

## Table of Contents

[**Table of Contents**](#_r65sr5prg6gd) **2**

[**Purpose**](#_svp6jivu97bx) **4**

[**Design Methodology**](#_3qmhvvaqj8ii) **4**

[System-on-Chip (SoC) Design](#_oaec9pdq06eg) 4

[Factorial Accelerator with Interface Wrapper](#_6q0i8nofs975) 7

[General Purpose I/O (GPIO) with Interface Wrapper](#_h98lq2fafn68) 11

[Single-cycle MIPS Processor](#_fw8c2lnm15zb) 13

[Pipelined MIPS Processor](#_c568wtvhsm8) 15

[Data Forwarding](#_13mezcf3agxj) 18

[**Performance Analysis**](#_sooaabhjgu7b) **19**

[**Waveforms**](#_7uhrv1ixky1w) **20**

[Factorial Accelerator with Interface Wrapper Testbench](#_i20dhd9ihcst) 21

[GPIO Unit with Interface Wrapper Testbench](#_i20dhd9ihcst) 22

[Single-Cycle SoC Testbench](#_i20dhd9ihcst) 23

[Pipelined SoC Testbench](#_i20dhd9ihcst) 24

[**Hardware Validation**](#_ezehtro2oukg) **27**

[**Accomplished Procedures**](#_gwc0abmu9rdr) **30**

[**Conclusion**](#_7a082pqvx7b7) **31**

[**Appendix A: Control Unit and Memory Module Diagrams**](#_16ybe1f5ittp) **32**

[**Appendix B: Single-Cycle MIPS Processor Design Diagrams and Truth Tables**](#_177gnpejjp1n) **33**

[**Appendix C: Pipelined MIPS Processor Design Diagrams and Truth Tables**](#_a6wwgri4w0y0) **37**

[**Appendix D: Single-Cycle MIPS Processor in SoC Hardware Validation**](#_y9mcbxchylum) **42**

[**Appendix E: Pipelined MIPS Processor in SoC Hardware Validation**](#_84jlbe45zqsx) **45**

[**Appendix F: Factorial Accelerator with Interface Wrapper Code**](#_5jrzzusev84w) **49**

[**Appendix G: GPIO Unit with Interface Wrapper Code**](#_yrdwzh5uz01p) **61**

[**Appendix H: Shared Code between Single-Cycle MIPS Processor and Pipelined MIPS Processor on SoC Implementations**](#_1k6wj98zgcom) **65**

[**Appendix I: Code for Single-Cycle MIPS Processor Implementation**](#_vthhoeddqjh) **73**

[**Appendix J: Code for Pipelined MIPS Processor Implementation**](#_m758gccrjvaj) **86**

[**Appendix K: Hardware Validation Code**](#_wpqqgpy68003) **109**

[**Appendix L: Data Forwarding - Attempt at Hazard Control**](#_shmhytc0g55h) **113**

[**Appendix M: Code for Attempted Data Forwarding**](#_k6ayfobrjbhx) **115**

[**Authors**](#_tmmzw9ig1j65) **126**

## 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.

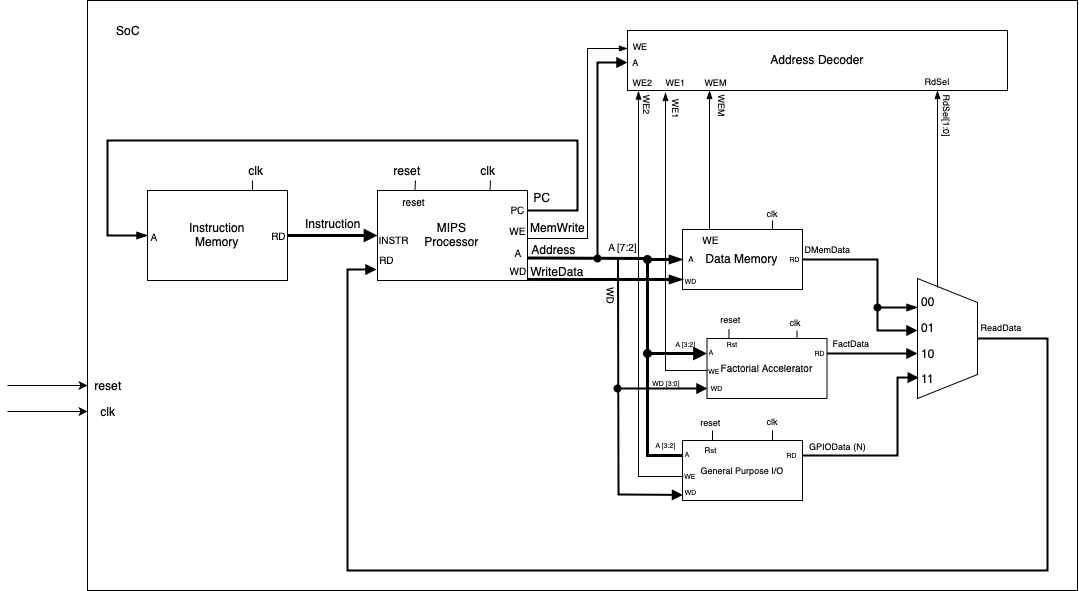


Figure 1: Design of the SoC that is shared by both CPU designs

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.

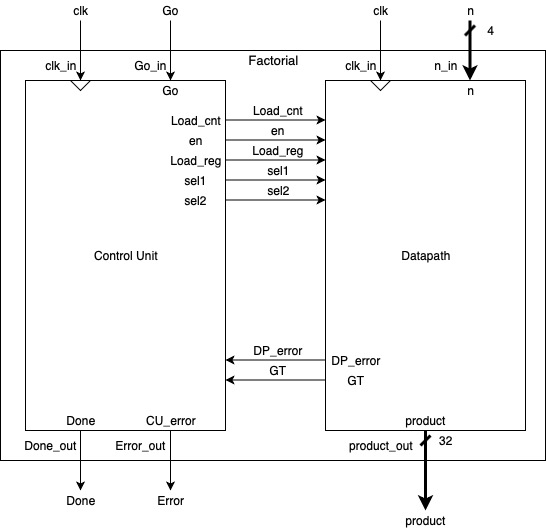


Figure 2: Factorial module containing control unit and datapath modules with their inputs and outputs.

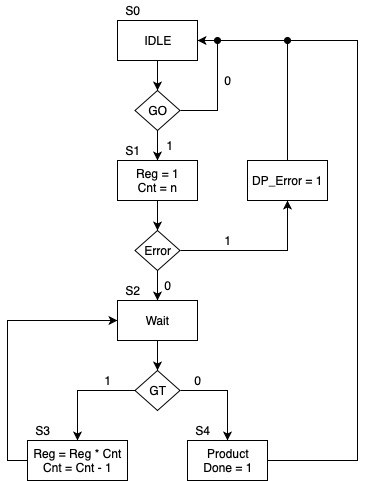
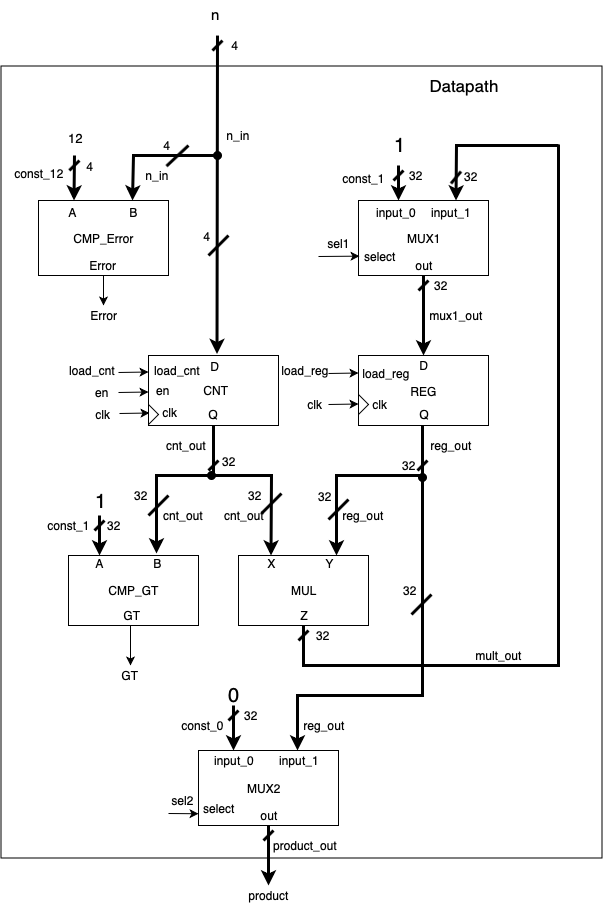


Figure 3 and 4. **Left**: Design of the datapath of the factorial accelerator. **Right**: Design of the control unit of the factorial accelerator.

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.

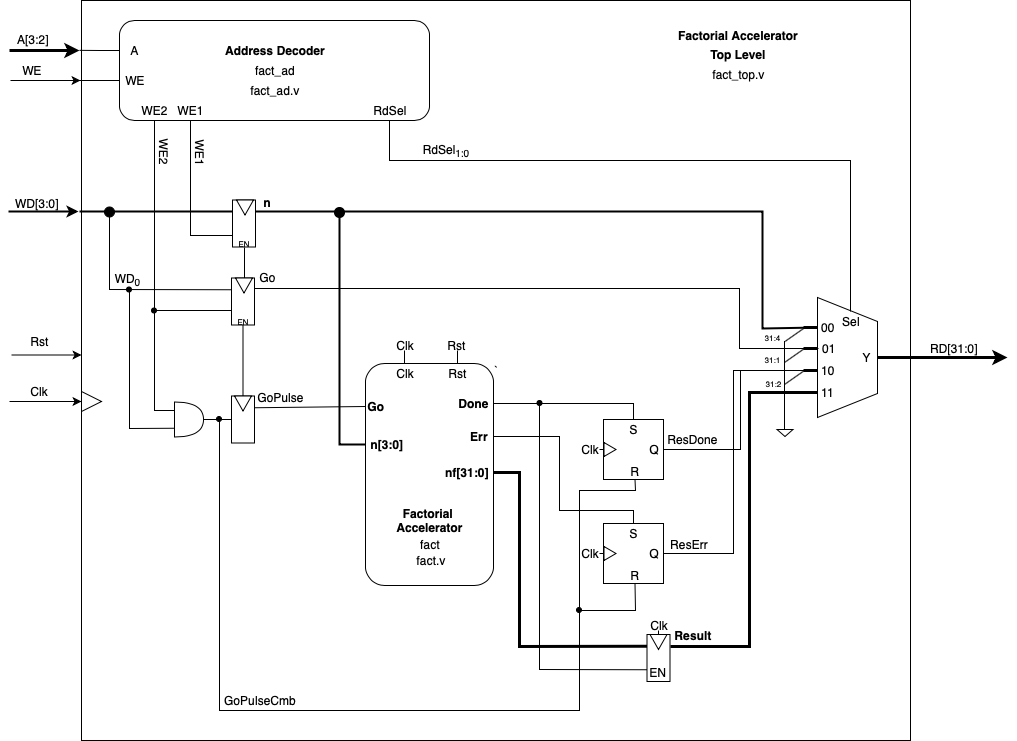


Figure 5: Diagram of the interface wrapper containing the factorial module from Assignment 1 with a few new components to control outputs of the factorial module and decode addresses given to the interface.

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.

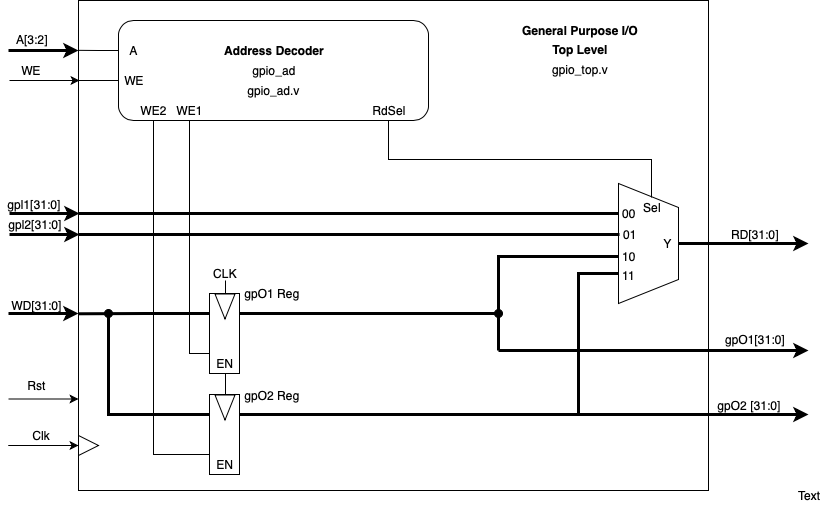


Figure 6: Diagram of the GPIO interface wrapper

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.

