Project 3 Report

Authors: Matthew Martin and Keaton Shelton

April 21st, 2022

ECE 4120-001 Fundamentals of Computer Design

Single-Cycle Implementation Modified Block Diagram

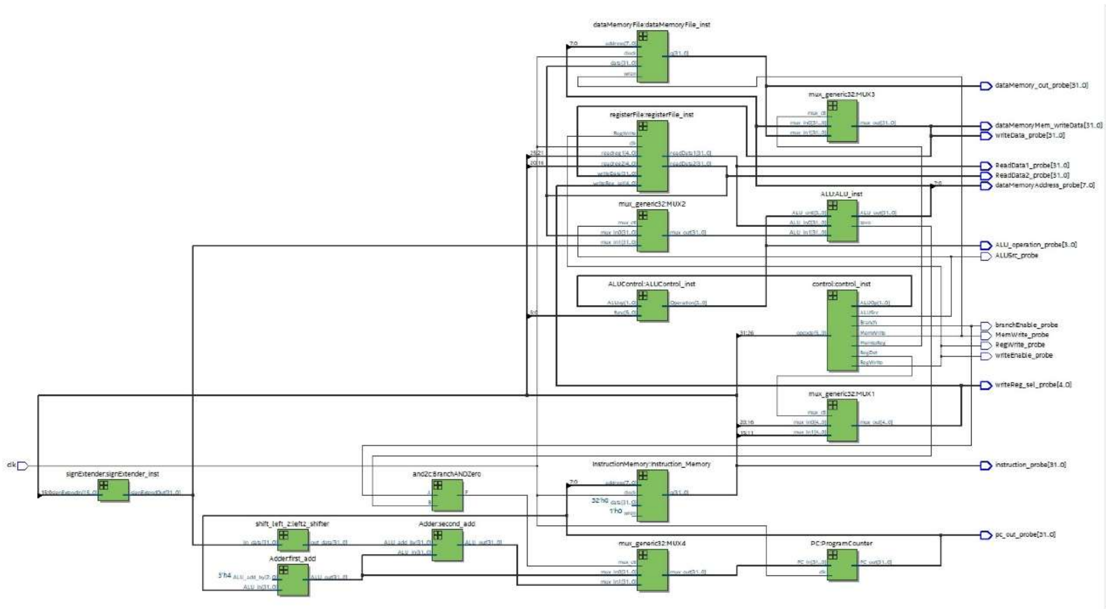

Figure 1. Modified Block Diagram of Single Cycle Implementation

Pipelined Implementation Modified Block Diagram

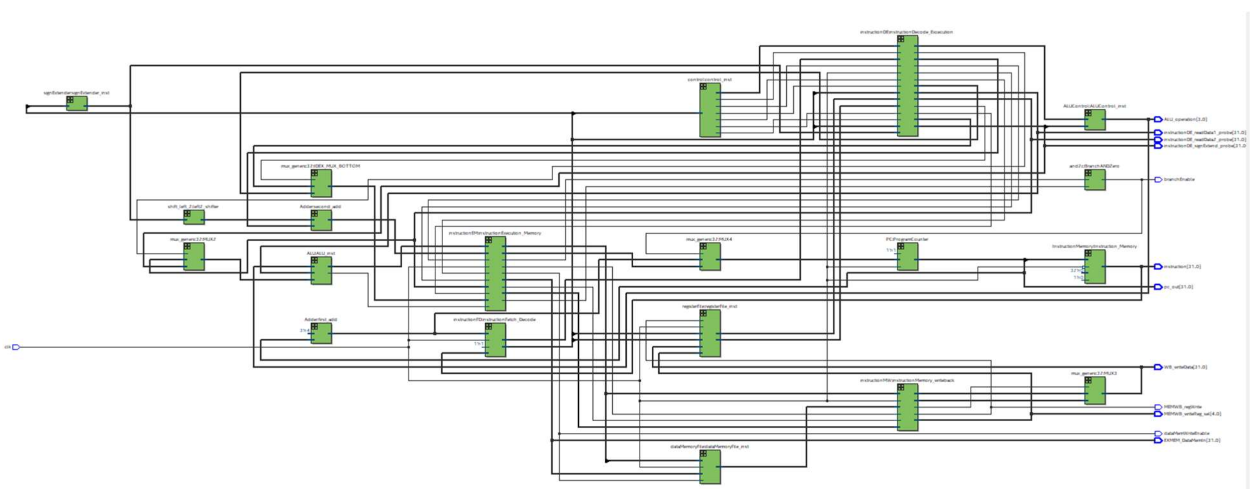

Figure 2. Modified Block Diagram of Pipelined Implementation

Pipelined Implementation with Hazard Detection & Full Forwarding

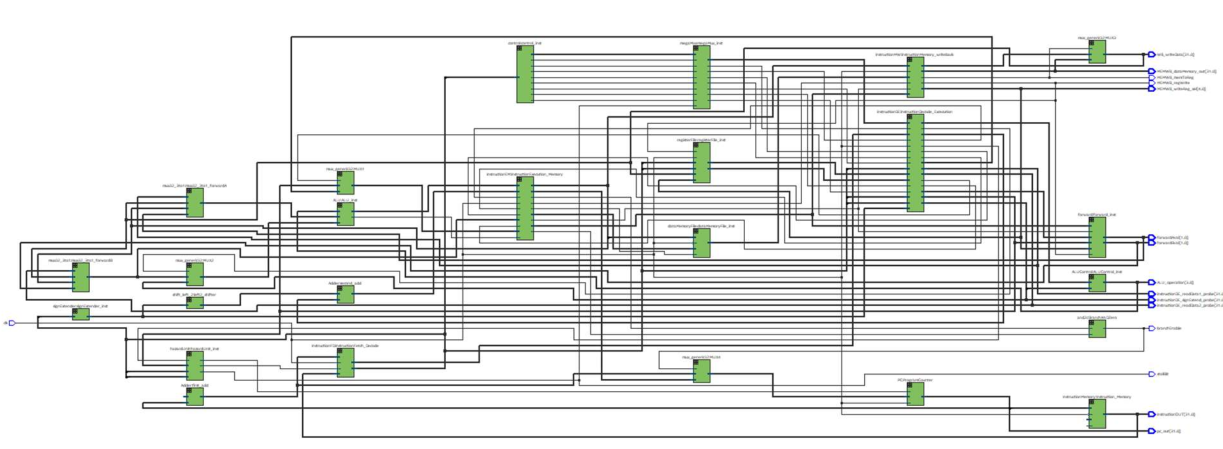

Figure 3. Modified Block Diagram of Pipelined Implementation with Hazard Detection & Full Forwarding

Single Cycle Implementation Components

PC Incrementor

The ALU or “add4” component is a simple generic combinational unit that adds 2 numbers. It is instantiated with a 3-bit input “100” to add 4 to the PC each cycle.

Program Counter

The “PC (Program Counter)” is one of the sequential modules in our design. It is a custom or “non-IP” block which takes an input called “PC in.” This input is simply the output of the PC + 4 which is loaded at the next rising clock edge. This functionally means that the program counts 4 bytes over a clock cycle equating to a 32-bit word per clock. It would be equivalent to defining a 32-bit word in the instruction memory and incrementing the PC by 1 each cycle. To achieve this, the output is loaded from the signal holding the input prior to the changing of this intermediary signal such that the incrementation is delayed by a clock cycle.

