example, 0x20090001 is an add instruction that adds 0x0 and 0xf. When this instruction reaches instrE, alu_out shows 0xf indicating proper execution of the instruction. This instruction's data is later written back to the register file in the next stage. This is evidenced by the probe on wd_rfW which is the bus to write data back to the register file. Continuing the example, when instrW = 0x20090001, wd_rfW = 0xf indicating that the correct data is being written back to the register file.

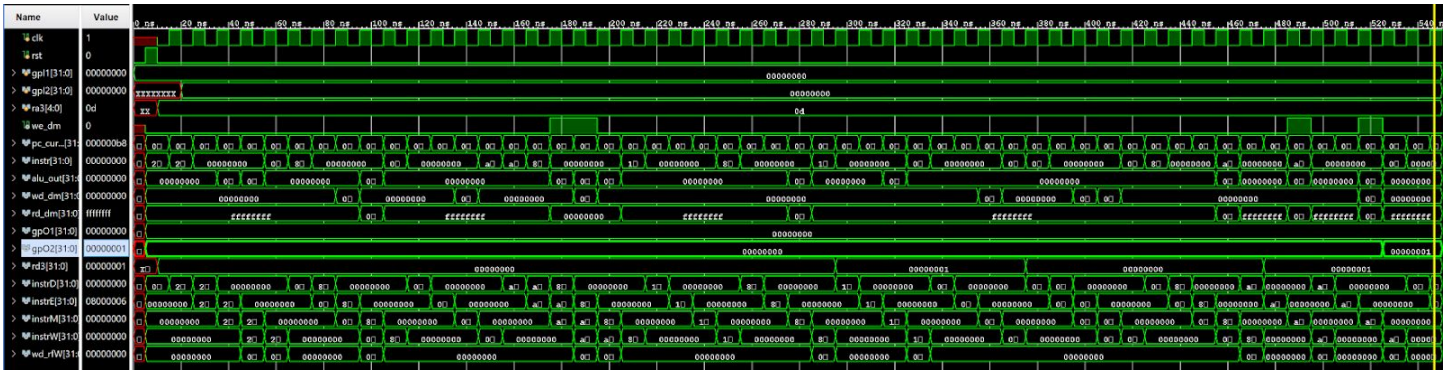


Figure 22: Another example of program execution where n = 0. gpO2 correctly shows the result of n! = 1.

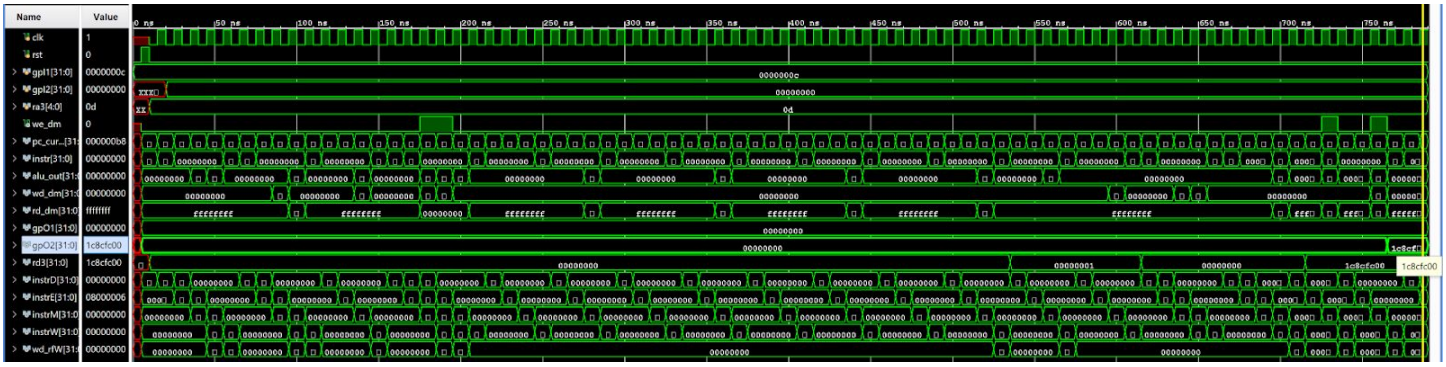


Figure 23: Another example of program execution where n = 12. gpO2 correctly shows the result of n! = 0x1c8cfc00.

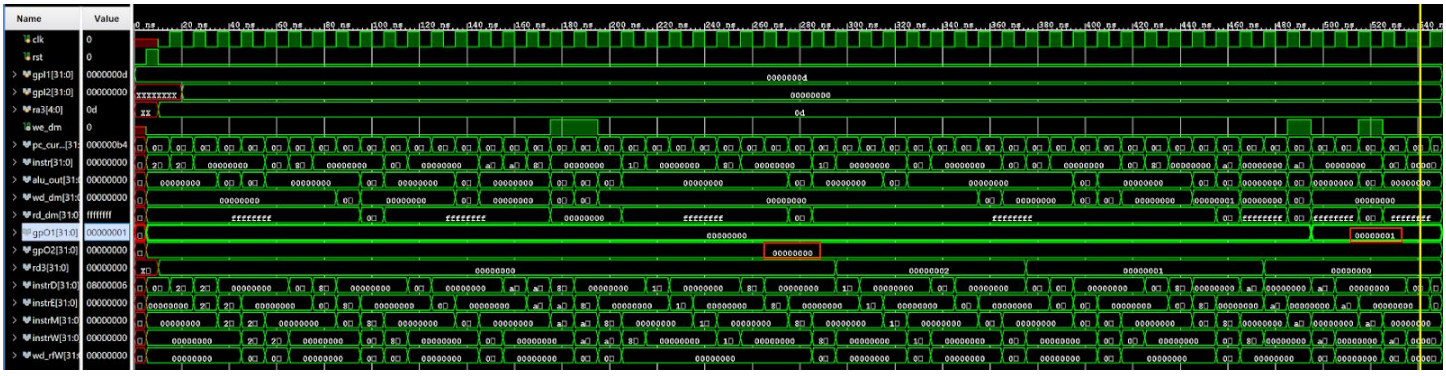


Figure 24: Another example of program execution where n = 13. In this case, no value for n! should be calculated. Instead, gpO2 should be 0 and gpO1 should show the Error bit on the first bit of its output (i.e. gpO1 = 0x1).

Hardware Validation

Validating the design of the single-cycle MIPS processor and SoC containing the new I/O devices on the Basys3 FPGA board was successful! The process for validating the pipelined MIPS processor on the same SoC was similar to validating the single-cycle SoC.

To validate the design, we constructed a hardware validation environment as shown in Figure 25. The components used are similar to components used in Assignment 7. These include a clock generator module, the SoC containing either the pipelined or single-cycle CPU, and a 7-seg LED display hex multiplexer. Table 9 below explains what these modules are. Table 10 below explains how the the hardware devices on the Basys3 board are connected in the FPGA validation environment. Figure 25 shows a picture of the hardware validation environment on the Basys3 board. Appendices D and E show the hardware validation pictures of the Basys3 board. Appendix D contains the hardware validation pictures for the single-cycle SoC while Appendix E contains the hardware validation pictures for the pipelined SoC.

To test the single-cycle SoC, the single-cycle MIPS processor takes the place of the processor within "System" in the hardware validation environment. To test the pipelined SoC, the pipelined MIPS processor takes the place of the processor within "System" in the hardware validation environment. In this way, only the processor changes while the validation environment is unchanged.

A snippet of the validation process is shown below to explain how both SoC designs were validated on the Basys3 board:

1) An input n is given on SW3 - SW0 in binary.
2) *n!* is calculated on a calculator, converted to hexadecimal, and compared to the value shown on the 7-segment display.
3) If the calculated value is large enough, both the lower and upper half-words are checked by flipping SW4 to toggle the display select.
4) Steps 1 through 2 are repeated to check edge cases: $n = 0$, $n = 1$, and $n = 12$. Also check $n = 4$, $n = 10$
5) Input $n = 13$ on the switches to confirm the error LEDs light up to indicate too large of an input.

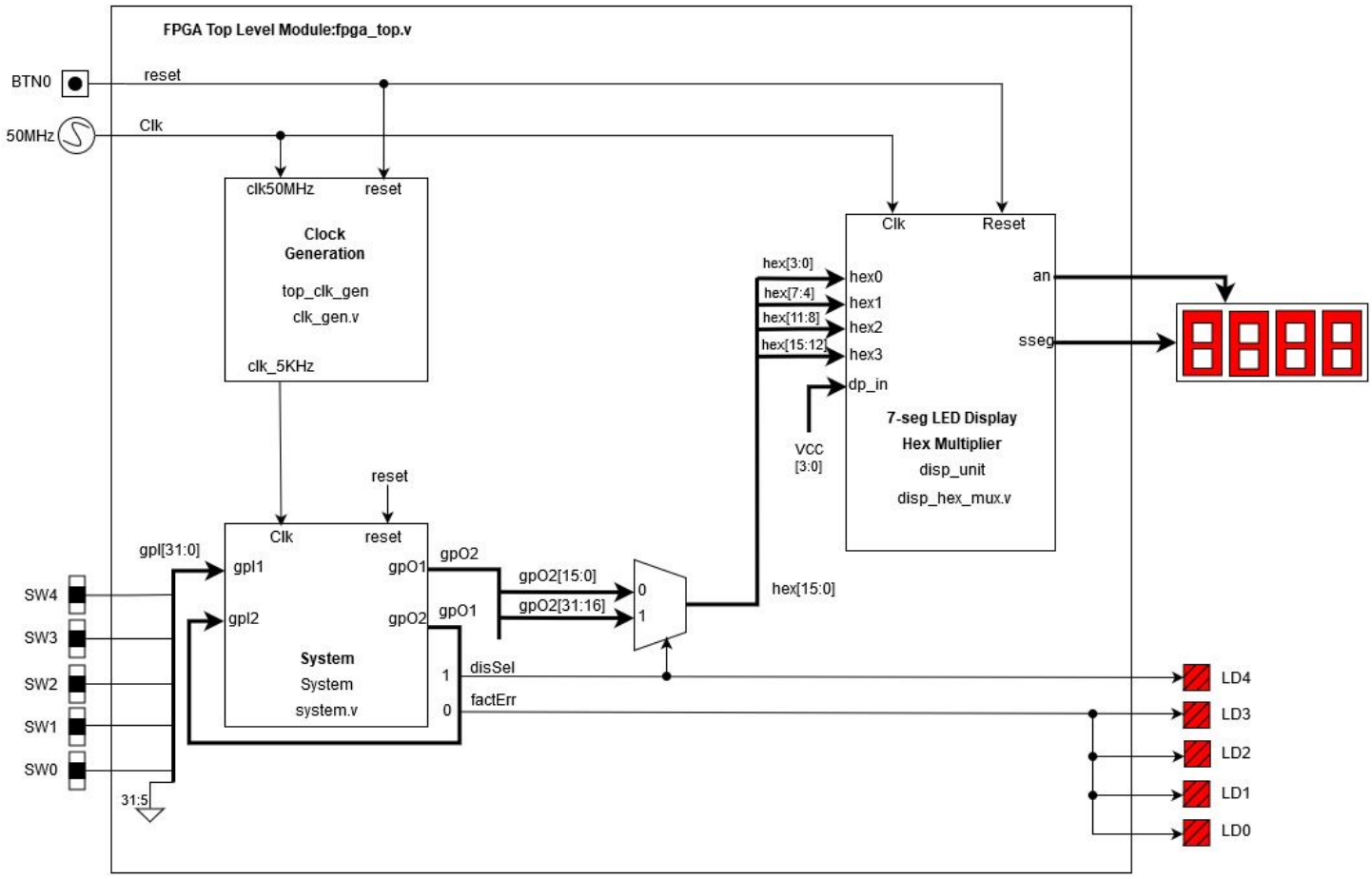


Figure 25. FPGA hardware validation environment diagram

Table 9: Hardware Validation Environment Modules

Module	Description
clk_gen	Takes an input 50MHz clock and converts it to a 5KHz clock. This clock signal is used to drive modules on the system-on-chip such as the MIPS processor and factorial accelerator.
SoC / System	The SoC takes an input clk and reset signal as well as general purpose inputs connected to external switches on the Basys3 board. It also outputs data via general purpose outputs to the 7-seg LED display hex mux and the connected LEDs. This module houses the MIPS processor, its I/O devices, and memory devices.
7-seg LED Display Hex Multiplexer	This module houses the logic for converting input data into signals to drive the 4 7-segment LED displays. It converts a digital hex digit into a visible hex digit on the LED display.

Button_debouncer (Not Used)	This module was not used in the final validation. However, it was used to debug problems with the processor as it allowed us to step through program execution. The module takes an input from a connected button on the Basys3 board, smooths out the input signal, and outputs a clean square wave as a clock signal for the SoC.

Table 10: Hardware Configuration

Hardware Device	**What It Does**
BTN0	This button acts as the reset signal for the SoC, clock generator, and 7-seg LED Display Hex Multiplexer.
SW4	This switch acts as the display select for the 7-segment display. It controls which half-word is shown on the display. For example, if the factorial accelerator computes a value that requires more than 16 bits to represent that value, only the lower or upper 16 bits (converted to hex) will be shown on the display depending on the position of this switch.
SW3 - SW0	These switches act as a way to input a value for n to perform n! on. Values for n are entered in binary. These switches are connected to the GPIO ports on the SoC.
LD4	This LED displays the status of the display select switch. The LED is lit when the switch is on.
LD3 - LD0	This LED displays the status of the Error bit output from the factorial accelerator via the GPIO output ports on the SoC. These LEDs are lit when an input n is greater than 12.
7-Segment Display	This display shows the result of the factorial calculation. The display select switch controls whether the upper or lower 16 bits of a calculated value are displayed on this display.

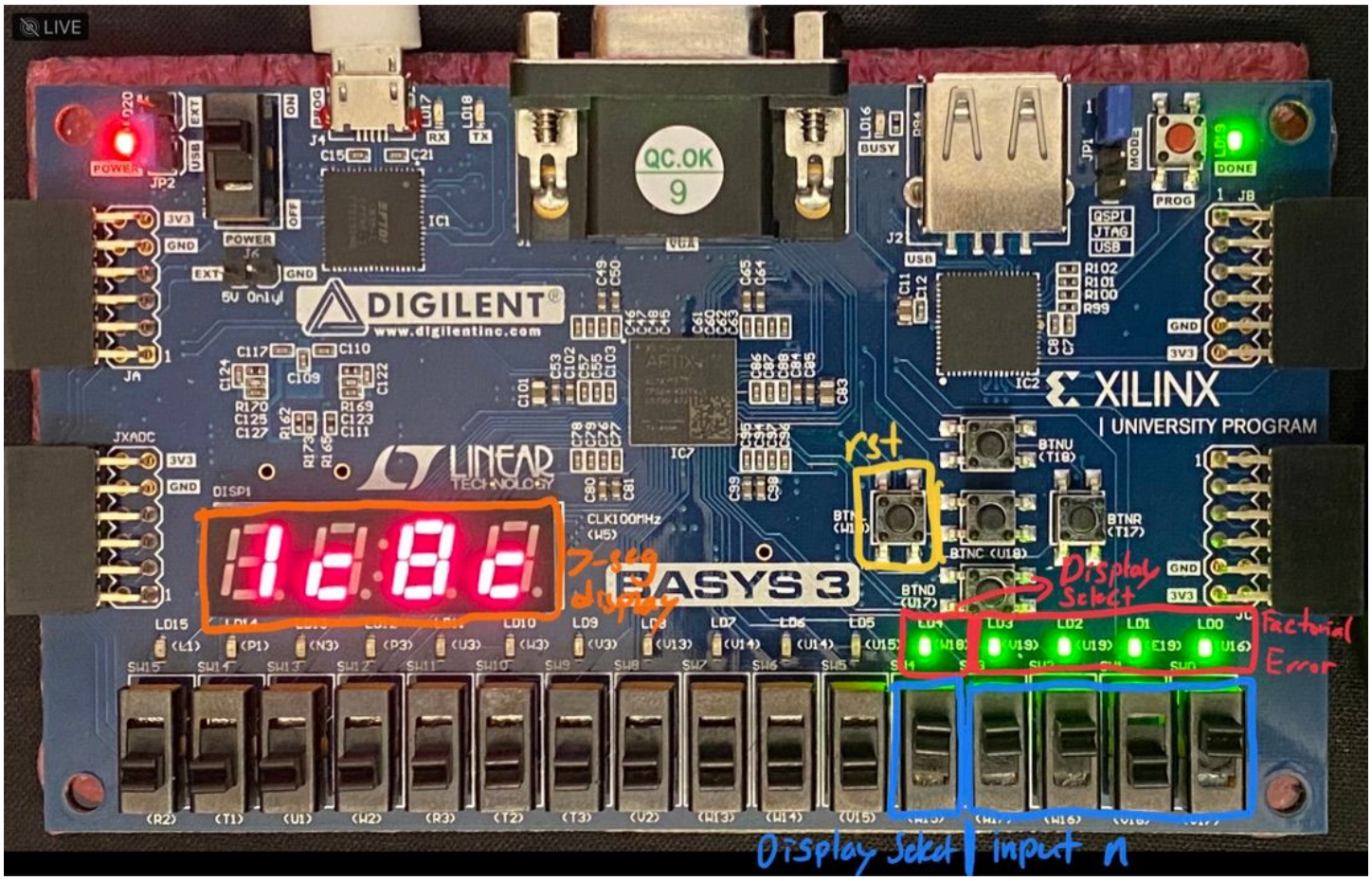


Figure 25: Hardware configuration of the Basys3 board via the constraints file

Accomplished Procedures

1. *Drafted the pipelined microarchitecture design*
2. *Drafted the SoC interface*
3. *CU tables and the memory maps for the SoC design*
4. *Performance analysis of the factorial design*
5. *Unit-level waveforms from the simulation which includes the interface wrappers*
6. *SoC interface design (includes the GPIO, single-cycle processor, and factorial unit)*
7. *Demonstration of the integrated SoC design upon the FPGA board*
8. *The integrated SoC with a pipelined design*
9. *Demonstration of the improved pipelined integration of the SoC upon the FPGA board*

Conclusion

The tasks required in this assignment were completed successfully. The team successfully designed, implemented, and tested a system-on-chip (SoC) design with an interchangeable CPU. The SoC was functionally verified and validated on hardware with both a single-cycle MIPS processor and a pipelined MIPS processor. The SoC was complete with memory modules used in previous assignments as well as new I/O devices such as the factorial accelerator and GPIO unit. These I/O devices were external to the CPU and required interface wrappers with an address decoder providing logic to address each device when necessary. The team successfully designed these interface wrappers to work with the internal I/O units and interface with the interchangeable CPU on the SoC. Through the design process learned in CMPE 125 and CMPE 140, the team started by designing the various modules we needed to implement (i.e. SoC, factorial accelerator, pipelined MIPS processor etc.), implementing the designs in Verilog using Vivado, functionally verifying the designs with a suite of testbenches, and validated the designs using a Basys3 FPGA device.

However, the tasks were not completed in one day nor were they completed on the first try. A great deal of time and mental fortitude went into tackling this assignment. From the second we demoed Assignment 7, the team began brainstorming how to approach this assignment. We knew from the get-go that we were at a disadvantage because we failed to group our single-cycle datapath by pipeline stage as demonstrated in lecture. Therefore, we knew we would have to dedicate more time and focus to ensure our new datapath (from scratch) was designed carefully and free from silly mistakes. By completing a first draft of our pipelined datapath for our pipelined MIPS processor, we grew more confident in our abilities to complete the assignment. After some feedback from our TA, we went back to the drawing board to edit our design. With that out of the way, we turned our focus to implementing our SoC with the single-cycle CPU. Implementing the newly designed factorial accelerator and GPIO units with interface wrappers was simple enough. However, we faced trouble when testing the factorial accelerator inside the interface wrapper. We had to fix issues with the logic of the registers within the wrapper and test until the design worked. Finally, we were able to functionally verify and validate the design of the single-cycle SoC in hardware. We ran into most of our problems when implementing the pipelined MIPS processor. We struggled with adding NOPs to our driver program until we remembered about data dependencies. We also found some small mistakes in our design such as switching the inputs to the ports of a multiplexer in the writeback stage when we copied the design from Assignment 7's datapath. After we spent several days sorting through these issues and getting external help when needed, we successfully finished all required tasks. We even had time to attempt data forwarding (but not enough to debug the design).

The end result was something we were extremely proud of. A lot of work went into getting a working pipelined CPU. A ton of work went into writing this report. Our team is happy to present this report detailing our efforts. We are also thankful to our lab TA, professor Hyeran Jeon, and Donald Hung for designing this assignment. We learned the design and validation process of creating a CPU based on its instruction set architecture. We learned how to improve the performance of a single-cycle CPU by using pipelining and hazard control to create a processor than can (in theory) execute 5 instructions per clock cycle when the pipeline is fully loaded.

Appendix A: Control Unit and Memory Module Diagrams

Note: These modules are shared between the single-cycle and pipelined MIPS processors

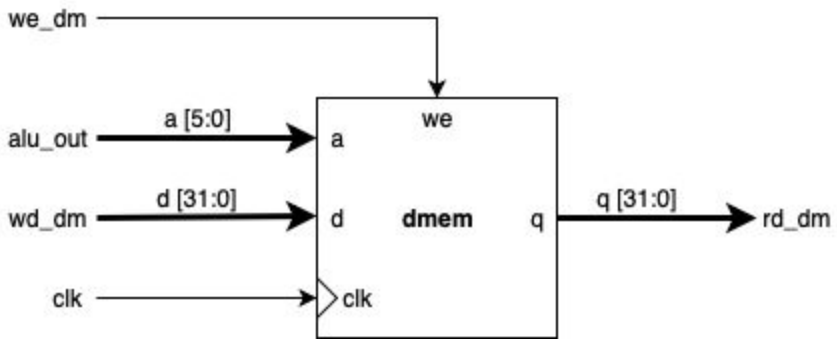


Figure 26: Data memory module used in both single-cycle and pipelined SoC designs

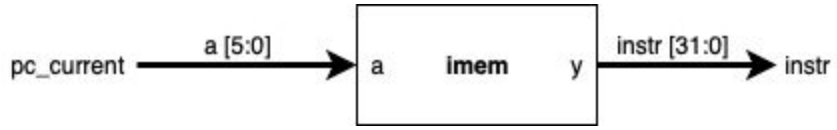


Figure 27: Instruction memory module used in both single-cycle and pipelined SoC designs

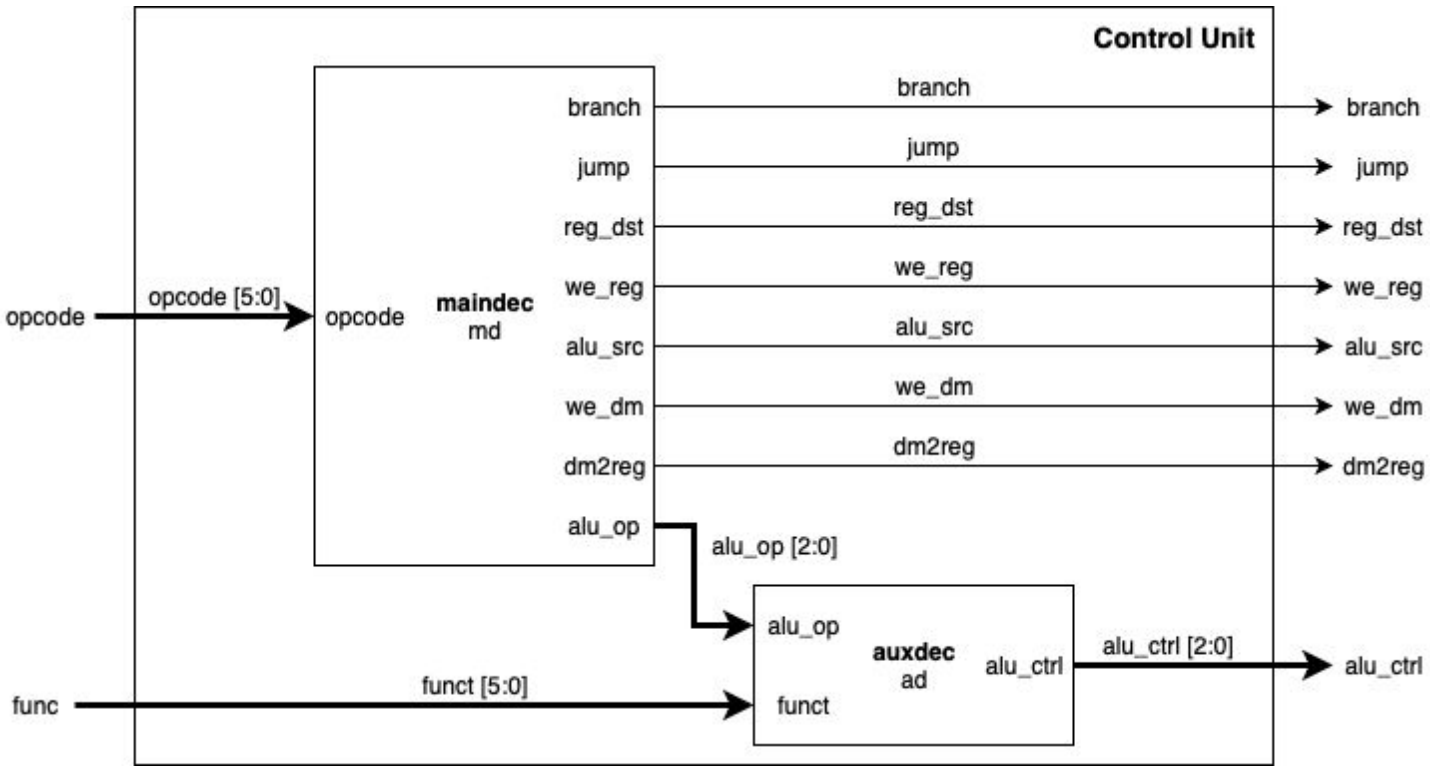


Figure 28: Control unit microarchitecture design used in both single-cycle and pipelined MIPS processor designs. Refer to the corresponding truth tables for each processor in the respective appendices