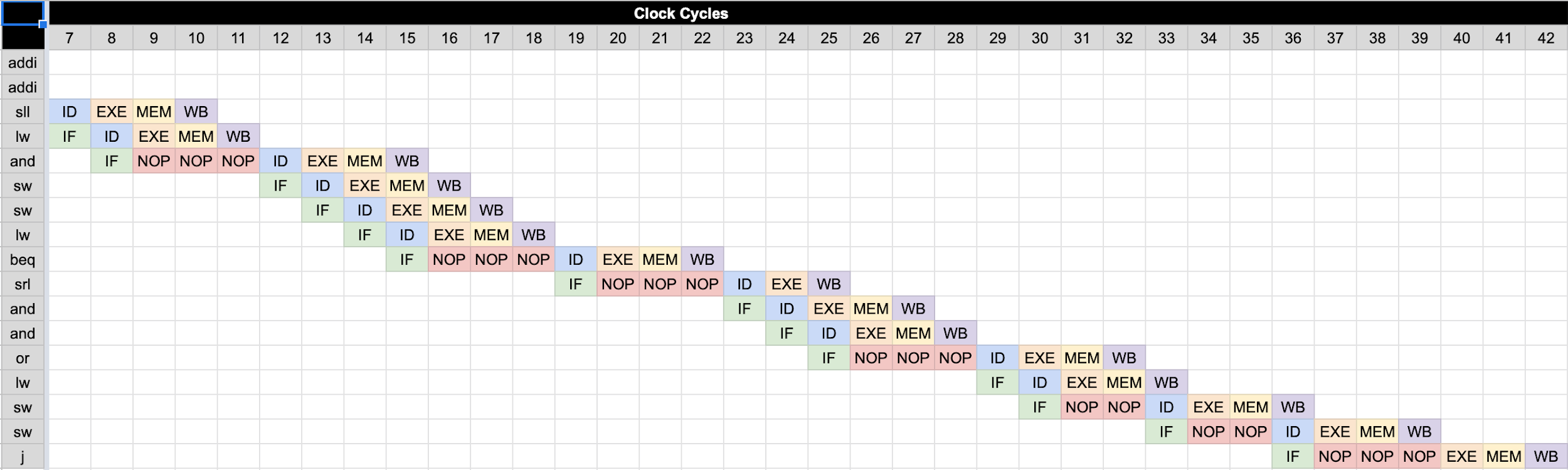


Figure 7: Time Space Diagram for modified MIPS Driver Code (code in Appendix J)

### 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.

(1)

(2)

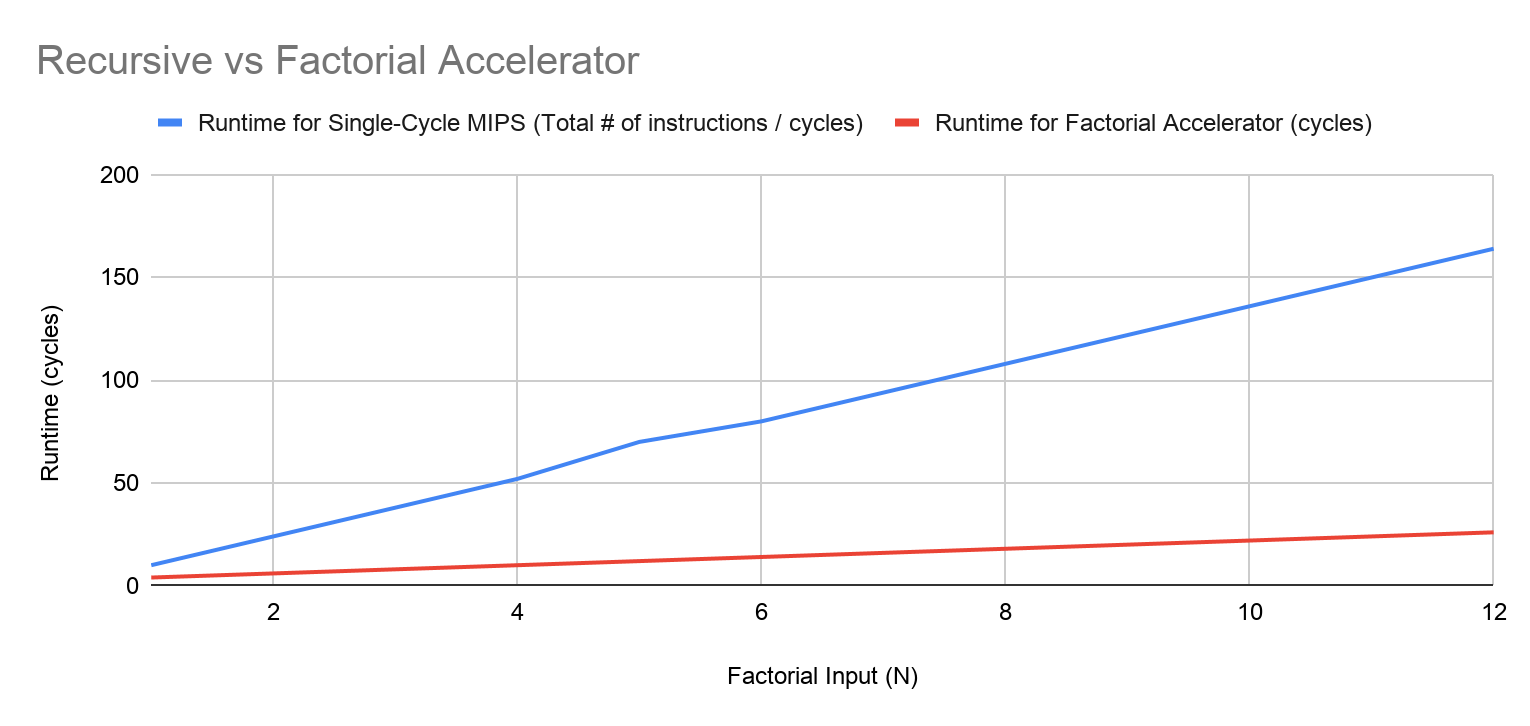


Figure 8. Performance Analysis line chart of *Recursive* vs. *Factorial accelerator*

## 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 |
| Pipelined SoC | End of Appendix J |

### 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.

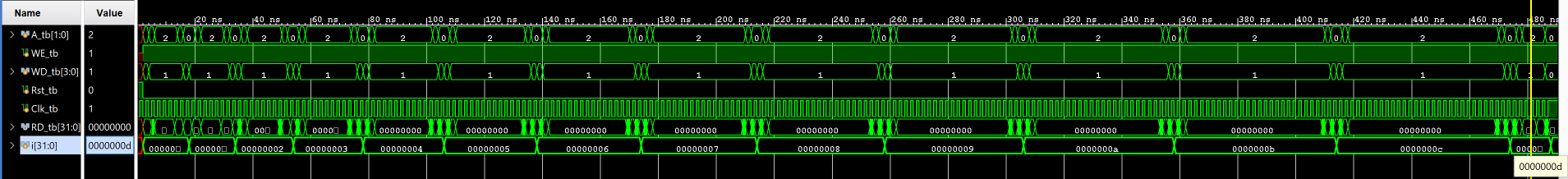


Figure 9: Waveform output for n = 0 through n = 13. Each iteration of n gets larger as more time is needed to calculate n! as n grows larger

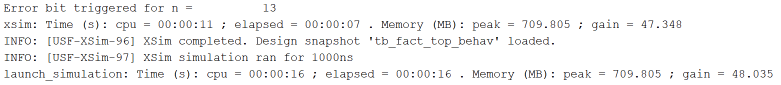


Figure 10: Console output showing the error bit triggered for n > 12

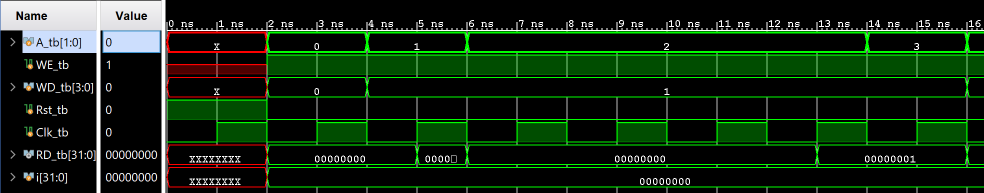


Figure 11: A close up view of n = 0. RD\_tb shows the output of the factorial interface wrapper. It changes as input A (address) changes according to Table 3. When A = 0, input n = 0 is stored internally in the wrapper from WD\_tb. When A = 1, the Go bit is stored in the wrapper from WD\_tb. When A = 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.

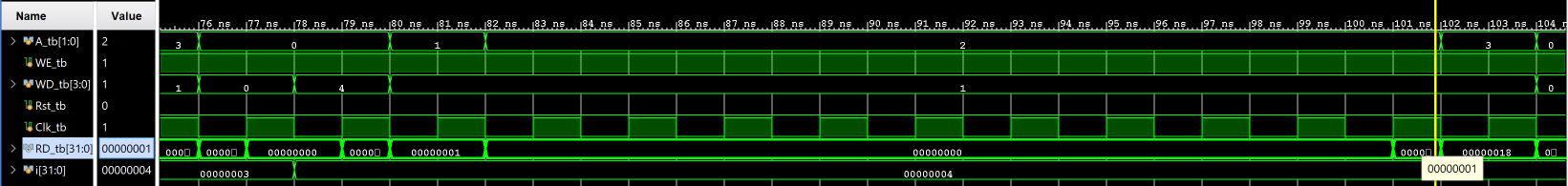


Figure 12: A close up view of n = 4. Same as in Figure 11, RD\_tb shows the output of the factorial interface wrapper. When A = 0, WD\_tb gives n = 4 to the wrapper, which is stored in a register. When A = 1, the Go bit is given to the wrapper, which is stored in a register. When A = 2, the calculation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

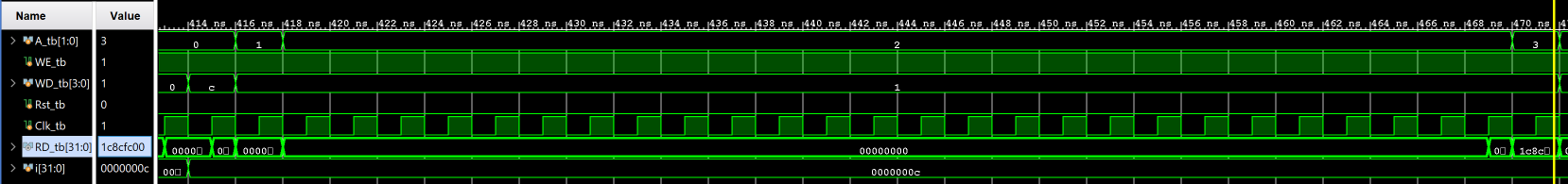


Figure 13: A close up view of n = 12. RD\_tb shows the output of the factorial interface, which changes depending on input A. When A = 0, WD\_tb gives 12 to the wrapper, which is stored in a register. When A = 1, WD\_tb gives the Go bit. When A = 2, the calculation takes place until RD\_tb shows 1 indicating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

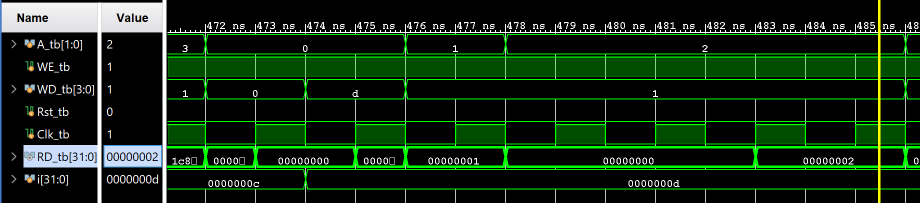


Figure 14: A close up view of n = 13. Note than no value is calculated here since n! is only calculated for n < 13. n = 13 is still loaded and stored in the wrapper when A = 0, and the Go bit is given when A =1. Of note is when A = 2. The calculation kicks off but stops when the error bit is triggered. 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.