Instruction Memory (Quartus IP)

The "Instruction Memory" is an IP (Intellectual Property) block using the 1-Port RAM block provided by Altera in Quartus. This memory block was initialized with a byte size of 8 bits and with a byte-width of 256 bytes. Interestingly, the 32-bits needed by the system are automatically read in a single clock cycle even though the address given to the RAM points only to the first of four bytes. If this were not the case, 32-bit words would need to be defined since it would require 4 clock cycles to read a single word as 8 bytes per word.

Data Memory

Data Memory is another IP 1-Port RAM block provided by Altera in Quartus. This unit is used for storage of data which can be accessed from the registers of the processor. This unit can be written to by the MIPs store word instruction and read from by the load word instruction. In the single cycle implementation, it must be provided with a delayed clock relative to the instruction memory such that the timing constraints are satisfied. In the pipelined implementation, it is altered or accessed in the MEM/WB stage and therefore is the source of data hazards.

Arithmetic Logic Unit

The "ALU" (ALU) [Arithmetic Logic Unit] is one of the combinational blocks of our design. It is a "non-IP" component with inputs 32-bit ALU_in0, 32-bit ALU_in1, and four-bit input ALU_cntl. The ALU has a 32-bit ALU_out output vector and a 1-bit "zero" output. The operations of this ALU are entirely dependent on the selected mode of the ALU using "ALU_cntl." If ALU_cntl is all zeros, then the ALU will perform an AND operation on the two inputs and return its output on ALU_out. If ALU_cntl is "0001" then the ALU will perform an OR operation on the two inputs and return the results into ALU_out. If ALU_cntl is "0010" then the ALU will add the two inputs and return the output to ALU_out. If ALU_cntl is "0110" then the ALU will subtract ALU_in1 from ALU_in0 and return the result to ALU_out. If ALU_cntl is "0111" then the ALU will XOR the two inputs together and place the result into ALU_out. If ALU_cntl is "1100" the ALU will perform a NOR operation on the two inputs and return the value into ALU_out. The "zero" output will go to one if the value of ALU_out zero, else it will be zero.

ALU Control Unit

The "ALUControl" is one of the combinational blocks of our design. It is a "non-IP" component with a six-bit input func (Function Code), two-bit input ALUop (ALU Op Code), and returns a four-bit output Operation (ALU Operation Mode). Using concurrent operations, we set each bit of ALU Operation Mode through a series of ORs, ANDs, and NOTs.

Control Unit

The "Control" (control) is one of the combinational blocks of our design. It is a "non-IP" component that takes one six-bit input opcode and returns a series one of one-bit outputs: RegDst, Branch, MemRead, MemWrite, MemtoReg, ALUOp, ALUSrc, RegWrite, and Jump as specified by the control diagram from our Zybooks homework. This block is the primary controller for deciphering an instructions opcode and sending the correct values to the currently implemented ALU, Register File, the mux going into "ALU_in1" and the mux going into Write Register.

Register File

The “Register File” is a sequential block in our design. It is also a “non-IP” block with inputs: readreg1 (Read Register 1), readreg2 (Read Register 2), writeReg (Write Register), writeData (Write Data), RegWrite (Register Write Enable), and clk (Clock). This block has outputs readData1 (Read Data 1) and readData2 (Read Data 2). Using sub-components and2c, decoder, register32, and mux32to1 we can create the diagram specified in Figure 2 and Figure 3 to create the main “Registers” specified in Figure 1. First, using the decoder we can take the Write Register number and put it into a 32-bit vector (decoder_out) to hold the intermediate values from the decoder to feed into the AND gates. Using a VHDL generate statement, we generate 32 instances of the “register32” and the “and2c” component. Each AND gate take one input of register write enable and a bitwise input from the 32-bit vector from the decoder (decoder_out(i)). Finally, this returns a 32-bit vector called “and_out”. The and_out vector contains the AND results of all the AND gate operations to be fed into each subsequent register. Within this generate statement are the “register32” instances which take an input from the clock, the bitwise input “and_out(i)” at location “i” vector, and the Write Data. The “register32” then outputs into an array of 32-bit vectors known as “regOut(i)” at location “i”. Finally, the data array regOut(i) at location “i” is then fed into each input of the “mux32to1” IN1-IN32. With the select line being the Read Register 1 and the output being Read Data 1. This process with “mux32to1” is then repeated for Read Register 2 and output Read Data 2.

Primitives

The “**AND Gate**” (and2c) is one of the combinational blocks of our design. It is modeled after the standard primitive AND gate. The AND gate takes two, one-bit inputs and returns a one-bit output. This is primarily used as a component in the Register File to help enable the write lines to a specific register from the write-enable line and the selected register.

The **“MUX”** (mux_generic32) is one of the combinational blocks of our design. It is a “non-IP” block that takes two data inputs of (N-1) bits, a one-bit control line, and outputs a (N-1) bits vector. Using the one-bit control line you can select which data-path needs to flow through the MUX. The size of the input data is specified using a generic block called N. For a 32-bit mux you would specify this blank to be 32 and the mux will auto initialize for 32-bits of data.

The **“Memory MUX”** (mux32to1) is one of the combinational blocks of our design. It is a “non-IP” block that takes 32, 32-bit inputs (IN1-IN32), a five-bit select line (sel), and a 32-bit output (F). This mux will pass one of the inputs selected to its output F. For example, if sel is all zeros then F will be IN1, or on the contrary if sel is all ones then F will be IN32.

The **“Decoder”** (decoder) is also another combination block in our design. It is a “non-IP” block with an input A that is five-bits and returns an output F that is 32-bits. Given an input A the output F will reflect on a bit-wise operation that selects the equivalent location on F to go to high. For example, if A is zero then F(0) will be one while the rest are zeros. Likewise, if A is all ones, then F(31) will be one while the rest are zeros. This component is primarily used as a sub-component for the Register File to help enable the write lines to a specific register from the write-enable line and the selected register.

The **“Registers”** (register32) is a primitive sequential block in our design. It is a “non-IP” block with inputs one-bit C, one-bit clock, and a 32-bit D. It has one 32-bit output named Q. This simple register file first checks if the clock is one a rising edge and if the C line (enable) is high. If these conditions are met, then D will over-write Q and the new data will be output. This sequential primitive is primarily used in the Register File as a sub-component to act as the registers.

The **“sign-extender”** unit is a simple combinational unit which receives a 16-bit VHDL vector and extends the MSb, resulting in a signed 32-bit output.

The **“Shift-left-2”** unit is another simple combinational unit which receives a 32-bit VHDL vector and shifts it to the left by 2 bits. This unit is implemented in a MIPs processor to increase the range of jump instructions.

Using these modules, we created we were able to then link and connect them together in our **“top_level”** to add onto our original design from Project Phase 1. The Instruction output from our Instruction Memory is then fed into the correct spots on the Register File. The subsequent lines from the Register File are then connected to the ALU and Mux to ALU. The output of the ALU result is then fed back into the Write Data of the Register File. Currently there are no hardware pins used as the processor is self-contained using only the initialized memory. There are virtual pins used to show the progress of the system as virtual time moves. The probes include: ALU in/out (This is the Add4 ALU), ALU add by (Add4 ALU), PC in/out, instruction in/out, Register File Read Data 1 and 2, Register Write Enable, Write Register Number, and finally Write Register Data.

Pipelined (No Hazard Detection or Forwarding)

- *In addition to the components explained above which were used in all 3 implementations, the following components were used in the pipelined implementation.*

IF/ID Register

The IF/ID register is the first auxiliary register, or “first-stage” in the pipelined implementation. It receives the incremented PC signal directly from the PC after incrementation, as well as the instruction directly from instruction memory.