Appendix B: Single-Cycle MIPS Processor Design Diagrams and Truth Tables

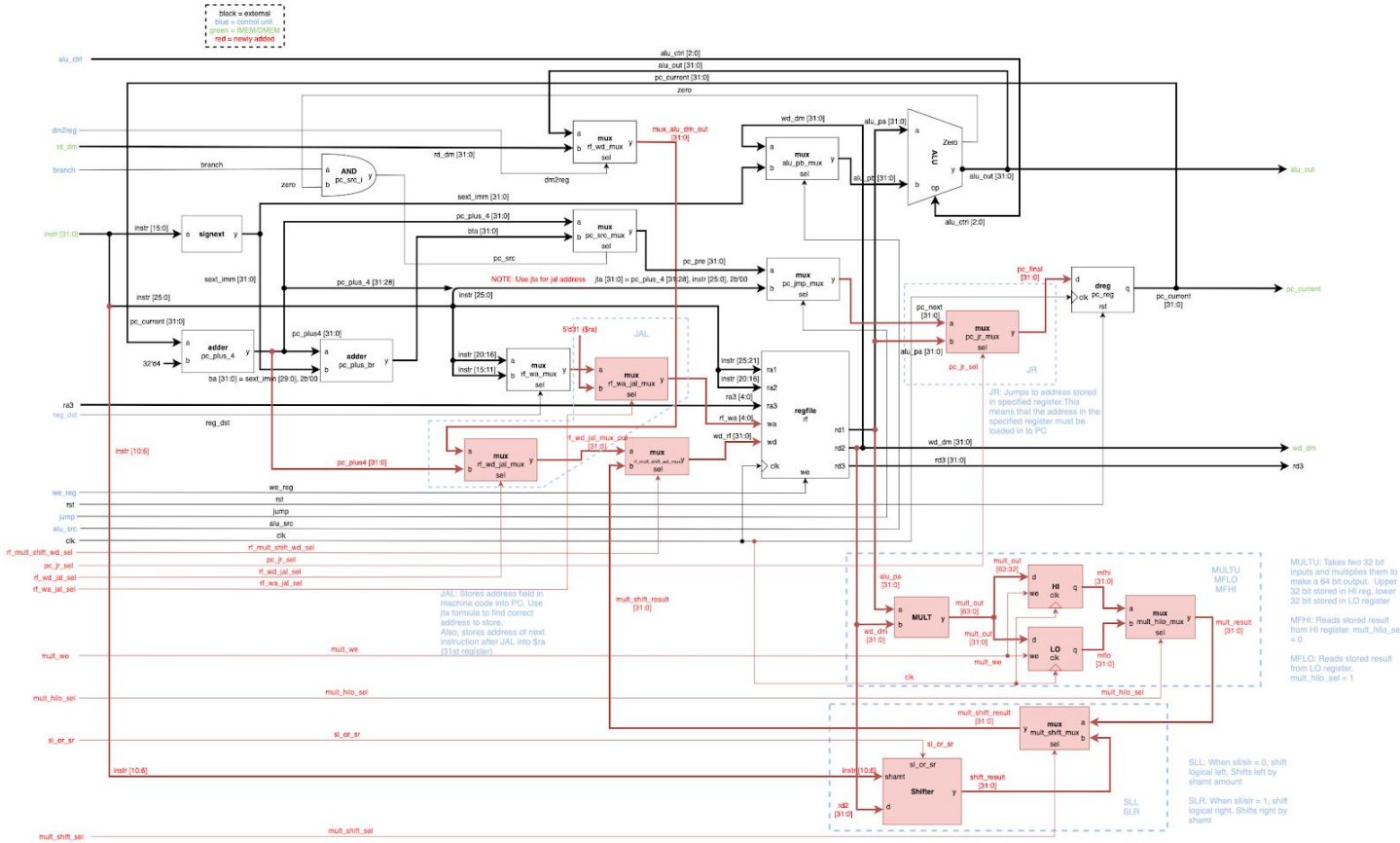
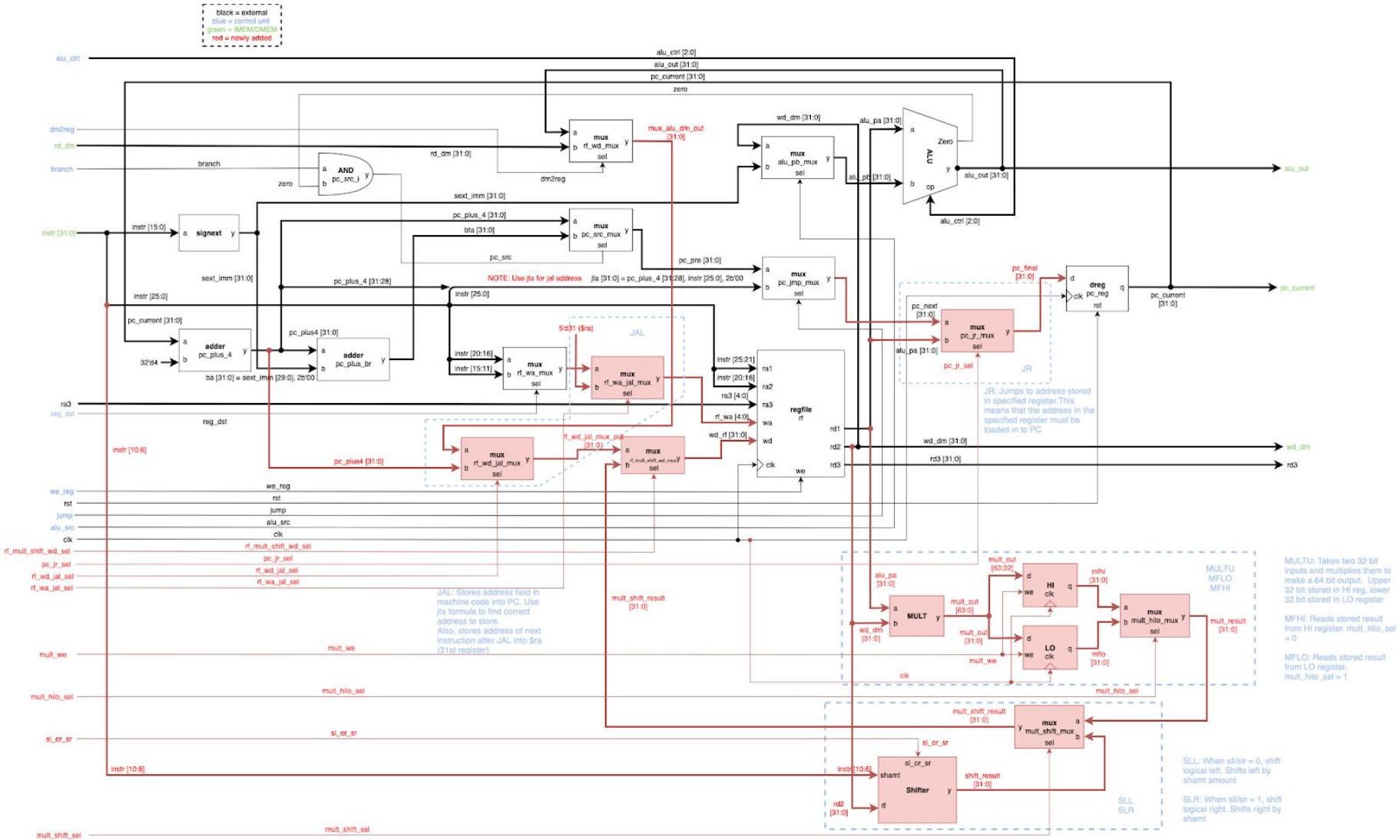


Figure 29: Datapath of the single-cycle MIPS processor. A larger, landscape diagram is shown on the next page.


black = external
blue = control unit
green = IMEM/DMEM
red = newly added
alu_ctrl
alu_ctrl [2:0]
alu_out [31:0]
pc_current [31:0]
zero
dm2reg
rd_dm
branch
AND
pc_src_i
mux
rf_wd_mux
sel
mux_alu_dm_out [31:0]
rd_dm [31:0]
dm2reg
wd_dm [31:0]
alu_pa [31:0]
mux
alu_pb_mux
sel
ALU
Zero
cp
alu_pb [31:0]
alu_out [31:0]
alu_out
sext_imm [31:0]
instr [31:0]
instr [15:0]
signext
pc_plus_4 [31:0]
bta [31:0]
mux
pc_src_mux
sel
pc_src
pc_pre [31:0]
mux
pc_jmp_mux
sel
NOTE: Use jta for jal address
jta [31:0] = pc_plus_4 [31:28], instr [25:0], 2'b'00
pc_final [31:0]
dreg
pc_reg
rst
pc_current [31:0]
pc_current
sext_imm [31:0]
pc_plus_4 [31:28]
instr [25:0]
pc_current [31:0]
adder
pc_plus_4
32'd4
pc_plus4 [31:0]
adder
pc_plus_br
ba [31:0] = sext_imm [29:0], 2'b'00
instr [20:16]
instr [15:11]
mux
rf_wa_mux
sel
5'd31 ($ra)
JAL
mux
rf_wa_jal_mux
sel
instr [25:21]
instr [20:16]
ra3 [4:0]
rf_wa [4:0]
wd_rf [31:0]
regfile
rf
ra1
ra2
ra3
wa
wd
clk
we
rd1
rd2
rd3
pc_next [31:0]
mux
pc_jr_mux
sel
alu_pa [31:0]
pc_jr_sel
JR
JR: Jumps to address stored in specified register.This means that the address in the specified register must be loaded in to PC
ra3
reg_dst
instr [10:6]
pc_plus4 [31:0]
rf_wd_jal_mux_out [31:0]
mux
rf_wd_jal_mux
sel
mux
rf_mult_shift_wd_mux
sel
wd_dm [31:0]
wd_dm
rd3 [31:0]
rd3
we_reg
rst
jump
alu_src
clk
rf_mult_shift_wd_sel
pc_jr_sel
rf_wd_jal_sel
rf_wa_jal_sel
JAL: Stores address field in machine code into PC. Use jta formula to find correct address to store.
Also, stores address of next instruction after JAL into $ra (31st register)
mult_shift_result [31:0]
alu_pa [31:0]
MULT
wd_dm [31:0]
mult_out [63:32]
mult_out [63:0]
mult_out [31:0]
HI
LO
we
clk
mfhi [31:0]
mflo [31:0]
mux
mult_hilo_mux
sel
mult_result [31:0]
MULTU
MFLO
MFHI
MULTU: Takes two 32 bit inputs and multiplies them to make a 64 bit output. Upper 32 bit stored in HI reg, lower 32 bit stored in LO register
MFHI: Reads stored result from HI register. mult_hilo_sel = 0
MFLO: Reads stored result from LO register. mult_hilo_sel = 1
mult_we
mult_hilo_sel
sl_or_sr
instr [10:6]
mult_shift_sel
mux
mult_shift_mux
sel
sl_or_sr
shamt
Shifter
d
rd2 [31:0]
shift_result [31:0]
SLL
SLR
SLL: When sll/slr = 0, shift logical left. Shifts left by shamt amount
SLR: When sll/slr = 1, shift logical right. Shifts right by shamt

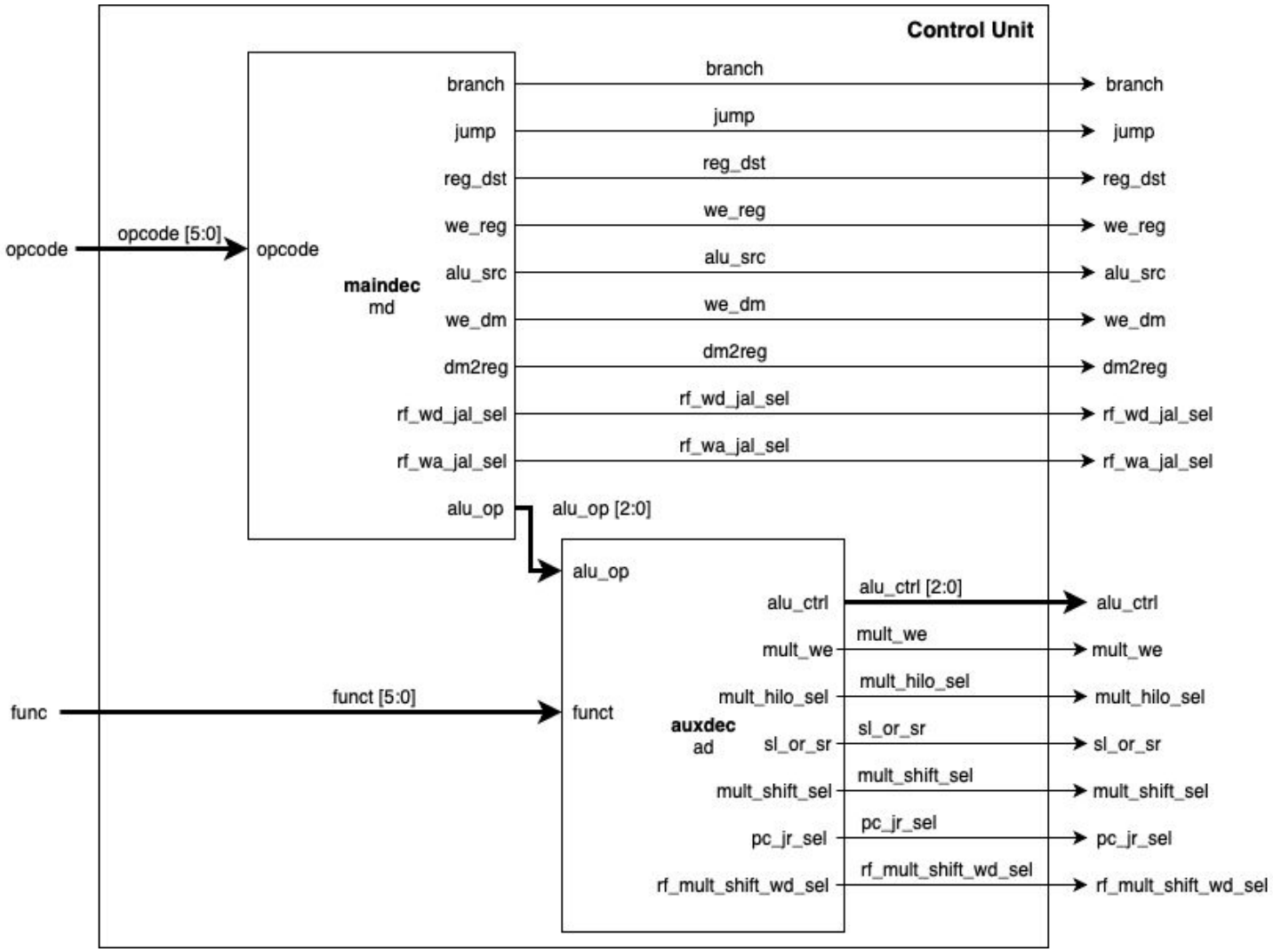


Figure 30: Control unit of single-cycle MIPS Processor

Table 11: Auxiliary Decoder Control Unit Truth Table

Type	Instruction	Input		Output						
		opcode [31:26]	*funct [5:0]*	*pc_jr_sel*	*rf_mult_shift_wd_sel*	*mult_we*	*mult_hilo_sel*	*mult_shift_sel*	*sl_or_sr*	*alu_ctrl [2:0]*
R	MULTU	00_0000	01_1000	0	0	1	0	0	0	0
	MFHI	00_0000	01_0000	0	1	0	0	0	0	0
	MFLO	00_0000	01_0010	0	1	0	1	0	0	0
	JR	00_0000	00_1000	1	0	0	0	0	0	0
	SLL	00_0000	00_0000	0	1	0	0	1	0	0
	SLR	00_0000	00_0010	0	1	0	0	1	1	0
J	JAL	00_0011	X	0	0	0	0	0	0	0
R	ADD	00_0000	10_0000	0	0	0	0	0	0	10
	SUB	00_0000	10_0010	0	0	0	0	0	0	110
	AND	00_0000	10_0100	0	0	0	0	0	0	0
	OR	00_0000	10_0101	0	0	0	0	0	0	1
	SLT	00_0000	10_1010	0	0	0	0	0	0	111
I	LW	10_0011	X	0	0	0	0	0	0	10
	SW	10_1011	X	0	0	0	0	0	0	10
	BEQ	00_0100	X	0	0	0	0	0	0	110
	ADDI	00_1000	X	0	0	0	0	0	0	10
J	J	00_0010	X	0	0	0	0	0	0	0

Table 12: Main Decoder Control Unit Truth Table

Type	Instruction	Input											
		opcode [31:26]	*funct [5:0]*	*rf_wd_jal_sel*	*rf_wa_jal_sel*	*dm2reg*	*we_dm*	*branch*	*alu_src*	*reg_dst*	*we_reg*	*jump*	*alu_op [1:0]*
R	MULTU	00_0000	01_1000	0	0	0	0	0	0	0	0	0	10
	MFHI	00_0000	01_0000	0	0	0	0	0	0	1	1	0	10
	MFLO	00_0000	01_0010	0	0	0	0	0	0	1	1	0	10
	JR	00_0000	00_1000	0	0	0	0	0	0	0	0	X	10
	SLL	00_0000	00_0000	0	0	0	0	0	0	1	1	0	10
	SLR	00_0000	00_0010	0	0	0	0	0	0	1	1	0	10
J	JAL	00_0011	X	1	1	0	0	0	0	0	1	1	0
R	ADD	00_0000	10_0000	0	0	0	0	0	0	1	1	0	10
	SUB	00_0000	10_0010	0	0	0	0	0	0	1	1	0	10
	AND	00_0000	10_0100	0	0	0	0	0	0	1	1	0	10
	OR	00_0000	10_0101	0	0	0	0	0	0	1	1	0	10
	SLT	00_0000	10_1010	0	0	0	0	0	0	1	1	0	10
I	LW	10_0011	X	0	0	1	0	0	1	0	1	0	0
	SW	10_1011	X	0	0	0	1	0	1	0	0	0	0
	BEQ	00_0100	X	0	0	0	0	1	0	0	0	0	1
	ADDI	00_1000	X	0	0	0	0	0	1	0	1	0	0
J	J	00_0010	X	0	0	0	0	0	0	0	0	1	0

Appendix C: Pipelined MIPS Processor Design Diagrams and Truth Tables

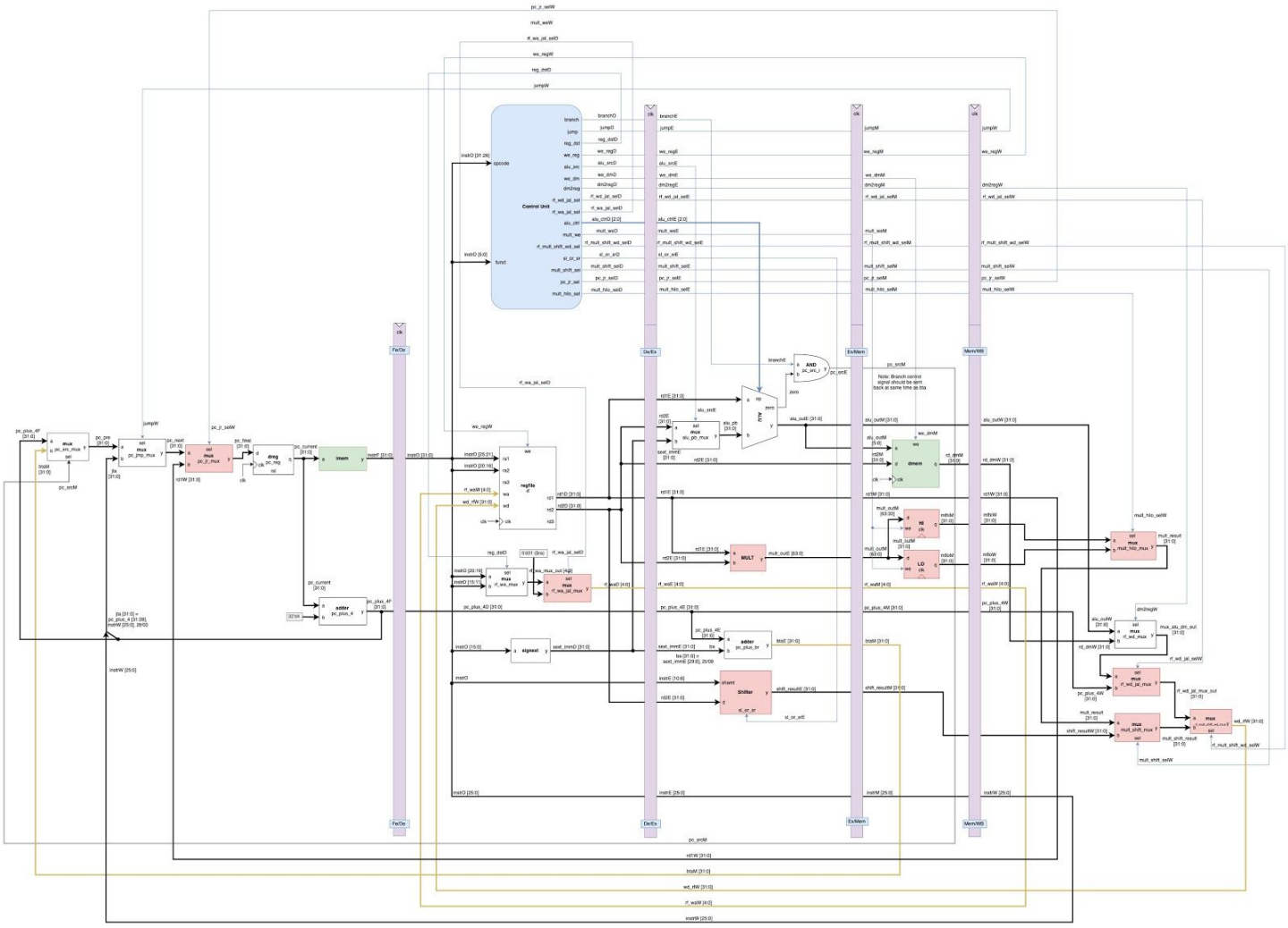


Figure 31: Final schematic of pipelined MIPS processor. Modules and wires in blue are from the control unit. Modules in red are modules added to support new instructions in Assignment 7. Modules in green are external to the processor. Wires in yellow show writeback buses to some portion in the datapath. Modules in purple are state registers that hold data in between stages. A larger, horizontal view is shown in the next page.