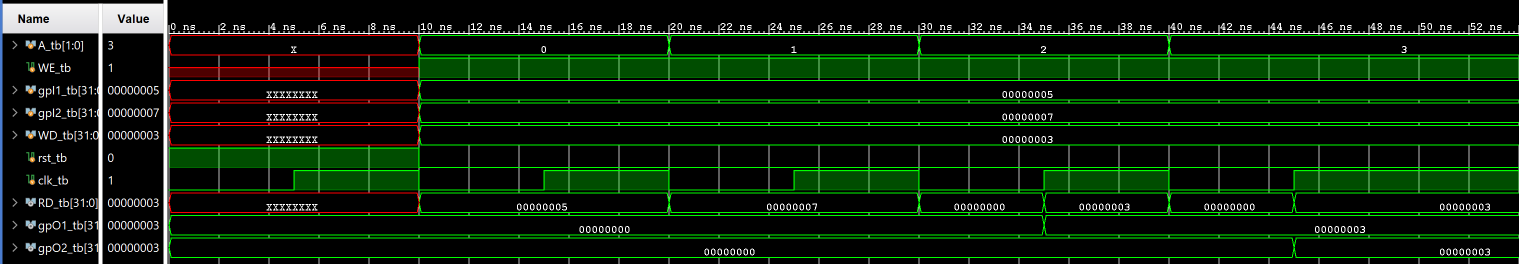


Figure 15: This waveform output shows the result of a testbench for the GPIO unit with interface wrapper. To safely confirm that our design works as intended, RD must output the following: the value of gpI1 when A = 0, the value of gpI2 when A = 1, the value of WD when A = 2, the value of WD when A =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

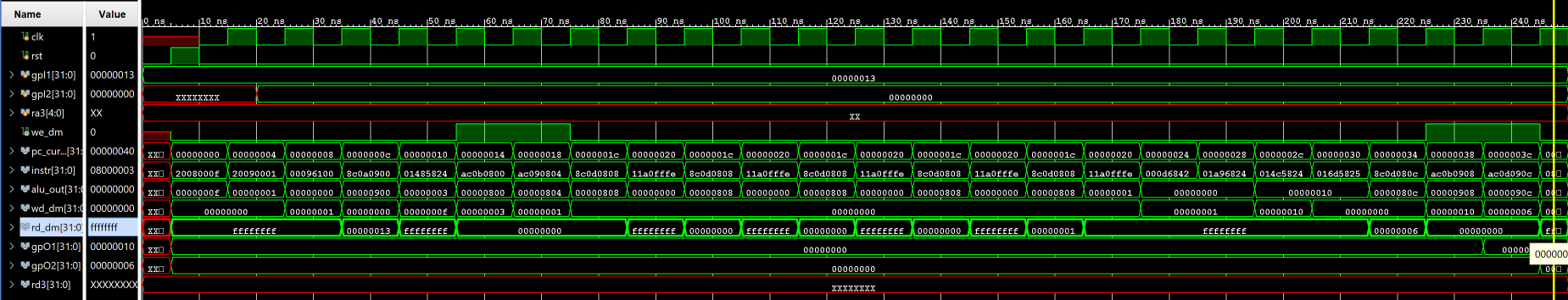


Figure 16: The full screenshot of the single-cycle SoC testbench. Execution continues until the final jump instruction is reached. For this run, input n is given on gpI1. gpI1 = 0x13 = 10011. The rightmost 4 bits encode n while the leftmost bit encodes the display select bit used later in hardware validation. 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.

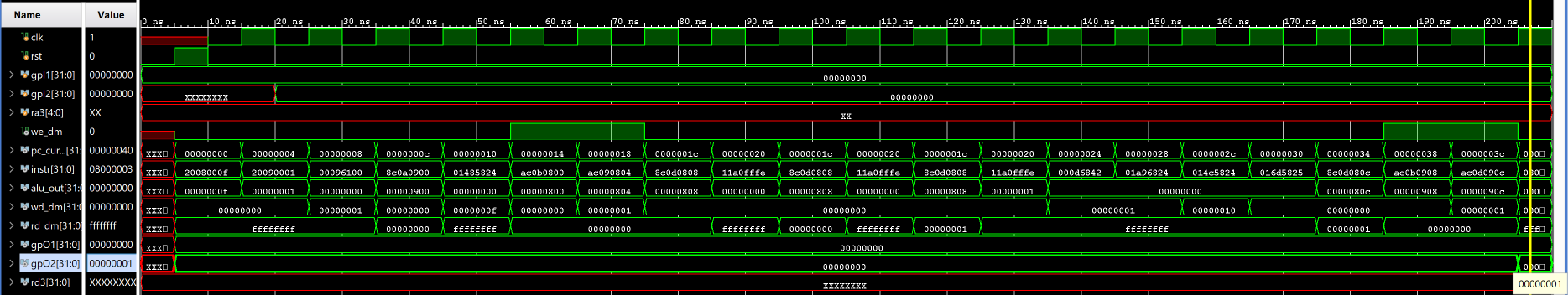


Figure 17: Another example of execution of the driver assembly program with n = 0. The result 0! = 0x1 is shown on gpO2 correctly.

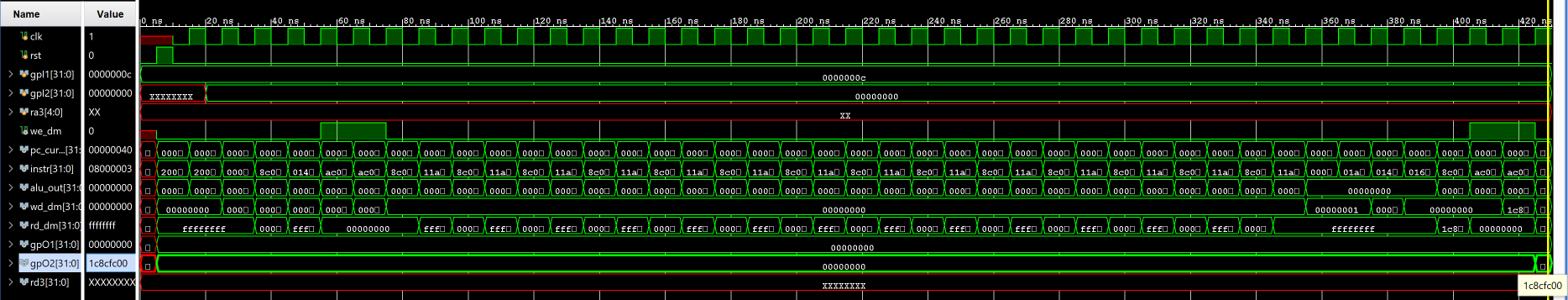


Figure 18: Another example of execution of the driver assembly program with n = 12. The result 12! = 0x1c8cfc00 is shown correctly. Note that program execution is much longer than before because the CPU waits for the factorial accelerator to finish its calculation before continuing program execution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

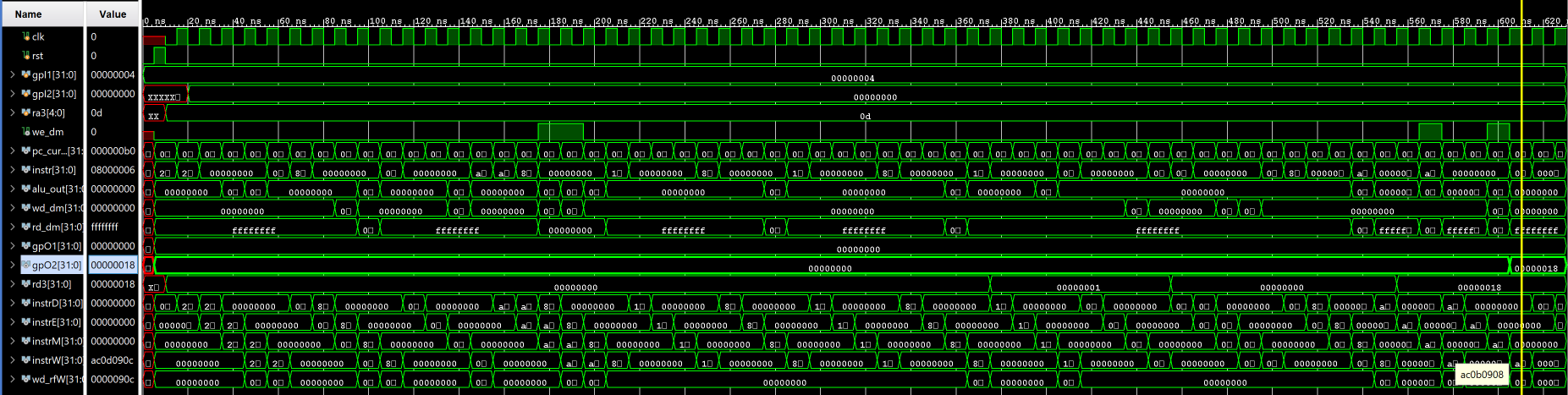


Figure 20: This massive waveform output shows the entire execution process of the pipelined MIPS processor. Of note is the input port gpI1 = 0x4 with n = 4 and the output port gpO2 with n! = 0x18 = 24. This is the correct result of the factorial calculation. The setup for this waveform is similar to 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.

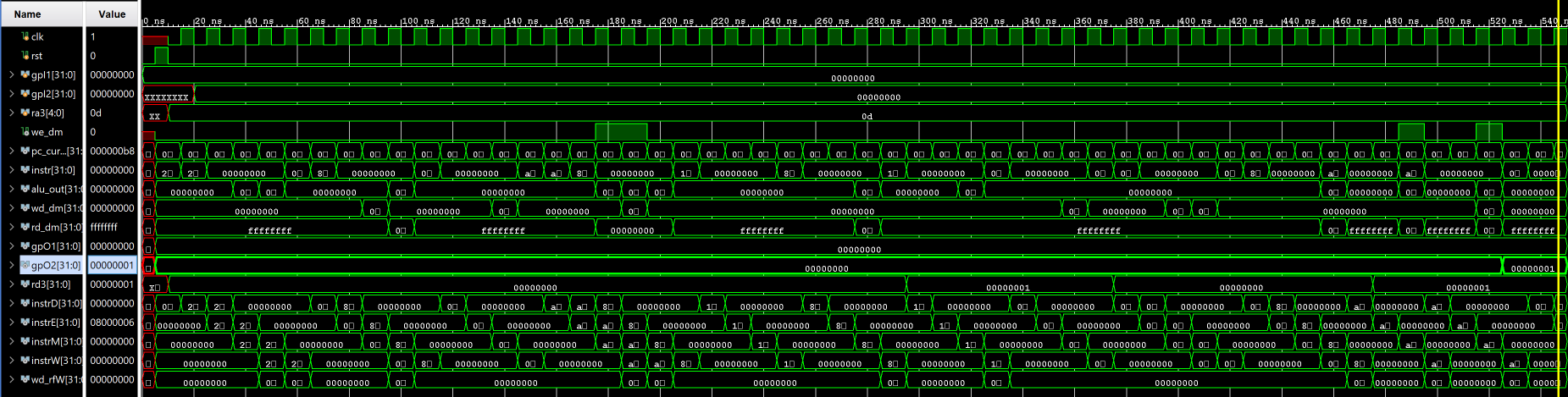


Figure 22: Another example of program execution where n = 0. gpO2 correctly shows the result of n! = 1.

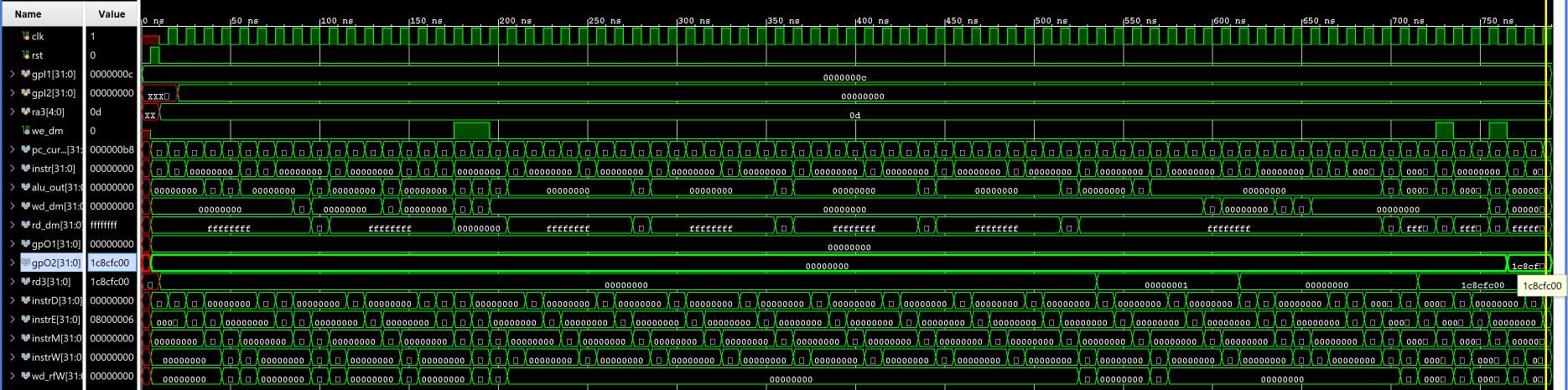


Figure 23: Another example of program execution where n = 12. gpO2 correctly shows the result of n! = 0x1c8cfc00.