ID/EX Register

The ID/Ex register is the second stage of the pipelined implementation. It receives the incremented PC count from IF/ID, the 2 outputs of the register file, the output of the sign-extender, and the Rd and Rt registers of the decoded instruction. It also receives all output signals of the control unit. On each rising clock edge, this register passes all control signals except those which are used in the execution stage to the Ex/Mem register.

EX/MEM Register

The Ex/Mem register is the third stage of the pipelined implementation. It receives the offset PC value, the ALU result and ALU zero signals, the “read data 2” signal passed from the Id/Ex register, and the output of the destination register selecting mux. It also receives the memWrite, memRead, and Branch control signals. On each rising clock edge, this register passes all control signals except those used in the memory accessing stage to the Mem/WB register.

MEM/WB Register

The Mem/WB register is the fourth and final stage of the pipelined implementation. It receives the MemToReg and RegWrite signals from the Ex/Mem register, as well as the output of data memory and ALU result passed from Ex/Mem. The Datapath outputs of this register are passed to the final multiplexer in the pipeline which is then sent back to the instruction decode stage for writing data back into the register file.

Pipelined Implementation (W/Hazard Detection and Full Forwarding)

Hazard Unit

The hazard unit is used in the pipelined bonus implementation to detect data hazards that require a stall. It receives the RT register from the Decode / Execution register, as well as the RS register from the Fetch / Decode register, the RT register from the Fetch / Decode register, and finally the MemRead control signal from the Decode / Execution register. It then checks to see if the MemRead signal is high and checks if the Decode / Exection RT register is equal to the Fetch / Decode RS register or if the Decode / Execution RT register is equal to the Fetch / Decode RT register. If that condition is met it then proceeds to set the stallSig output to one, sets the Fetch / Decode Write signal to zero and sets the PC write signal to zero as well. With stallSig set to one that forces all the control signals to be zero to be written to the Decode / Execution register which then implements our stall.

Forward Unit

The forward unit is used in the pipelined bonus implementation to detect data hazards that require forwarding. It receives the RT register from the Decode / Execution register, the RS register from the Decode / Execution register, the RD register from the Execution / Memory register, the RD register from the Memory / Writeback register, the RegWrite control signal from the Execution / Memory register and finally the RegWrite control signal from the Memory / Writeback register. It then proceeds to check several different conditions. It first checks if any data needs to be forwarded from the Execution / Memory register and will set the outputs forwardA and forwardB to "10" if needed. Both do not need to be set, only one of them can be set if necessary. The next set of checks determine if any data needs to be forwarded from the Memory / Writeback register and will set the outputs forwardA and forwardB to "01" if needed. As stated, both do not need to be set, only one of them can be set if necessary.

Device Selection

10M50DAF484C7G	1.2V	49760	360	360	1677312	288

Figure 4. Snapshot of Device Selection (Same For all three Implementations)

Flow summaries

Flow Summary

<<Filter>>

Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Blake_Proj3
Top-level Entity Name	top_level
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	1,862 / 49,760 (4 %)
Total registers	1054
Total pins	247 / 360 (69 %)
Total virtual pins	0
Total memory bits	16,384 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 5. Single-cycle Flow Summary

Flow Summary

<<Filter>>

Flow Status	Successful - Thu Apr 21 21:22:53 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MIPS_Project1
Top-level Entity Name	top_level
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,001 / 49,760 (4 %)
Total registers	1380
Total pins	237 / 360 (66 %)
Total virtual pins	0
Total memory bits	16,384 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 6. Pipelined Flow Summary

Flow Summary	
<<Filter>>	
Flow Status	Successful - Thu Apr 21 20:31:15 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MIPS_Project1
Top-level Entity Name	top_level
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,081 / 49,760 (4 %)
Total registers	1385
Total pins	242 / 360 (67 %)
Total virtual pins	0
Total memory bits	16,384 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 7. Bonus Flow Summary

