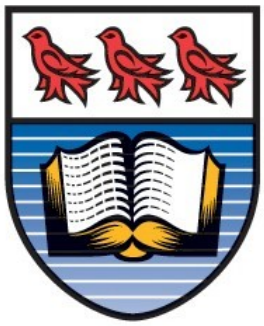
University of Victoria

Electrical and Computer Engineering



CENG 450

Computer Systems and Architecture

Spring 2018

Lab Project Report

Morgan Williams - V00804732
Brosnan Yuen - V00799522

Submitted: January 24, 2024

Abstract

Contents

List of Figures

List of Tables

1 Introduction

The purpose of this lab project was to design and implement a 16-bit CPU for a specified instruction set architecture using a Xilinx Spartan-3E family FPGA. The CPU design is to be implemented by programming the FPGA in VHDL using the Xilinx ISE. The project was introduced January 24th and was to be completed by April 5th 2018.

The specified 16-bit instruction set architecture is comprised of formats A, B, and L. Format A includes arithmetic, logical, as well as, some I/O command instructions to be completed by the ALU. Format B consists of primarily branch instructions but also includes the return instruction. Branches instructions are categorized as relative (BRR), absolute (BR), conditional (BR.(Z or N)), and subroutine (BR.SUB). Format L consists of register and memory instructions like MOV, LOAD, STORE.

2 Considerations

Given the complexity of this project, it is important to outline the criteria and challenges to convey the design approach. This section covers design requirements, project time line, and a brief discussion of the inherent design challenges.

2.1 Design Requirements

The goal of this project was to implement a 16-bit pipelined CPU capable of running the provided 16-bit instruction set architecture (ISA). The project was to be completed according to the project time line outlined in the eponymous section. The basic CPU schematic is shown in Figure 1. The optional implementation of interrupt handling was not implemented in the final design.

The 16-bit ISA is composed of Formats A, B, and L. Format A is composed of primarily of arithmetic operations. Format B consists of branching operations and format L contains all register and memory operations. ISA specifications can be found in the Appendix. Pipeline hazards are resolved using hardware techniques instead of compilation methods.

Final CPU designs are tested using the provided format tests, as well as, 3 final test codes that are traditional programs employing all instruction formats.

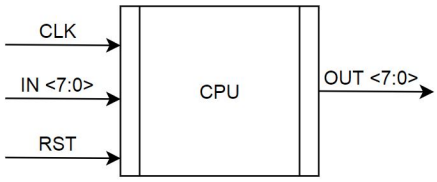


Figure 1: CPU System Schematic.

2.2 Project Time Line

1. Preliminary Design Review: February 21, 2018
    - High Level Block Design
    - Format A Instructions
2. Format B Instructions: March 7, 2018
3. Format L Instructions: March 28, 2018
4. Final Design Review and Project Demonstration: April 5, 2018

2.3 Pipeline Hazards

Pipeline hazards are the most difficult design challenge when implementing a pipelined CPU. Hazards are manifested by the pipeline operation itself. Managing pipeline hazards is incumbent when designing a pipelined CPU. These hazards can be categorized into three subsets: structural, control, and data.

2.3.1 Structural Hazards

Pipeline structural hazards occur when 2 or more separate instructions attempt to access the same hardware component simultaneously. In the example seen in Table. 1, a structural hazard occurs as instruction i attempts to access memory at the same time as $i + 3$ attempts to fetch an instruction from memory. This same hazard occurs between $i + 1$ and $i + 4$.

	Clock Cycle Number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
i	IF	D	EX	M	WB				
$i + 1$		IF	D	EX	M	WB			
$i + 2$			IF	D	EX	M	WB		
$i + 3$				IF	D	EX	M	WB	
$i + 4$					IF	D	EX	M	WB

Table 1: Pipeline Structural Hazard Example.

Structural hazards do not exist within this CPU design because the components used in the fetch and memory access stages of the pipeline are unique. In this CPU design, a ROM component stores the program and a RAM module is used as the memory storage solution. A popular method of circumventing this hazard is to employ a dual-ported memory module.