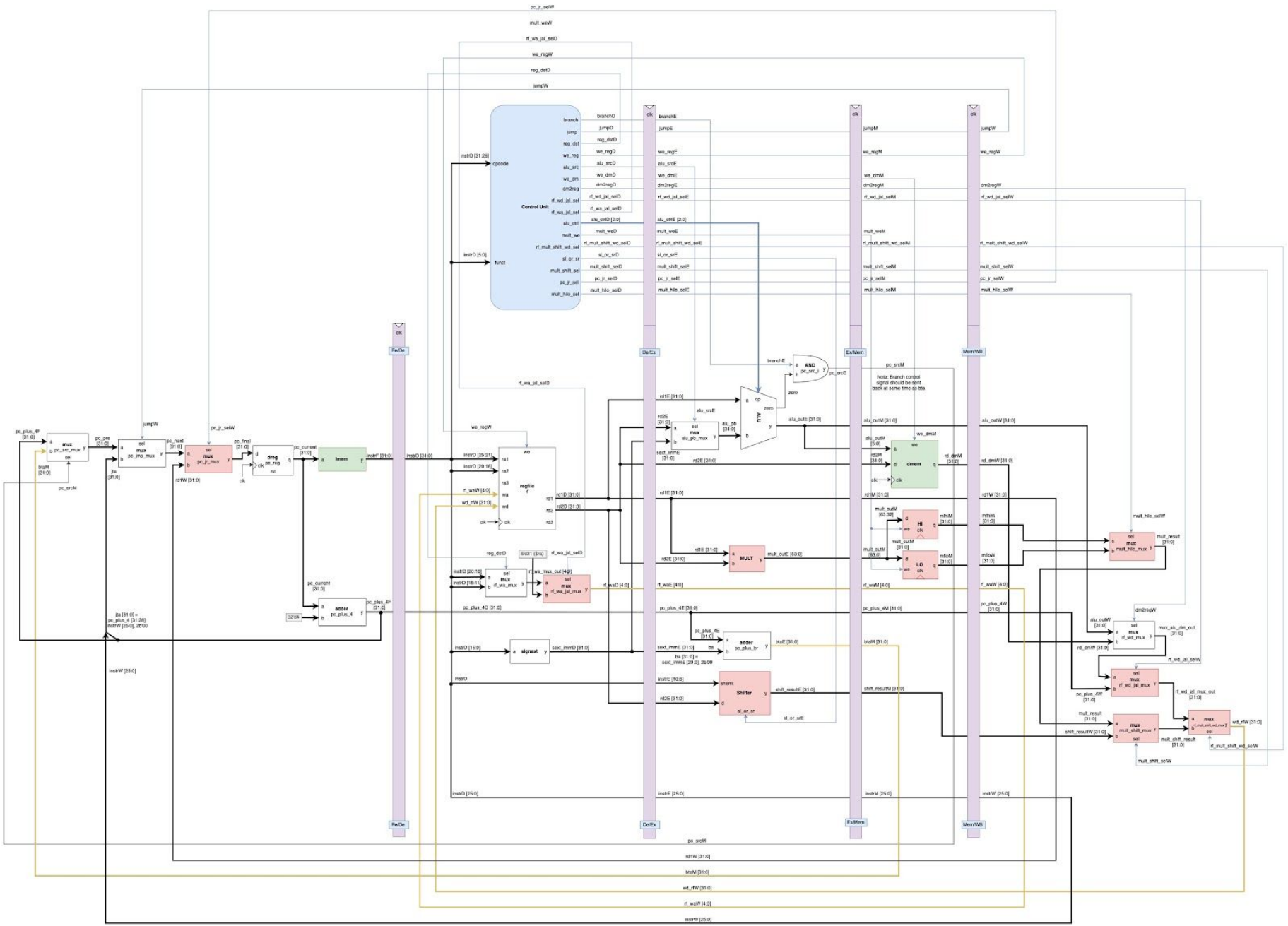

Control Unit
regfile
Shifter
ALU
MULT
dmem
Fe/De
De/Ex
Ex/Mem
Mem/WB
Note: Branch control signal should be sent back at same time as bta

Table 13: Control Unit truth table for the pipelined MIPS processor organized by stage

		Input		Decode			Execute				Memory	Writeback							
Type	Instruction	opcode [31:26]	funct [5:0]	mult_hilo_sel	rf_wa_jal_sel	reg_dst	alu_src	branch	alu_ctrl [2:0]	sl_or_sr	we_dm	jump	we_reg	dm2reg	rf_wd_jal_sel	mult_we	rf_mult_shift_wd_sel	mult_shift_sel	pc_jr_sel
R	MULTU	00_0000	01_1000	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	MFHI	00_0000	01_0000	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
	MFLO	00_0000	01_0010	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
	JR	00_0000	00_1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	SLL	00_0000	00_0000	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0
	SLR	00_0000	00_0010	0	0	1	0	0	0	1	0	0	1	0	0	0	1	1	0
J	JAL	00_0011	X	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0
R	ADD	00_0000	10_0000	0	0	1	0	0	10	0	0	0	1	0	0	0	0	0	0
	SUB	00_0000	10_0010	0	0	1	0	0	110	0	0	0	1	0	0	0	0	0	0
	AND	00_0000	10_0100	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
	OR	00_0000	10_0101	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0
	SLT	00_0000	10_1010	0	0	1	0	0	111	0	0	0	1	0	0	0	0	0	0
I	LW	10_0011	X	0	0	0	1	0	10	0	0	0	1	1	0	0	0	0	0
	SW	10_1011	X	0	0	0	1	0	10	0	1	0	0	0	0	0	0	0	0
	BEQ	00_0100	X	0	0	0	0	1	110	0	0	0	0	0	0	0	0	0	0
	ADDI	00_1000	X	0	0	0	1	0	10	0	0	0	1	0	0	0	0	0	0
J	J	00_0010	X	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Control Signals from MA INDEC
Control Signals from AU XDEC
R-Type Instructions
I-Type Instructions
J-Type Instructions

Table 14: Decode portion of the truth table

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

Table 15: Execute portion of the truth table

Type	Instruction	Input		Execute			
		opcode [31:26]	*funct [5:0]*	*alu_src*	*branch*	*alu_ctrl [2:0]*	*sl_or_sr*
R	MFHI	00_0000	01_0000	0	0	0	0
	MFLO	00_0000	01_0010	0	0	0	0
	JR	00_0000	00_1000	0	0	0	0
	SLL	00_0000	00_0000	0	0	0	0
	SLR	00_0000	00_0010	0	0	0	1
J	JAL	00_0011	X	0	0	0	0
R	ADD	00_0000	10_0000	0	0	10	0
	SUB	00_0000	10_0010	0	0	110	0
	AND	00_0000	10_0100	0	0	0	0
	OR	00_0000	10_0101	0	0	1	0
	SLT	00_0000	10_1010	0	0	111	0
I	LW	10_0011	X	1	0	10	0
	SW	10_1011	X	1	0	10	0
	BEQ	00_0100	X	0	1	110	0
	ADDI	00_1000	X	1	0	10	0
J	J	00_0010	X	0	0	0	0

Table 16: Memory portion of the truth table

Type	Instruction	Input		Memory
		opcode [31:26]	*funct [5:0]*	*we_dm*
R	MFHI	00_0000	01_0000	0
	MFLO	00_0000	01_0010	0
	JR	00_0000	00_1000	0
	SLL	00_0000	00_0000	0
	SLR	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	1
	BEQ	00_0100	X	0
	ADDI	00_1000	X	0
J	J	00_0010	X	0

Table 17: Writeback portion of the truth table

Type	Instruction	Input		Writeback							
		opcode [31:26]	*funct [5:0]*	*jump*	*we_reg*	*dm2reg*	*rf_wd_jal_sel*	*mult_we*	*rf_mult_shift_wd_sel*	*mult_shift_sel*	*pc_jr_sel*
R	MFHI	00_0000	01_0000	0	1	0	0	0	1	0	0
	MFLO	00_0000	01_0010	0	1	0	0	0	1	0	0
	JR	00_0000	00_1000	0	0	0	0	0	0	0	1
	SLL	00_0000	00_0000	0	1	0	0	0	1	1	0
	SLR	00_0000	00_0010	0	1	0	0	0	1	1	0
J	JAL	00_0011	X	1	1	0	1	0	0	0	0
R	ADD	00_0000	10_0000	0	1	0	0	0	0	0	0
	SUB	00_0000	10_0010	0	1	0	0	0	0	0	0
	AND	00_0000	10_0100	0	1	0	0	0	0	0	0
	OR	00_0000	10_0101	0	1	0	0	0	0	0	0
	SLT	00_0000	10_1010	0	1	0	0	0	0	0	0
I	LW	10_0011	X	0	1	1	0	0	0	0	0
	SW	10_1011	X	0	0	0	0	0	0	0	0
	BEQ	00_0100	X	0	0	0	0	0	0	0	0
	ADDI	00_1000	X	0	1	0	0	0	0	0	0
J	J	00_0010	X	1	0	0	0	0	0	0	0

Appendix D: Single-Cycle MIPS Processor in SoC Hardware Validation

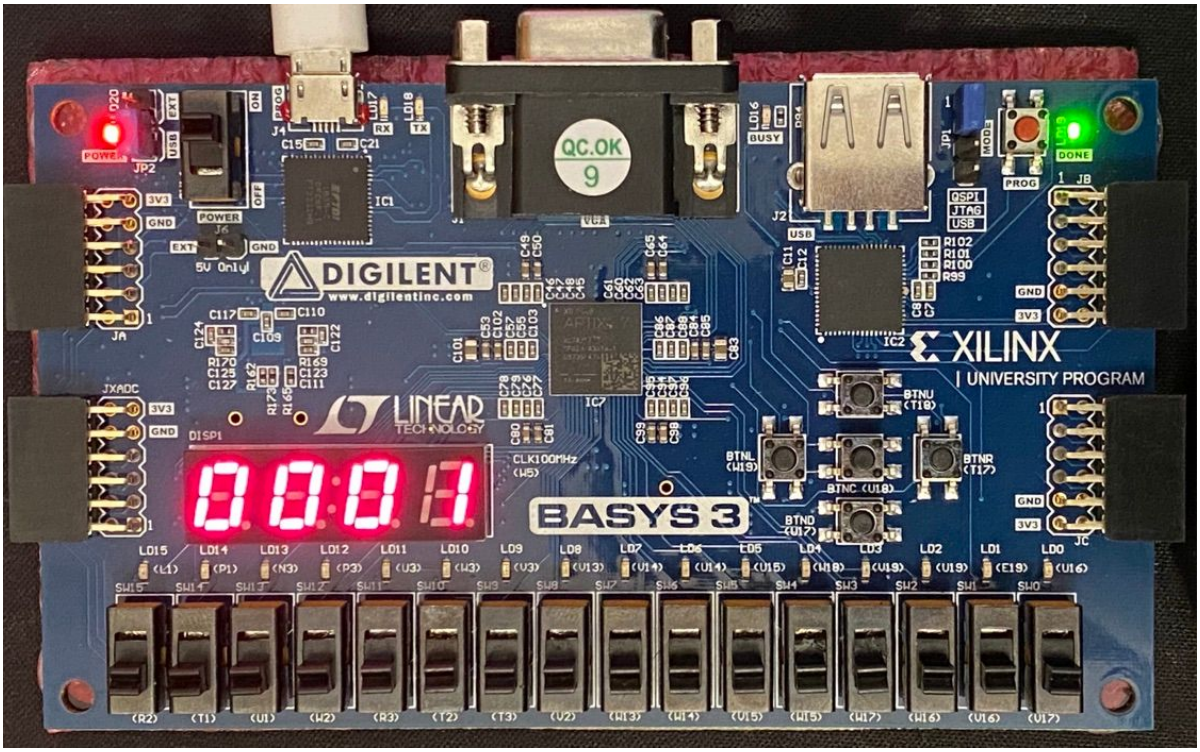


Figure 32: Picture showing the result of input n = 0. 0! = 1 = 0x1as shown on the display.

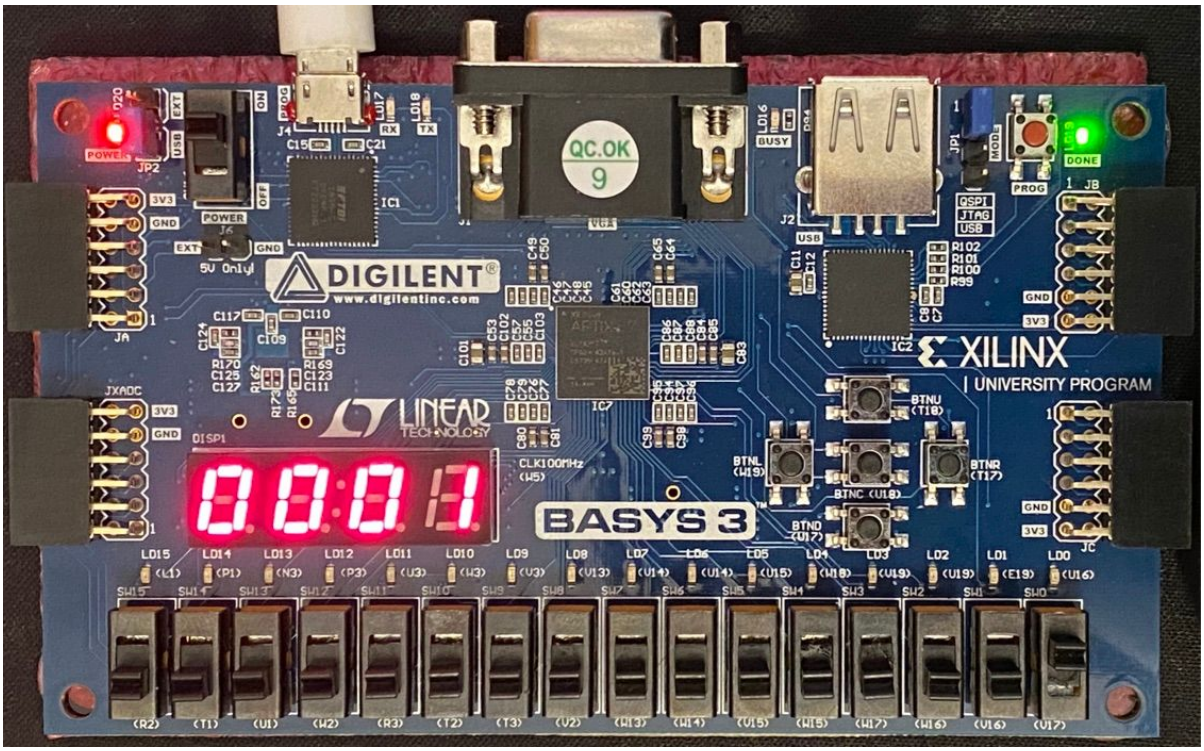


Figure 33: Picture showing the result of input n = 1. 1! = 1 = 0x1 as shown on the display.

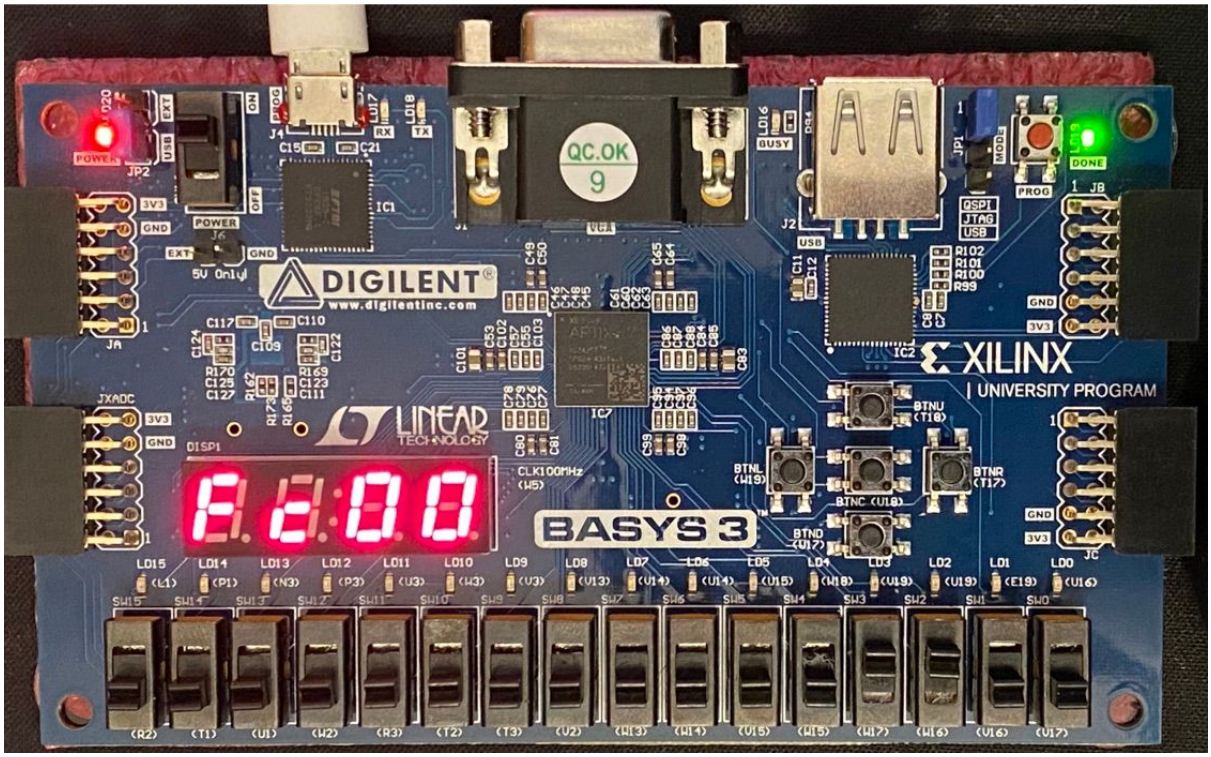


Figure 34: Picture showing the result of input n = 12 with display select = 0. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 0, the display only shows 0xFC00.

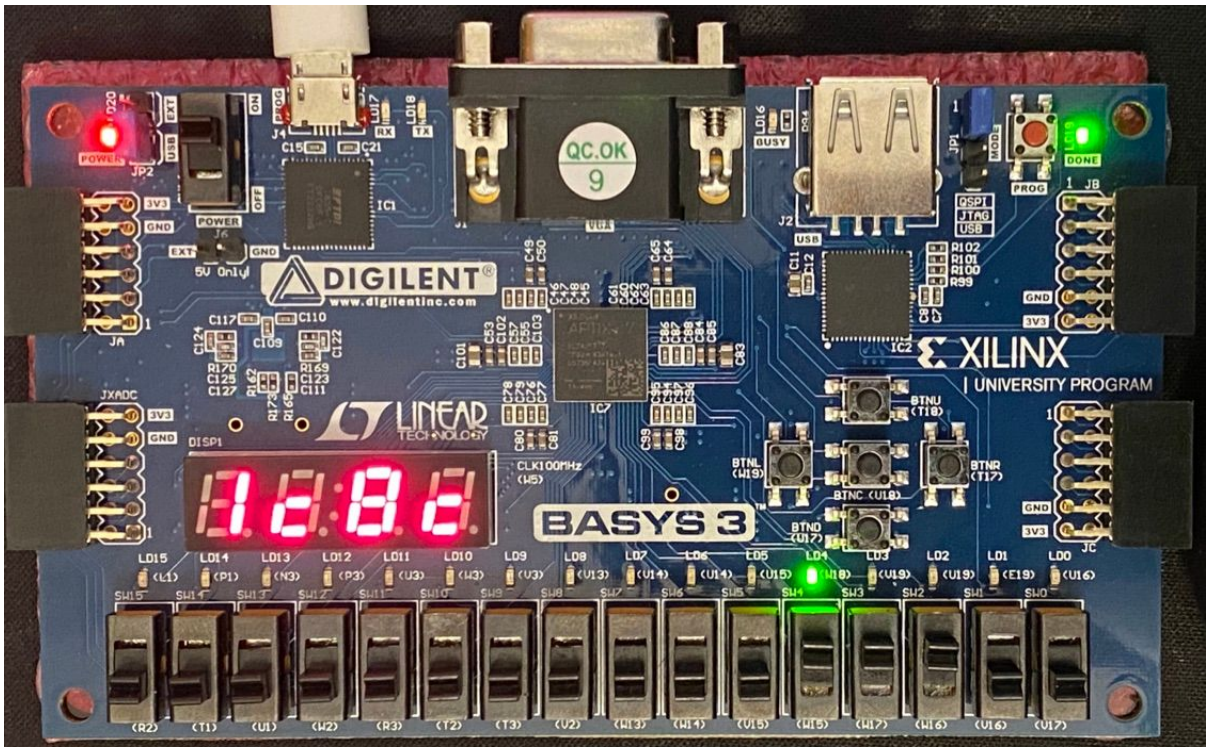


Figure 35: Picture showing the result of input n = 12 with display select = 1. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 1, the display only shows 0x1C8C with the LED above the display select switch lit.

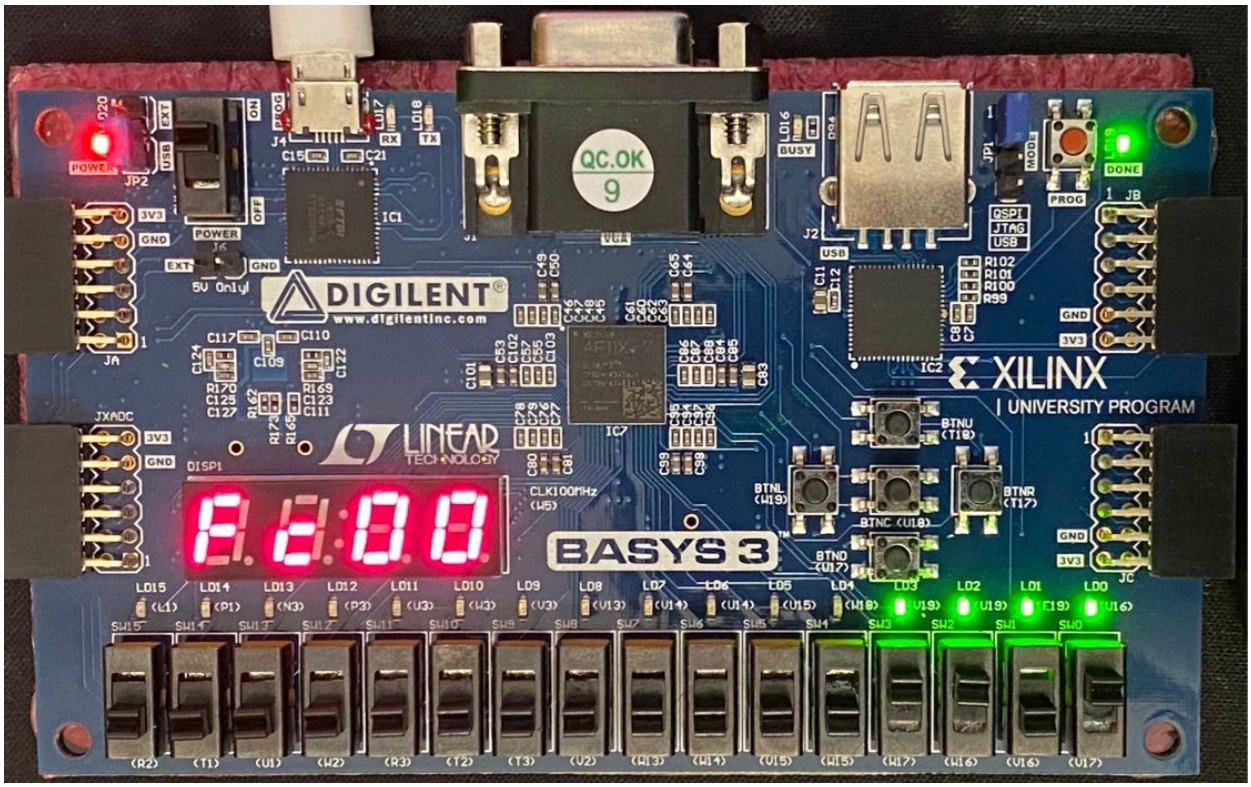


Figure 36: Picture showing the result of input n = 13. Note that n! is not calculated for this input as 13 is greater than 12. Instead, the LEDs connected to FactErr are lit up to indicate an error with the input.

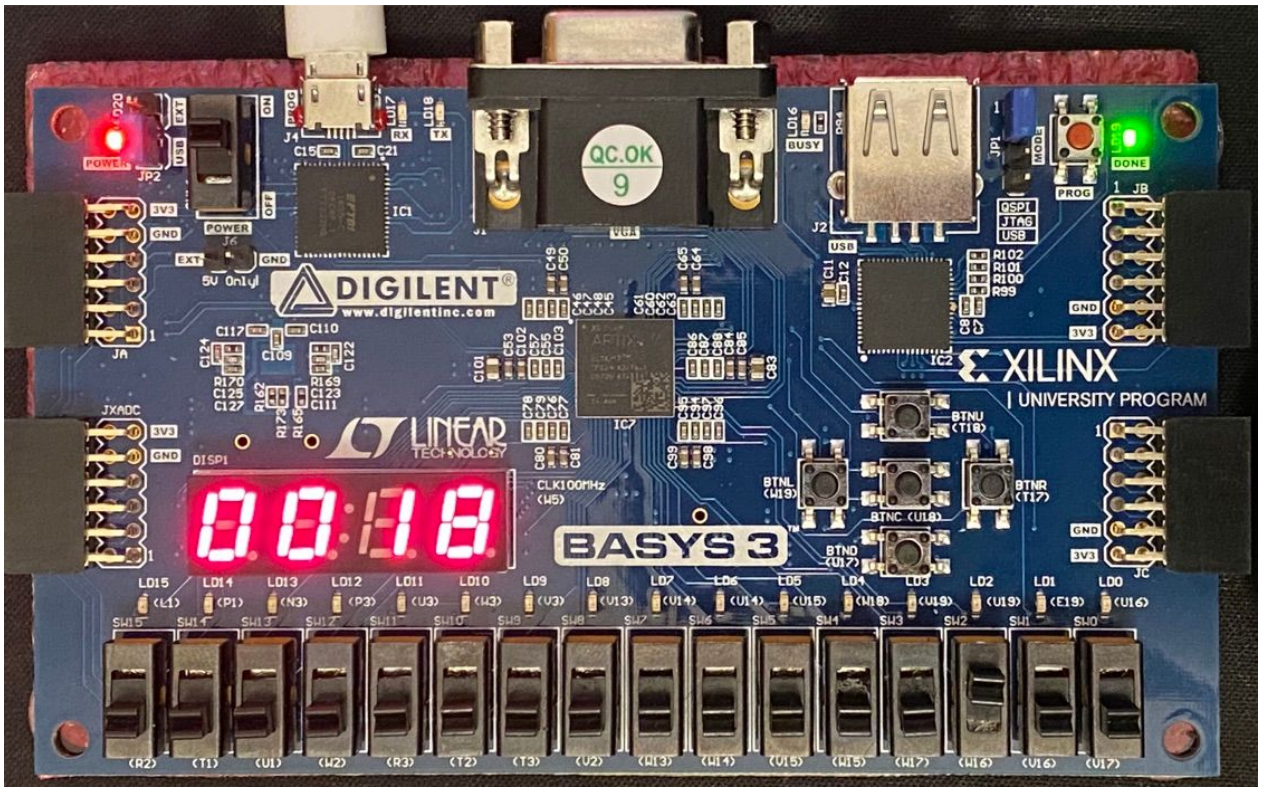


Figure 37: Picture showing the result of input n = 4. 4! = 24 = 0x18 as shown on the display