RTL Views

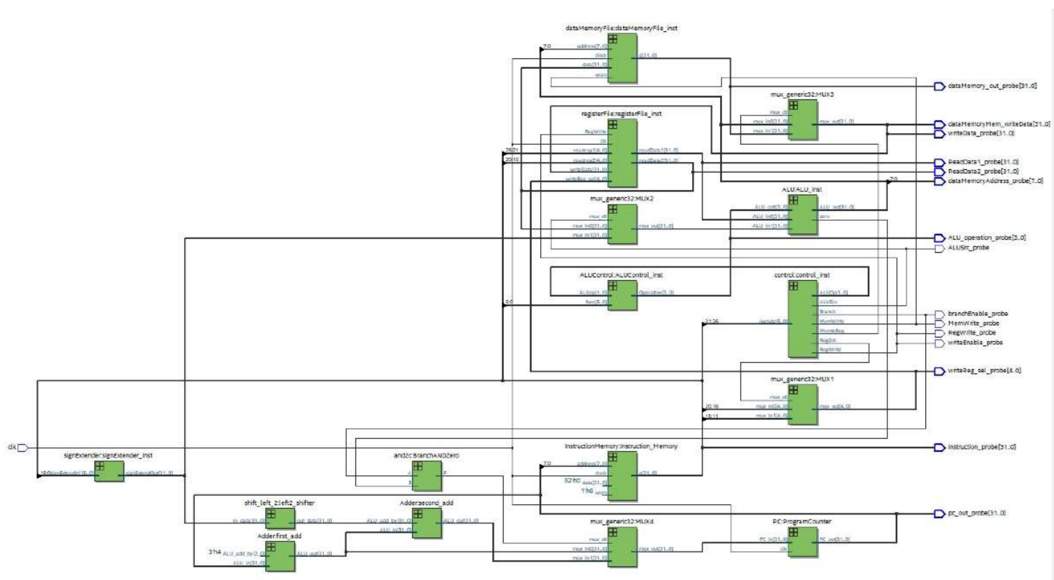

Figure 8. Single-cycle RTL Viewer

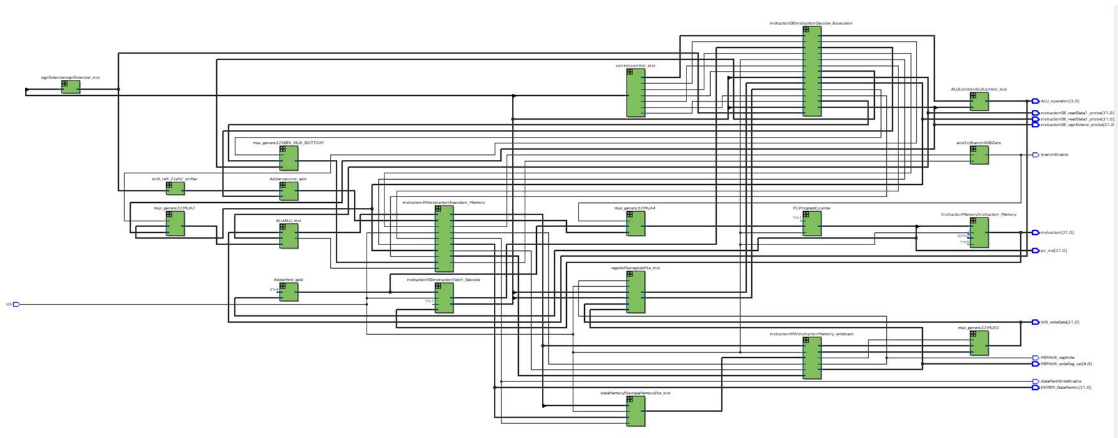

Figure 9. Pipelined RTL Viewer

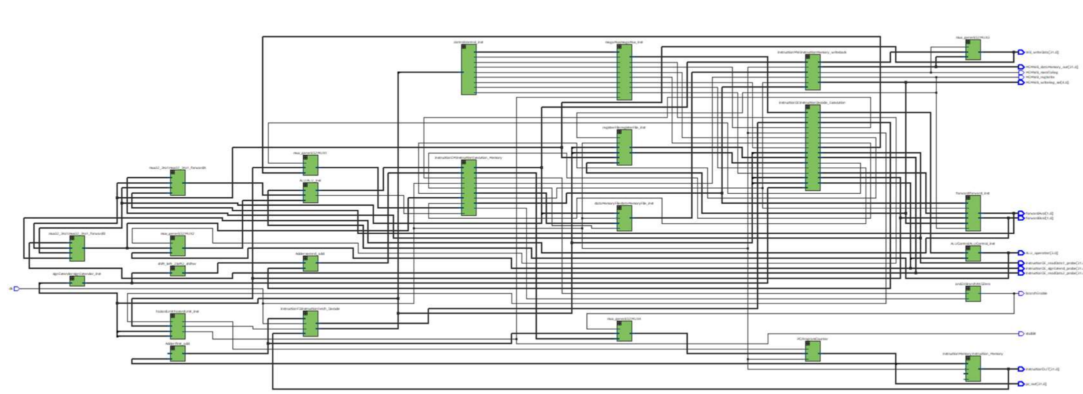

Figure 10. Bonus RTL Viewer

Technology Mapping

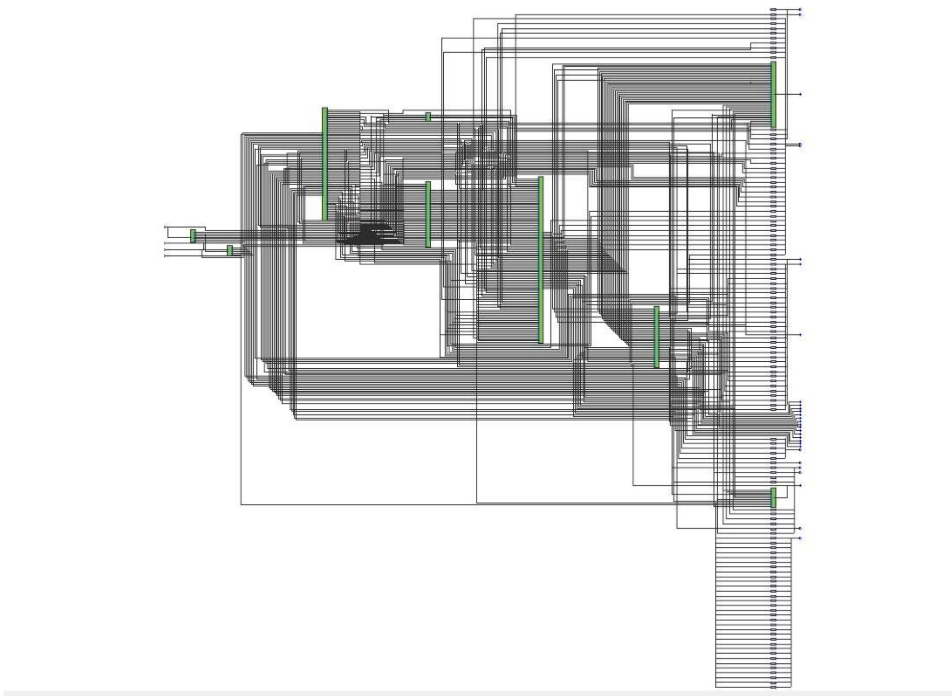

Figure 11. Single-Cycle Technology Map (Post-Fitting)

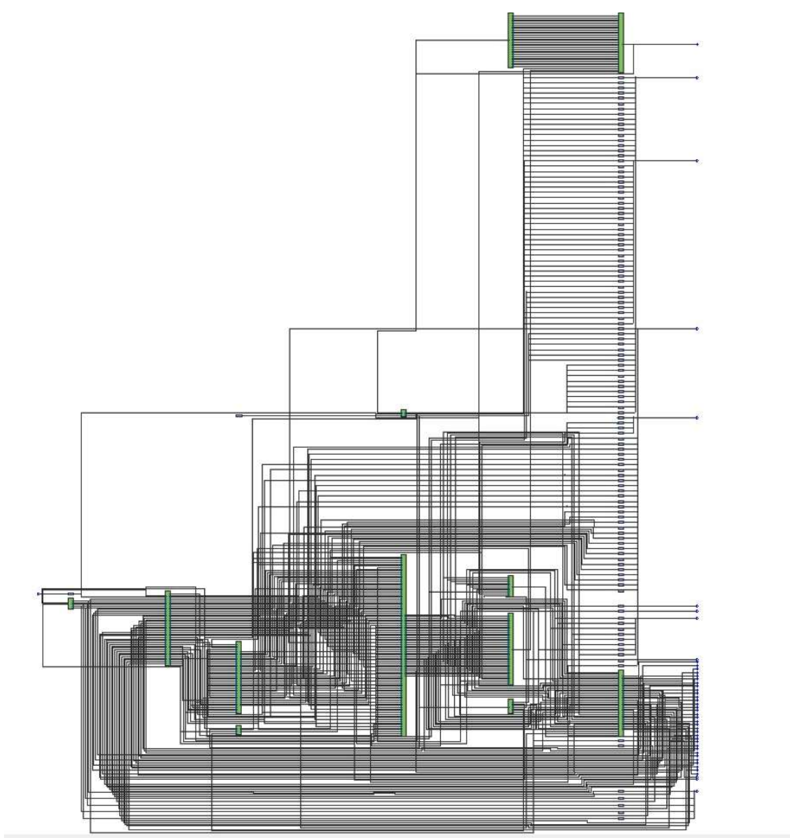

Figure 12. Single-Cycle Technology Map (Post-Mapping)

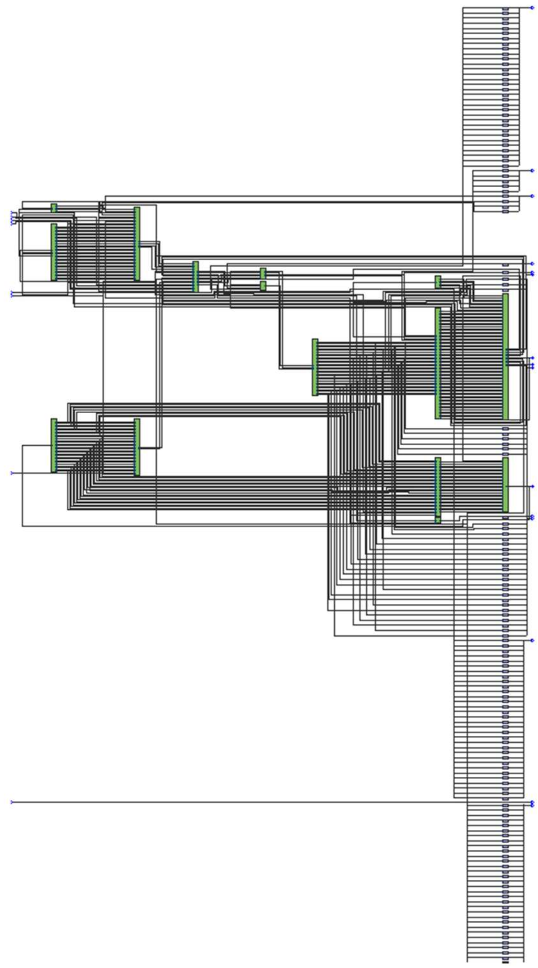

Figure 13. Pipelined Technology Map (Post-Fitting)