2.3.2 Control Hazards

Control hazards manifest from pipelined branching operations and other operations that change the program counter. The easiest method of handling pipelined branching operations is to implement a stalling mechanism. The purpose of this stalling or "bubbling" mechanism is to permit the resolution of the conditional branch so that the new computed destination can be loaded into the program counter. Table. 2 illustrates a stalling mechanism. As a branch instruction is decoded, the successive instruction is stalled and the branch instruction carries through the pipeline to be resolved. Once the branch is resolved, the PC counter is updated and the pipeline continues.

	Clock Cycle Number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
Branch	IF	**D**	EX	M	WB					
Branch + 1		**Stall**	**Stall**	**Stall**	IF	D	EX	M	WB	
Branch + 2						IF	D	EX	M	WB

Table 2: Pipeline Control Hazard Example.

2.3.3 Data Hazards

Pipeline data hazards occur as dependent instruction results are not executed with correct timing to satisfy data dependencies due to the nature of pipelining. This hazard manifests in a number of scenarios, in particular with read after write (RAW) errors. Table. 3 illustrates a RAW error as the highlighted decode stage will not deliver the correct ***R1*** into the execute stage, which will result in an erroneous value to be stored in ***R3***.

	Clock Cycle Number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
R1 <= R1+R2	IF	D	EX	M	WB					
R3 <= R1 + R2		IF	D	EX	M	WB				

Table 3: Pipeline Data Hazard Example.

A method used to solve this common RAW scenario is operand forwarding. This technique derives from the observation that the newly computed value for ***R1*** is available following the execution stage. Control logic detects this hazard by checking if the previous instruction's source destination is the same as a current instruction's source. If this is true, then the previous instruction's execution result is fed back into the current instruction's execute stage, maintaining data integrity. Additional logic is required to select the correct forwarding operand in order to maintain data integrity.

3 Pipeline Components

The pipelined CPU is composed of a number of components that accomplish each stage's function. This section does not cover stage specific controllers and multiplexers, and rather focuses on the functional units.

3.1 ROM Module

The CPU makes use of a ROM module that stores all program instructions. By using a unique ROM module instead of a traditional singular memory unit, all structural hazards are circumvented. Complicated dual-ported memory configurations are also avoided, at the expense of additional hardware. It should be noted that this memory configuration is not the modern approach and is has only been implemented due to its convenience.

The ROM module is found in the fetch stage of the pipeline. The ROM receives an address from the program counter and outputs the corresponding instruction. The ROM array is byte addressable and 2 KB in size.

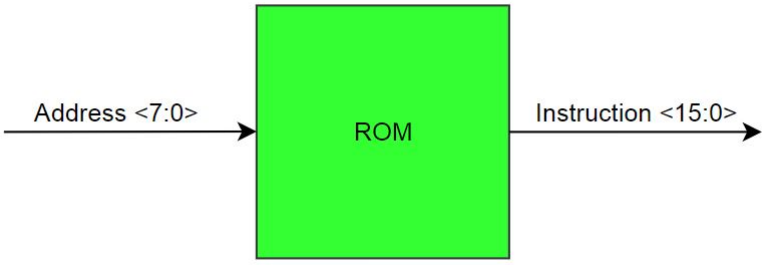


Figure 2: ROM Diagram.

3.2 Program Counter

3.3 Register File

3.4 ALU

3.5 RAM

4 Pipeline

4.1 Control Unit

The Control unit is a finite state machine (FSM) that interacts with all pipeline stages. The FSM is controlled using the previous state and a signal from the decode stage. The Control Unit FSM consists of several internal states and 4 output states. The internal states govern the sequence and selection of output state signals. The primary function of the Control Unit is to insert "bubbles" to enable branch handling.

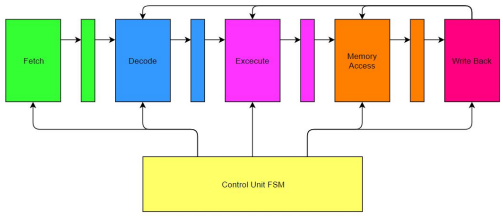


Figure 3: CPU System Diagram.