Appendix E: Pipelined MIPS Processor in SoC Hardware Validation

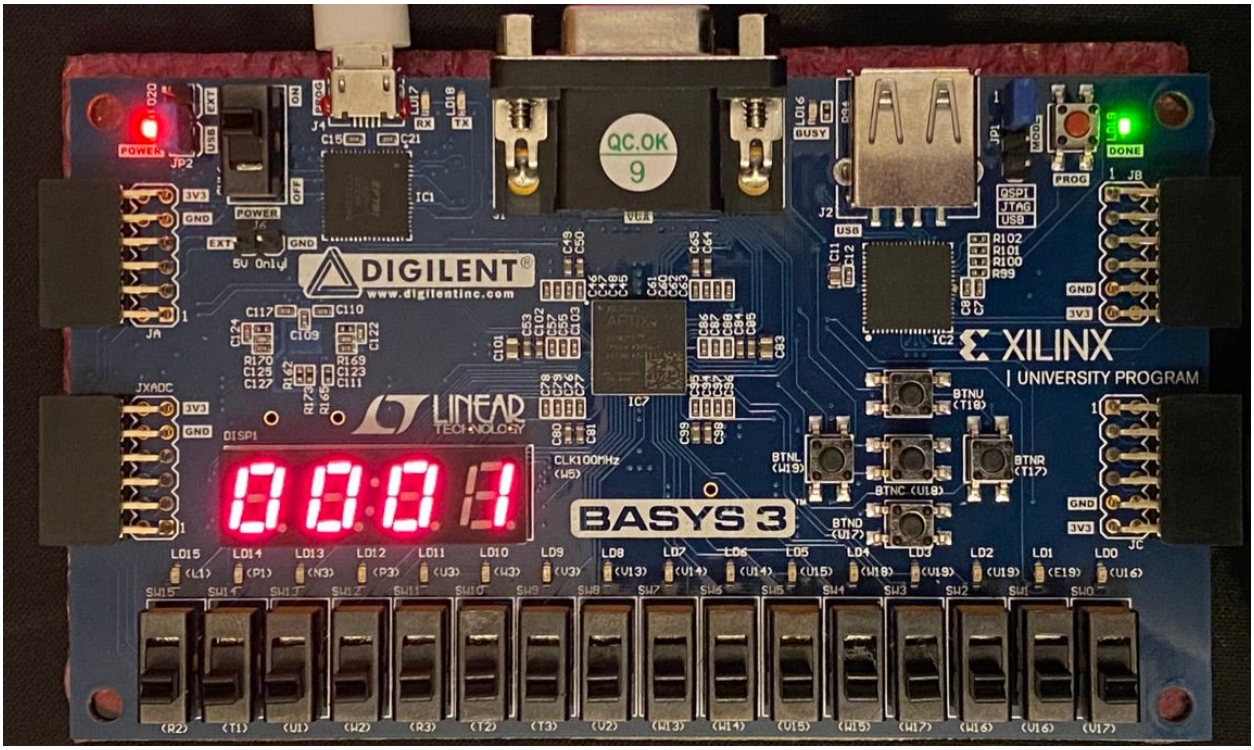


Figure 38: Picture showing the result of input n = 0. 0! = 1 = 0x1 as shown on the display

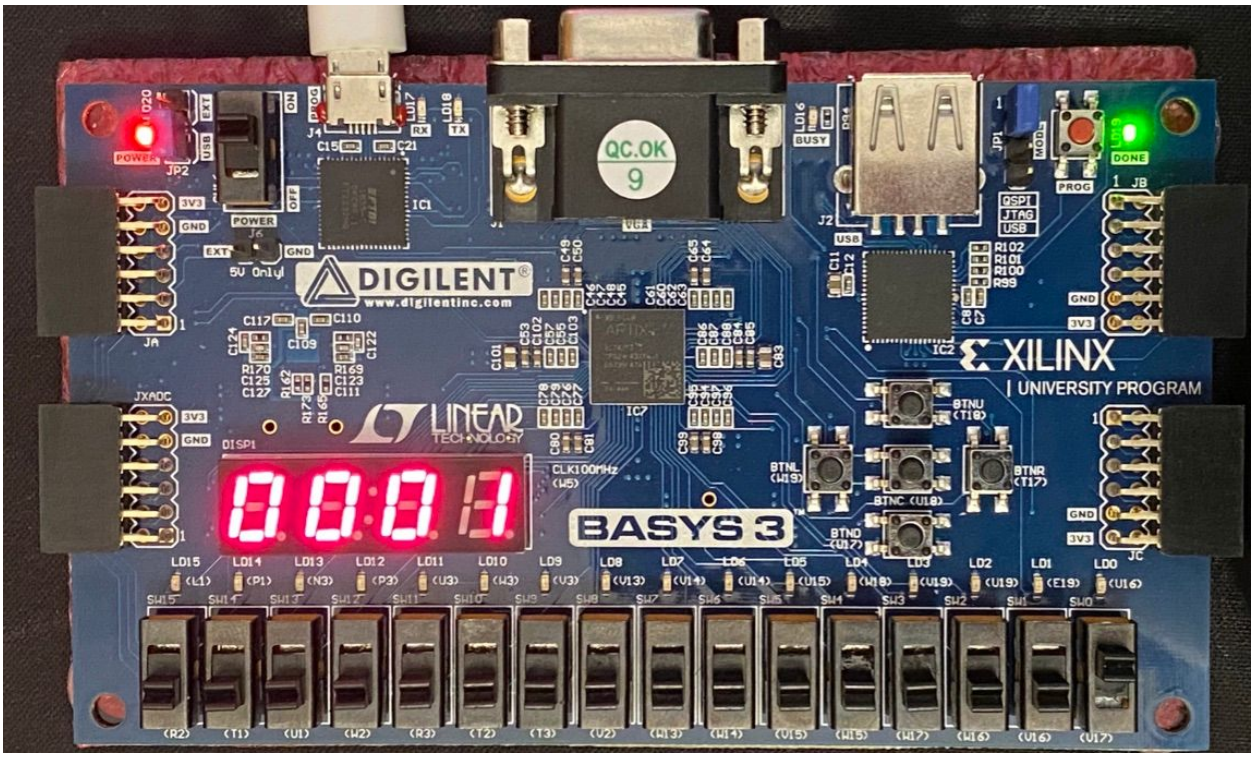


Figure 39: Picture showing the result of input n = 1. 1! = 1 = 0x1 as shown on the display

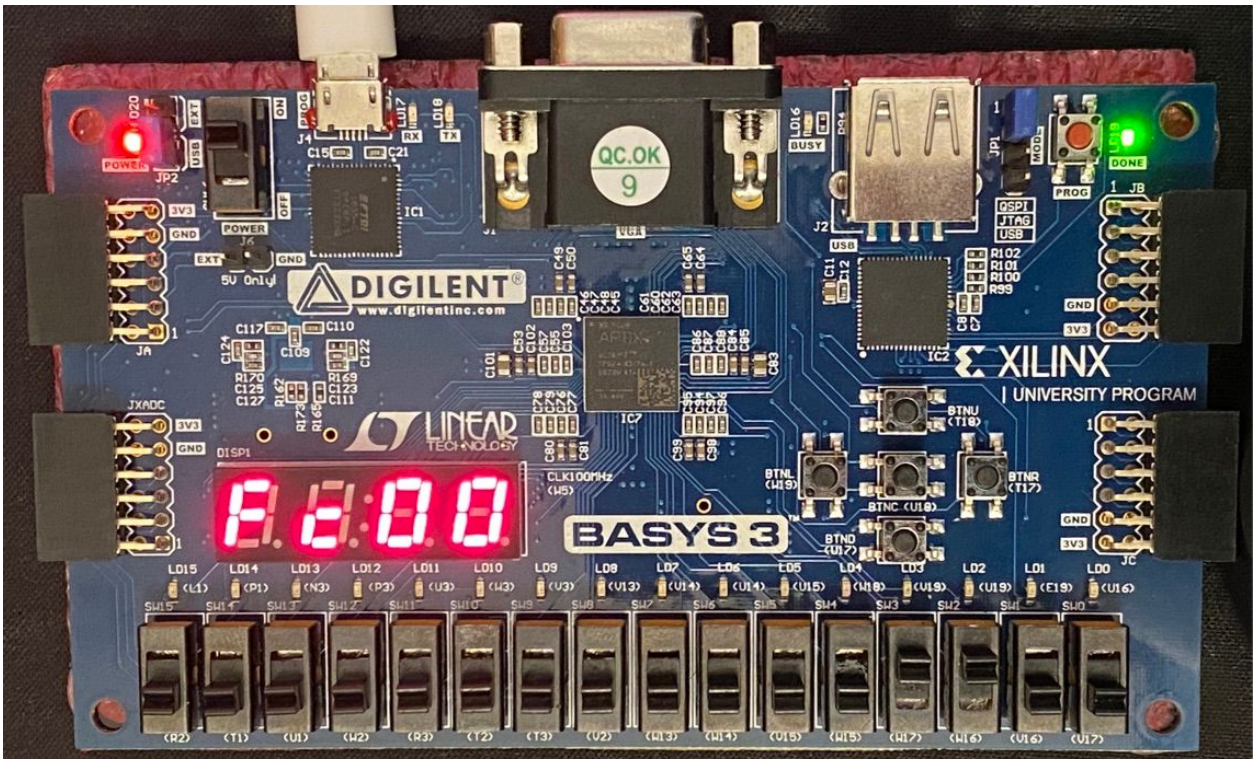


Figure 40: Picture showing the result of input n = 12 with display select = 0. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 0, only 0xFC00 is shown on the display.

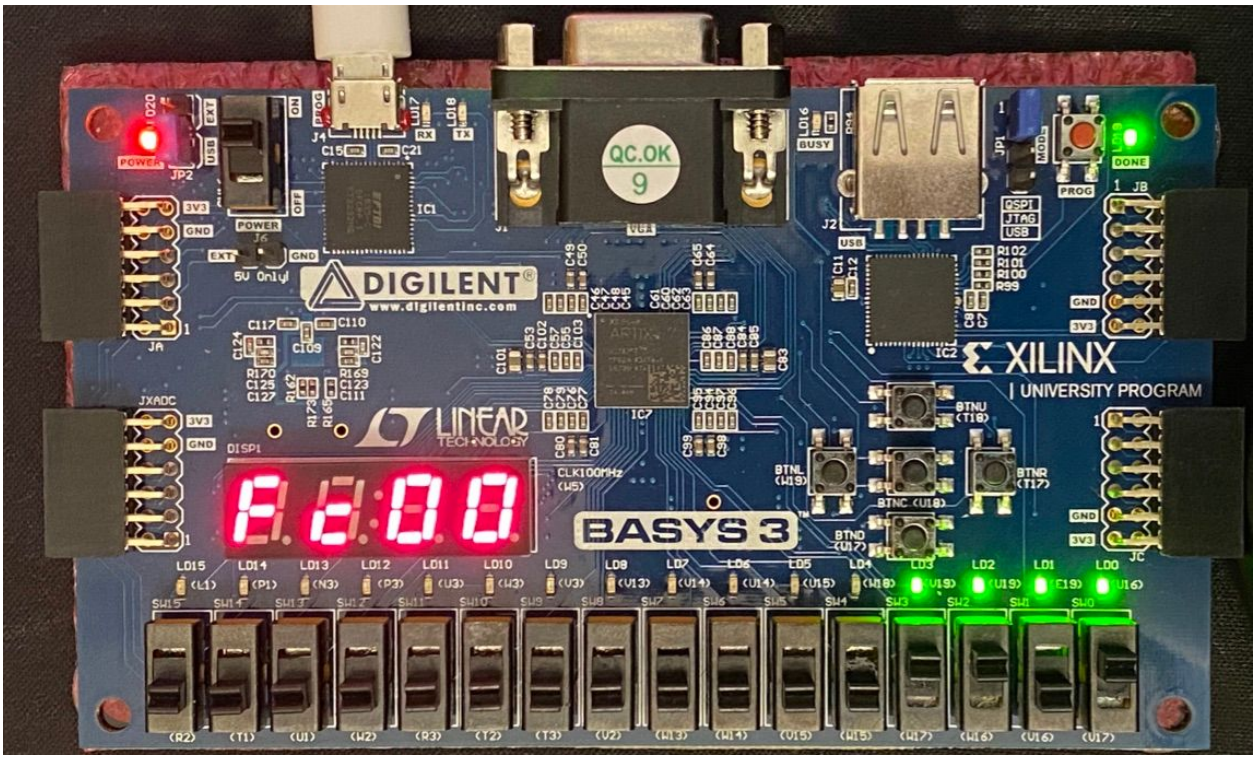


Figure 41: Picture showing the result of input n = 13. Since 13 > 12, the factorial accelerator does not calculate 13!. Instead, the factorial error bit lights up the shown LEDs to indicate an error with the input.

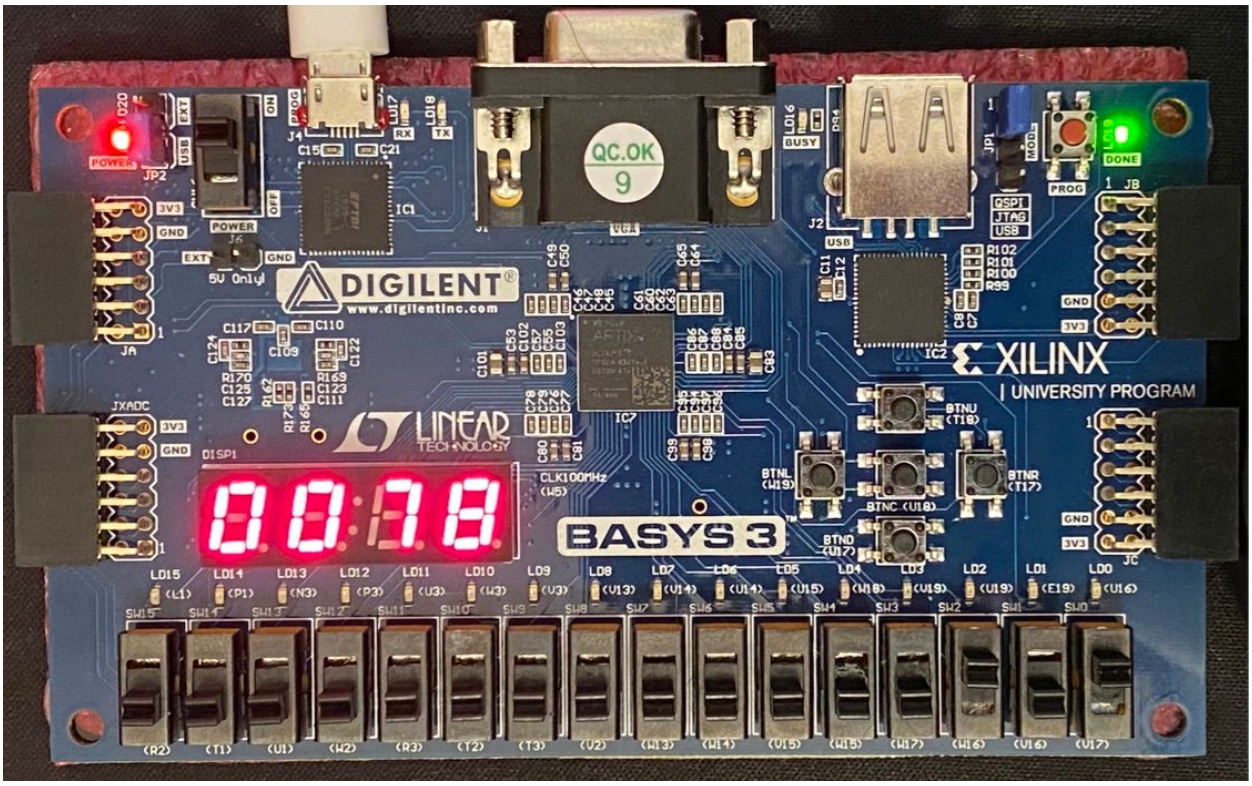


Figure 42: Picture showing the result of input n = 5. 5! = 120 = 0x78 as shown on the display

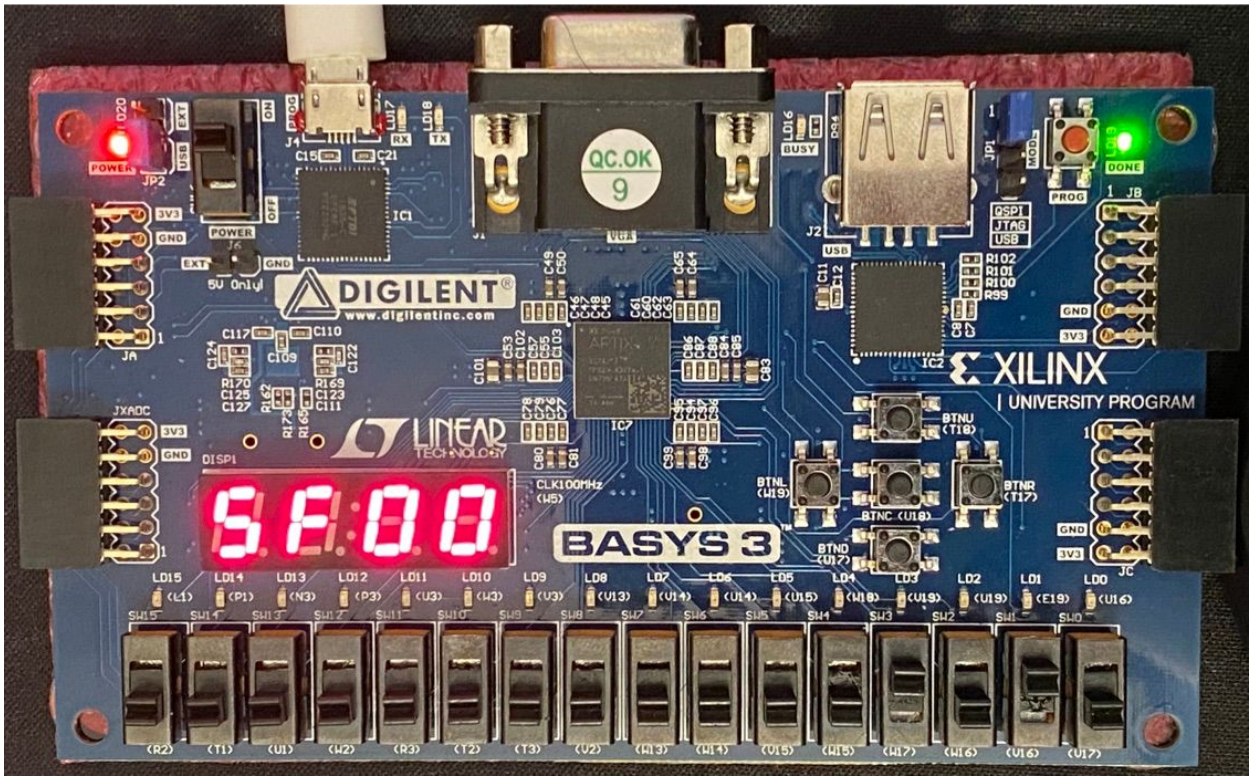


Figure 43: Picture showing the result of input n = 10 with display select = 0. 10! = 3,628,800 = 0x00375F00. Since the display select is 0, the display only shows 0x5F00.

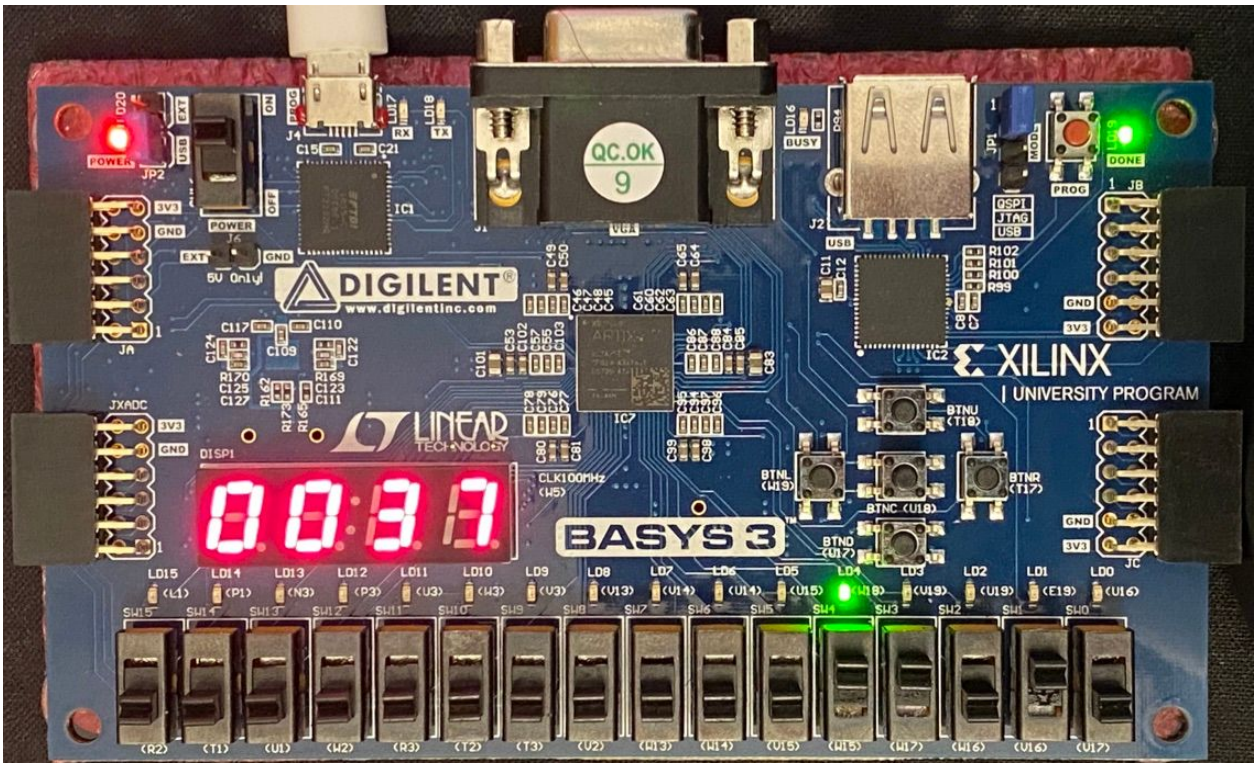


Figure 44: Picture showing the result of input n = 10 with display select = 1. 10! = 3,628,800 = 0x00375F00. Since the display select is 1, the display only shows 0x0037 with the LED above the display select switch lit.

Appendix F: Factorial Accelerator with Interface Wrapper Code

fact_top.v

```
module fact_top(
        input   wire    [1:0]   A,
        input   wire            WE,
        input   wire    [3:0]   WD,
        input   wire            Rst,
        input   wire            Clk,
        output          [31:0]  RD
    );

    wire        WE1, WE2;
    wire [1:0]  RdSel;
    wire [3:0]  n;
    wire        Go;
    wire        GoPulseCmb;
    wire        GoPulse;
    wire [31:0] nf;
    wire        Done;
    wire [31:0] Result;
    wire        Err;
    wire        ResDone, ResErr;

    fact_ad fact_ad (
        .A(A),
        .WE(WE),
        .WE1(WE1),
        .WE2(WE2),
        .RdSel(RdSel)
    );

    fact_reg #(4) n_reg (
        .Clk(Clk),
        .Rst(Rst),
        .D(WD),
        .Load_Reg(WE1),
        .Q(n)
    );

    fact_reg #(1) go_reg (
        .Clk(Clk),
        .Rst(Rst),
        .D(WD[0]),
        .Load_Reg(WE2),
        .Q(Go)
    );

    fact_reg #(1) go_pulse_reg (
        .Clk(Clk),
        .Rst(Rst),
```

```
        .D(GoPulseCmb),
        .Load_Reg(1'b1),
        .Q(GoPulse)
    );

    fact_reg #(32) result_reg (
        .Clk(Clk),
        .Rst(Rst),
        .D(nf),
        .Load_Reg(Done),
        .Q(Result)
    );

    Factorial factorial (
        .go(GoPulse),
        .clk(Clk),
        .n(n),
        .done(Done),
        .error(Err),
        .product(nf)
    );

    fact_and fact_and (
        .a(WE2),
        .b(WD[0]),
        .c(GoPulseCmb)
    );

    fact_mux4 fact_mux4 (
        .RdSel(RdSel),
        .n(n),
        .Go(Go),
        .ResDone(ResDone),
        .ResErr(ResErr),
        .Result(Result),
        .RD(RD)
    );

    fact_done_stat_reg fact_done_stat_reg (
        .Clk(Clk),
        .Rst(Rst),
        .GoPulseCmb(GoPulseCmb),
        .Done(Done),
        .ResDone(ResDone)
    );

    fact_err_stat_reg fact_err_stat_reg (
        .Clk(Clk),
        .Rst(Rst),
        .GoPulseCmb(GoPulseCmb),
        .Err(Err),
        .ResErr(ResErr)
    );
endmodule
```

fact_ad.v

```
module fact_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 = 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;
                WE2 = 1'bx;
            end
        endcase
    end

    assign RdSel = A;

endmodule
```

fact_reg.v

```
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)
        if (Rst)
```

```
            Q <= 0;
        else if (Load_Reg)
            Q <= D;
        else
            Q <= Q;
endmodule
```

fact_and.v

```
module fact_and(
        input   wire    a, b,
        output          c
    );

    assign c = a & b;

endmodule
```

fact_mux4.v

```
module fact_mux4(
        input   wire    [1:0]   RdSel,
        input   wire    [3:0]   n,
        input   wire            Go,
        input   wire            ResDone, ResErr,
        input   wire    [31:0]  Result,
        output  reg     [31:0]  RD
    );

    always @ (*) begin
        case (RdSel)
            2'b00:      RD = {{(32-4){1'b0}}, n};   // 32 - width of n; {1'b0} assigns
all upper 28 bits = 0
            2'b01:      RD = {{31{1'b0}}, Go};
            2'b10:      RD = {{30{1'b0}}, ResErr, ResDone};
            2'b11:      RD = Result;
            default:    RD = {31{1'bx}};
        endcase
    end
endmodule
```

fact_done_stat_reg.v

```
module fact_done_stat_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
```

fact_err_stat_reg.v

```
module fact_err_stat_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
```