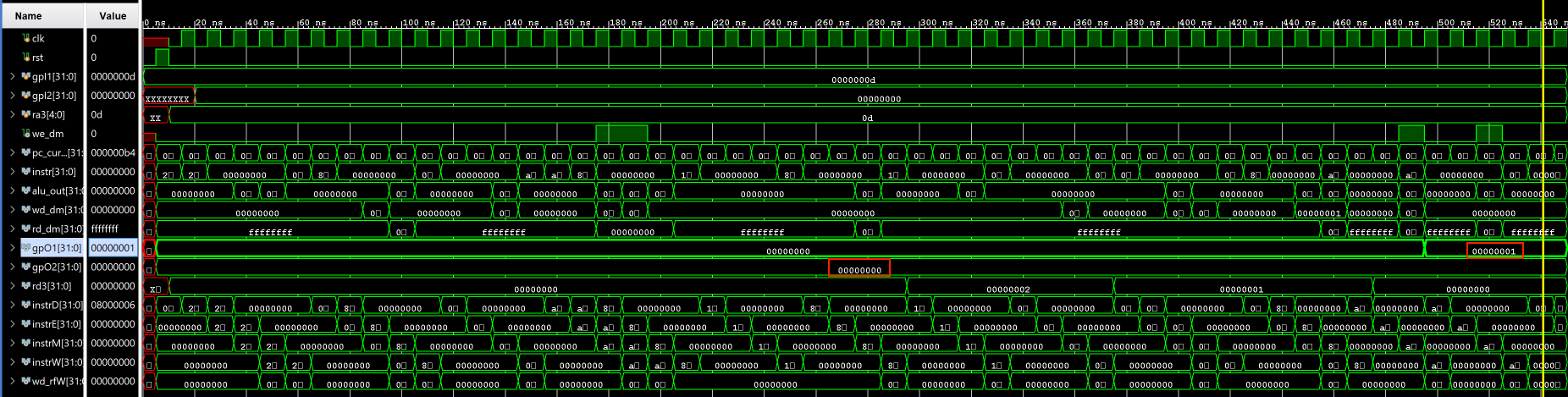


Figure 24: Another example of program execution where n = 13. In this case, no value for n! should be calculated. Instead, gpO2 should be 0 and gpO1 should show the Error bit on the first bit of its output (i.e. gpO1 = 0x1).

## 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. |

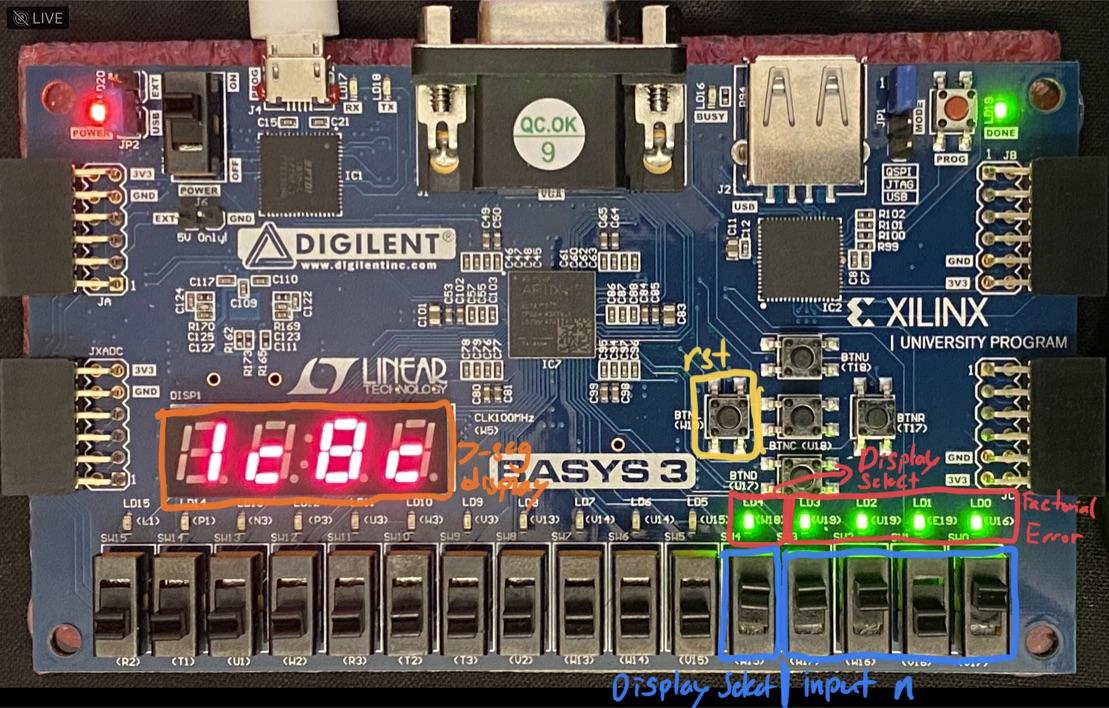


Figure 25: Hardware configuration of the Basys3 board via the constraints file

## 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

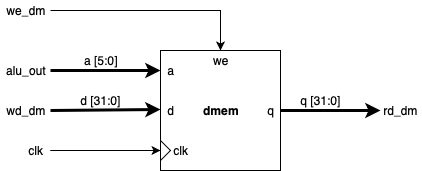


Figure 26: Data memory module used in both single-cycle and pipelined SoC designs



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

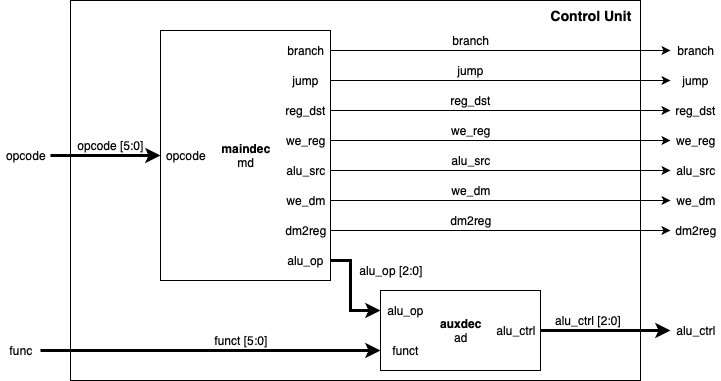


Figure 28: Control unit microarchitecture design used in both single-cycle and pipelined MIPS processor designs. Refer to the corresponding truth tables for each processor in the respective appendices

## 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.

## 

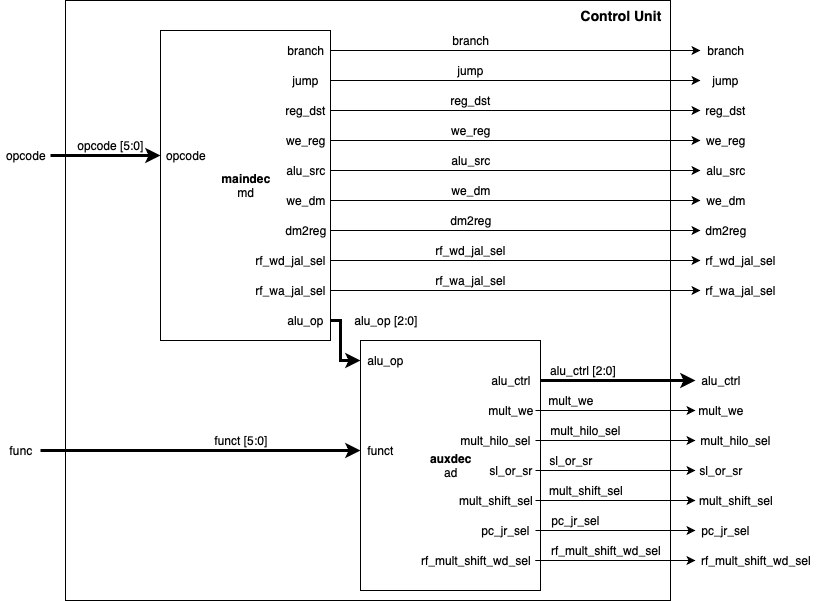


Figure 30: Control unit of single-cycle MIPS Processor

Table 11: Auxiliary Decoder Control Unit Truth Table

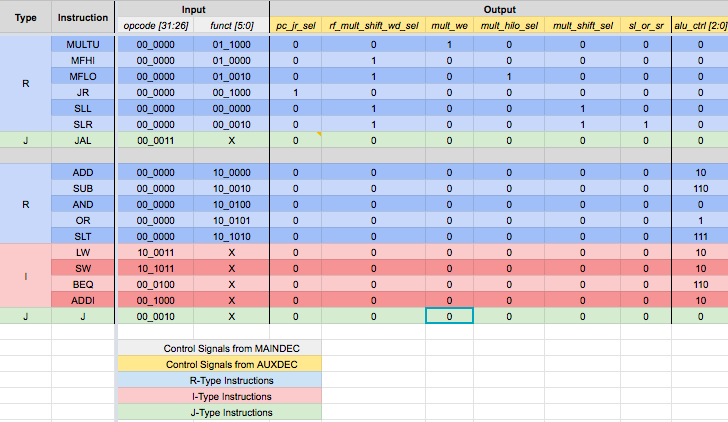
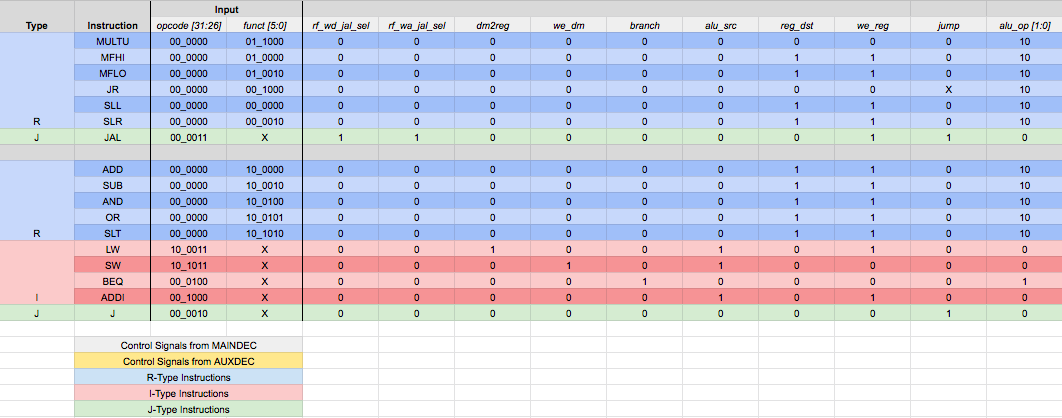