The flow chart seen in Figure. 4 illustrates the FSM internal state sequence. Following a Control Unit RESET state, a RUN state is always engaged. The RUN state is only disengaged when

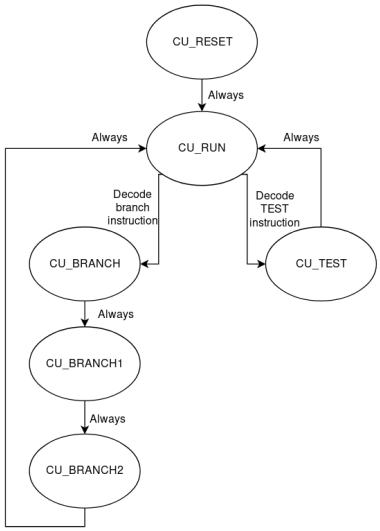


Figure 4: Control Unit FSM Flow Chart.

The following tables 4 and 5 illustrate the resulting Control Unit outputs to the pipeline stages given the Control Unit internal states seen in Figure 4:

	Control Unit State		
Pipeline Stage	*RESET*	*RUN*	*TEST*
Fetch	Reset	Run	Reset
Decode	Reset	Run	Stall
Execute	Reset	Run	Run
Memory	Reset	Run	Run
Write Back	Reset	Run	Run
Next State	*RUN*	*RUN/TEST/BRANCH*	*RUN*

Table 4: Control Unit Outputs to Pipeline Stages.

	Control Unit State		
Pipeline Stage	*BRANCH*	*BRANCH 1*	*BRANCH 2*
Fetch	Stall	Stall	Reset
Decode	Reset	Reset	Write PC
Execute	Run	Stall	Stall
Memory	Run	Run	Stall
Write Back	Run	Run	Run
Next State	*BRANCH 1*	*BRANCH 2*	*RUN*

Table 5: Control Unit Outputs to Pipeline Stages cont'd.

Table. 6 shows all possible states of the stage FSM and the interaction with the RAM and Register File, Program Counter value, and stage output. When the RESET state is asserted, all data in all the stages is cleared and the PC value is set to zero. The RUN state operates the stages normally. The STALL state retains the current values inside the stages, as well as, their outputs. The WRITE PC state acts identically to STALL, except that the PC value is updated.

	State Stage			
Component	*RESET*	*RUN*	*STALL*	*WRITE PC*
Mem. & Reg.	Set to 0.	Run normally	Keep current value	Keep current value
PC Value	Set to 0.	Run normally	Keep current value	Update PC if branch taken
Stage Output	Set to 0.	Run normally	Keep current value	Output previous value

Table 6: Control Unit FSM State Table

4.2 Fetch

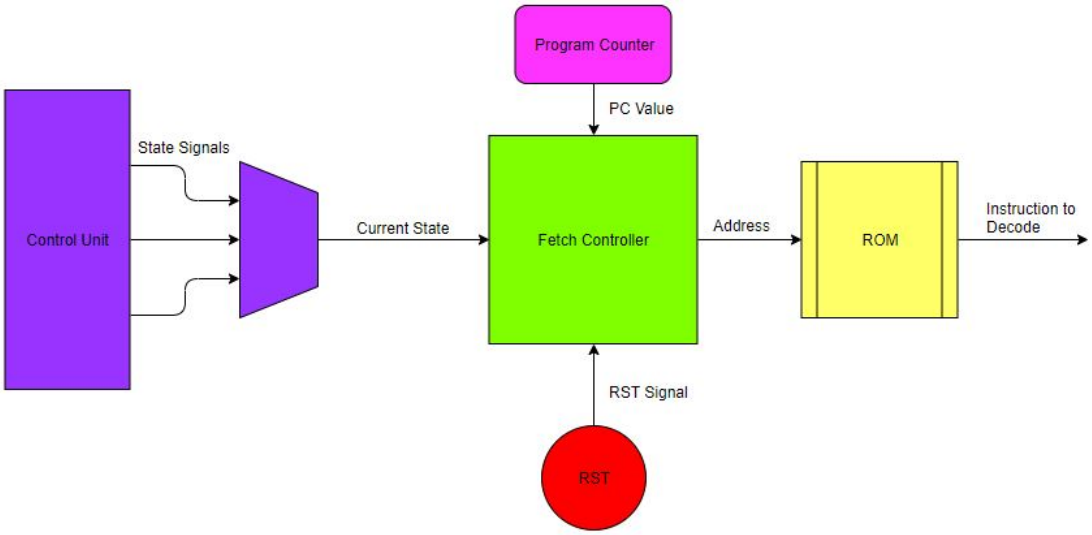


Figure 5: Fetch Block Diagram.