Factorial.v

```
module Factorial(
    input wire go, clk,
    input wire [3:0] n,
    output wire done, error,
    output wire [31:0] product
);

wire load_cnt, en, load_reg, sel1, sel2, DP_error, GT;

DP DP0(
    .n(n),
    .load_cnt(load_cnt),
    .en(en),
    .load_reg(load_reg),
    .sel1(sel1),
    .sel2(sel2),
    .clk(clk),
    .Error(DP_error),
    .GT(GT),
    .product(product)
);

CU CU0(
    .go(go),
    .clk(clk),
    .GT(GT),
    .DP_error(DP_error),
    .load_cnt(load_cnt),
    .en(en),
```

```
    .load_reg(load_reg),
    .sel1(sel1),
    .sel2(sel2),
    .done(done),
    .CU_error(error)
);

endmodule
```

DP.v

```
module DP(
    input [3:0] n,
    input load_cnt,
    input en,
    input load_reg,
    input sel1,
    input sel2,
    input clk,
    output Error,
    output GT,
    output [31:0] product///reg or wire
);


wire [31:0] mux1_out;
wire [31:0] reg_out;
wire [31:0] cnt_out;
wire [31:0] mult_out;

// instantiate the building blocks, within parenthesis is the wire or include same name
    CMP_ERROR U0(
        .A(4'b1100),
        .B(n),
        .ERROR(Error)
    );

    CMP_GT U1(
        .A(32'h00000001),
        .B(cnt_out),
        .GT(GT)
    );

    CNT U2(
        .D(n),
        .load_cnt(load_cnt),
        .en(en),
        .clk(clk),
        .Q(cnt_out)
    );

    MUL U3(
        .X(cnt_out),
        .Y(reg_out),
```

```
        .Z(mult_out)
    );

    MUX M1(
        .input_0(1),
        .input_1(mult_out),
        .select(sel1),
        .out(mux1_out)
    );

    MUX M2(
        .input_0(0),
        .input_1(reg_out),
        .select(sel2),
        .out(product)
    );

    REG R1(
        .D(mux1_out),
        .Q(reg_out),
        .clk(clk),
        .load_reg(load_reg)
    );

endmodule //DP
```

CU.v

```
module CU(
    input go,
    input clk,
    input GT,
    input DP_error,
    output reg load_cnt,
    output reg en,
    output reg load_reg,
    output reg sel1,
    output reg sel2,
    output reg done,
    output reg CU_error
);

integer state0 = 3'b000;
integer state1 = 3'b001;
integer state2 = 3'b010;
integer state3 = 3'b011;
integer state4 = 3'b100;
integer state5 = 3'b101;

reg [2:0] cs = 3'b000;
reg [2:0] ns;

//reg [6:0] cw;
//always@(cw){load_cnt, en, load_reg, sel1, sel2, done, CU_error} = cw;
```

```
// cw stands for control word. This control word refers to each output bit sent by the
CU to the DP

always@(posedge clk)
    cs <= ns;

always@(*) begin
    case (cs)
        state0: begin
                    done = 0;
                    CU_error = 0;
                    load_cnt = 0;
                    en = 0;
                    load_reg = 0;
                    sel1 = 0;
                    sel2 = 0;
                    if(!go)
                        ns = state0;
                    else
                        ns = state1;
                end
        state1: begin
                    done = 0;
                    CU_error = 0;
                    load_cnt = 1;
                    en = 1;
                    load_reg = 1;
                    sel1 = 0;
                    sel2 = 0;
                    if(DP_error)
                        ns = state5;
                    else
                        ns = state2;
                end
        state2: begin
                    done = 0;
                    CU_error = 0;
                    load_cnt = 0;
                    en = 0;
                    load_reg = 0;  // updated
                    sel1 = 0;   //updated
                    sel2 = 0;
                    if(GT)
                        ns = state3;
                    else
                        ns = state4;
                end
        state3: begin
                    done = 0;
                    CU_error = 0;
                    load_cnt = 0;
                    en = 1;
                    load_reg = 1;
                    sel1 = 1;
                    sel2 = 0;
                    ns = state2;
```

```
                end
        state4: begin
                    done = 1;
                    CU_error = 0;
                    load_cnt = 0;
                    en = 0;
                    load_reg = 0;
                    sel1 = 0;
                    sel2 = 1;
                    ns = state0;
                end
        state5: begin
                    done = 0;
                    CU_error = 1;
                    load_cnt = 0;
                    en = 0;
                    load_reg = 0;
                    sel1 = 0;
                    sel2 = 0;
                    ns = state0;
                end
         default: begin
                    done = 0;
                    CU_error = 1;
                    load_cnt = 0;
                    en = 0;
                    load_reg = 0;
                    sel1 = 0;
                    sel2 = 0;
                    ns = state0;
                end

    endcase
    end
endmodule
```

CMP_ERROR.v

```
module CMP_ERROR(
    input [3:0] A, B,
    output reg ERROR
    );

always @ (A, B) begin
    if (B > A)
        ERROR = 1;
    else
        ERROR = 0;
    end
endmodule
```

CMP_GT.v

```
module CMP_GT(
    input [31:0] A, B,
    output reg GT
    );

always @ (A, B) begin
    if (B > A)
        GT = 1;
    else
        GT = 0;
    end
endmodule
```

CNT.v

```
module CNT(
    input [3:0] D,
    input load_cnt, en, clk,
    output reg [31:0] Q
    );

always @ (posedge clk) begin
    if (en) begin
        if (load_cnt)
            Q <= D;
        else
            Q <= Q - 1;
    end else
        Q <= Q;
end

endmodule
```

MUL.v

```
module MUL(
    input [31:0] X, Y,
    output reg [31:0] Z
    );

always @ (X, Y) begin
    Z = X * Y;
end

endmodule
```

MUX.v

```
module MUX(
    input [31:0] input_0, input_1,
    input select,
```

```
    output reg [31:0] out
);

always @ (input_0, input_1, select)
begin
    if (select)
        out = input_1;
    else
        out = input_0;
end
endmodule
```

REG.v

```
module REG(
    input [31:0] D,
    input clk, load_reg,
    output reg [31:0] Q
    );

always @ (posedge clk)
    if(load_reg)
        Q <= D;
    else
        Q <= Q;
endmodule
```

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
```

Appendix G: GPIO Unit with Interface Wrapper Code

gpio_top.v

```
module gpio_top(
        input   wire    [1:0]   A,
        input   wire            WE,
        input   wire    [31:0]  gpI1, gpI2,
        input   wire    [31:0]  WD,
        input   wire            Rst, Clk,
        output          [31:0]  gpO1, gpO2,
        output          [31:0]  RD
    );

    wire            WE1, WE2;
    wire    [1:0]   RdSel;

    gpio_ad gpio_ad (
        .A(A),
        .WE(WE),
        .WE1(WE1),
        .WE2(WE2),
        .RdSel(RdSel)
    );

    gpio_reg gpO1_reg (
        .D(WD),
        .Clk(Clk),
        .Rst(Rst),
        .Load_Reg(WE1),
        .Q(gpO1)
    );

    gpio_reg gpO2_reg (
        .D(WD),
        .Clk(Clk),
        .Rst(Rst),
        .Load_Reg(WE2),
        .Q(gpO2)
    );

    gpio_mux4 gpio_mux4(
        .gpI1(gpI1),
        .gpI2(gpI2),
        .gpO1(gpO1),
        .gpO2(gpO2),
        .RdSel(RdSel),
        .RD(RD)
    );
endmodule
```

gpio_ad.v

```
module gpio_ad(
        input   wire    [3:2]   A,
        input   wire            WE,
        output  reg             WE1, 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
```

gpio_reg.v

```
// Used for gpO1 and gpO2
module gpio_reg #(parameter w = 32) (
        input   wire    [w-1:0] D,
        input   wire            Clk, Rst,
        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
```

gpio_mux4.v

```
module gpio_mux4(
        input   wire    [31:0]  gpI1,
        input   wire    [31:0]  gpI2,
        input   wire    [31:0]  gpO1,
        input   wire    [31:0]  gpO2,
        input   wire    [1:0]   RdSel,
        output  reg     [31:0]  RD
    );

    always @ (*) begin
        case (RdSel)
            2'b00: RD = gpI1;
            2'b01: RD = gpI2;
            2'b10: RD = gpO1;
            2'b11: RD = gpO2;
            default: RD = {31{1'bx}};
        endcase
    end
endmodule
```

tb_gpio_top.v

```
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 DUT1(
        .A(A_tb),
        .WE(WE_tb),
        .gpI1(gpI1_tb),
        .gpI2(gpI2_tb),
        .WD(WD_tb),
```

```
        .Rst(rst_tb),
        .Clk(clk_tb),
        .gpO1(gpO1_tb),
        .gpO2(gpO2_tb),
        .RD(RD_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
```

Appendix H: Shared Code between Single-Cycle MIPS Processor and Pipelined MIPS Processor on SoC Implementations

imem.v

```
module imem (
        input  wire [5:0]  a,
        output wire [31:0] y
    );

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

    initial begin
        $readmemh ("memfile.dat", rom);
    end

    assign y = rom[a];

endmodule
```

dmem.v

```
 module dmem (
        input  wire        clk,
        input  wire        we,
        input  wire [5:0]  a,
        input  wire [31:0] d,
        output wire [31:0] q,
        input  wire        rst
    );

    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 rst) begin
    end

    always @ (posedge clk, posedge rst) begin
        if (rst) for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;
        else if (we) ram[a] <= d;
    end

    assign q = ram[a];

endmodule
```

SoC_ad.v

```
module SoC_ad(
        input   wire            WE,
        input   wire    [31:0]  A,
        output  reg             WE1, WE2, WEM,
        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
```

SoC_mux4.v

```
module SoC_mux4(
        input   wire    [31:0]  DMemData, FactData, GPIOData,
        input   wire    [1:0]   RdSel,
        output  reg     [31:0]  ReadData
    );

    always @ (*) begin
        case (RdSel)
            2'b00: ReadData = DMemData;
            2'b01: ReadData = DMemData;
            2'b10: ReadData = FactData;
            2'b11: ReadData = GPIOData;
            default: ReadData = {31{1'bx}};
```

```
        endcase
    end
endmodule
```

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 [2:0]  alu_ctrl,

        // New outputs
        output wire        pc_jr_sel,
        output wire        rf_mult_shift_wd_sel,
        output wire        mult_we,
        output wire        mult_hilo_sel,
        output wire        mult_shift_sel,
        output wire        sl_or_sr,
        output wire        rf_wd_jal_sel,
        output wire        rf_wa_jal_sel
    );

    wire [1:0] alu_op;

    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),
        .rf_wd_jal_sel  (rf_wd_jal_sel),
        .rf_wa_jal_sel  (rf_wa_jal_sel)
    );

    auxdec ad (
        .alu_op         (alu_op),
        .funct          (funct),
        .alu_ctrl       (alu_ctrl),
        .pc_jr_sel      (pc_jr_sel),
        .rf_mult_shift_wd_sel (rf_mult_shift_wd_sel),
        .mult_we        (mult_we),
        .mult_hilo_sel  (mult_hilo_sel),
        .mult_shift_sel (mult_shift_sel),
```

```
        .sl_or_sr       (sl_or_sr)
    );

endmodule
```

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,

        // New outputs
        output wire       rf_wd_jal_sel,
        output wire       rf_wa_jal_sel
    );

    reg [10:0] ctrl;

    assign {branch, jump, reg_dst, we_reg, alu_src, we_dm, dm2reg, alu_op,
rf_wd_jal_sel, rf_wa_jal_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'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'b00_0011: ctrl = 11'b0_1_0_1_0_0_0_00_1_1; // NEW: JAL
            default:    ctrl = 11'bx_x_x_x_x_x_x_xx_x_x;
        endcase
    end

endmodule
```

auxdec.v

```
module auxdec (
        input  wire [1:0] alu_op,
        input  wire [5:0] funct,
        output wire [2:0] alu_ctrl,

        // New outputs
        output wire       pc_jr_sel,
        output wire       rf_mult_shift_wd_sel,
```

```
        output wire       mult_we,
        output wire       mult_hilo_sel,
        output wire       mult_shift_sel,
        output wire       sl_or_sr
    );

    reg [9:0] ctrl;

    assign {alu_ctrl, pc_jr_sel, rf_mult_shift_wd_sel, mult_we, mult_hilo_sel,
mult_shift_sel, sl_or_sr} = ctrl;

    always @ (alu_op, funct) begin
        case (alu_op)
            2'b00: ctrl = 9'b010_0_0_0_0_0_0;          // ADD (LW or SW)
            2'b01: ctrl = 9'b110_0_0_0_0_0_0;          // SUB (BEQ)
            default: case (funct)
                9'b10_0100: ctrl = 9'b000_0_0_0_0_0_0; // AND
                9'b10_0101: ctrl = 9'b001_0_0_0_0_0_0; // OR
                9'b10_0000: ctrl = 9'b010_0_0_0_0_0_0; // ADD
                9'b10_0010: ctrl = 9'b110_0_0_0_0_0_0; // SUB
                9'b10_1010: ctrl = 9'b111_0_0_0_0_0_0; // SLT
                9'b01_1001: ctrl = 9'b000_0_0_1_0_0_0; // NEW: MULTU
                9'b01_0000: ctrl = 9'b000_0_1_0_0_0_0; // NEW: MFHI
                9'b01_0010: ctrl = 9'b000_0_1_0_1_0_0; // NEW: MFLO
                9'b00_1000: ctrl = 9'b000_1_0_0_0_0_0; // NEW: JR
                9'b00_0000: ctrl = 9'b000_0_1_0_0_1_0; // NEW: SLL
                9'b00_0010: ctrl = 9'b000_0_1_0_0_1_1; // NEW: SLR
                default:    ctrl = 9'bxxx_x;
            endcase
        endcase
    end

endmodule
```

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
```

adder.v

```
module adder (
        input  wire [31:0] a,
```

```
        input  wire [31:0] b,
        output wire [31:0] y
    );

    assign y = a + b;

endmodule
```

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
```

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,
        input  wire        rst
    );

    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; // Initialze $sp
    end

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
            for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;
            rf[29] = 32'h100; // Initialze $sp
        end
        else 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
```

signext.v

```
module signext (
        input  wire [15:0] a,
        output wire [31:0] y
    );

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

endmodule
```

alu.v

```
module alu (
        input  wire [2:0]  op,
        input  wire [31:0] a,
        input  wire [31:0] b,
        output wire        zero,
        output reg  [31:0] y
    );

    assign zero = (y == 0);

    always @ (op, a, b) begin
        case (op)
            3'b000: y = a & b;
            3'b001: y = a | b;
            3'b010: y = a + b;
            3'b110: y = a - b;
            3'b111: y = (a < b) ? 1 : 0;
        endcase
    end

endmodule
```

mult.v

```
module mult (
        input wire [31:0] a, b,
        output wire [63:0] y
    );

    assign y = a * b;
```

```
endmodule
```

we_dreg.v

```
module we_dreg # (parameter WIDTH = 32) (
        input  wire             clk,
        input  wire             we,
        input  wire [WIDTH-1:0] d,
        output reg  [WIDTH-1:0] q
    );

    always @ (posedge clk) begin
        if (we) q <= d;
        else    q <= q;
    end
endmodule
```

shifter.v

```
module shifter # (parameter WIDTH = 32) (
        input   wire [4:0]       shamt,
        input   wire             sl_or_sr,
        input   wire [WIDTH-1:0] d,
        output  reg  [WIDTH-1:0] y
    );

    always @ (shamt, sl_or_sr, d) begin
        if  (sl_or_sr)  y = d >> shamt;
        else            y = d << shamt;
    end

endmodule
```

Appendix I: Code for Single-Cycle MIPS Processor Implementation

Single_Cycle_MIPS_Driver_Code.asm

```
# Label     Assembly          Description
main:       addi  $t0, $0, 0x0F     # $t0 = 0x0F
            addi  $t1, $0, 1  # $t1 = 1
            sll   $t4, $t1, 4 # $t4 = $t1 << 4

fact:       lw    $t2, 0x0900($0)   # read switches
            and   $t3, $t2, $t0     # get input data n
            sw    $t3, 0x0800($0)   # write input data n
            sw    $t1, 0x0804($0)   # write control Go bit


poll:       lw    $t5, 0x0808($0)   # read status Done bit
            beq   $t5, $0, poll     # wait until Done == 1

            srl   $t5, $t5, 1 # $t5 = $t5 >> 1
            and   $t5, $t5, $t1     # get status Error bit
            and   $t3, $t2, $t4     # get display Select
            or    $t3, $t3, $t5     # combine Sel and Err
            lw    $t5, 0x080C($0)   # read result data nf
            sw    $t3, 0x0908($0)   # display Sel and Err
            sw    $t5, 0x090C($0)   # display result nf

done:       j     fact        # repeat fact loop
```

Memfile.dat (generated from Single_Cycle_MIPS_Driver_Code.asm)

```
2008000f
20090001
00096100
8c0a0900
01485824
ac0b0800
ac090804
8c0d0808
11a0fffe
000d6842
01a96824
014c5824
016d5825
8c0d080c
ac0b0908
ac0d090c
08000003
```

fpga_Single_Cycle_SoC.v

```
module fpga_Single_Cycle_SoC (
        input  wire       clk,
        input  wire       rst,
        input  wire       button,
        input  wire [4:0] switches,
        output wire [3:0] factErr,  // 4 bits wide to trigger 4 LEDs
        output wire       dispSel,
        output wire [3:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    wire  [15:0] reg_hex;
    wire        clk_sec;
    wire        clk_5KHz;
    wire        clk_pb;

    wire [7:0]  digit0;
    wire [7:0]  digit1;
    wire [7:0]  digit2;
    wire [7:0]  digit3;

    wire [31:0] gpO1_wire;
    wire [31:0] gpO2_wire;

    assign dispSel = gpO1_wire[4];  // Not bit 0 of GPO1 as the slides would have you
believe
    assign factErr[0] = gpO1_wire[0];
    assign factErr[1] = factErr[0];
    assign factErr[2] = factErr[0];
    assign factErr[3] = factErr[0];

    clk_gen clk_gen (
            .clk100MHz          (clk),
            .rst                (rst),
            .clk_4sec           (clk_sec),
            .clk_5KHz           (clk_5KHz)
        );

    button_debouncer bd (
            .clk                (clk_5KHz),
            .button             (button),
            .debounced_button   (clk_pb)
        );

    Single_Cycle_SoC Single_Cycle_SoC (
            .clk                (clk_5KHz), //clk_pb//
            .rst                (rst),
            .ra3                (),
            .gpI1               ({27'b0,switches[4:0]}),
            .gpI2               (gpO1_wire),
            .we_dm              (),
            .pc_current         (),
            .instr              (),
```

```
            .alu_out            (),
            .wd_dm              (),
            .rd_dm              (),
            .rd3                (),
            .gpO1               (gpO1_wire),
            .gpO2               (gpO2_wire)
        );

    mux2 #(16) fgpa_mux (
            .a                  (gpO2_wire[15:0]),
            .b                  (gpO2_wire[31:16]),
            .y                  (reg_hex),
            .sel                (dispSel)
        );

    hex_to_7seg hex3 (
            .HEX                (reg_hex[15:12]),
            .s                  (digit3)
        );

    hex_to_7seg hex2 (
            .HEX                (reg_hex[11:8]),
            .s                  (digit2)
        );

    hex_to_7seg hex1 (
            .HEX                (reg_hex[7:4]),
            .s                  (digit1)
        );

    hex_to_7seg hex0 (
            .HEX                (reg_hex[3:0]),
            .s                  (digit0)
        );

    led_mux led_mux (
            .clk                (clk_5KHz),
            .rst                (rst),
            .LED3               (digit3),
            .LED2               (digit2),
            .LED1               (digit1),
            .LED0               (digit0),
            .LEDSEL             (LEDSEL),
            .LEDOUT             (LEDOUT)
        );
endmodule
```

Single_Cycle_SoC.v

```
module Single_Cycle_SoC (
        input  wire        clk,
        input  wire        rst,
        input  wire [4:0]  ra3,
        input  wire [31:0] gpI1, gpI2,
```

```
        output wire        we_dm,
        output wire [31:0] pc_current,
        output wire [31:0] instr,
        output wire [31:0] alu_out,
        output wire [31:0] wd_dm,
        output wire [31:0] rd_dm,
        output wire [31:0] rd3,
        output wire [31:0] gpO1, gpO2
    );

    wire            WE1, WE2, WEM;
    wire    [1:0]   RdSel;
    wire    [31:0]  DMemData, FactData, GPIOData;

    mips mips (
            .clk            (clk),
            .rst            (rst),
            .ra3            (ra3),
            .instr          (instr),
            .rd_dm          (rd_dm),
            .we_dm          (we_dm),
            .pc_current     (pc_current),
            .alu_out        (alu_out),
            .wd_dm          (wd_dm),
            .rd3            (rd3)
        );

    imem imem (
            .a              (pc_current[7:2]),
            .y              (instr)
        );

    dmem dmem (
            .clk            (clk),
            .we             (WEM),
            .a              (alu_out[7:2]),
            .d              (wd_dm),
            .q              (DMemData),
            .rst            (rst)
        );

    SoC_ad SoC_ad (
            .WE             (we_dm),
            .A              (alu_out),
            .WE1            (WE1),
            .WE2            (WE2),
            .WEM            (WEM),
            .RdSel          (RdSel)
        );

    SoC_mux4 SoC_mux4 (
            .DMemData       (DMemData),
            .FactData       (FactData),
            .GPIOData       (GPIOData),
            .RdSel          (RdSel),
            .ReadData       (rd_dm)
```

```
        );

    fact_top factorial_accelerator (
            .A              (alu_out[3:2]),
            .WE             (WE1),
            .WD             (wd_dm[3:0]),
            .Rst            (rst),
            .Clk            (clk),
            .RD             (FactData)
        );

    gpio_top gpio_module (
            .A              (alu_out[3:2]),
            .WE             (WE2),
            .gpI1           (gpI1),
            .gpI2           (gpI2),
            .WD             (wd_dm),
            .Rst            (rst),
            .Clk            (clk),
            .gpO1           (gpO1),
            .gpO2           (gpO2),
            .RD             (GPIOData)
        );
endmodule
```

mips.v

```
module mips (
        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       branch;
    wire       jump;
    wire       reg_dst;
    wire       we_reg;
    wire       alu_src;
    wire       dm2reg;
    wire [2:0] alu_ctrl;

    // New wires Moved all control signals to output of this module to see in waveforms.
To remove, uncomment below and remove outputs from module definition
    wire       pc_jr_sel;
    wire       rf_mult_shift_wd_sel;
    wire       mult_we;
    wire       mult_hilo_sel;
```

```
    wire       mult_shift_sel;
    wire       sl_or_sr;
    wire       rf_wa_jal_sel;
    wire       rf_wd_jal_sel;

    datapath dp (
            .clk            (clk),
            .rst            (rst),
            .branch         (branch),
            .jump           (jump),
            .reg_dst        (reg_dst),
            .we_reg         (we_reg),
            .alu_src        (alu_src),
            .dm2reg         (dm2reg),
            .alu_ctrl       (alu_ctrl),
            .ra3            (ra3),
            .instr          (instr),
            .rd_dm          (rd_dm),
            .pc_current     (pc_current),
            .alu_out        (alu_out),
            .wd_dm          (wd_dm),
            .rd3            (rd3),
            .pc_jr_sel      (pc_jr_sel),
            .rf_mult_shift_wd_sel (rf_mult_shift_wd_sel),
            .mult_we        (mult_we),
            .mult_hilo_sel  (mult_hilo_sel),
            .mult_shift_sel (mult_shift_sel),
            .sl_or_sr       (sl_or_sr),
            .rf_wa_jal_sel  (rf_wa_jal_sel),
            .rf_wd_jal_sel  (rf_wd_jal_sel)
        );

    controlunit cu (
            .opcode         (instr[31:26]),
            .funct          (instr[5:0]),
            .branch         (branch),
            .jump           (jump),
            .reg_dst        (reg_dst),
            .we_reg         (we_reg),
            .alu_src        (alu_src),
            .we_dm          (we_dm),
            .dm2reg         (dm2reg),
            .alu_ctrl       (alu_ctrl),
            .pc_jr_sel      (pc_jr_sel),
            .rf_mult_shift_wd_sel (rf_mult_shift_wd_sel),
            .mult_we        (mult_we),
            .mult_hilo_sel  (mult_hilo_sel),
            .mult_shift_sel (mult_shift_sel),
            .sl_or_sr       (sl_or_sr),
            .rf_wa_jal_sel  (rf_wa_jal_sel),
            .rf_wd_jal_sel  (rf_wd_jal_sel)
        );

endmodule
```