Table 12: Main Decoder Control Unit Truth Table



## 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.

## 

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



Table 14: Decode portion of the truth table

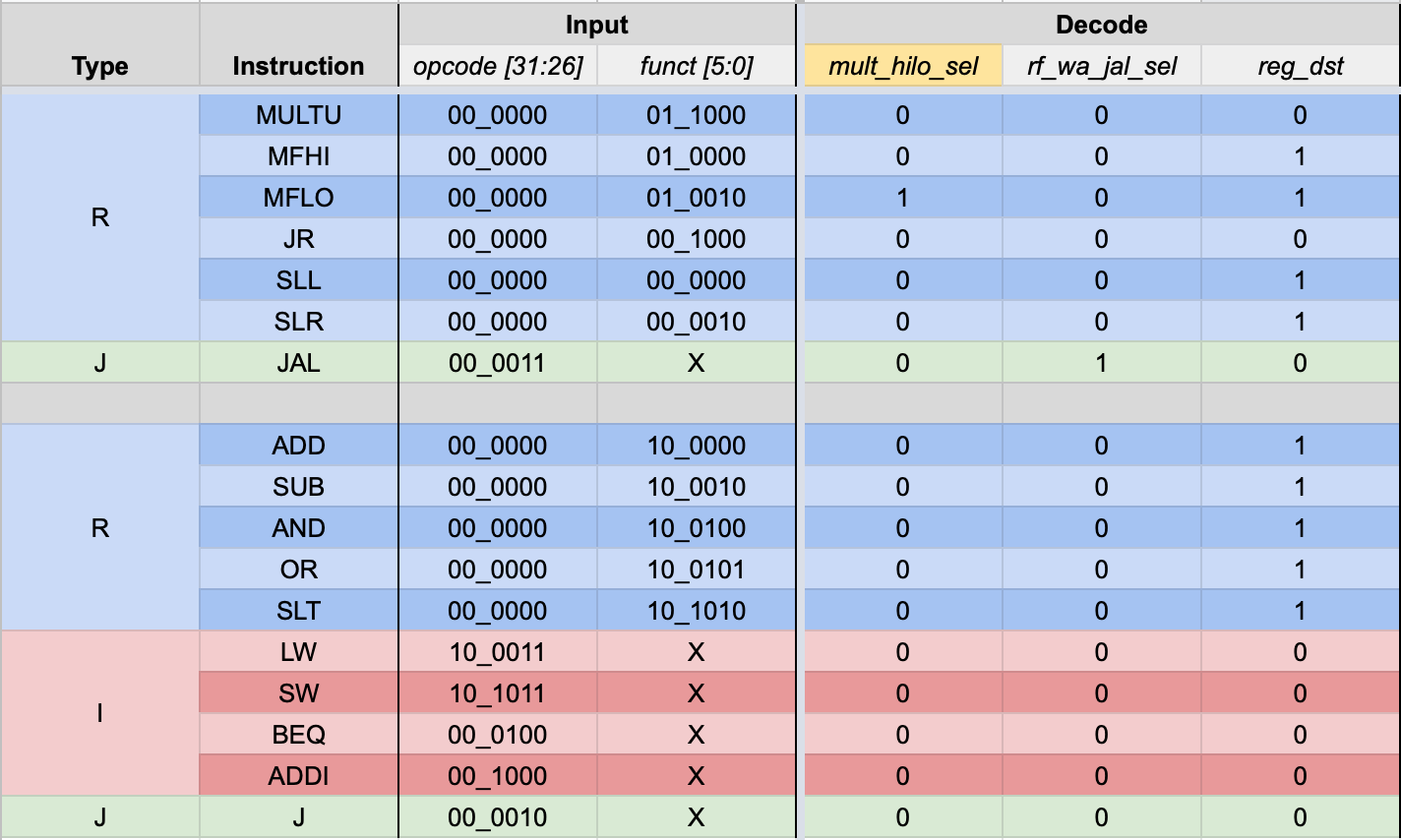


Table 15: Execute portion of the truth table

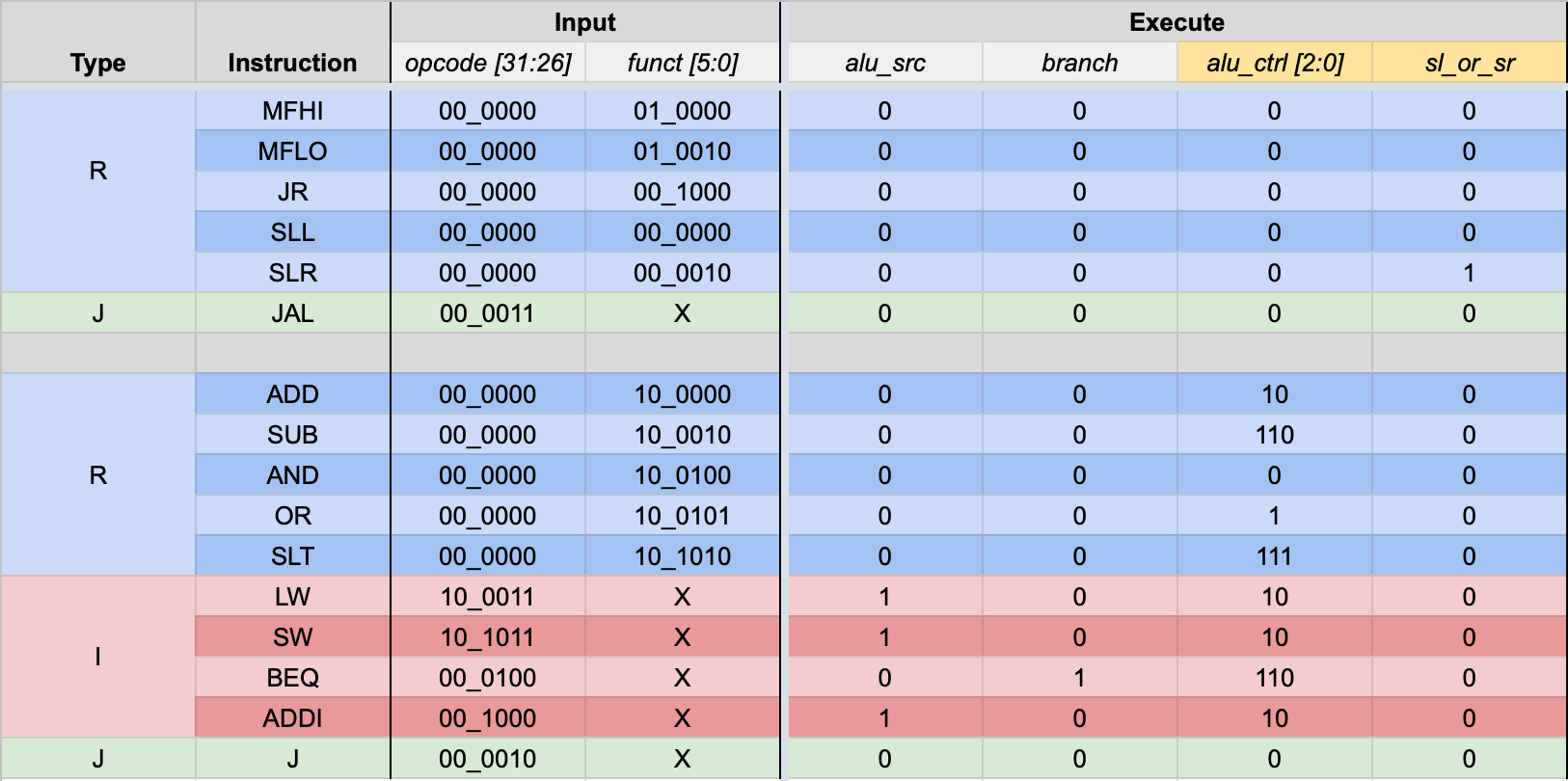


Table 16: Memory portion of the truth table

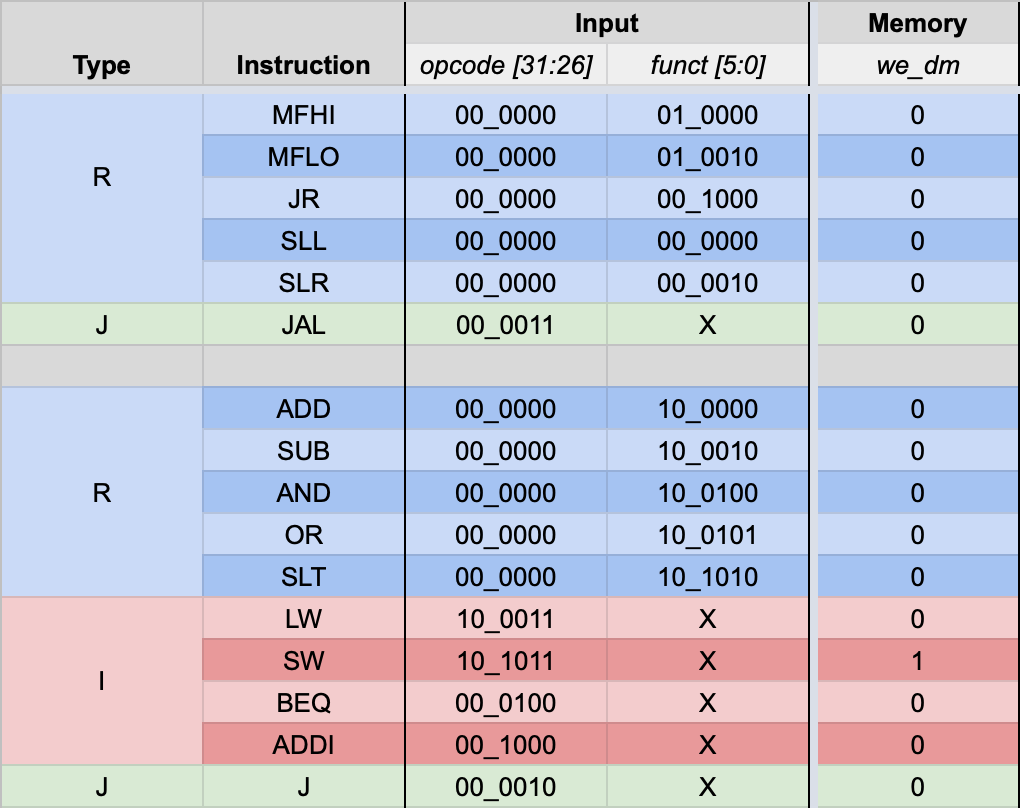
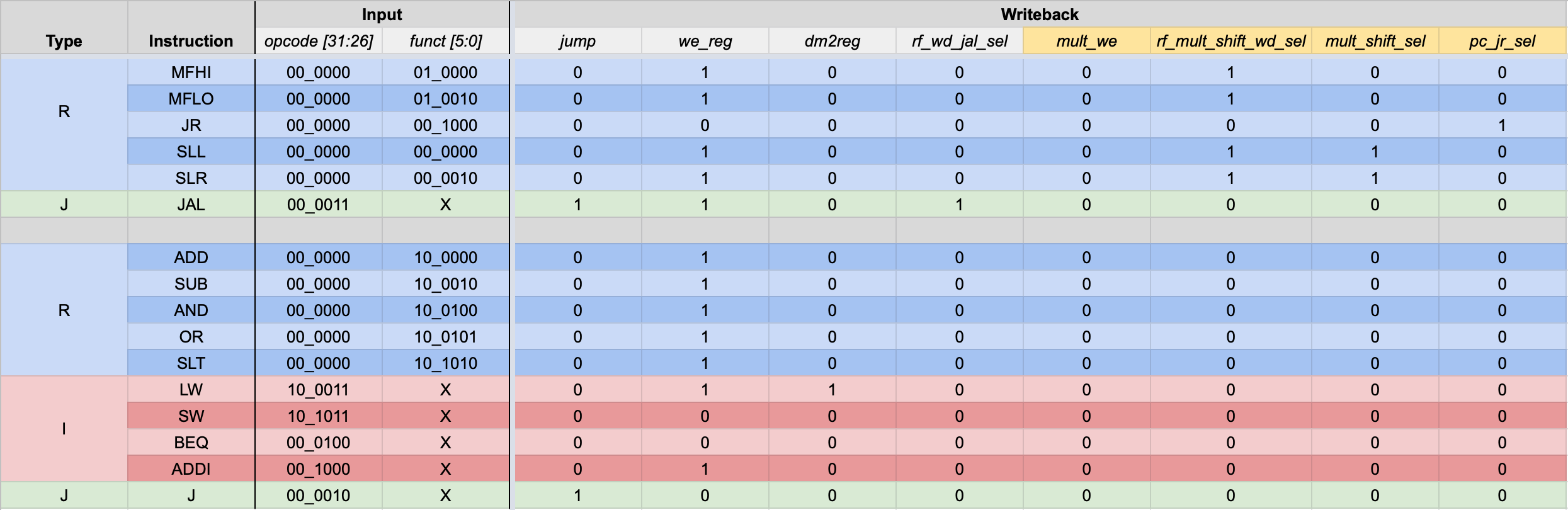


Table 17: Writeback portion of the truth table



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

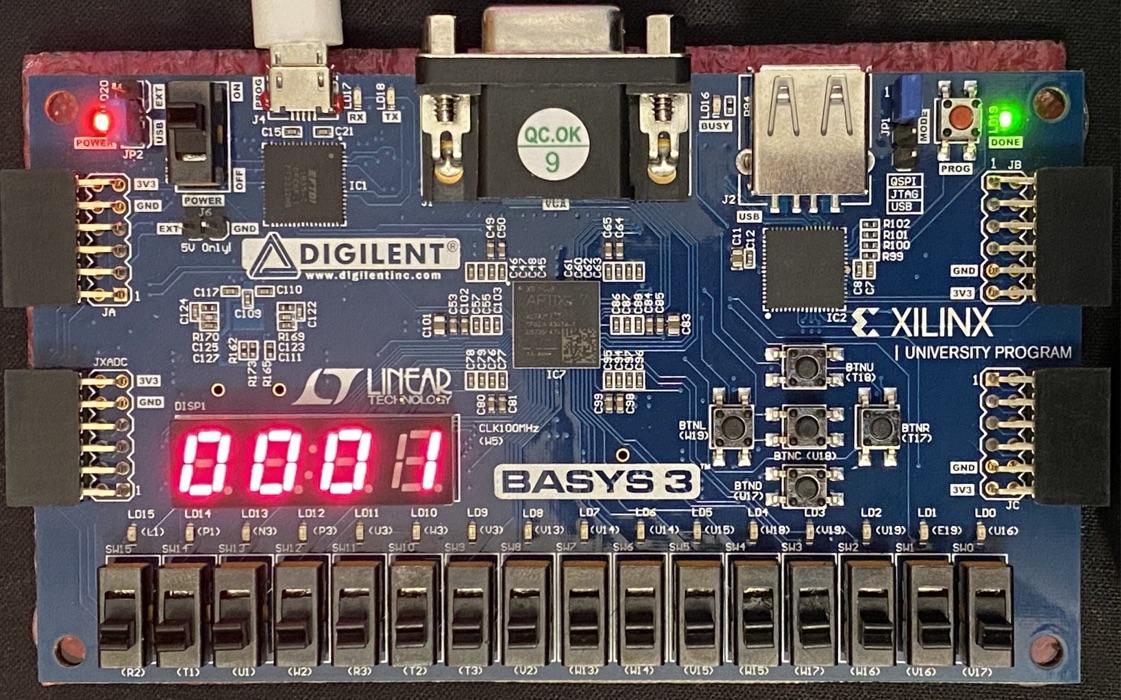


Figure 32: Picture showing the result of input n = 0. 0! = 1 = 0x1as shown on the display.