4.3 Decode

4.4 Execute

4.5 Memory Access

4.6 Write Back

4.7 ALU

5 Hazard Mitigation Techniques

Decode, execute, and memory access stages each has an independent hazard detection unit. The hazard detection unit has a write destination FIFO and a validity indicator vector. Write destination FIFO stores the write destinations of the instructions as they come in. Validity indicator vector contains validity bits to indicate which items on the write destination FIFO are valid. At rising edge, the current instruction is checked for hazards. The opcode is decoded and is compared against a list of opcodes that modifies data. If the opcode modifies a register or the RAM then the first bit on the validity indicator vector is marked valid. At the same time, the write destination value is written into the write destinations FIFO. The other data in the write destinations FIFO moves along until they expire. Hazard detection unit will also compare the current instruction's read destination to the write destination FIFO. If a write destination on the write destination FIFO matches the read destination of the current instruction then the hazard is flagged. Hazards will trigger data forwarding in the pipeline.

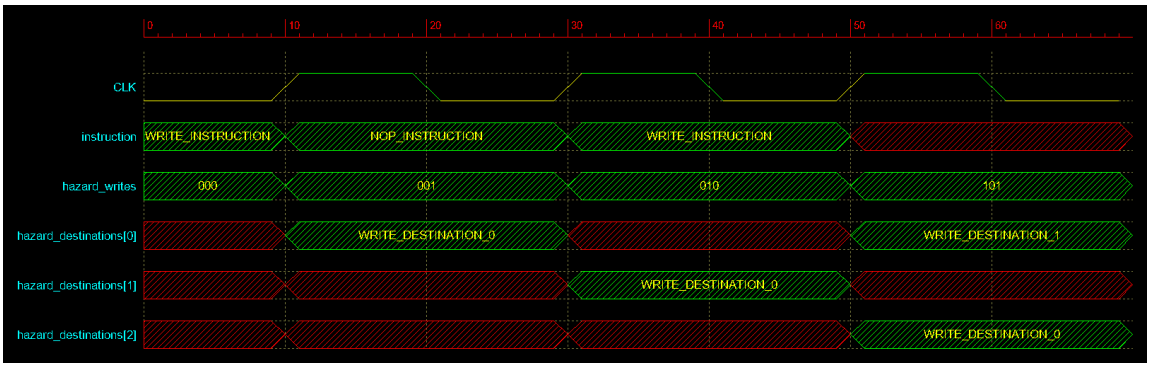


Figure 6: Hazard detection FIFO and valid bits.

Figure. 6 contains a hazard detection example. When the write instruction is clocked in at t = 10 ns, the write destination is stored in the write destination FIFO (hazard_destinations). The first bit of the validity indicator vector (hazard_writes) is set to 1. All the other bits on the validity indicator vector is set to 0. At t = 30 ns, a NOP

instruction is clocked in. Nothing new is written in the FIFO. However, the previous write destination moves through the FIFO. At t = 50 ns, a new write instruction is clocked in. The write destination FIFO and validity indicator vector is updated with new data.

5.1 Operand Forwarding

The simplest RAW hazard is an ALU instruction after an ALU instruction. The former ALU instruction does not have enough time to write into the register file before the latter ALU instruction reads the register file. The solution to the problem is operand forwarding. Operands from the former instruction are forwarded to the latter instruction using muxes.

	Clock Cycle Number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
ADD R1, R4, R0	IF	D	EX	**M**	**WB**			
ADD R0, R1, R3		IF	D	**EX**	**M**	**WB**		
ADD R2, R1, R0			IF	D	**EX**	M	WB	
ADD R3, R1, R0				IF	**D**	**EX**	M	WB

Table 7: ALU instruction after ALU instruction hazard.

Figure. 12 shows an example of ALU instruction after ALU instruction hazard. At clock 3, R1's value is computed. At clock 4, R1's value is forwarded from the memory access stage (instruction 1) to the execute stage (instruction 2). The forwarding uses muxes from the memory acccess stage to the execute stage. R0's value is also computed at clock 4. At clock 5, R1's value is forwarded from write back stage (instruction 1) to execute stage (instruction 3). At clock 5, R1's value is forwarded from write back stage (instruction 1) to decode stage (instruction 4). At clock 5, R0's value is forwarded from memory access stage (instruction 2) to execute stage (instruction 3). At clock 6, R0's value is forwarded from write back stage (instruction 2) to execute stage (instruction 4). Operands are satisfied at every execute stage of each instruction.