datapath.v

```
module datapath (
        input  wire        clk,
        input  wire        rst,
        input  wire        branch,
        input  wire        jump,
        input  wire        reg_dst,
        input  wire        we_reg,
        input  wire        alu_src,
        input  wire        dm2reg,
        input  wire [2:0]  alu_ctrl,
        input  wire [4:0]  ra3,
        input  wire [31:0] instr,
        input  wire [31:0] rd_dm,

        // New inputs
        input wire        pc_jr_sel,
        input wire        rf_mult_shift_wd_sel,
        input wire        mult_we,
        input wire        mult_hilo_sel,
        input wire        mult_shift_sel,
        input wire        sl_or_sr,
        input wire        rf_wd_jal_sel,
        input wire        rf_wa_jal_sel,

        output wire [31:0] pc_current,
        output wire [31:0] alu_out,
        output wire [31:0] wd_dm,
        output wire [31:0] rd3

        // Outputs for simulation (to remove, remove from here, remove from mips.v
outputs and port connect, mips_top.v outputs and port connect, tb_mips_top.v
    );

    wire [4:0]  rf_wa;
    wire        pc_src;
    wire [31:0] pc_plus4;
    wire [31:0] pc_pre;
    wire [31:0] pc_next;
    wire [31:0] sext_imm;
    wire [31:0] ba;
    wire [31:0] bta;
    wire [31:0] jta;
    wire [31:0] alu_pa;
    wire [31:0] alu_pb;
    wire [31:0] wd_rf;
    wire        zero;

    // Newly added wires
    wire [31:0] pc_final;
    wire [63:0] mult_out;
    wire [31:0] mfhi;
    wire [31:0] mflo;
```

```
    wire [31:0] mult_result;
    wire [31:0] shift_result;
    wire [31:0] mult_shift_result;
    wire [31:0] rf_wd_jal_mux_out;
    wire [31:0] mux_alu_dm_out;
    wire [4:0]  rf_wa_mux_out;

    assign pc_src = branch & zero;
    assign ba = {sext_imm[29:0], 2'b00};
    assign jta = {pc_plus4[31:28], instr[25:0], 2'b00};

    // --- PC Logic --- //
    dreg pc_reg (
            .clk            (clk),
            .rst            (rst),
            .d              (pc_final),
            .q              (pc_current)
        );

    adder pc_plus_4 (
            .a              (pc_current),
            .b              (32'd4),
            .y              (pc_plus4)
        );

    adder pc_plus_br (
            .a              (pc_plus4),
            .b              (ba),
            .y              (bta)
        );

    mux2 #(32) pc_src_mux (
            .sel            (pc_src),
            .a              (pc_plus4),
            .b              (bta),
            .y              (pc_pre)
        );

    mux2 #(32) pc_jmp_mux (
            .sel            (jump),
            .a              (pc_pre),
            .b              (jta),
            .y              (pc_next)
        );

    // --- RF Logic --- //
    mux2 #(5) rf_wa_mux (
            .sel            (reg_dst),
            .a              (instr[20:16]),
            .b              (instr[15:11]),
            .y              (rf_wa_mux_out)
        );

    regfile rf (
            .clk            (clk),
            .we             (we_reg),
```

```
            .ra1            (instr[25:21]),
            .ra2            (instr[20:16]),
            .ra3            (ra3),
            .wa             (rf_wa),
            .wd             (wd_rf),
            .rd1            (alu_pa),
            .rd2            (wd_dm),
            .rd3            (rd3),
            .rst            (rst)
        );

    signext se (
            .a              (instr[15:0]),
            .y              (sext_imm)
        );

    // --- ALU Logic --- //
    mux2 #(32) alu_pb_mux (
            .sel            (alu_src),
            .a              (wd_dm),
            .b              (sext_imm),
            .y              (alu_pb)
        );

    alu alu (
            .op             (alu_ctrl),
            .a              (alu_pa),
            .b              (alu_pb),
            .zero           (zero),
            .y              (alu_out)
        );

    // --- MEM Logic --- //
    mux2 #(32) rf_wd_mux (
            .sel            (dm2reg),
            .a              (alu_out),
            .b              (rd_dm),
            .y              (mux_alu_dm_out)
        );

   // -- JR Logic --- //
   mux2 #(32) pc_jr_mux (
            .sel            (pc_jr_sel),
            .a              (pc_next),
            .b              (alu_pa),
            .y              (pc_final)
        );

   // -- MULTU, MFLO, MFHI Logic -- //
   mult multu (
            .a              (alu_pa),
            .b              (wd_dm),
            .y              (mult_out)
        );

   we_dreg #(32) hi (
```

```
          .d              (mult_out[63:32]),
          .q              (mfhi),
          .we             (mult_we),
          .clk            (clk)
     );

  we_dreg #(32) lo (
          .d              (mult_out[31:0]),
          .q              (mflo),
          .we             (mult_we),
          .clk            (clk)
     );

  mux2 #(32) mult_hilo_mux (
          .sel            (mult_hilo_sel),
          .a              (mfhi),
          .b              (mflo),
          .y              (mult_result)
     );

  // -- SLL, SLR Logic -- //
  shifter #(32) sll_slr_shifter (
          .sl_or_sr       (sl_or_sr),
          .shamt          (instr[10:6]),
          .d              (wd_dm),
          .y              (shift_result)
     );

  // -- MULTU or SHIFT MUX -- //
  mux2 #(32) mult_shift_mux (
          .sel            (mult_shift_sel),
          .a              (mult_result),
          .b              (shift_result),
          .y              (mult_shift_result)
     );

  mux2 #(32) rf_mult_shift_wd_mux (
          .sel            (rf_mult_shift_wd_sel),
          .a              (rf_wd_jal_mux_out),
          .b              (mult_shift_result),
          .y              (wd_rf)
     );

  // -- JAL Logic -- //
  mux2 #(32) rf_wd_jal_mux (
          .sel            (rf_wd_jal_sel),
          .a              (mux_alu_dm_out),
          .b              (pc_plus4),
          .y              (rf_wd_jal_mux_out)
     );

  mux2 #(5) rf_wa_jal_mux (
          .sel            (rf_wa_jal_sel),
          .a              (rf_wa_mux_out),
          .b              (5'd31),
          .y              (rf_wa)
```

```
        );


endmodule
```

tb_Single_Cycle_SoC.v

```
module tb_Single_Cycle_SoC;
    reg             clk;
    reg             rst;
    reg     [31:0]  gpI1, gpI2;
    reg     [4:0]   ra3;
    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]  gpO1, gpO2;
    wire    [31:0]  rd3;

    Single_Cycle_SoC DUT1 (
        .clk            (clk),
        .rst            (rst),
        .ra3            (ra3),
        .gpI1           (gpI1),
        .gpI2           (gpI2),
        .we_dm          (we_dm),
        .pc_current     (pc_current),
        .instr          (instr),
        .alu_out        (alu_out),
        .wd_dm          (wd_dm),
        .rd_dm          (rd_dm),
        .rd3            (rd3),
        .gpO1           (gpO1),
        .gpO2           (gpO2)
    );

    task tick;
    begin
        clk = 0; #5;
        clk = 1; #5;
    end
    endtask

    initial begin
        gpI1 = 32'b10011;   // n = 3, sel = 0
        rst = 1'b0; #5;
        rst = 1'b1; #5;
        rst = 1'b0;
        tick;

        gpI2 = gpO2;
```

```
        tick;

        while(pc_current != 32'h40) tick;
//        tick;
        $finish;
    end
endmodule
```

tb_fpga_Single_Cycle_SoC.v

```
module tb_fpga_Single_Cycle_SoC;
    reg             clk_tb, rst_tb, button_tb;
    reg     [4:0]   switches_tb;
    wire            factErr_tb;
    wire            dispSel_tb;
    wire    [3:0]   LEDSEL_tb;
    wire    [7:0]   LEDOUT_tb;
    integer         i;

    fpga_Single_Cycle_SoC DUT (
        .clk        (clk_tb),
        .rst        (rst_tb),
        .button     (),
        .switches   (switches_tb),
        .factErr    (factErr_tb),
        .dispSel    (dispSel_tb),
        .LEDSEL     (LEDSEL_tb),
        .LEDOUT     (LEDOUT_tb)
    );

    task tick;
    begin
        clk_tb = 0; #5;
        clk_tb = 1; #5;
    end
    endtask

    initial begin
        rst_tb = 1;
        tick;
        rst_tb = 0;
        i = 0;

        switches_tb = 5'b11010; // n = 10, sel = 1

        while(i < 50) begin
            tick;
            i = i+1;
        end

    end

endmodule
```

Appendix J: Code for Pipelined MIPS Processor Implementation

Pipelined_MIPS_Driver_Code.asm

```
# Label     Assembly                Description
main:       addi  $t0, $0, 0x0F     # $t0 = 0x0F
            addi  $t1, $0, 1        # $t1 = 1
            nop
            nop
            nop
            sll   $t4, $t1, 4       # $t4 = $t1 << 4

fact:       lw    $t2, 0x0900($0)   # read switches
            nop                     # $t2 data is required in following and instruction
            nop
            nop
            and   $t3, $t2, $t0     # get input data n
            nop                     # $t3 data is required in following sw instruction
            nop
            nop
            sw    $t3, 0x0800($0)   # write input data n
            sw    $t1, 0x0804($0)   # write control Go bit


poll:       lw    $t5, 0x0808($0)   # read status Done bit
            nop                     # $t5 data is required in following beq instruction
            nop
            nop
            beq   $t5, $0, poll     # wait until Done == 1

            nop
            nop
            nop

            srl   $t5, $t5, 1        # $t5 = $t5 >> 1
            nop                      # $t5 data is required in following and instruction
            nop
            nop
            and   $t5, $t5, $t1      # get status Error bit
            and   $t3, $t2, $t4      # get display Select
            nop                      # $t3 data is required in following or instruction
            nop
            nop
            or    $t3, $t3, $t5      # combine Sel and Err
            lw    $t5, 0x080C($0)    # read result data nf
            nop
            nop
            sw    $t3, 0x0908($0)    # display Sel and Err
            nop                     # $t5 from lw is required in following sw
instruction
            nop
            sw    $t5, 0x090C($0)    # display result nf
            nop
```

```
            nop
            nop

 done:      j     fact              # repeat fact loop
            nop
            nop
            nop
            nop
```

Memfile.dat (generated from Pipelined_MIPS_Driver_Code.asm)

```
2008000f
20090001
00000000
00000000
00000000
00096100
8c0a0900
00000000
00000000
00000000
01485824
00000000
00000000
00000000
ac0b0800
ac090804
8c0d0808
00000000
00000000
00000000
11a0fffb
00000000
00000000
00000000
000d6842
00000000
00000000
00000000
01a96824
014c5824
00000000
00000000
00000000
016d5825
8c0d080c
00000000
00000000
ac0b0908
00000000
00000000
ac0d090c
00000000
00000000
```

```
00000000
08000006
00000000
00000000
00000000
00000000
```

fpga_Pipelined_SoC.v

```
module fpga_Pipelined_SoC (
        input  wire       clk,
        input  wire       rst,
        input  wire       button,
        input  wire [4:0] switches,
        output wire [3:0] factErr,  // 4 bits wide to trigger 4 LEDs
        output wire       dispSel,
        output wire [3:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    wire  [15:0] reg_hex;
    wire         clk_sec;
    wire         clk_5KHz;
    wire         clk_pb;

    wire [7:0]  digit0;
    wire [7:0]  digit1;
    wire [7:0]  digit2;
    wire [7:0]  digit3;

    wire [31:0] gpO1_wire;
    wire [31:0] gpO2_wire;

    assign dispSel = gpO1_wire[4];  // Not bit 0 of GPO1 as the slides would have you
believe
    assign factErr[0] = gpO1_wire[0];
    assign factErr[1] = factErr[0];
    assign factErr[2] = factErr[0];
    assign factErr[3] = factErr[0];

    clk_gen clk_gen (
            .clk100MHz          (clk),
            .rst                (rst),
            .clk_4sec           (clk_sec),
            .clk_5KHz           (clk_5KHz)
        );

    button_debouncer bd (
            .clk                (clk_5KHz),
            .button             (button),
            .debounced_button   (clk_pb)
        );

    Pipelined_SoC Pipelined_SoC (
```

```
        .clk                (clk_5KHz), //clk_pb or clk_5KHz//
        .rst                (rst),
        .ra3                (),
        .gpI1               ({27'b0,switches[4:0]}),
        .gpI2               (gpO1_wire),
        .we_dm              (),
        .pc_current         (),
        .instr              (),
        .alu_out            (),
        .wd_dm              (),
        .rd_dm              (),
        .rd3                (),
        .gpO1               (gpO1_wire),
        .gpO2               (gpO2_wire),

        // From simulation
        .instrD             (),
        .instrE             (),
        .instrM             (),
        .instrW             (),
        .wd_rfW             ()
    );

mux2 #(16) fgpa_mux (
        .a                  (gpO2_wire[15:0]),
        .b                  (gpO2_wire[31:16]),
        .y                  (reg_hex),
        .sel                (dispSel)
    );

hex_to_7seg hex3 (
        .HEX                (reg_hex[15:12]),
        .s                  (digit3)
    );

hex_to_7seg hex2 (
        .HEX                (reg_hex[11:8]),
        .s                  (digit2)
    );

hex_to_7seg hex1 (
        .HEX                (reg_hex[7:4]),
        .s                  (digit1)
    );

hex_to_7seg hex0 (
        .HEX                (reg_hex[3:0]),
        .s                  (digit0)
    );

led_mux led_mux (
        .clk                (clk_5KHz),
        .rst                (rst),
        .LED3               (digit3),
        .LED2               (digit2),
        .LED1               (digit1),
```

```
            .LED0               (digit0),
            .LEDSEL             (LEDSEL),
            .LEDOUT             (LEDOUT)
        );
endmodule
```

Pipelined_SoC.v

```
module Pipelined_SoC (
        input  wire        clk,
        input  wire        rst,
        input  wire [4:0]  ra3,
        input  wire [31:0] gpI1, gpI2,

        output wire        we_dm,
        output wire [31:0] pc_current,
        output wire [31:0] instr,
        output wire [31:0] alu_out,
        output wire [31:0] wd_dm,
        output wire [31:0] rd_dm,
        output wire [31:0] rd3,
        output wire [31:0] gpO1, gpO2,

        // Outputs for simulation
        output wire [31:0] instrD, instrE, instrM, instrW,
        output wire [31:0] wd_rfW
    );

    wire            WE1, WE2, WEM;
    wire    [1:0]   RdSel;
    wire    [31:0]  DMemData, FactData, GPIOData;

    mips mips (
            .clk            (clk),
            .rst            (rst),
            .ra3            (ra3),
            .instrF         (instr),
            .rd_dmM         (rd_dm),
            .we_dmM         (we_dm),
            .pc_current     (pc_current),
            .alu_outM       (alu_out),
            .rd2M           (wd_dm),
            .rd3            (rd3),

            // Outputs for simulation
            .instrD         (instrD),
            .instrE         (instrE),
            .instrM         (instrM),
            .instrW         (instrW),
            .wd_rfW         (wd_rfW)
        );

    imem imem (
            .a              (pc_current[7:2]),
```

```
            .y              (instr)
        );

    dmem dmem (
            .clk            (clk),
            .we             (WEM),
            .a              (alu_out[7:2]),
            .d              (wd_dm),
            .q              (DMemData),
            .rst            (rst)
        );

    SoC_ad SoC_ad (
            .WE             (we_dm),
            .A              (alu_out),
            .WE1            (WE1),
            .WE2            (WE2),
            .WEM            (WEM),
            .RdSel          (RdSel)
        );

    SoC_mux4 SoC_mux4 (
            .DMemData       (DMemData),
            .FactData       (FactData),
            .GPIOData       (GPIOData),
            .RdSel          (RdSel),
            .ReadData       (rd_dm)
        );

    fact_top factorial_accelerator (
            .A              (alu_out[3:2]),
            .WE             (WE1),
            .WD             (wd_dm[3:0]),
            .Rst            (rst),
            .Clk            (clk),
            .RD             (FactData)
        );

    gpio_top gpio_module (
            .A              (alu_out[3:2]),
            .WE             (WE2),
            .gpI1           (gpI1),
            .gpI2           (gpI2),
            .WD             (wd_dm),
            .Rst            (rst),
            .Clk            (clk),
            .gpO1           (gpO1),
            .gpO2           (gpO2),
            .RD             (GPIOData)
        );
endmodule
```

mips.v

```
module mips (
        input  wire        clk,
        input  wire        rst,
        input  wire [4:0]  ra3,
        input  wire [31:0] instrF,      // Instruction
        input  wire [31:0] rd_dmM,      // RD

        output wire        we_dmM,      // MemWrite
        output wire [31:0] pc_current,  // PC
        output wire [31:0] alu_outM,    // Address
        output wire [31:0] rd2M,        // WD
        output wire [31:0] rd3,

        // Outputs for simulation. To remove, remove from here, uncomment from wire
declaration below, module instatiation in Pipelined_SoC.v and tb_Pipelined_SoC.v
        output wire [31:0] instrD, instrE, instrM, instrW,
        output wire [31:0] wd_rfW
    );

    // Fetch Stage
        // Datapath Wires
        wire    [31:0]  pc_final;
        wire    [31:0]  pc_next;
        wire    [31:0]  pc_pre;
        wire    [31:0]  pc_plus_4F;
        wire    [31:0]  jta;

    // Decode Stage
        // Datapath Wires
 //       wire    [31:0]  instrD;       // Added to output of MIPS for simulation
        wire    [31:0]  pc_plus_4D;
        wire    [31:0]  sext_immD;
        wire    [31:0]  rd1D;
        wire    [31:0]  rd2D;
        wire    [4:0]   rf_wa_mux_out;
        wire    [4:0]   rf_waD;

        // Control Signals
        wire            branchD;
        wire            jumpD;
        wire            reg_dstD;
        wire            we_regD;
        wire            alu_srcD;
        wire            we_dmD;
        wire            dm2regD;
        wire            rf_wd_jal_selD;
        wire            rf_wa_jal_selD;
        wire    [2:0]   alu_ctrlD;
        wire            mult_weD;
        wire            rf_mult_shift_wd_selD;
        wire            sl_or_srD;
        wire            mult_shift_selD;
        wire            pc_jr_selD;
        wire            mult_hilo_selD;

    // Execute Stage
```

```
        // Datapath Wires
        wire    [31:0]  rd1E;
        wire    [31:0]  rd2E;
        wire    [31:0]  sext_immE;
        wire    [4:0]   rf_waE;
        wire    [31:0]  pc_plus_4E;
        wire    [31:0]  ba;
//        wire    [31:0]  instrE;       // Added to output of MIPS for simulation
        wire            zero;
        wire            pc_srcE;
        wire    [31:0]  alu_outE;
        wire    [31:0]  alu_pb;
        wire    [63:0]  mult_outE;
        wire    [31:0]  btaE;
        wire    [31:0]  shift_resultE;

        // Control Signals
        wire            branchE;
        wire            jumpE;
        wire            we_regE;
        wire            alu_srcE;
        wire            we_dmE;
        wire            dm2regE;
        wire            rf_wd_jal_selE;
        wire     [2:0]  alu_ctrlE;
        wire            mult_weE;
        wire            rf_mult_shift_wd_selE;
        wire            sl_or_srE;
        wire            mult_shift_selE;
        wire            pc_jr_selE;
        wire            mult_hilo_selE;

    // Memory Stage
        // Datapath Wires
        wire            pc_srcM;
//        wire    [31:0]  alu_outM;     // Output of MIPS to DM
//        wire    [31:0]  rd2M;         // Output of MIPS to DM
        wire    [31:0]  rd1M;
        wire    [63:0]  mult_outM;
        wire    [4:0]   rf_waM;
        wire    [31:0]  pc_plus_4M;
        wire    [31:0]  btaM;
        wire    [31:0]  shift_resultM;
//        wire    [31:0]  instrM;       // Added to output of MIPS for simulation
        wire    [31:0]  mfhiM;
        wire    [31:0]  mfloM;


        // Control Signals
        wire            jumpM;
        wire            we_regM;
//        wire              we_dmM;       // Output of MIPS to DM
        wire            dm2regM;
        wire            rf_wd_jal_selM;
        wire            mult_weM;
        wire            rf_mult_shift_wd_selM;
```

```
        wire            mult_shift_selM;
        wire            pc_jr_selM;
        wire            mult_hilo_selM;

    // Writeback Stage
        // Datapath Wires
        wire    [31:0]  alu_outW;
        wire    [31:0]  rd_dmW;
        wire    [31:0]  rd1W;
        wire    [31:0]  mfhiW;
        wire    [31:0]  mfloW;
        wire    [4:0]   rf_waW;
        wire    [31:0]  pc_plus_4W;
        wire    [31:0]  shift_resultW;
//        wire    [31:0]  instrW;       // Added to output of MIPS for simulation
        wire    [31:0]  mult_result;
        wire    [31:0]  mux_alu_dm_out;
        wire    [31:0]  rf_wd_jal_mux_out;
        wire    [31:0]  mult_shift_result;
//        wire    [31:0]  wd_rfW;       // Added to output of MIPS for simulation

        // Control Signals
        wire            jumpW;
        wire            we_regW;
        wire            dm2regW;
        wire            rf_wd_jal_selW;
        wire            rf_mult_shift_wd_selW;
        wire            mult_shift_selW;
        wire            pc_jr_selW;
        wire            mult_hilo_selW;

     // Modules and connections
/////// Fetch
        assign jta = {pc_plus_4F[31:28], instrW[25:0], 2'b00};

        dreg pc_reg (
                .clk            (clk),
                .rst            (rst),
                .d              (pc_final),
                .q              (pc_current)
            );

        mux2 #(32) pc_jr_mux (
                .sel            (pc_jr_selW),
                .a              (pc_next),
                .b              (rd1W),
                .y              (pc_final)
            );

        mux2 #(32) pc_jmp_mux (
                .sel            (jumpW),
                .a              (pc_pre),
                .b              (jta),
                .y              (pc_next)
            );
```

```
        mux2 #(32) pc_src_mux (
                .sel            (pc_srcM),
                .a              (pc_plus_4F),
                .b              (btaM),
                .y              (pc_pre)
            );

        adder pc_plus_4 (
                .a              (pc_current),
                .b              (32'd4),
                .y              (pc_plus_4F)
            );

        FE_DE_Stage_Reg FE_DE_Stage_Reg (
                .clk            (clk),
                .rst            (rst),
                .instrF         (instrF),
                .pc_plus_4F     (pc_plus_4F),
                .instrD         (instrD),
                .pc_plus_4D     (pc_plus_4D)
            );

/////// Decode
        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),
                .pc_jr_sel          (pc_jr_selD),
                .rf_mult_shift_wd_sel (rf_mult_shift_wd_selD),
                .mult_we            (mult_weD),
                .mult_hilo_sel      (mult_hilo_selD),
                .mult_shift_sel     (mult_shift_selD),
                .sl_or_sr           (sl_or_srD),
                .rf_wa_jal_sel      (rf_wa_jal_selD),
                .rf_wd_jal_sel      (rf_wd_jal_selD)
            );

        regfile rf (
                .clk            (clk),
                .we             (we_regW),
                .ra1            (instrD[25:21]),
                .ra2            (instrD[20:16]),
                .ra3            (ra3),
                .wa             (rf_waW),
                .wd             (wd_rfW),
                .rd1            (rd1D),
                .rd2            (rd2D),
                .rd3            (rd3),
                .rst            (rst)
```

```
        );

    mux2 #(5) rf_wa_mux (
            .sel            (reg_dstD),
            .a              (instrD[20:16]),
            .b              (instrD[15:11]),
            .y              (rf_wa_mux_out)
        );

    mux2 #(5) rf_wa_jal_mux (
            .sel            (rf_wa_jal_selD),
            .a              (rf_wa_mux_out),
            .b              (5'd31),
            .y              (rf_waD)
        );

    signext se (
            .a              (instrD[15:0]),
            .y              (sext_immD)
        );

    DE_EX_State_Reg DE_EX_State_Reg (
            // Datapath
            .clk            (clk),
            .rst            (rst),
            .rd1D           (rd1D),
            .rd2D           (rd2D),
            .sext_immD      (sext_immD),
            .rf_waD         (rf_waD),
            .pc_plus_4D     (pc_plus_4D),
            .instrD         (instrD),
            .rd1E           (rd1E),
            .rd2E           (rd2E),
            .sext_immE      (sext_immE),
            .rf_waE         (rf_waE),
            .pc_plus_4E     (pc_plus_4E),
            .instrE         (instrE),

            // Control Signals
            .branchD        (branchD),
            .jumpD          (jumpD),
            .we_regD        (we_regD),
            .alu_srcD       (alu_srcD),
            .we_dmD         (we_dmD),
            .dm2regD        (dm2regD),
            .rf_wd_jal_selD (rf_wd_jal_selD),
            .alu_ctrlD      (alu_ctrlD),
            .mult_weD       (mult_weD),
            .rf_mult_shift_wd_selD  (rf_mult_shift_wd_selD),
            .sl_or_srD      (sl_or_srD),
            .mult_shift_selD    (mult_shift_selD),
            .pc_jr_selD     (pc_jr_selD),
            .mult_hilo_selD (mult_hilo_selD),
            .branchE        (branchE),
            .jumpE          (jumpE),
            .we_regE        (we_regE),
```

```
                .alu_srcE       (alu_srcE),
                .we_dmE         (we_dmE),
                .dm2regE        (dm2regE),
                .rf_wd_jal_selE (rf_wd_jal_selE),
                .alu_ctrlE      (alu_ctrlE),
                .mult_weE       (mult_weE),
                .rf_mult_shift_wd_selE  (rf_mult_shift_wd_selE),
                .sl_or_srE      (sl_or_srE),
                .mult_shift_selE    (mult_shift_selE),
                .pc_jr_selE     (pc_jr_selE),
                .mult_hilo_selE (mult_hilo_selE)
            );

/////// Execute
        assign pc_srcE = branchE & zero;
        assign ba = {sext_immE[29:0], 2'b00};

        mux2 #(32) alu_pb_mux (
                .sel            (alu_srcE),
                .a              (rd2E),
                .b              (sext_immE),
                .y              (alu_pb)
            );

        alu alu (
                .op             (alu_ctrlE),
                .a              (rd1E),
                .b              (alu_pb),
                .zero           (zero),
                .y              (alu_outE)
            );

        mult multu (
                .a              (rd1E),
                .b              (rd2E),
                .y              (mult_outE)
            );

        adder pc_plus_br (
                .a              (pc_plus_4E),
                .b              (ba),
                .y              (btaE)
            );

        shifter #(32) sll_slr_shifter (
                .sl_or_sr       (sl_or_srE),
                .shamt          (instrE[10:6]),
                .d              (rd2E),
                .y              (shift_resultE)
            );

        EX_MEM_State_Reg EX_MEM_State_Reg(
                .clk            (clk),
                .rst            (rst),
                .pc_srcE        (pc_srcE),
                .alu_outE       (alu_outE),
```

```
                .rd1E           (rd1E),
                .rd2E           (rd2E),
                .mult_outE      (mult_outE),
                .rf_waE         (rf_waE),
                .pc_plus_4E     (pc_plus_4E),
                .btaE           (btaE),
                .shift_resultE  (shift_resultE),
                .instrE         (instrE),
                .pc_srcM        (pc_srcM),
                .alu_outM       (alu_outM),
                .rd1M           (rd1M),
                .rd2M           (rd2M),
                .mult_outM      (mult_outM),
                .rf_waM         (rf_waM),
                .pc_plus_4M     (pc_plus_4M),
                .btaM           (btaM),
                .shift_resultM  (shift_resultM),
                .instrM         (instrM),

                // Control Signals
                .jumpE          (jumpE),
                .we_regE        (we_regE),
                .we_dmE         (we_dmE),
                .dm2regE        (dm2regE),
                .rf_wd_jal_selE (rf_wd_jal_selE),
                .mult_weE       (mult_weE),
                .rf_mult_shift_wd_selE  (rf_mult_shift_wd_selE),
                .mult_shift_selE    (mult_shift_selE),
                .pc_jr_selE     (pc_jr_selE),
                .mult_hilo_selE (mult_hilo_selE),
                .jumpM          (jumpM),
                .we_regM        (we_regM),
                .we_dmM         (we_dmM),
                .dm2regM        (dm2regM),
                .rf_wd_jal_selM (rf_wd_jal_selM),
                .mult_weM       (mult_weM),
                .rf_mult_shift_wd_selM (rf_mult_shift_wd_selM),
                .mult_shift_selM    (mult_shift_selM),
                .pc_jr_selM     (pc_jr_selM),
                .mult_hilo_selM (mult_hilo_selM)
            );

/////// Memory
        we_dreg #(32) hi (
                .d              (mult_outM[63:32]),
                .q              (mfhiM),
                .we             (mult_weM),
                .clk            (clk)
            );

        we_dreg #(32) lo (
                .d              (mult_outM[31:0]),
                .q              (mfloM),
                .we             (mult_weM),
                .clk            (clk)
            );
```

```
        MEM_WB_State_Reg MEM_WB_State_Reg (
                // Datapath
                .clk            (clk),
                .rst            (rst),
                .alu_outM       (alu_outM),
                .rd_dmM         (rd_dmM),
                .rd1M           (rd1M),
                .mfhiM          (mfhiM),
                .mfloM          (mfloM),
                .rf_waM         (rf_waM),
                .pc_plus_4M     (pc_plus_4M),
                .shift_resultM  (shift_resultM),
                .instrM         (instrM),
                .alu_outW       (alu_outW),
                .rd_dmW         (rd_dmW),
                .rd1W           (rd1W),
                .mfhiW          (mfhiW),
                .mfloW          (mfloW),
                .rf_waW         (rf_waW),
                .pc_plus_4W     (pc_plus_4W),
                .shift_resultW  (shift_resultW),
                .instrW         (instrW),

                // Control Signals
                .jumpM          (jumpM),
                .we_regM        (we_regM),
                .dm2regM        (dm2regM),
                .rf_wd_jal_selM (rf_wd_jal_selM),
                .rf_mult_shift_wd_selM  (rf_mult_shift_wd_selM),
                .mult_shift_selM    (mult_shift_selM),
                .pc_jr_selM     (pc_jr_selM),
                .mult_hilo_selM (mult_hilo_selM),
                .jumpW          (jumpW),
                .we_regW        (we_regW),
                .dm2regW        (dm2regW),
                .rf_wd_jal_selW (rf_wd_jal_selW),
                .rf_mult_shift_wd_selW  (rf_mult_shift_wd_selW),
                .mult_shift_selW(mult_shift_selW),
                .pc_jr_selW     (pc_jr_selW),
                .mult_hilo_selW (mult_hilo_selW)
            );

/////// Writeback
        mux2 #(32) mult_hilo_mux (
                .sel            (mult_hilo_selW),
                .a              (mfhiW),
                .b              (mfloW),
                .y              (mult_result)
            );

        mux2 #(32) rf_wd_mux (
                .sel            (dm2regW),
                .a              (alu_outW),
                .b              (rd_dmW),
                .y              (mux_alu_dm_out)
```

```
            );

        mux2 #(32) mult_shift_mux (
                .sel            (mult_shift_selW),
                .a              (mult_result),
                .b              (shift_resultW),
                .y              (mult_shift_result)
            );

        mux2 #(32) rf_mult_shift_wd_mux (
                .sel            (rf_mult_shift_wd_selW),
                .a              (rf_wd_jal_mux_out),
                .b              (mult_shift_result),
                .y              (wd_rfW)
            );

        mux2 #(32) rf_wd_jal_mux (
                .sel            (rf_wd_jal_selW),
                .a              (mux_alu_dm_out),
                .b              (pc_plus_4W),
                .y              (rf_wd_jal_mux_out)
            );
endmodule
```