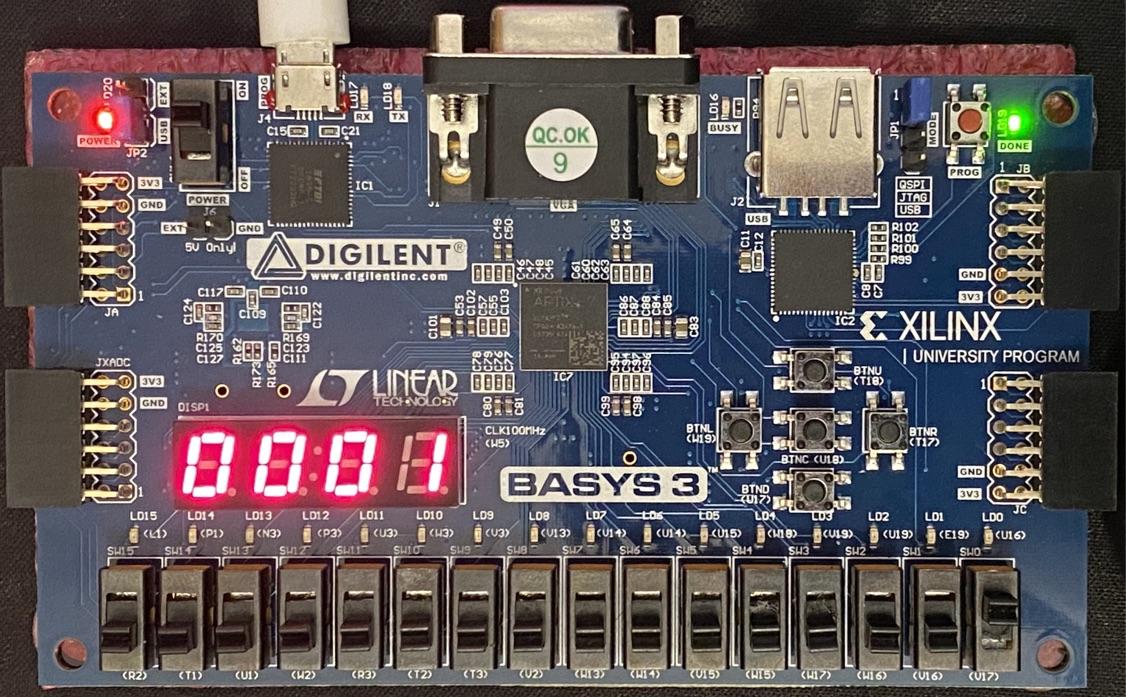


Figure 33: Picture showing the result of input n = 1. 1! = 1 = 0x1 as shown on the display.

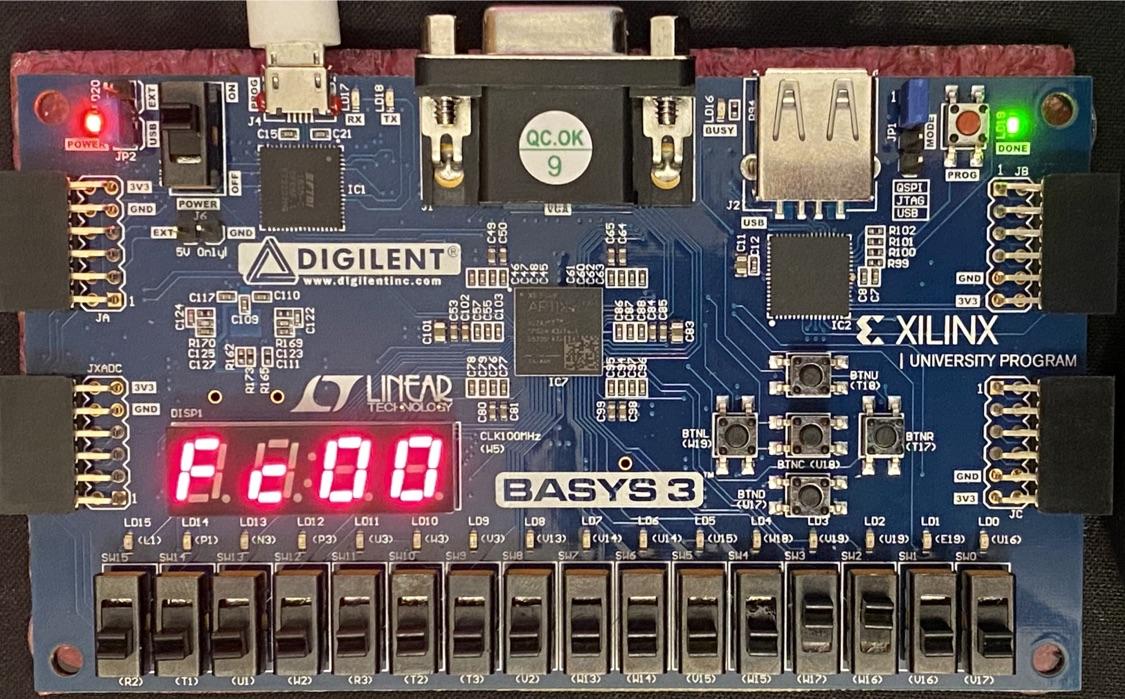


Figure 34: Picture showing the result of input n = 12 with display select = 0. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 0, the display only shows 0xFC00.

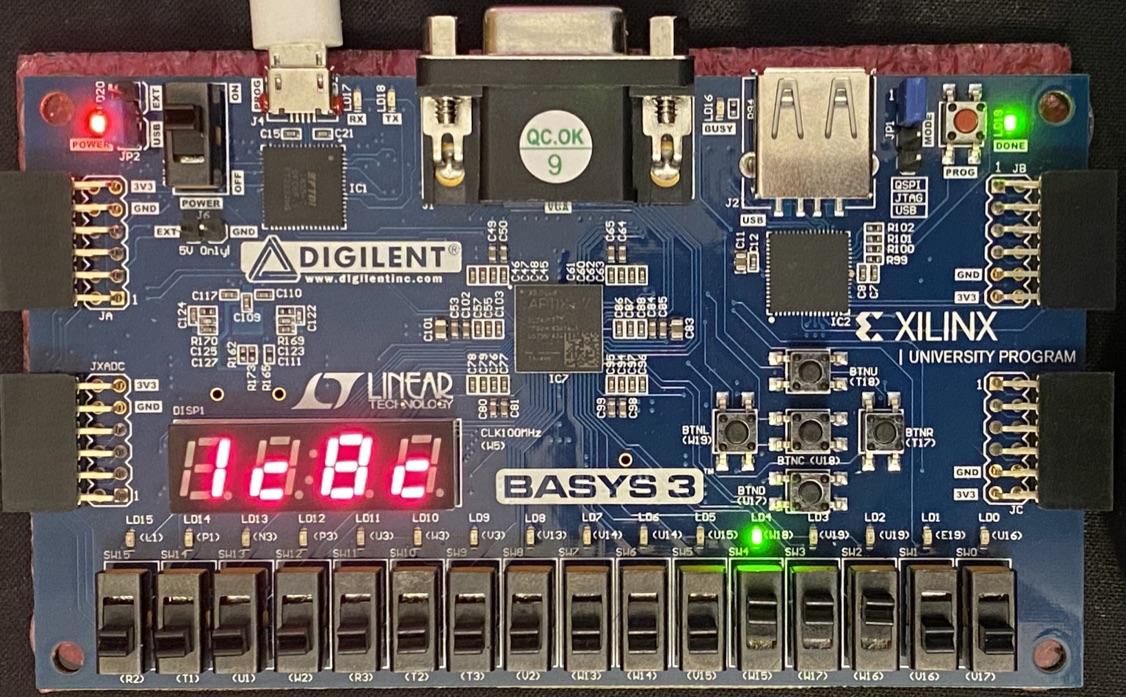


Figure 35: Picture showing the result of input n = 12 with display select = 1. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 1, the display only shows 0x1C8C with the LED above the display select switch lit.

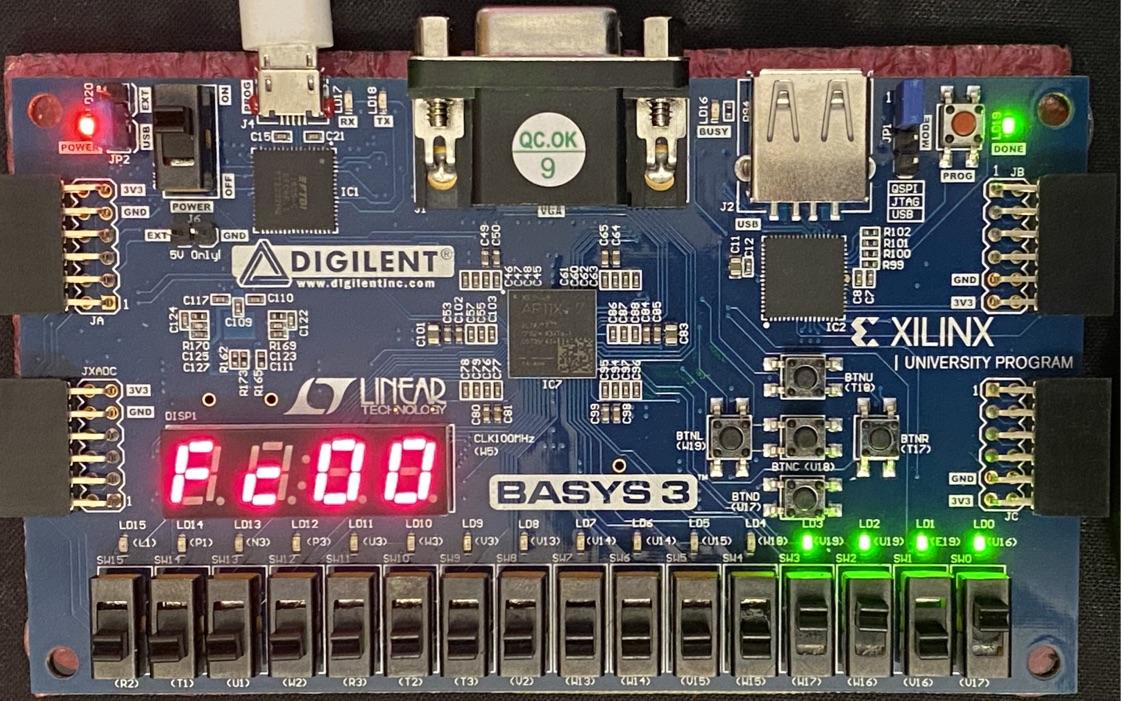


Figure 36: Picture showing the result of input n = 13. Note that n! is not calculated for this input as 13 is greater than 12. Instead, the LEDs connected to FactErr are lit up to indicate an error with the input.