	Clock Cycle Number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
IN R1	IF	D	EX	**M**	**WB**			
ADD R0, R1, R3		IF	D	**EX**	M	WB		
ADD R2, R1, R0			IF	D	**EX**	M	WB	
ADD R3, R1, R0				IF	**D**	EX	M	WB

Table 8: IN instruction hazard.

Another hazard is any instruction after the IN instruction . The hazard is shown in Figure. 8. At clock 4, memory access stage (instruction 1) latches the IN data. R1's value becomes the IN data. At the same time in clock 4, the IN data is forwarded to the

execute stage (instruction 2). At clock 5, the R1's value is forwarded to instruction 3 and instruction 4. This resolves the IN data hazard.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
LOADIMM.lower #9	IF	D	**EX**	M	WB		
LOADIMM.upper #4		IF	**D**	**EX**	M	WB	
MOV R2, R7			IF	**D**	EX	M	WB

Table 9: LOADIMM hazard.

Figure. 9 shows the LOADIMM hazard. The hazard occurs when any other instruction arrives after the LOADIMM instruction. In clock 2, R7's value is retrieved and the upper bits are updated to the value 9. In clock 3, R7's value is forwarded from execute stage (instruction 1) to decode stage (instruction 2). At the same time in clock 3, the lower bits of R7 are updated to the value 4. The muxes in the decode stage handle the forwarding. In clock 4, the correct R7 value is forwarded again from execute stage (instruction 2) to decode stage (instruction 3). The forwarding resolves all hazards.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
MOV R2, R7	IF	D	**EX**	M	WB		
ADD R1, R2, R3		IF	**D**	EX	M	WB	

Table 10: MOV hazard.

Figure. 10 shows the MOV hazard. The hazard occurs when any other instruction arrives after the MOV instruction. At clock 2, R2's value is retrieved. At clock 3, the R2's value is forwarded from execute stage (instruction 1) to decode stage (instruction 2). At clock 4, R2's value is available to the ALU.

	Clock Cycle Number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
ADD R1, R4, R0	IF	D	EX	M	WB				
ADD R0, R1, R3		IF	D	EX	**M**	WB			
TEST R0			IF	D	**EX**	**M**	WB		
BR.Z R1, #6				IF	S	**D**	EX	M	WB

Table 11: TEST branch instruction hazard.

Figure. 11 shows the TEST branch instruction hazard. The hazard occurs when a combination of ALU instructions, TEST instructions, and branch instructions happen in-succession. At clock 4, R0's value is computed. At clock 5, R0's value is forwarded from

memory access stage (instruction 2) to execute stage (instruction 3). At the same time in clock 5, the branch instruction is stalled by one cycle. In clock 6, R1's value is forwarded from the register file to the decode stage of branch instruction. In clock 6, negative zero flags are forwarded from the test instruction to branch instruction. The branch destination is computed in clock 7.

5.2 Load and Store Bypassing

If a load/store operation occurs after an ALU instruction then load and store bypassing is done. At the decode stage of the store instruction, the source register value is captured and forwarded. The destination register value is captured in a special FIFO when ALU instruction passes through the memory access stage. At the memory access stage of the store instruction, the correct values for the destination and source register are available. FIFO value retrieval only requires 3 searches in parallel. The search time is only a few nanoseconds.

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
ADD R1, R4, R0	IF	D	EX	**M**	WB		
ADD R0, R1, R3		IF	D	EX	**M**	WB	
STORE R1, R0			IF	**D**	EX	**M**	WB

Table 12: Store after ALU instruction hazard.

Figure 12 shows store after ALU instruction hazard. At clock 3, R1's value is computed by the ALU. At clock 4, decode stage of the store instruction captured R1's value. At the same time in clock 4, R0's value is computed. In clock 5, R0's value captured by a special FIFO in the memory access stage. At clock 6, R0's value and R1's value are available to the memory access stage. This allows the store function to store R1's value at the memory location of R0's value. A similar hazard mitigation procedure happens for the load after ALU hazard.