FE_DE_State_Reg.v

```
module FE_DE_Stage_Reg(
        input   wire            clk, rst,
        input   wire    [31:0]  instrF,
        input   wire    [31:0]  pc_plus_4F,

        output  reg     [31:0]  instrD,
        output  reg     [31:0]  pc_plus_4D
    );

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
            instrD      <= 0;
            pc_plus_4D  <= 0;
        end
        else begin
            instrD      <= instrF;
            pc_plus_4D  <= pc_plus_4F;
        end
    end
endmodule
```

DE_EX_State_Reg.v

```
module DE_EX_State_Reg(
        // Datapath
        input   wire            clk, rst,
        input   wire    [31:0]  rd1D,
```

```
        input   wire    [31:0]  rd2D,
        input   wire    [31:0]  sext_immD,
        input   wire    [4:0]   rf_waD,
        input   wire    [31:0]  pc_plus_4D,
        input   wire    [31:0]  instrD,

        output  reg     [31:0]  rd1E,
        output  reg     [31:0]  rd2E,
        output  reg     [31:0]  sext_immE,
        output  reg     [4:0]   rf_waE,
        output  reg     [31:0]  pc_plus_4E,
        output  reg     [31:0]  instrE,

        // Control Signals
        input   wire            branchD,
        input   wire            jumpD,
        input   wire            we_regD,
        input   wire            alu_srcD,
        input   wire            we_dmD,
        input   wire            dm2regD,
        input   wire            rf_wd_jal_selD,
        input   wire    [2:0]   alu_ctrlD,
        input   wire            mult_weD,
        input   wire            rf_mult_shift_wd_selD,
        input   wire            sl_or_srD,
        input   wire            mult_shift_selD,
        input   wire            pc_jr_selD,
        input   wire            mult_hilo_selD,

        output   reg            branchE,
        output   reg            jumpE,
        output   reg            we_regE,
        output   reg            alu_srcE,
        output   reg            we_dmE,
        output   reg            dm2regE,
        output   reg            rf_wd_jal_selE,
        output   reg    [2:0]   alu_ctrlE,
        output   reg            mult_weE,
        output   reg            rf_mult_shift_wd_selE,
        output   reg            sl_or_srE,
        output   reg            mult_shift_selE,
        output   reg            pc_jr_selE,
        output   reg            mult_hilo_selE
    );

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
            rd1E                    <= 0;
            rd2E                    <= 0;
            sext_immE               <= 0;
            rf_waE                  <= 0;
            pc_plus_4E              <= 0;
            instrE                  <= 0;

            branchE                 <= 0;
            jumpE                   <= 0;
```

```
            we_regE                 <= 0;
            alu_srcE                <= 0;
            we_dmE                  <= 0;
            dm2regE                 <= 0;
            rf_wd_jal_selE          <= 0;
            alu_ctrlE               <= 0;
            mult_weE                <= 0;
            rf_mult_shift_wd_selE   <= 0;
            sl_or_srE               <= 0;
            mult_shift_selE         <= 0;
            pc_jr_selE              <= 0;
            mult_hilo_selE          <= 0;
        end
        else begin
            rd1E                    <= rd1D;
            rd2E                    <= rd2D;
            sext_immE               <= sext_immD;
            rf_waE                  <= rf_waD;
            pc_plus_4E              <= pc_plus_4D;
            instrE                  <= instrD;

            branchE                 <= branchD;
            jumpE                   <= jumpD;
            we_regE                 <= we_regD;
            alu_srcE                <= alu_srcD;
            we_dmE                  <= we_dmD;
            dm2regE                 <= dm2regD;
            rf_wd_jal_selE          <= rf_wd_jal_selD;
            alu_ctrlE               <= alu_ctrlD;
            mult_weE                <= mult_weD;
            rf_mult_shift_wd_selE   <= rf_mult_shift_wd_selD;
            sl_or_srE               <= sl_or_srD;
            mult_shift_selE         <= mult_shift_selD;
            pc_jr_selE              <= pc_jr_selD;
            mult_hilo_selE          <= mult_hilo_selD;
        end
    end
endmodule
```

EX_MEM_State_Reg.v

```
module EX_MEM_State_Reg (
        // Datapath
        input   wire            clk, rst,
        input   wire            pc_srcE,
        input   wire    [31:0]  alu_outE,
        input   wire    [31:0]  rd1E,
        input   wire    [31:0]  rd2E,
        input   wire    [63:0]  mult_outE,
        input   wire    [4:0]   rf_waE,
        input   wire    [31:0]  pc_plus_4E,
        input   wire    [31:0]  btaE,
        input   wire    [31:0]  shift_resultE,
        input   wire    [31:0]  instrE,
```

```
    output  reg             pc_srcM,
    output  reg     [31:0]  alu_outM,
    output  reg     [31:0]  rd1M,
    output  reg     [31:0]  rd2M,
    output  reg     [63:0]  mult_outM,
    output  reg     [4:0]   rf_waM,
    output  reg     [31:0]  pc_plus_4M,
    output  reg     [31:0]  btaM,
    output  reg     [31:0]  shift_resultM,
    output  reg     [31:0]  instrM,

    // Control Signals
    input   wire            jumpE,
    input   wire            we_regE,
    input   wire            we_dmE,
    input   wire            dm2regE,
    input   wire            rf_wd_jal_selE,
    input   wire            mult_weE,
    input   wire            rf_mult_shift_wd_selE,
    input   wire            mult_shift_selE,
    input   wire            pc_jr_selE,
    input   wire            mult_hilo_selE,

    output  reg             jumpM,
    output  reg             we_regM,
    output  reg             we_dmM,
    output  reg             dm2regM,
    output  reg             rf_wd_jal_selM,
    output  reg             mult_weM,
    output  reg             rf_mult_shift_wd_selM,
    output  reg             mult_shift_selM,
    output  reg             pc_jr_selM,
    output  reg             mult_hilo_selM
);

always @ (posedge clk, posedge rst) begin
    if (rst) begin
        pc_srcM         <= 0;
        alu_outM        <= 0;
        rd1M            <= 0;
        rd2M            <= 0;
        mult_outM       <= 0;
        rf_waM          <= 0;
        pc_plus_4M      <= 0;
        btaM            <= 0;
        shift_resultM   <= 0;
        instrM          <= 0;

        jumpM           <= 0;
        we_regM         <= 0;
        we_dmM          <= 0;
        dm2regM         <= 0;
        rf_wd_jal_selM  <= 0;
        mult_weM        <= 0;
        rf_mult_shift_wd_selM   <= 0;
```

```
            mult_shift_selM <= 0;
            pc_jr_selM      <= 0;
            mult_hilo_selM  <= 0;
        end
        else begin
            pc_srcM         <= pc_srcE;
            alu_outM        <= alu_outE;
            rd1M            <= rd1E;
            rd2M            <= rd2E;
            mult_outM       <= mult_outE;
            rf_waM          <= rf_waE;
            pc_plus_4M      <= pc_plus_4E;
            btaM            <= btaE;
            shift_resultM   <= shift_resultE;
            instrM          <= instrE;

            jumpM           <= jumpE;
            we_regM         <= we_regE;
            we_dmM          <= we_dmE;
            dm2regM         <= dm2regE;
            rf_wd_jal_selM  <= rf_wd_jal_selE;
            mult_weM        <= mult_weE;
            rf_mult_shift_wd_selM   <= rf_mult_shift_wd_selE;
            mult_shift_selM <= mult_shift_selE;
            pc_jr_selM      <= pc_jr_selE;
            mult_hilo_selM  <= mult_hilo_selE;
        end
    end
endmodule
```

MEM_WB_State_Reg.v

```
module MEM_WB_State_Reg (
        // Datapath
        input   wire            clk, rst,
        input   wire    [31:0]  alu_outM,
        input   wire    [31:0]  rd_dmM,
        input   wire    [31:0]  rd1M,
        input   wire    [31:0]  mfhiM,
        input   wire    [31:0]  mfloM,
        input   wire    [4:0]   rf_waM,
        input   wire    [31:0]  pc_plus_4M,
        input   wire    [31:0]  shift_resultM,
        input   wire    [31:0]  instrM,

        output  reg     [31:0]  alu_outW,
        output  reg     [31:0]  rd_dmW,
        output  reg     [31:0]  rd1W,
        output  reg     [31:0]  mfhiW,
        output  reg     [31:0]  mfloW,
        output  reg     [4:0]   rf_waW,
        output  reg     [31:0]  pc_plus_4W,
        output  reg     [31:0]  shift_resultW,
        output  reg     [31:0]  instrW,
```

```
    // Control Signals
    input   wire            jumpM,
    input   wire            we_regM,
    input   wire            dm2regM,
    input   wire            rf_wd_jal_selM,
    input   wire            rf_mult_shift_wd_selM,
    input   wire            mult_shift_selM,
    input   wire            pc_jr_selM,
    input   wire            mult_hilo_selM,

    output  reg             jumpW,
    output  reg             we_regW,
    output  reg             dm2regW,
    output  reg             rf_wd_jal_selW,
    output  reg             rf_mult_shift_wd_selW,
    output  reg             mult_shift_selW,
    output  reg             pc_jr_selW,
    output  reg             mult_hilo_selW
);

always @ (posedge clk, posedge rst) begin
    if (rst) begin
        alu_outW            <= 0;
        rd_dmW              <= 0;
        rd1W                <= 0;
        mfhiW               <= 0;
        mfloW               <= 0;
        rf_waW              <= 0;
        pc_plus_4W          <= 0;
        shift_resultW       <= 0;
        instrW              <= 0;

        jumpW               <= 0;
        we_regW             <= 0;
        dm2regW             <= 0;
        rf_wd_jal_selW      <= 0;
        rf_mult_shift_wd_selW   <= 0;
        mult_shift_selW     <= 0;
        pc_jr_selW          <= 0;
        mult_hilo_selW      <= 0;
    end
    else begin
        alu_outW            <= alu_outM;
        rd_dmW              <= rd_dmM;
        rd1W                <= rd1M;
        mfhiW               <= mfhiM;
        mfloW               <= mfloM;
        rf_waW              <= rf_waM;
        pc_plus_4W          <= pc_plus_4M;
        shift_resultW       <= shift_resultM;
        instrW              <= instrM;

        jumpW               <= jumpM;
        we_regW             <= we_regM;
        dm2regW             <= dm2regM;
```

```
            rf_wd_jal_selW      <= rf_wd_jal_selM;
            rf_mult_shift_wd_selW   <= rf_mult_shift_wd_selM;
            mult_shift_selW     <= mult_shift_selM;
            pc_jr_selW          <= pc_jr_selM;
            mult_hilo_selW      <= mult_hilo_selM;
        end
    end
endmodule
```

tb_Pipelined_SoC.v

```
module tb_Pipelined_SoC;
    reg             clk;
    reg             rst;
    reg     [31:0]  gpI1, gpI2;
    reg     [4:0]   ra3;
    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]  gpO1, gpO2;
    wire    [31:0]  rd3;

    // Outputs for simulation
    wire    [31:0]  instrD, instrE, instrM, instrW;
    wire    [31:0]  wd_rfW;

    Pipelined_SoC DUT1 (
        .clk            (clk),
        .rst            (rst),
        .ra3            (ra3),
        .gpI1           (gpI1),
        .gpI2           (gpI2),
        .we_dm          (we_dm),
        .pc_current     (pc_current),
        .instr          (instr),
        .alu_out        (alu_out),
        .wd_dm          (wd_dm),
        .rd_dm          (rd_dm),
        .rd3            (rd3),
        .gpO1           (gpO1),
        .gpO2           (gpO2),

        // Outputs for simulation
        .instrD         (instrD),
        .instrE         (instrE),
        .instrM         (instrM),
        .instrW         (instrW),
        .wd_rfW         (wd_rfW)
    );

    task tick;
```

```
    begin
        clk = 0; #5;
        clk = 1; #5;
    end
    endtask

    initial begin
        // SoC_ad is outputting RdSel = 01 when alu_out (A) is 0x900???
        gpI1 = 32'b00000;   // n = 5, sel = 0
        rst = 1'b0; #5;
        rst = 1'b1; #5;
        rst = 1'b0;
        ra3 = 13;           // probe a register value
        tick;

        gpI2 = gpO2;
        tick;

        while(pc_current != 32'hb8) // tick until end of program
            tick;
        $finish;
    end
endmodule
```

tb_fpga_Pipelined_SoC.v

```
module tb_fpga_Pipelined_SoC;
    reg             clk_tb, rst_tb, button_tb;
    reg     [4:0]   switches_tb;
    wire            factErr_tb;
    wire            dispSel_tb;
    wire    [3:0]   LEDSEL_tb;
    wire    [7:0]   LEDOUT_tb;
    integer         i;

    fpga_Pipelined_SoC DUT (
        .clk        (clk_tb),
        .rst        (rst_tb),
        .button     (),
        .switches   (switches_tb),
        .factErr    (factErr_tb),
        .dispSel    (dispSel_tb),
        .LEDSEL     (LEDSEL_tb),
        .LEDOUT     (LEDOUT_tb)
    );

    task tick;
    begin
        clk_tb = 0; #5;
        clk_tb = 1; #5;
    end
    endtask

    initial begin
```

```
        rst_tb = 1;
        tick;
        rst_tb = 0;
        i = 0;

        switches_tb = 5'b00100; // n = 4, sel = 1

        while(i < 100) begin
            tick;
            i = i+1;
        end

    end

endmodule
```

Appendix K: Hardware Validation Code

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
```

hex_to_7seg.v

```
module hex_to_7seg (
            input  wire [3:0] HEX,
            output reg  [7:0] s
        );

    always @ (HEX) begin
        case (HEX)
                4'h0: s = 8'b11000000;
                4'h1: s = 8'b11111001;
                4'h2: s = 8'b10100100;
                4'h3: s = 8'b10110000;
                4'h4: s = 8'b10011001;
                4'h5: s = 8'b10010010;
                4'h6: s = 8'b10000010;
                4'h7: s = 8'b11111000;
                4'h8: s = 8'b10000000;
                4'h9: s = 8'b10010000;
                4'hA: s = 8'b10100000; // b10001000; This is capital A
                4'hB: s = 8'b10000011; // b10000000; This is capital B, but is the same
as 8
                4'hC: s = 8'b10100111; // b11000110; This is capital C
                4'hD: s = 8'b10100001; // b11000000; This is capital D, but is the same
as 0
                4'hE: s = 8'b10000110;
                4'hF: s = 8'b10001110;
            default: s = 8'b01111111;
       endcase
    end

endmodule
```

led_mux.v

```
module led_mux (
        input  wire       clk,
        input  wire       rst,
        input  wire [7:0] LED3,
        input  wire [7:0] LED2,
        input  wire [7:0] LED1,
        input  wire [7:0] LED0,
        output wire [3:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    reg [1:0]  index;
    reg [11:0] led_ctrl;

    assign {LEDSEL, LEDOUT} = led_ctrl;

    always @ (posedge clk) index <= (rst) ? 2'b0 : (index + 2'd1);

    always @ (index, LED0, LED1, LED2, LED3) begin
        case (index)
               2'd0: led_ctrl <= {4'b1110, LED0};
               2'd1: led_ctrl <= {4'b1101, LED1};
               2'd2: led_ctrl <= {4'b1011, LED2};
               2'd3: led_ctrl <= {4'b0111, LED3};
            default: led_ctrl <= {4'b1111, 8'hFF};
        endcase
    end

endmodule
```

clk_gen.v

```
module clk_gen (
        input  wire clk100MHz,
        input  wire rst,
        output reg  clk_4sec,
        output reg  clk_5KHz
    );

    integer count1, count2;

    always @ (posedge clk100MHz) begin
        if (rst) begin
            count1 = 0;
            count2 = 0;
            clk_5KHz = 0;
            clk_4sec = 0;
        end
        else begin
            if (count1 == 200000000) begin
                clk_4sec = ~clk_4sec;
                count1 = 0;
            end
```

```
            if (count2 == 10000) begin
                clk_5KHz = ~clk_5KHz;
                count2 = 0;
            end

            count1 = count1 + 1;
            count2 = count2 + 1;
        end
    end

endmodule
```

button_debouncer.v

```
module button_debouncer #(parameter depth = 16) (
        input  wire clk,                /* 5 KHz clock */
        input  wire button,             /* Input button from constraints */
        output reg  debounced_button
    );

    localparam history_max = (2**depth)-1;

    /* History of sampled input button */
    reg [depth-1:0] history;

    always @ (posedge clk) begin
        /* Move history back one sample and insert new sample */
        history <= { button, history[depth-1:1] };

        /* Assert debounced button if it has been in a consistent state throughout
history */
        debounced_button <= (history == history_max) ? 1'b1 : 1'b0;
    end

endmodule
```

fpga_top.xdc

```
# Clock Signal
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33}           [get_ports {clk}];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

# Buttons
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports {button}]; # Center
Button
set_property -dict {PACKAGE_PIN W19 IOSTANDARD LVCMOS33} [get_ports {rst}]; # Left
Button

# Switches
```

```
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports {switches[0]}];  #
Switch 0
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports {switches[1]}];  #
Switch 1
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports {switches[2]}];  #
Switch 2
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33} [get_ports {switches[3]}];  #
Switch 3
set_property -dict {PACKAGE_PIN W15 IOSTANDARD LVCMOS33} [get_ports {switches[4]}];  #
Switch 4

# LEDs
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports {factErr[0]}];  #
LED 0
set_property -dict {PACKAGE_PIN E19 IOSTANDARD LVCMOS33} [get_ports {factErr[1]}];  #
LED 1
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports {factErr[2]}];  #
LED 2
set_property -dict {PACKAGE_PIN V19 IOSTANDARD LVCMOS33} [get_ports {factErr[3]}];  #
LED 3
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports {dispSel}];  # LED 4