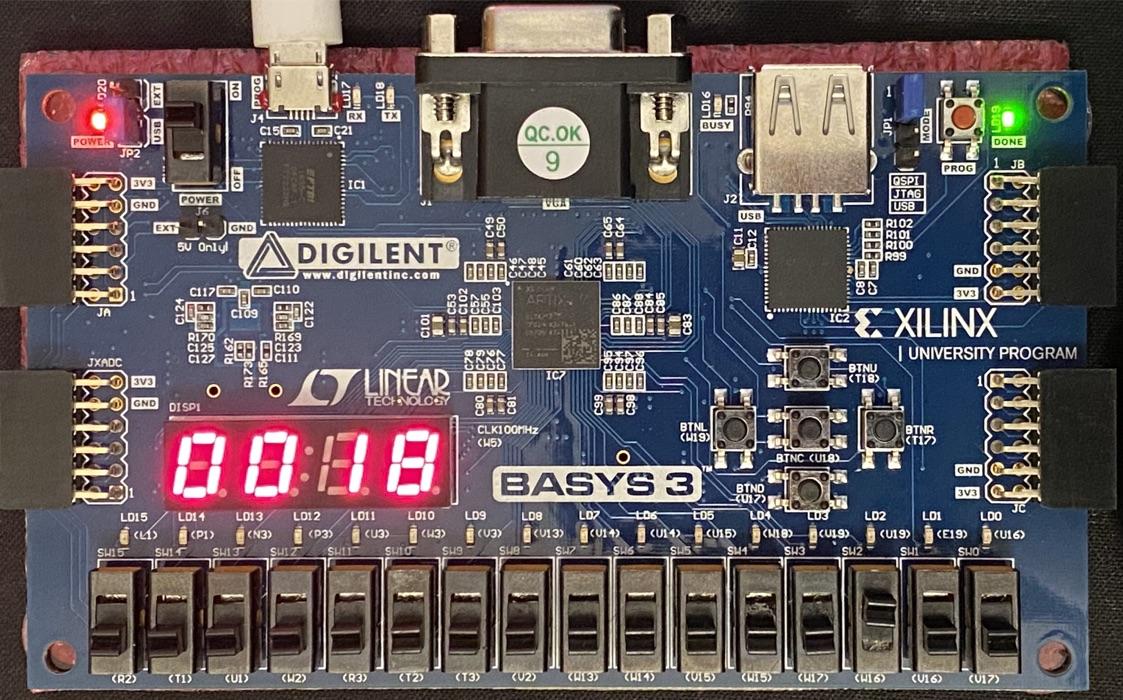


Figure 37: Picture showing the result of input n = 4. 4! = 24 = 0x18 as shown on the display

## Appendix E: Pipelined MIPS Processor in SoC Hardware Validation

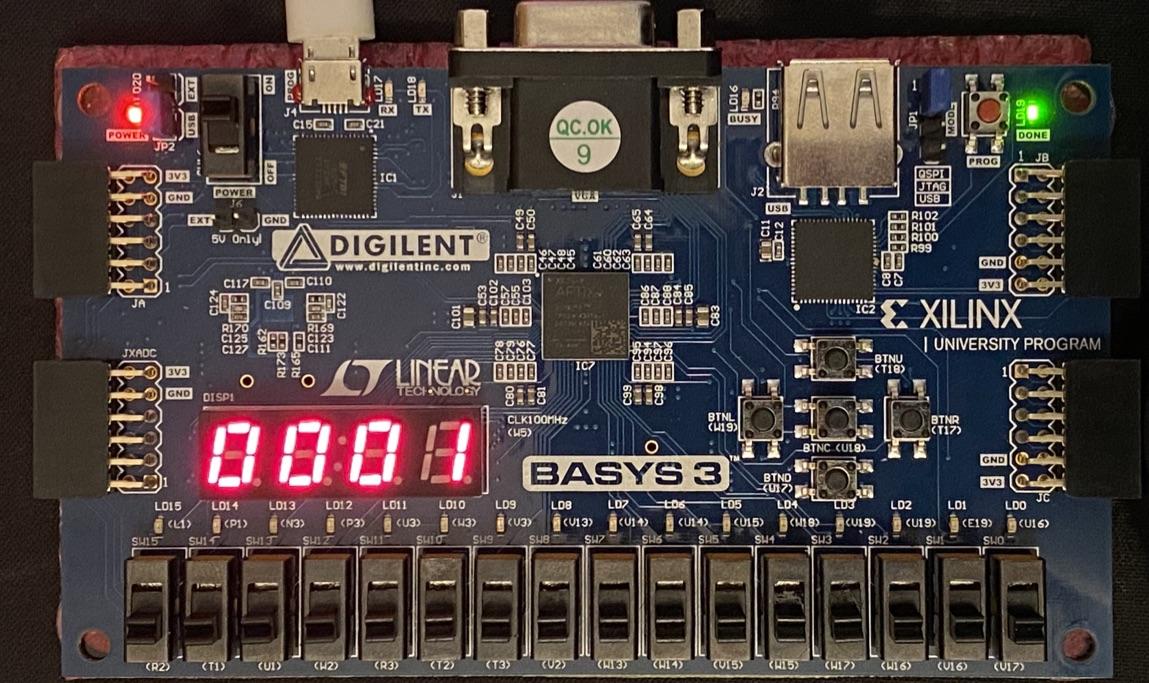


Figure 38: Picture showing the result of input n = 0. 0! = 1 = 0x1 as shown on the display

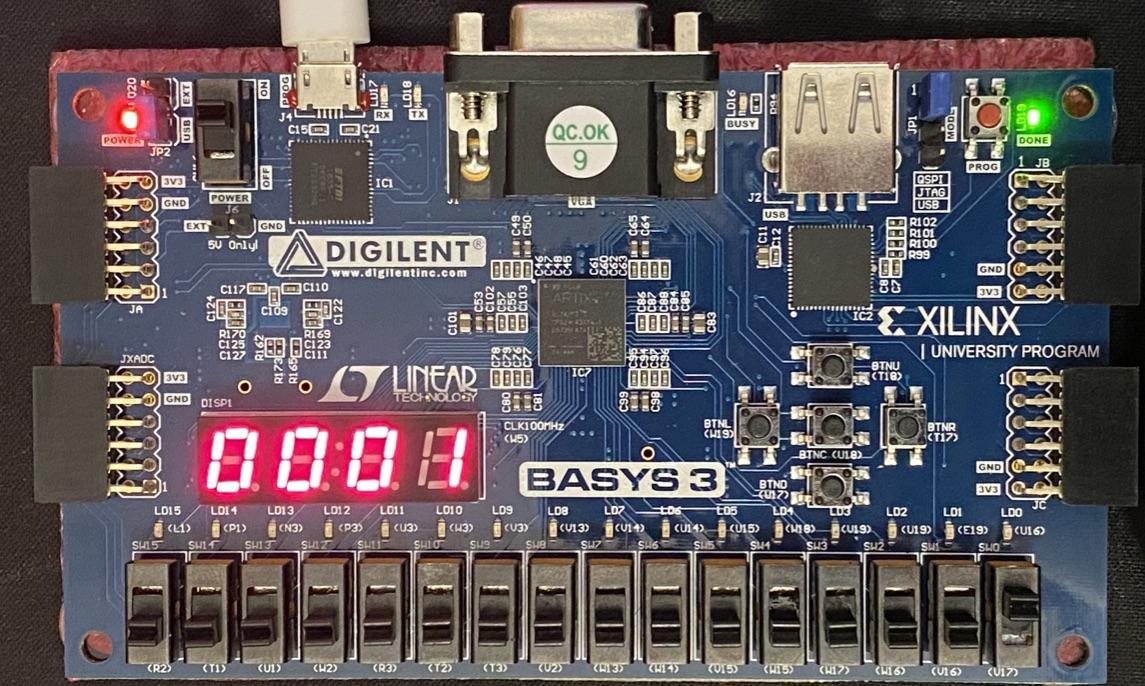


Figure 39: Picture showing the result of input n = 1. 1! = 1 = 0x1 as shown on the display

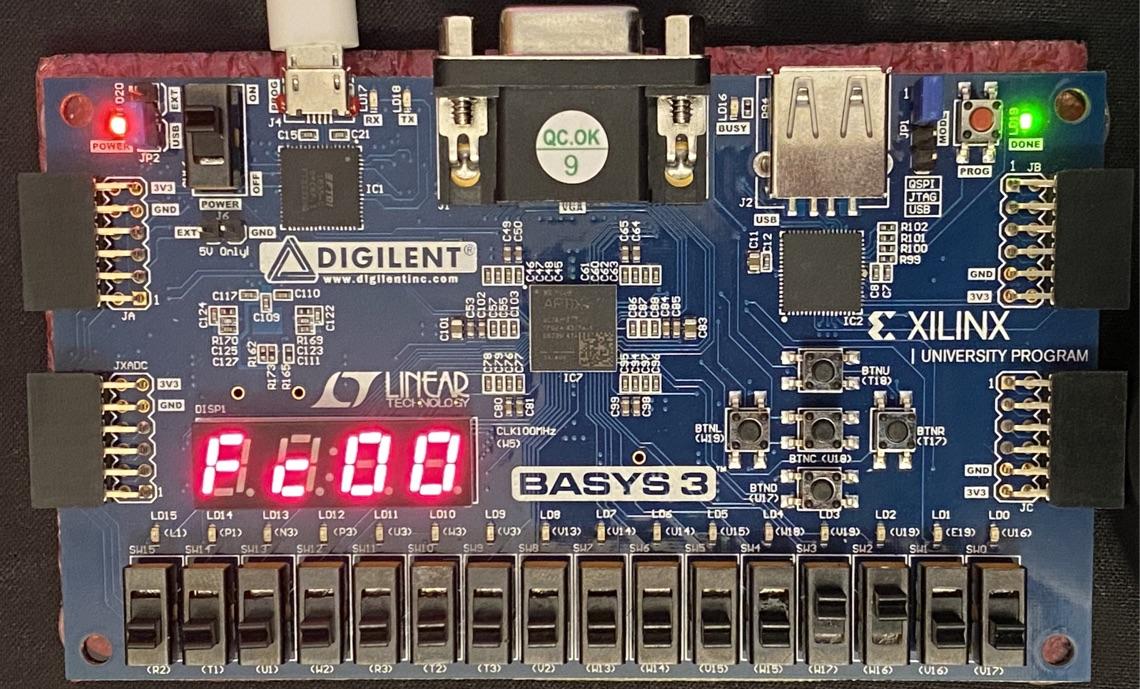


Figure 40: Picture showing the result of input n = 12 with display select = 0. 12! = 479,001,600 = 0x1C8CFC00. Since the display select is 0, only 0xFC00 is shown on the display.

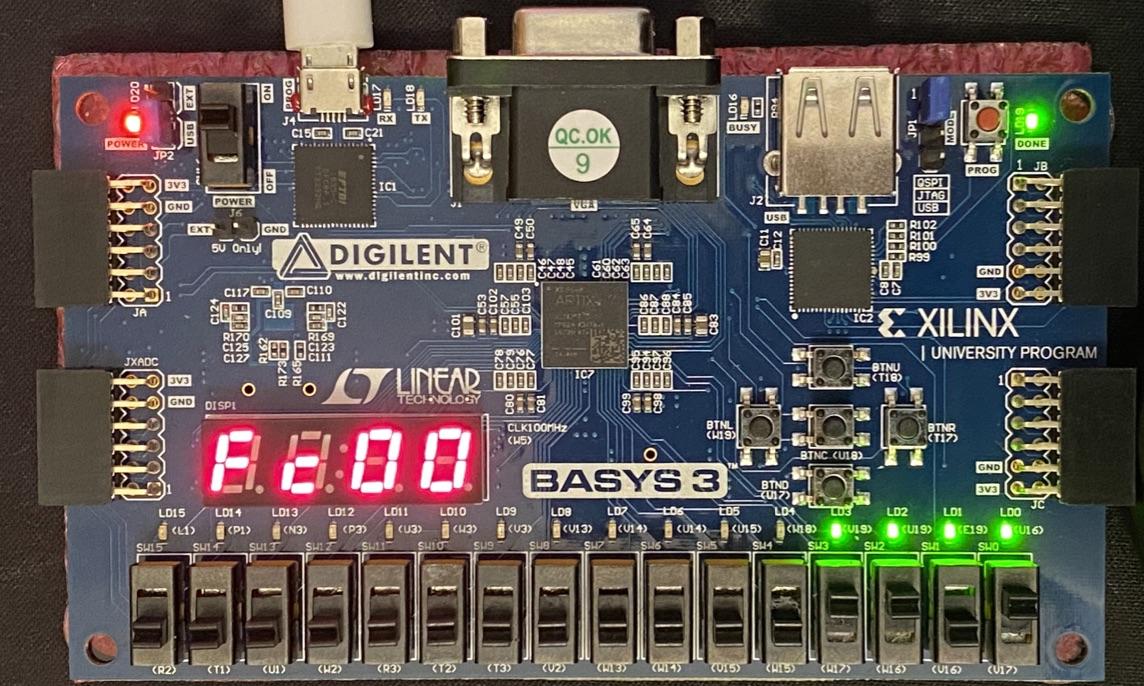


Figure 41: Picture showing the result of input n = 13. Since 13 > 12, the factorial accelerator does not calculate 13!. Instead, the factorial error bit lights up the shown LEDs to indicate an error with the input.