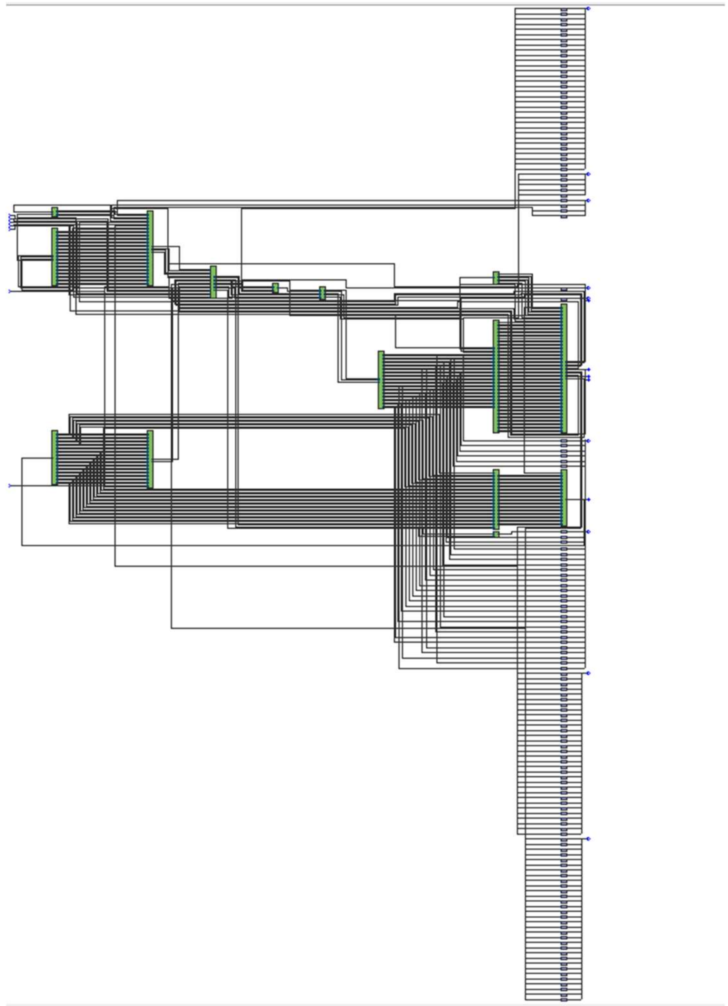

Figure 14. Pipelined Technology Map (Post-Mapping)

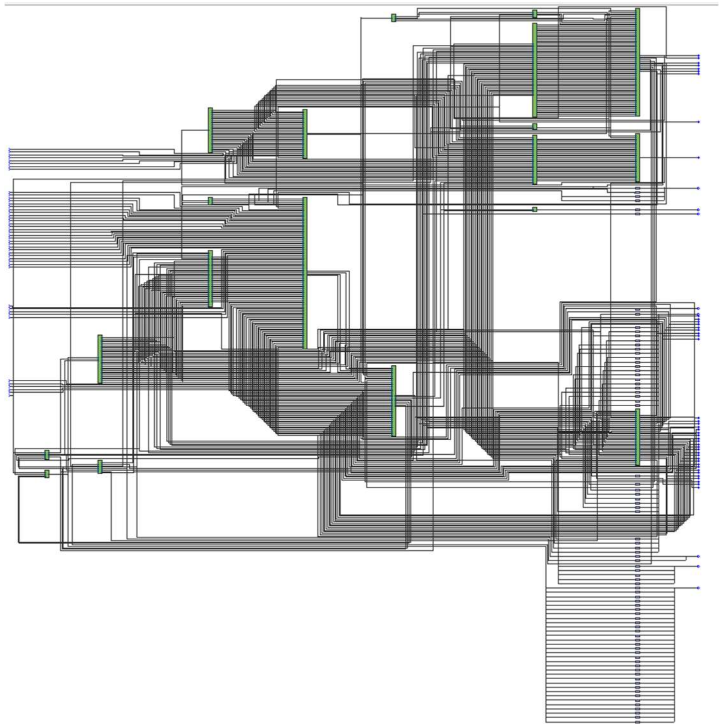

Figure 15. Bonus Technology Map (Post-Fitting)

Figure 16. Bonus Technology Map (Post-Mapping)

Elaboration on Testbenches

As was done for the previous phases of the project, the instructions (and this time the data memory) were loaded by use of "mif" files, and so the testbench is simply the process of providing a clock signal to the input of the system and reading/comparing the waveforms with the expected outputs. This is true for all 3 implementations and their associated testbenches/waveforms.

Waveform Elaborations

Single-Cycle Implementation

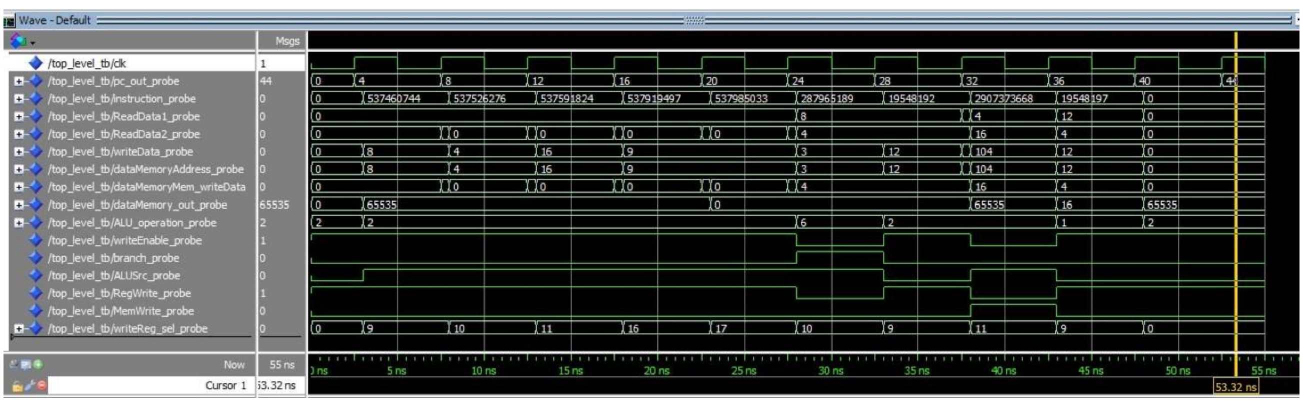

Figure 17. Single-cycle Waveforms

Signal Breakdown

From the above waveform we can see that we have the following signals: clk (the incoming clock), pc_out_probe (the output of the program counter), instruction_probe (the ouput of the instruction memory), readData1_probe (the Read Data 1 output out of the register file), readData2_probe (the Read Data 2 output out of the register file), writeData_probe (the data that gets written to the register file), dataMemoryAddress_probe (the address input to the data memory), dataMemory_writeData (the write data input to the data memory), dataMemory_out_probe (the output of the data memory), ALU_operation_probe (the control signal to set the ALU operation), writeEnable_probe (control line that enables writes to the register file), branch_probe (this is the control signal branch, not the true branch signal PCSrc), ALUSrc_probe (the control signal ALUSrc), RegWrite_probe (the control write enable for the register file), MemWrite_probe (the control write enable for the data memory), and finally writeReg_sel_probe (the destination register number for the register file).