The second hazard is a store instruction after a load instruction. If the load instruction loads a value into a register and that register contains the value for write memory location in the store instruction then a hazard occurs. The hazard is mitigated when the loaded value is stored in a special FIFO in the memory access stage. At the memory access stage of the store instruction, the value is retrieved from the special FIFO.

Figure 13 shows an example of the store after load hazard. At clock 4, the loaded R1 value is written into a special FIFO. At the clock 5, R1's value is retrieved from the special FIFO. Then R4's value is stored at the memory location of R1's value.

5.3 Multiplication

	Clock Cycle Number						
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***
LOAD R1, R0	IF	D	EX	**M**	WB		
STORE R4, R1		IF	D	EX	**M**	WB	

Table 13: Store after load hazard.

6 Observations

7 Performance

7.1 Testing Methodology

Final Tests 1-3 ROM files provided by Lab TA Ibrahim Hazmi will be used as the CPU test code. Additionally, Ibrahim's 7-Segment Display Controller will be used to display CPU output data in hexadecimal format. The 7-segment display is integrated into the FPGA evaluation board. CPU input data is controlled via the mapped FPGA evaluation board SPST switches. A function generator outputting a 3.3V square wave with variable frequency is used as the CPU clock signal. The CPU reset signal is also mapped to an FPGA evaluation board push switch. The Testing system schematic is illustrated in Figure 7.

Figure 7: Testing System Schematic.

7.2 Results

7.2.1 Critical Path

The critical path is the execution stage of the pipeline. The execution stage contains the ALU, 4 muxes, forwarding and a direct path from the register file. The multiplier is the slowest component of the ALU. Multiplier uses the DSP48A block without any pipelining and zero stalls. The DSP48A multiplier mode maximum delay is 6 ns. The maximum delay for the 2 layers of muxes is 3 ns. The read period from the register file is 4 ns. Therefore, the maximum delay is 13 ns. As a result, the period of the execute stage correlates with the maximum frequency of 78 MHz.

7.2.2 CPI

Given an input of 7, Test 1 program requires 70 instructions to complete. Using timing analysis from the Vivado ISE, we were able to calculate the number of elapsed clock cycles required to produce the correct resultant. The equations below demonstrate the calculated CPI of 1.58

$CPI_{Test1} = \frac{Cycles}{Instructions} = \frac{20+3+3+1}{20} = 1.35$

Test 2 has 9 instructions per loop. The total penalties are fixed at a 7 cycles per loop. Therefore the average CPI is 1.77.

$CPI_{Test2} = \frac{Cycles}{Instructions} = \frac{9+3+3+1}{9} = 1.77$

Test 3 has 5 instructions per loop. The total penalties are fixed at a 7 cycles per loop. Therefore the average CPI is 2.4.

$$CPI_{Test3} = \frac{Cycles}{Instructions} = \frac{5+3+3+1}{5} = 2.4$$

7.2.3 Clock Rate

As the maximum clock rate of the CPU exceeds the input buffer slew rate of 66 MHz, an internal PLL was used. The internal PLL stepped up the on-board clock of 50 MHz to 78 MHz. The 78 MHz clock was used to drive the entire CPU. 78 MHz is the maximum clock rate of the CPU. Aggressive optimizations was done by Xilinx ISE to increase the clock rate. All the tests were done on the CPU at 78 MHz. The tests used the LEDs for output instead of the 10 segment display. The 10 segment display has issues displaying digits at high frequencies.

7.2.4 Simulations

8 Contributions

9 Conclusion

10 Recommendation

11 References

12 Appendices

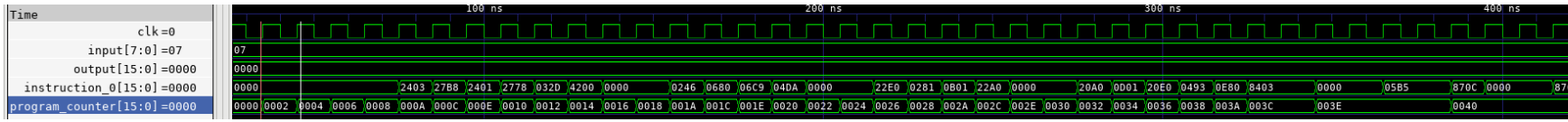


Figure 8: Test 1 time 1.