# 7 segment display
set_property -dict {PACKAGE_PIN W7 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[0]}]; # CA
set_property -dict {PACKAGE_PIN W6 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[1]}]; # CB
set_property -dict {PACKAGE_PIN U8 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[2]}]; # CC
set_property -dict {PACKAGE_PIN V8 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[3]}]; # CD
set_property -dict {PACKAGE_PIN U5 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[4]}]; # CE
set_property -dict {PACKAGE_PIN V5 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[5]}]; # CF
set_property -dict {PACKAGE_PIN U7 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[6]}]; # CG
set_property -dict {PACKAGE_PIN V7 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[7]}]; # DP

set_property -dict {PACKAGE_PIN U2 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[0]}]; # AN0
set_property -dict {PACKAGE_PIN U4 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[1]}]; # AN1
set_property -dict {PACKAGE_PIN V4 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[2]}]; # AN2
set_property -dict {PACKAGE_PIN W4 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[3]}]; # AN3
```

Appendix L: Data Forwarding - Attempt at Hazard Control

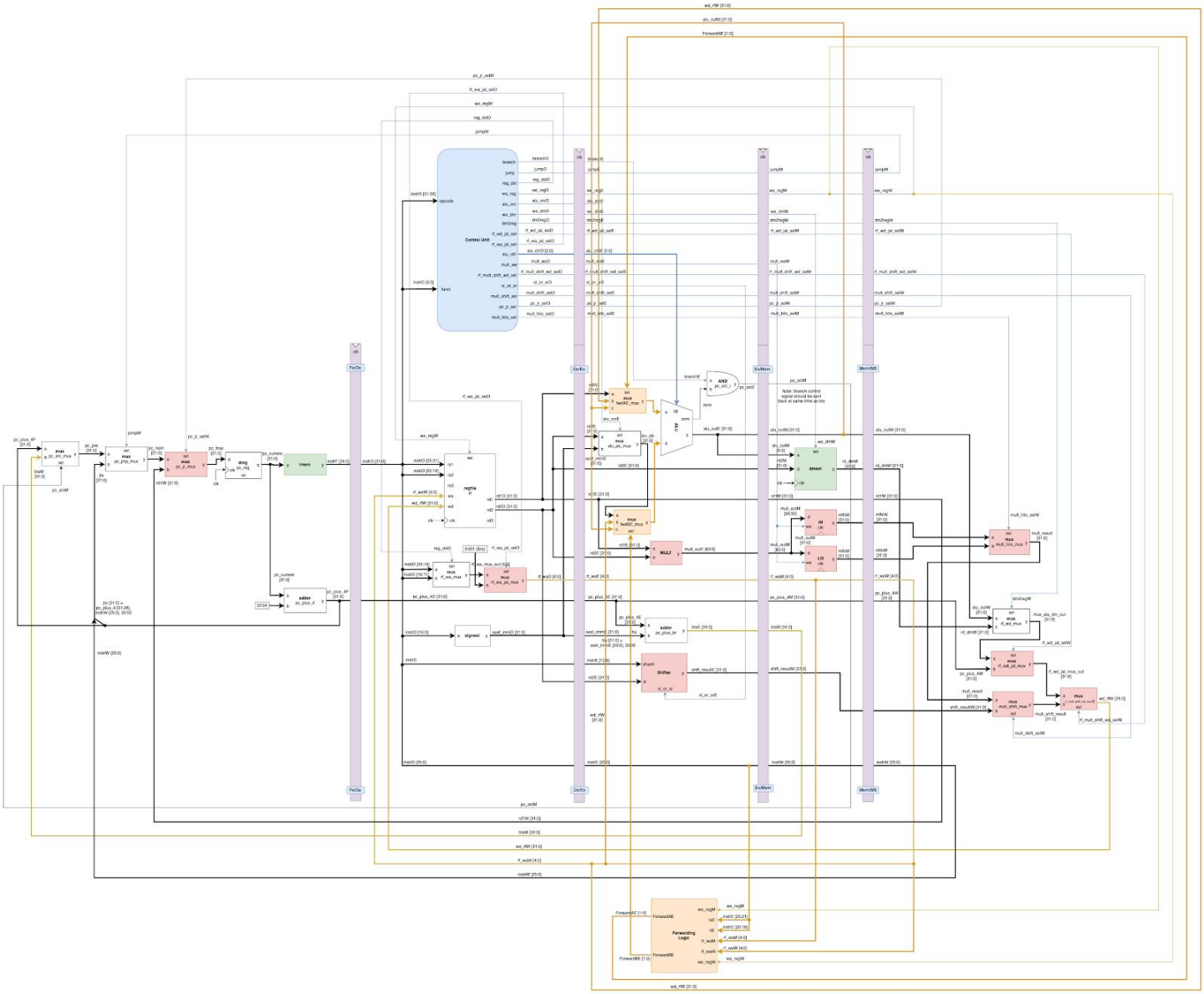
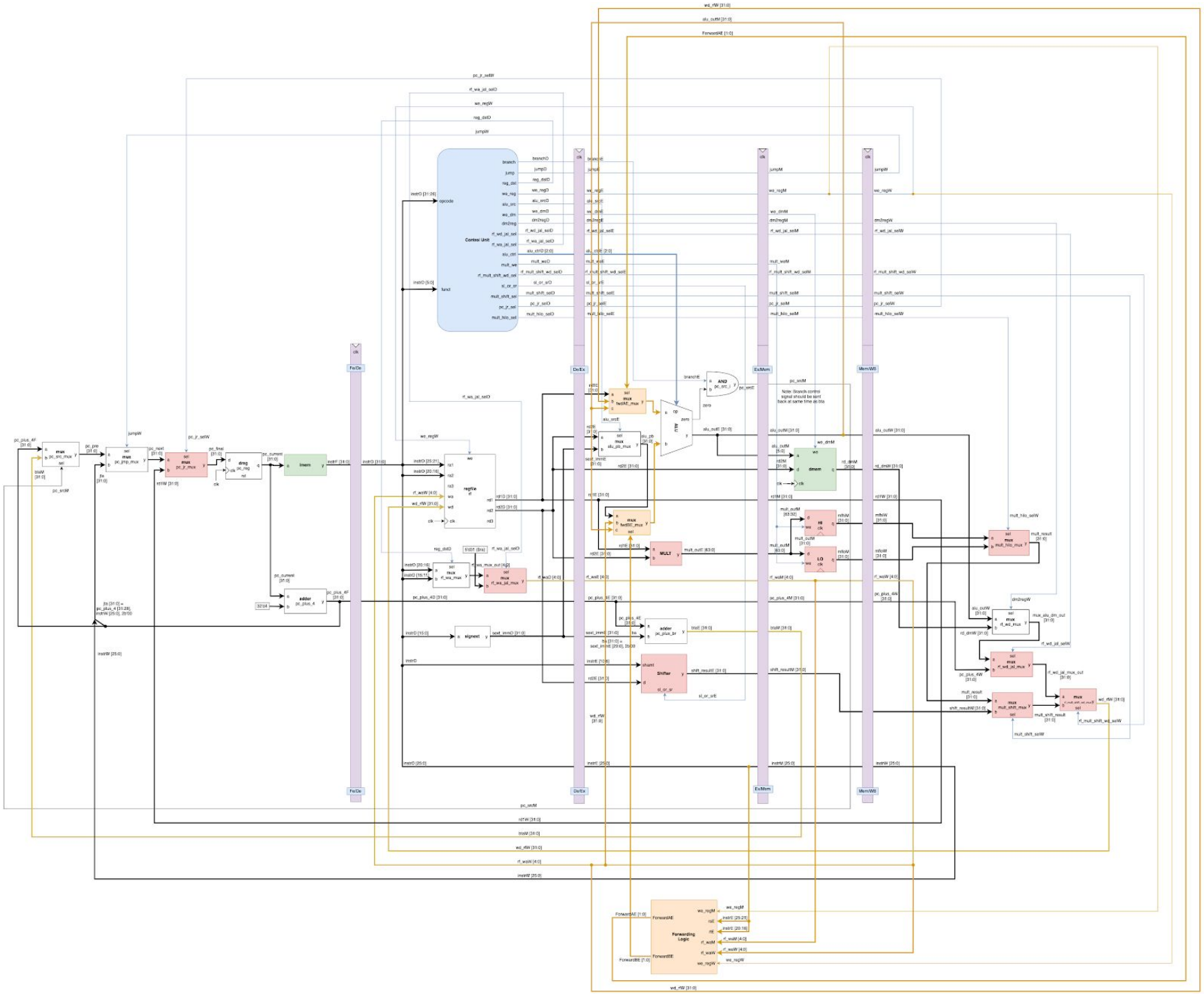


Figure 45: Pipelined MIPS CPU with data forwarding logic and multiplexers (in orange). A larger image is present on the next page


Control Unit
opcode
funct
branch
jump
reg_dst
we_reg
alu_src
we_dm
dm2reg
rf_wd_jal_sel
rf_wa_jal_sel
alu_ctrl
mult_we
rf_mult_shift_wd_sel
sl_or_sr
mult_shift_sel
pc_jr_sel
mult_hilo_sel
Fet/De
De/Ex
Ex/Mem
Mem/WB
regfile
Forwarding Logic
ALU
Shifter
MULT
HI
LO
dmem
imem
signext
adder
Note: Branch control signal should be sent back at same time as this

Appendix M: Code for Attempted Data Forwarding

forwarding_logic.v

```
module forwarding_logic(
        input   wire            we_regM,
        input   wire    [4:0]   rsE,        // instrE [25:21]
        input   wire    [4:0]   rtE,        // instrE [20:16]
        input   wire    [4:0]   rf_waM,
        input   wire    [4:0]   rf_waW,
        input   wire            we_regW,

        output  reg    [1:0]   ForwardAE,
        output  reg    [1:0]   ForwardBE
    );

    always @ (*) begin
        // ForwardAE
        if(we_regM && (rsE != 0) && (rf_waM == rsE))
            ForwardAE = 10;
        else if(we_regW && (rsE != 0) && (rf_waW == rsE))
            ForwardAE = 01;
        else
            ForwardAE = 00;

        // ForwardBE
        if(we_regM && (rtE != 0) && (rf_waM == rtE))
            ForwardBE = 10;
        else if(we_regW && (rtE != 0) && (rf_waW == rtE))
            ForwardBE = 01;
        else
            ForwardBE = 00;
    end
endmodule
```

mux3.v

```
module mux3 #(parameter WIDTH = 32) (
         input  [WIDTH-1:0] a,
         input  [WIDTH-1:0] b,
         input  [WIDTH-1:0] c,
         input  [WIDTH-1:0] d,
         input  [1:0]       sel,
         output [WIDTH-1:0] out);

   // When sel[1] is 0, (sel[0]? b:a) is selected and when sel[1] is 1, (sel[0] ? d:c)
is taken
   // When sel[0] is 0, a is sent to output, else b and when sel[0] is 0, c is sent to
output, else d
   assign out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);
```

```
endmodule
```

mips.v

```
module mips (
        input  wire        clk,
        input  wire        rst,
        input  wire [4:0]  ra3,
        input  wire [31:0] instrF,      // Instruction
        input  wire [31:0] rd_dmM,      // RD

        output wire        we_dmM,      // MemWrite
        output wire [31:0] pc_current,  // PC
        output wire [31:0] alu_outM,    // Address
        output wire [31:0] rd2M,        // WD
        output wire [31:0] rd3,

        // Outputs for simulation. To remove, remove from here, uncomment from wire
declaration below, module instatiation in Pipelined_SoC.v and tb_Pipelined_SoC.v
        output wire [31:0] instrD, instrE, instrM, instrW,
        output wire [31:0] wd_rfW,
        output wire [1:0]  ForwardAE, ForwardBE,
        output wire [31:0] fwdAE_mux_out, fwdBE_mux_out
    );

    // Fetch Stage
        // Datapath Wires
        wire    [31:0]  pc_final;
        wire    [31:0]  pc_next;
        wire    [31:0]  pc_pre;
        wire    [31:0]  pc_plus_4F;
        wire    [31:0]  jta;

    // Decode Stage
        // Datapath Wires
 //       wire    [31:0]  instrD;       // Added to output of MIPS for simulation
        wire    [31:0]  pc_plus_4D;
        wire    [31:0]  sext_immD;
        wire    [31:0]  rd1D;
        wire    [31:0]  rd2D;
        wire    [4:0]   rf_wa_mux_out;
        wire    [4:0]   rf_waD;

        // Control Signals
        wire            branchD;
        wire            jumpD;
        wire            reg_dstD;
        wire            we_regD;
        wire            alu_srcD;
        wire            we_dmD;
        wire            dm2regD;
        wire            rf_wd_jal_selD;
        wire            rf_wa_jal_selD;
```

```
        wire    [2:0]   alu_ctrlD;
        wire            mult_weD;
        wire            rf_mult_shift_wd_selD;
        wire            sl_or_srD;
        wire            mult_shift_selD;
        wire            pc_jr_selD;
        wire            mult_hilo_selD;

    // Execute Stage
        // Datapath Wires
        wire    [31:0]  rd1E;
        wire    [31:0]  rd2E;
        wire    [31:0]  sext_immE;
        wire    [4:0]   rf_waE;
        wire    [31:0]  pc_plus_4E;
        wire    [31:0]  ba;
//        wire    [31:0]  instrE;       // Added to output of MIPS for simulation
        wire            zero;
        wire            pc_srcE;
        wire    [31:0]  alu_outE;
        wire    [31:0]  alu_pb;
        wire    [63:0]  mult_outE;
        wire    [31:0]  btaE;
        wire    [31:0]  shift_resultE;

        // Control Signals
        wire            branchE;
        wire            jumpE;
        wire            we_regE;
        wire            alu_srcE;
        wire            we_dmE;
        wire            dm2regE;
        wire            rf_wd_jal_selE;
        wire     [2:0]  alu_ctrlE;
        wire            mult_weE;
        wire            rf_mult_shift_wd_selE;
        wire            sl_or_srE;
        wire            mult_shift_selE;
        wire            pc_jr_selE;
        wire            mult_hilo_selE;

    // Memory Stage
        // Datapath Wires
        wire            pc_srcM;
//        wire    [31:0]  alu_outM;     // Output of MIPS to DM
//        wire    [31:0]  rd2M;         // Output of MIPS to DM
        wire    [31:0]  rd1M;
        wire    [63:0]  mult_outM;
        wire    [4:0]   rf_waM;
        wire    [31:0]  pc_plus_4M;
        wire    [31:0]  btaM;
        wire    [31:0]  shift_resultM;
//        wire    [31:0]  instrM;       // Added to output of MIPS for simulation
        wire    [31:0]  mfhiM;
        wire    [31:0]  mfloM;
```

```
        // Control Signals
        wire            jumpM;
        wire            we_regM;
//        wire            we_dmM;       // Output of MIPS to DM
        wire            dm2regM;
        wire            rf_wd_jal_selM;
        wire            mult_weM;
        wire            rf_mult_shift_wd_selM;
        wire            mult_shift_selM;
        wire            pc_jr_selM;
        wire            mult_hilo_selM;

    // Writeback Stage
        // Datapath Wires
        wire    [31:0]  alu_outW;
        wire    [31:0]  rd_dmW;
        wire    [31:0]  rd1W;
        wire    [31:0]  mfhiW;
        wire    [31:0]  mfloW;
        wire    [4:0]   rf_waW;
        wire    [31:0]  pc_plus_4W;
        wire    [31:0]  shift_resultW;
//        wire    [31:0]  instrW;       // Added to output of MIPS for simulation
        wire    [31:0]  mult_result;
        wire    [31:0]  mux_alu_dm_out;
        wire    [31:0]  rf_wd_jal_mux_out;
        wire    [31:0]  mult_shift_result;
//        wire    [31:0]  wd_rfW;       // Added to output of MIPS for simulation

        // Control Signals
        wire            jumpW;
        wire            we_regW;
        wire            dm2regW;
        wire            rf_wd_jal_selW;
        wire            rf_mult_shift_wd_selW;
        wire            mult_shift_selW;
        wire            pc_jr_selW;
        wire            mult_hilo_selW;

    //  Forwarding Logic
//        wire    [1:0]   ForwardAE, ForwardBE;
//        wire    [31:0]  fwdAE_mux_out, fwdBE_mux_out;

     // Modules and connections
/////// Fetch
        assign jta = {pc_plus_4F[31:28], instrW[25:0], 2'b00};

        dreg pc_reg (
                .clk            (clk),
                .rst            (rst),
                .d              (pc_final),
                .q              (pc_current)
            );

        mux2 #(32) pc_jr_mux (
```

```
                .sel            (pc_jr_selW),
                .a              (pc_next),
                .b              (rd1W),
                .y              (pc_final)
            );

        mux2 #(32) pc_jmp_mux (
                .sel            (jumpW),
                .a              (pc_pre),
                .b              (jta),
                .y              (pc_next)
            );

        mux2 #(32) pc_src_mux (
                .sel            (pc_srcM),
                .a              (pc_plus_4F),
                .b              (btaM),
                .y              (pc_pre)
            );

        adder pc_plus_4 (
                .a              (pc_current),
                .b              (32'd4),
                .y              (pc_plus_4F)
            );

        FE_DE_Stage_Reg FE_DE_Stage_Reg (
                .clk            (clk),
                .rst            (rst),
                .instrF         (instrF),
                .pc_plus_4F     (pc_plus_4F),
                .instrD         (instrD),
                .pc_plus_4D     (pc_plus_4D)
            );

/////// Decode
        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),
                .pc_jr_sel          (pc_jr_selD),
                .rf_mult_shift_wd_sel (rf_mult_shift_wd_selD),
                .mult_we            (mult_weD),
                .mult_hilo_sel      (mult_hilo_selD),
                .mult_shift_sel     (mult_shift_selD),
                .sl_or_sr           (sl_or_srD),
                .rf_wa_jal_sel      (rf_wa_jal_selD),
                .rf_wd_jal_sel      (rf_wd_jal_selD)
            );
```

```
    regfile rf (
            .clk            (clk),
            .we             (we_regW),
            .ra1            (instrD[25:21]),
            .ra2            (instrD[20:16]),
            .ra3            (ra3),
            .wa             (rf_waW),
            .wd             (wd_rfW),
            .rd1            (rd1D),
            .rd2            (rd2D),
            .rd3            (rd3),
            .rst            (rst)
        );

    mux2 #(5) rf_wa_mux (
            .sel            (reg_dstD),
            .a              (instrD[20:16]),
            .b              (instrD[15:11]),
            .y              (rf_wa_mux_out)
        );

    mux2 #(5) rf_wa_jal_mux (
            .sel            (rf_wa_jal_selD),
            .a              (rf_wa_mux_out),
            .b              (5'd31),
            .y              (rf_waD)
        );

    signext se (
            .a              (instrD[15:0]),
            .y              (sext_immD)
        );

    DE_EX_State_Reg DE_EX_State_Reg (
            // Datapath
            .clk            (clk),
            .rst            (rst),
            .rd1D           (rd1D),
            .rd2D           (rd2D),
            .sext_immD      (sext_immD),
            .rf_waD         (rf_waD),
            .pc_plus_4D     (pc_plus_4D),
            .instrD         (instrD),
            .rd1E           (rd1E),
            .rd2E           (rd2E),
            .sext_immE      (sext_immE),
            .rf_waE         (rf_waE),
            .pc_plus_4E     (pc_plus_4E),
            .instrE         (instrE),

            // Control Signals
            .branchD        (branchD),
            .jumpD          (jumpD),
            .we_regD        (we_regD),
            .alu_srcD       (alu_srcD),
```

```
                .we_dmD         (we_dmD),
                .dm2regD        (dm2regD),
                .rf_wd_jal_selD (rf_wd_jal_selD),
                .alu_ctrlD      (alu_ctrlD),
                .mult_weD       (mult_weD),
                .rf_mult_shift_wd_selD  (rf_mult_shift_wd_selD),
                .sl_or_srD      (sl_or_srD),
                .mult_shift_selD    (mult_shift_selD),
                .pc_jr_selD     (pc_jr_selD),
                .mult_hilo_selD (mult_hilo_selD),
                .branchE        (branchE),
                .jumpE          (jumpE),
                .we_regE        (we_regE),
                .alu_srcE       (alu_srcE),
                .we_dmE         (we_dmE),
                .dm2regE        (dm2regE),
                .rf_wd_jal_selE (rf_wd_jal_selE),
                .alu_ctrlE      (alu_ctrlE),
                .mult_weE       (mult_weE),
                .rf_mult_shift_wd_selE  (rf_mult_shift_wd_selE),
                .sl_or_srE      (sl_or_srE),
                .mult_shift_selE    (mult_shift_selE),
                .pc_jr_selE     (pc_jr_selE),
                .mult_hilo_selE (mult_hilo_selE)
            );

/////// Execute
        assign pc_srcE = branchE & zero;
        assign ba = {sext_immE[29:0], 2'b00};

        mux2 #(32) alu_pb_mux (
                .sel            (alu_srcE),
                .a              (rd2E),
                .b              (sext_immE),
                .y              (alu_pb)
            );

        alu alu (
                .op             (alu_ctrlE),
                .a              (fwdAE_mux_out),
                .b              (fwdBE_mux_out),
                .zero           (zero),
                .y              (alu_outE)
            );

        mult multu (
                .a              (rd1E),
                .b              (rd2E),
                .y              (mult_outE)
            );

        adder pc_plus_br (
                .a              (pc_plus_4E),
                .b              (ba),
                .y              (btaE)
            );
```

```
        shifter #(32) sll_slr_shifter (
                .sl_or_sr       (sl_or_srE),
                .shamt          (instrE[10:6]),
                .d              (rd2E),
                .y              (shift_resultE)
            );

        EX_MEM_State_Reg EX_MEM_State_Reg(
                .clk            (clk),
                .rst            (rst),
                .pc_srcE        (pc_srcE),
                .alu_outE       (alu_outE),
                .rd1E           (rd1E),
                .rd2E           (rd2E),
                .mult_outE      (mult_outE),
                .rf_waE         (rf_waE),
                .pc_plus_4E     (pc_plus_4E),
                .btaE           (btaE),
                .shift_resultE  (shift_resultE),
                .instrE         (instrE),
                .pc_srcM        (pc_srcM),
                .alu_outM       (alu_outM),
                .rd1M           (rd1M),
                .rd2M           (rd2M),
                .mult_outM      (mult_outM),
                .rf_waM         (rf_waM),
                .pc_plus_4M     (pc_plus_4M),
                .btaM           (btaM),
                .shift_resultM  (shift_resultM),
                .instrM         (instrM),

                // Control Signals
                .jumpE          (jumpE),
                .we_regE        (we_regE),
                .we_dmE         (we_dmE),
                .dm2regE        (dm2regE),
                .rf_wd_jal_selE (rf_wd_jal_selE),
                .mult_weE       (mult_weE),
                .rf_mult_shift_wd_selE  (rf_mult_shift_wd_selE),
                .mult_shift_selE    (mult_shift_selE),
                .pc_jr_selE     (pc_jr_selE),
                .mult_hilo_selE (mult_hilo_selE),
                .jumpM          (jumpM),
                .we_regM        (we_regM),
                .we_dmM         (we_dmM),
                .dm2regM        (dm2regM),
                .rf_wd_jal_selM (rf_wd_jal_selM),
                .mult_weM       (mult_weM),
                .rf_mult_shift_wd_selM (rf_mult_shift_wd_selM),
                .mult_shift_selM    (mult_shift_selM),
                .pc_jr_selM     (pc_jr_selM),
                .mult_hilo_selM (mult_hilo_selM)
            );

/////// Memory
```

```
        we_dreg #(32) hi (
                .d              (mult_outM[63:32]),
                .q              (mfhiM),
                .we             (mult_weM),
                .clk            (clk)
            );

        we_dreg #(32) lo (
                .d              (mult_outM[31:0]),
                .q              (mfloM),
                .we             (mult_weM),
                .clk            (clk)
            );

        MEM_WB_State_Reg MEM_WB_State_Reg (
                // Datapath
                .clk            (clk),
                .rst            (rst),
                .alu_outM       (alu_outM),
                .rd_dmM         (rd_dmM),
                .rd1M           (rd1M),
                .mfhiM          (mfhiM),
                .mfloM          (mfloM),
                .rf_waM         (rf_waM),
                .pc_plus_4M     (pc_plus_4M),
                .shift_resultM  (shift_resultM),
                .instrM         (instrM),
                .alu_outW       (alu_outW),
                .rd_dmW         (rd_dmW),
                .rd1W           (rd1W),
                .mfhiW          (mfhiW),
                .mfloW          (mfloW),
                .rf_waW         (rf_waW),
                .pc_plus_4W     (pc_plus_4W),
                .shift_resultW  (shift_resultW),
                .instrW         (instrW),

                // Control Signals
                .jumpM          (jumpM),
                .we_regM        (we_regM),
                .dm2regM        (dm2regM),
                .rf_wd_jal_selM (rf_wd_jal_selM),
                .rf_mult_shift_wd_selM  (rf_mult_shift_wd_selM),
                .mult_shift_selM    (mult_shift_selM),
                .pc_jr_selM     (pc_jr_selM),
                .mult_hilo_selM (mult_hilo_selM),
                .jumpW          (jumpW),
                .we_regW        (we_regW),
                .dm2regW        (dm2regW),
                .rf_wd_jal_selW (rf_wd_jal_selW),
                .rf_mult_shift_wd_selW  (rf_mult_shift_wd_selW),
                .mult_shift_selW(mult_shift_selW),
                .pc_jr_selW     (pc_jr_selW),
                .mult_hilo_selW (mult_hilo_selW)
            );
```

```
/////// Writeback
        mux2 #(32) mult_hilo_mux (
                .sel            (mult_hilo_selW),
                .a              (mfhiW),
                .b              (mfloW),
                .y              (mult_result)
            );

        mux2 #(32) rf_wd_mux (
                .sel            (dm2regW),
                .a              (alu_outW),
                .b              (rd_dmW),
                .y              (mux_alu_dm_out)
            );

        mux2 #(32) mult_shift_mux (
                .sel            (mult_shift_selW),
                .a              (mult_result),
                .b              (shift_resultW),
                .y              (mult_shift_result)
            );

        mux2 #(32) rf_mult_shift_wd_mux (
                .sel            (rf_mult_shift_wd_selW),
                .a              (rf_wd_jal_mux_out),
                .b              (mult_shift_result),
                .y              (wd_rfW)
            );

        mux2 #(32) rf_wd_jal_mux (
                .sel            (rf_wd_jal_selW),
                .a              (mux_alu_dm_out),
                .b              (pc_plus_4W),
                .y              (rf_wd_jal_mux_out)
            );

 ////// Forwarding Logic
        mux3 #(32) fwdAE_mux (
                .sel            (ForwardAE),
                .a              (rd1E),
                .b              (wd_rfW),
                .c              (alu_outM),
                .d              (),
                .out            (fwdAE_mux_out)
            );

        mux3 #(32) fwdBE_mux (
                .sel            (ForwardBE),
                .a              (rd2E),
                .b              (wd_rfW),
                .c              (alu_outM),
                .d              (),
                .out            (fwdBE_mux_out)
            );

        forwarding_logic fwd_logic (
```

```
                .we_regM        (we_regM),
                .rsE            (instrE[25:21]),
                .rtE            (instrE[20:16]),
                .rf_waM         (rf_waM),
                .rf_waW         (rf_waW),
                .we_regW        (we_regW),
                .ForwardAE      (ForwardAE),
                .ForwardBE      (ForwardBE)
            );
endmodule
```

Authors

Sidarth Shahri

I am a senior majoring in Computer Engineering at San Jose State University. I plan to graduate in Spring 2020. I hope to go into firmware engineering or embedded systems development in my career. In my free time, I enjoy playing video games, rock climbing, and playing board games with friends. CMPE 140 was extremely challenging for me, but I'm proud of myself for rising to the challenge.

Manuel Chavez Sanchez

I am a senior Computer Engineering student at San Jose State University, graduating this Fall 2019. My career interests are embedded software or firmware development, computer networking, and cybersecurity. In my free time I enjoy watching TV series, reading nonfiction books, or explore outdoor events with friends. CMPE140 has made me appreciate, better, the inner workings of computer hardware.

Akash Sindhu

I am a BS senior at San Jose State University majoring in Computer Engineering. My area of interest is Machine learning and Data Science. I will graduate in May 2020. In my free time, I usually learn something new in machine learning and try to implement it new projects. Other than ML, I do like to go to the gym and eat healthy food. Things I want to learn in the future are Reinforcement learning and Quantum machine learning.

Karine Worley

I am also a senior majoring in computer engineering at San Jose State University, graduating this Fall 2019. Regarding career interests, I am interested in software development, machine learning, AI, and mobile applications. I would like to further develop ML based applications in the future. In my free time I also enjoy cooking, learning machine learning, and reading books.