Waveform Analysis

The first four instructions preload our registers (addi t1 zero 0x8, addi t2 zero 0x4, addi t3 zero 0x10, addi s0 zero 0x9, addi s1 zero 0x9). We can see at time 2.5 ns that an eight (writeData_probe) gets written to register nine (writeReg_sel_probe) (t1). This is confirmed as RegWrite_probe goes high at the same time. The next instruction then executes at time 7.5 ns that shows that a four (writeData_probe) gets written to register ten (writeReg_sel_probe) (t2). This is confirmed as RegWrite_probe goes high at the same time. The next instruction then executes at time 12.5 ns that shows that a sixteen (writeData_probe) gets written to register eleven (writeReg_sel_probe) (t3). This is confirmed as RegWrite_probe goes high at the same time. The next instruction then executes at time 17.5 ns that shows that a nine (writeData_probe) gets written to register sixteen (writeReg_sel_probe) (s0). This is confirmed as RegWrite_probe goes high at the same time. The next instruction then executes at time 22.5 that shows that a nine (writeData_probe) gets written to register seventeen (writeReg_sel_probe) (s1). This is confirmed as RegWrite_probe goes high at the same time. The next instruction is beq t1 t1 t2. This is executed at time 27.5 ns and we can see that the control line branchEnable_probe goes high but the PC does not jump. The value on the writeData_porbe does not get written as the RegWrite_probe goes low at that time. The next instruction is add t1 t1 t2. This instruction gets executed at 32.5 ns. We can see that twelve (writeData_probe) is set to write to register nine (writeReg_sel_probe) (t1). This is confirmed by RegWrite_probe going high at the time which is expected. The next instruction to be executed is sw t3 0x64(t2). This is executed at time 37.5 ns and we can see that dataMemoryMem_write is set to sixteen at the time. This is confirmed to write as the control signal MemWrite_probe goes high at the time. The last instruction to execute is or t1 t1 t2. This executes at time 42.5 ns. We see on the waveform that writeData_probe writes a twelve to register number nine (writeReg_sel_probe) (t1) which is the correct result of twelve or four is equal to twelve.

Pipelined Implementation

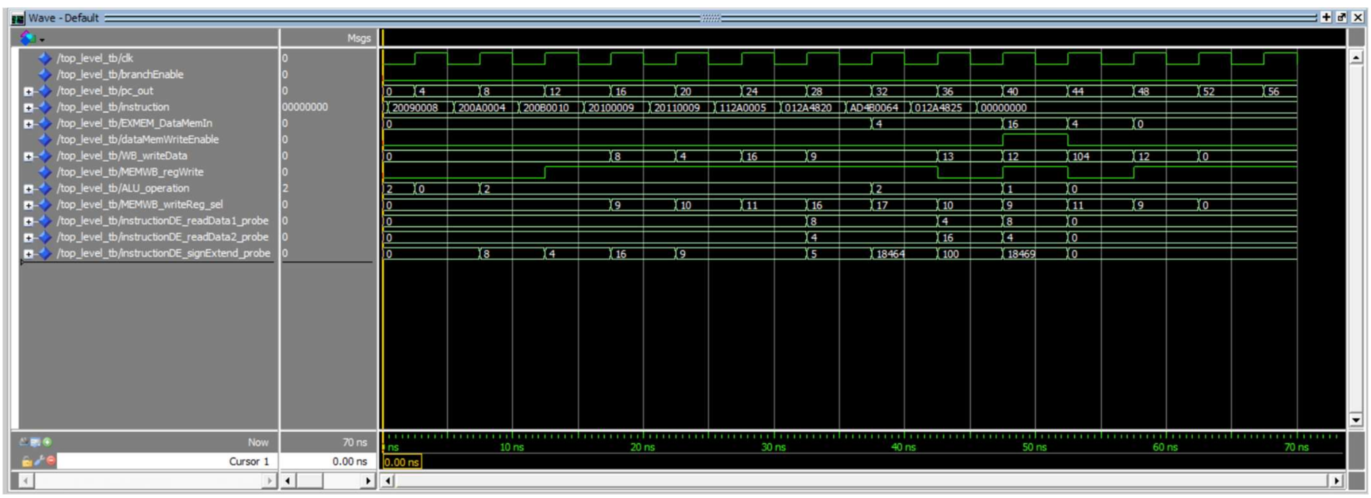

Figure 18. Pipeline Waveforms

Signal Breakdown

From the above waveform we can see that we have the following signals: clk (the incoming clock), branchEnable (from the pipeline diagram this is PCSrc), pc_out (the program counter output), instruction (the output of the instruction memory), EXMEM_DataMemIn (the write data input to the data memory), dataMemWriteEnable (the control write enable line to write to the data memory), WB_writeData (the data that gets fed into the write data port on the register file), MEMWB_RegWrite (the write enable line for the register file), ALU_Operation (the ALU control signal), MEMWB_writeReg_sel (the destination register to be written to on the register file), instructionDE_readData1_probe (data coming immediately off of Read Data 1 of the register file from the Decode / Execution pipeline register), instuctionDE_readData2_probe (data coming immediately off of Read Data 2 of the register file from the Decode / Execution pipeline register), and finally instructionDE_signExtend_probe (data coming off of the sign extender from the Decode / Execution pipeline register).

Waveform Analysis

The first four instructions we have executing are preloading the registers (addi t1 zero 0x8, addi t2 zero 0x4, addi t3 zero 0x10, addi s0 zero 0x9, addi s1 zero 0x9). Due to the nature of pipelining this does not truly begin unit four clock cycles after the loading of the first instruction. Beginning at 17.5 ns we can see that an eight (WB_writeData) is being stored to register number nine (MEMWB_writeReg_sel) (t1) as expected. One clock cycle later at time 22.5 ns we can see that a four (WB_writeData) is being written to register number ten (MEMWB_writeReg_sel) (t2) as expected. One clock cycle later at time 27.5 ns we can see that a sixteen (WB_writeData) is being written to register number eleven (MEMWB_writeReg_sel) (t3) as expected. One clock cycle later at time 32.5 ns we can see that a nine is being written to register number sixteen (MEMWB_writeReg_sel) (s0) as expected. One clock cycle later at time 37.5 ns we can see that a nine (WB_writeData) is also being written to register number seventeen (MEMWB_writeReg_sel) (s1) as expected. We know that it has been successfully written due to the MEMWB_regWrite signal going high through these commands. The next instruction to execute is beq t1 t2 0x5. This command will check if t1 and t2 are equal then the PC will jump ahead by the commands offset, otherwise the PC will continue to count normally. Since t1 is loaded with eight and t2 is loaded with four the command fails and branchEnable does not go high at time 42.5 ns. MEMWB_regWrite also goes low at this time to prevent the calculated value from being written to register ten (t2). The next instruction to execute is add t1 t1 t2. This should result in a twelve as t1 currently holds eight and t2 currently holds a four. Looking at our waveform at time 47.5 ns we can see that it does correctly calculate a twelve and proceeds to write the twelve (WB_writeData) to register number nine (MEMWB_writeReg_sel) (t1) as expected. The next instruction to execute is sw t3 0x64(t2). Looking at our waveform we can see the value of WB_writeData goes to 104 as it is the immediate value of 0x64 added to the value of t2 which is four. This value is being thrown from the Execution/Memory register to the address input of the data memory. This is shown at time 52.5 ns but since the value shown is from the Memory / Writeback register the correct time of arrival to the data memory is at time 47.5 ns. At time 47.5 ns we can see that the value on EXMEM_dataMemIn goes to sixteen as expected due to the value of t3 being sixteen. The write is shown to be successful as the control value of dataMemWriteEnable goes high at time 47.5 ns. The final instruction to be executed is or t1,t1, t2. This should cause an or condition for the registers t1 and t2 and store into t1. The correct result of 8 (t1) or 4 (t2) should result in twelve which it does. This is shown at time 57.5 ns with the twelve (WB_writeData) being written to register number eleven (MEMWB_writeReg_sel) (t1) which is expected.