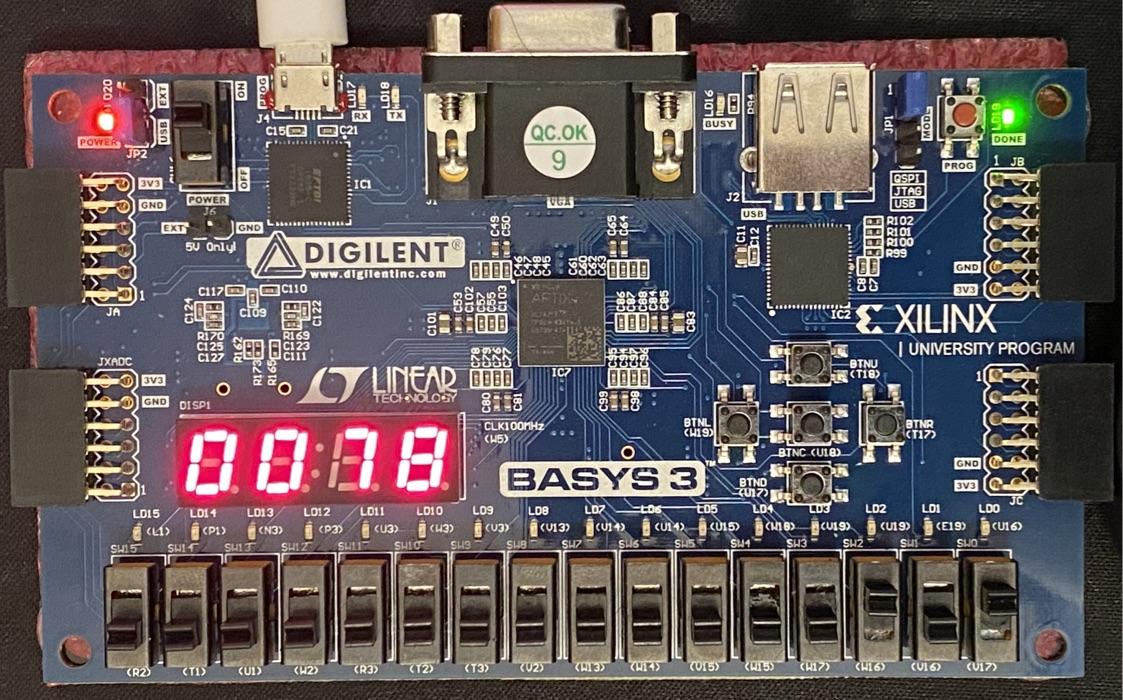


Figure 42: Picture showing the result of input n = 5. 5! = 120 = 0x78 as shown on the display

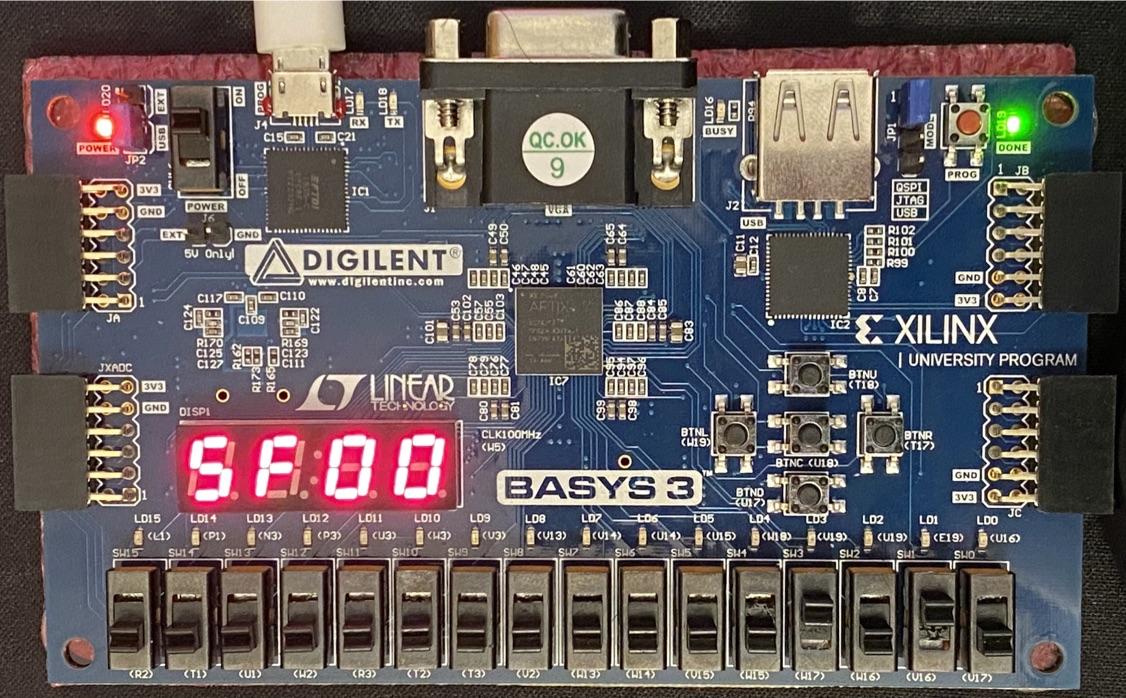


Figure 43: Picture showing the result of input n = 10 with display select = 0. 10! = 3,628,800 = 0x00375F00. Since the display select is 0, the display only shows 0x5F00.

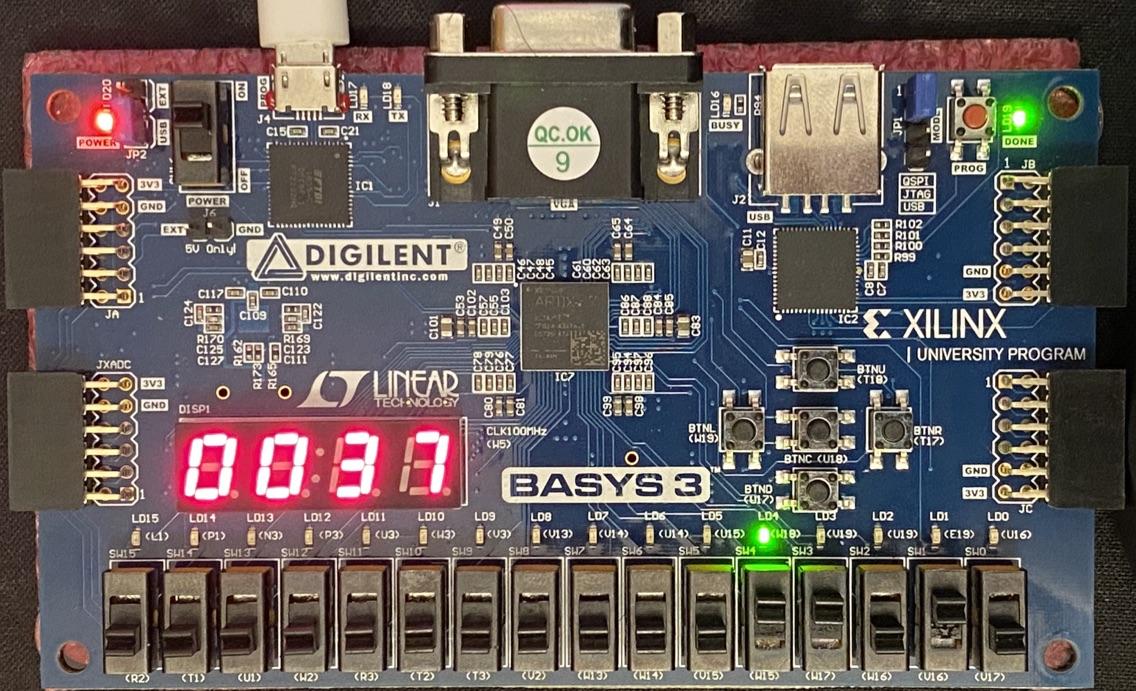


Figure 44: Picture showing the result of input n = 10 with display select = 1. 10! = 3,628,800 = 0x00375F00. Since the display select is 1, the display only shows 0x0037 with the LED above the display select switch lit.

## 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 |

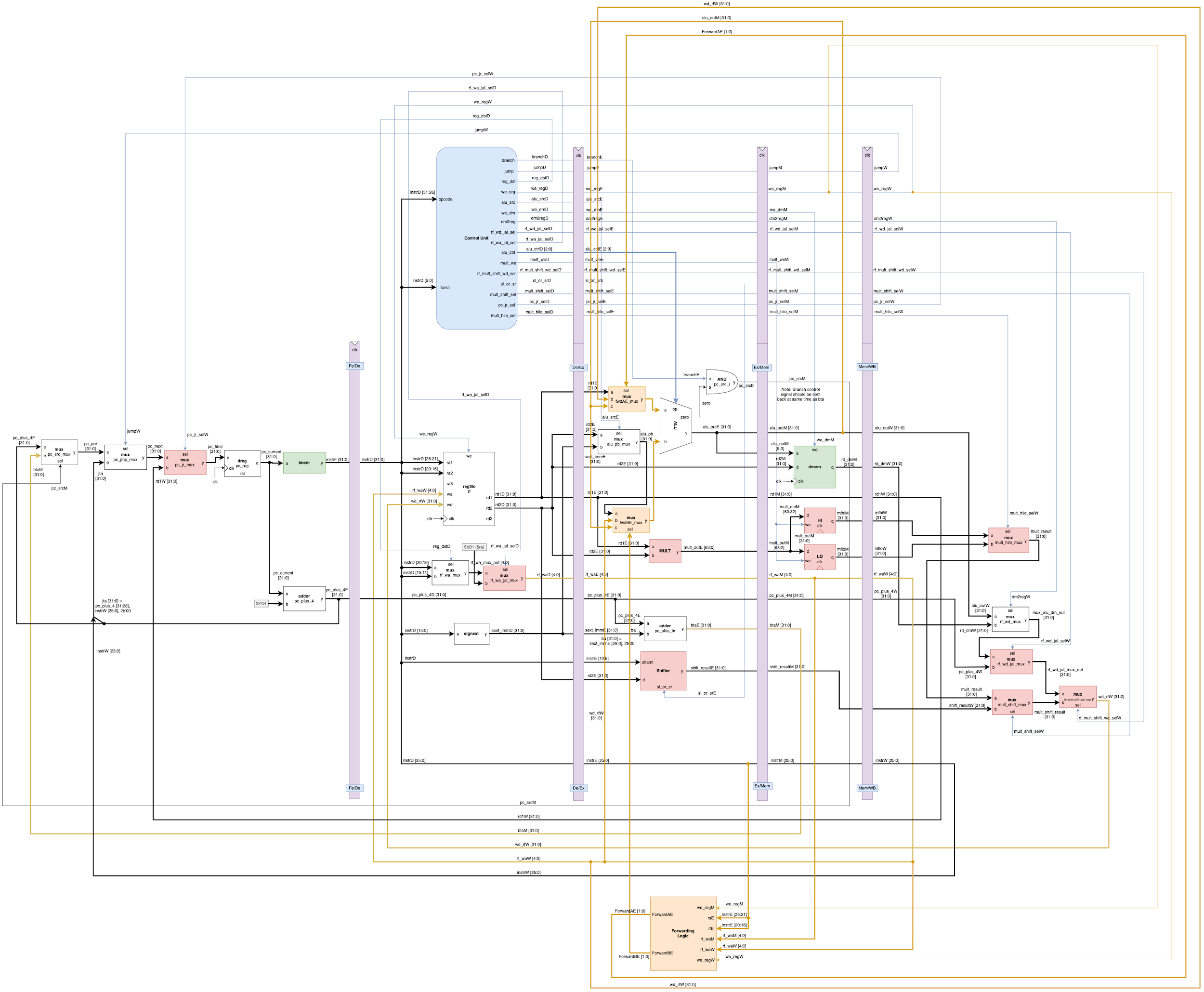
|  |
| --- |
| **fpga\_top.xdc** |
| # Clock Signal  set\_property -dict {PACKAGE\_PIN W5 IOSTANDARD LVCMOS33} [get\_ports {clk}];  create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clk}];  # Buttons  set\_property -dict {PACKAGE\_PIN U18 IOSTANDARD LVCMOS33} [get\_ports {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



## 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.