Bonus Implementation

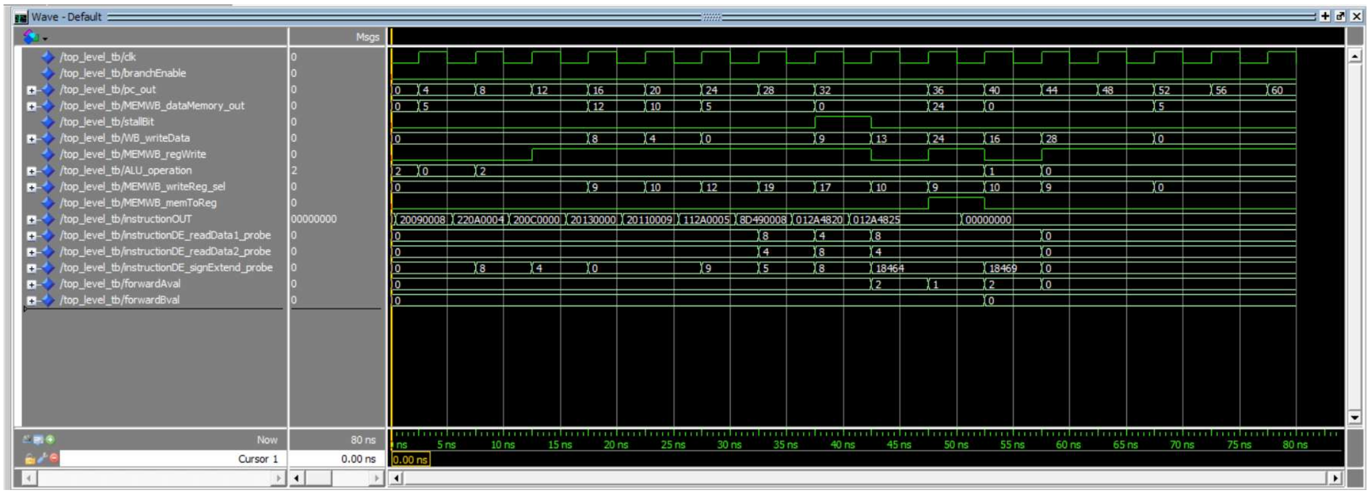

Figure 19. Pipeline Bonus Waveform

Signal Breakdown

From the above waveform we can see that we have the following signals: clk (the incoming clock), branchEnable (From the Pipeline diagram this is PCSrc), pc_out (The program counter output), instructOUT (the output of the instruction memory), MEMWB_dataMemory_out (the output read data of the data me), dataMemWriteEnable (the control write enable line to write to the data memory), stallBit (one of the control signals off of the hazard unit to show that a stall has been issued), WB_writeData (the data that gets fed into the write data port on the register file), MEMWB_RegWrite (the write enable line for the register file), ALU_Operation (the ALU control signal), MEMWB_writeReg_sel (the destination register to be written to on the register file), MEMWB_memToReg (control value coming out of the Memory / Writeback register that controls MUX 3 to determine which data needs be forwarded to the write data of the register file), instructionDE_readData1_probe (data coming immediately off of Read Data 1 of the register file from the Decode / Execution pipeline register), instuctionDE_readData2_probe (data coming immediately off of Read Data 2 of the register file from the Decode / Execution pipeline register), instructionDE_signExtend_probe (data coming off of the sign extender from the Decode / Execution pipeline register), forwardAval (the output of the forward unit to control the forward A mux), and finally fowardBval (the output of the forward unit to control the forward B mux).

Waveform Analysis

The first four instructions we have executing are preloading the registers (addi t1 zero 0x8, addi t2 zero 0x4, addi t4 zero 0x0, addi s3 zero 0x0, addi s1 zero 0x9). Due to the nature of pipelining this does not truly begin till four clock cycles after the loading of the first instruction. Beginning at 17.5 ns we can see that an eight (WB_writeData) is being stored to register number nine (MEMWB_writeReg_sel) (t1) as expected. One clock cycle later at time 22.5 ns we can see that a four (WB_writeData) is being written to register number ten (MEMWB_writeReg_sel) (t2) as expected. One clock cycle later at time 27.5 ns we can see that a zero (WB_writeData) is being written to register number twelve (MEMWB_writeReg_sel) (t4) as expected. One clock cycle later at time 32.5 ns we can see that a zero is being written to register number nineteen (MEMWB_writeReg_sel) (s3) as expected. One clock cycle later at time 37.5 ns we can see that a nine (WB_writeData) is also being written to register number seventeen (MEMWB_writeReg_sel) (s1) as expected. We know that it has been successfully written due to the MEMWB_regWrite signal going high through these commands. The next instruction to execute is beq t1 t2 0x5. This command will check if t1 and t2 are equal then the PC will jump ahead by the commands offset, otherwise the PC will continue to count normally. Since t1 is loaded with eight and t2 is loaded with four the command fails and branchEnable does not go high at time 42.5 ns. MEMWB_regWrite also goes low at this time to prevent the calculated value from being written to register ten (t2). The next instruction to execute is lw t1 0x8(t2). This instruction will load the value in the data memory at location twelve as it takes the immediate value of eight and adds it with the four in t2. In our data memory mif file we have location twelve set to 24. We can see at time 47.5 ns that the signal MEMWB_dataMemory_out goes to twenty-four. We can see that this value (WB_writeData) is being written to register nine (MEMWB_writeReg_sel) (t1) with MEMWB_regWrite going high to enable write as well as the control signal MEMWB_memToReg going high to forward the data from the data memory to the register file using MUX 3. The next instruction to execute is add t1 t1 t2. However, since this is after a load word this presents a data hazard, to resolve a data hazard a stall must have been introduced to allow for the value from the data, then this value needs to be forwarded to the execution unit during the execution of the add t1 t1 t2 instruction. We can see that stallBit goes high at time 37.5 ns earlier in the pipeline. This is then shown on writeback at time 52.5 ns as there is the wrong result of six-teen that is trying to write to register 10 (t2) but is blocked by the stall as the control line MEMWB_regWrite has been set to zero. The correct result is then shown on the next cycle at time 57.5 ns of a twenty-eight (WB_writeData) being written to register nine (MEMWB_writeReg_sel) (t1). This is confirmed to write as MEMWB_regWrite goes high at this time. This is only possible due to the stall letting the value of twenty-four being written to the Memory / Writeback register and then being forwarded to the execution unit by forwardA with a value of one as shown on the waveform at time 47.5 ns. The next instruction is or t1 t1 t2. For this instruction to execute we need to forward the result of the previous instruction (add t1 t1 t2) back to the execution unit so that the right value can be calculated, as the twenty-eight has not been written to t1 yet. For this to execute correctly we need to take the twenty-eight that is now in the Execution / Memory register and forward its value to the Forward A Mux. This is shown on the waveform as the value of forwardA goes to two earlier in the pipeline at time 52.5 ns. The correct result of twenty-eight or four is equal to twenty-eight (WB_writeData) is then shown to be written to register nine (MEMWB_writeReg_sel) (t1) at time 62.5 ns. This is confirmed to write as MEMWB_regWrite goes high at this time.

Setup Slack Analysis

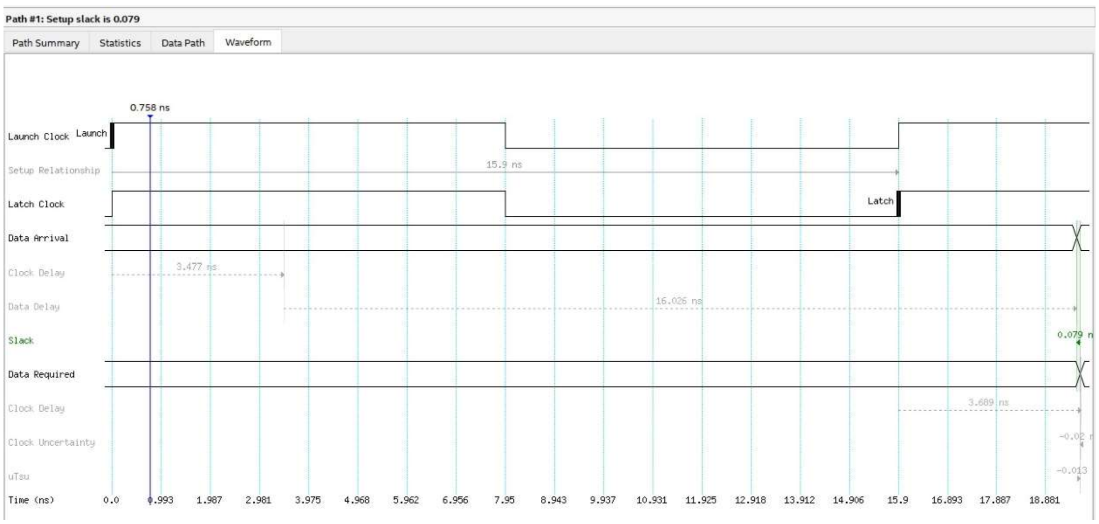

Figure 20. Single-Cycle Setup Slack Analysis

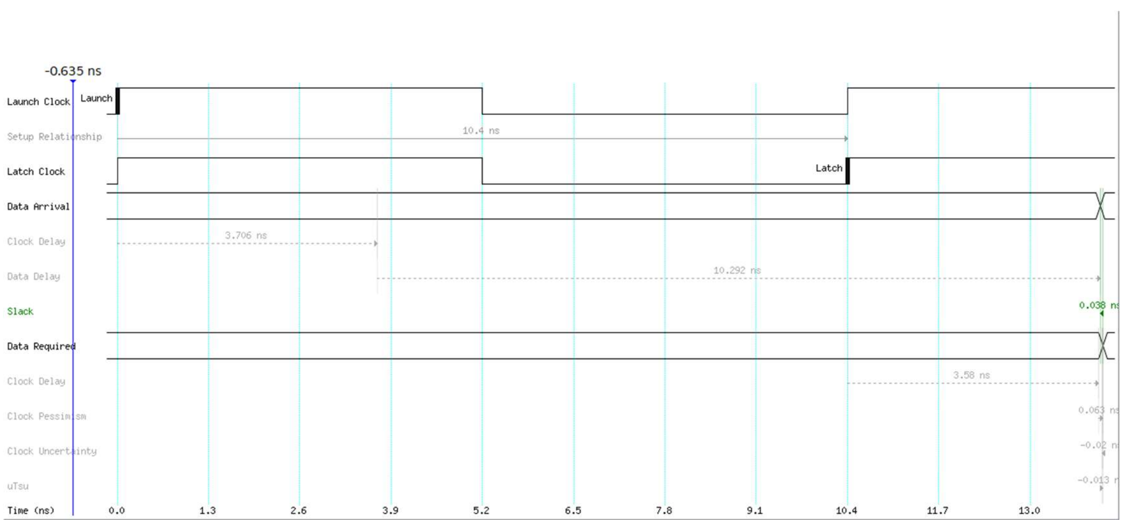

Figure 21. Pipelined Setup Slack Analysis

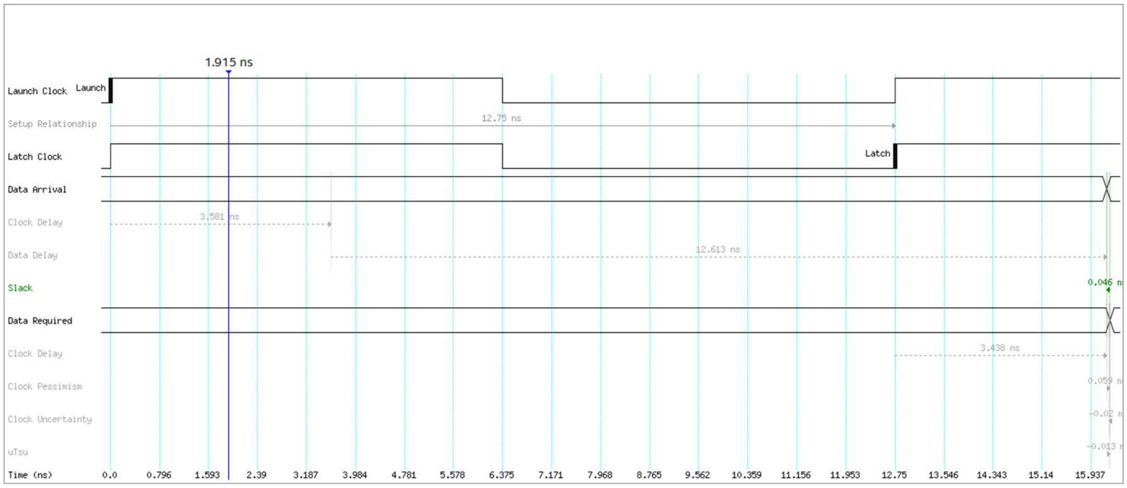

Figure 22. Bonus Setup Slack Analysis

Hold Slack Analysis:

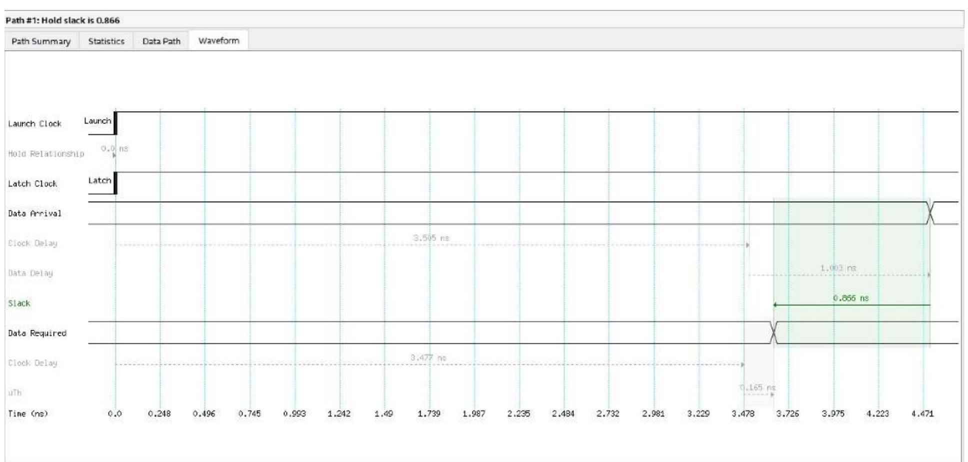

Figure 23. Single-Cycle Hold Slack Analysis

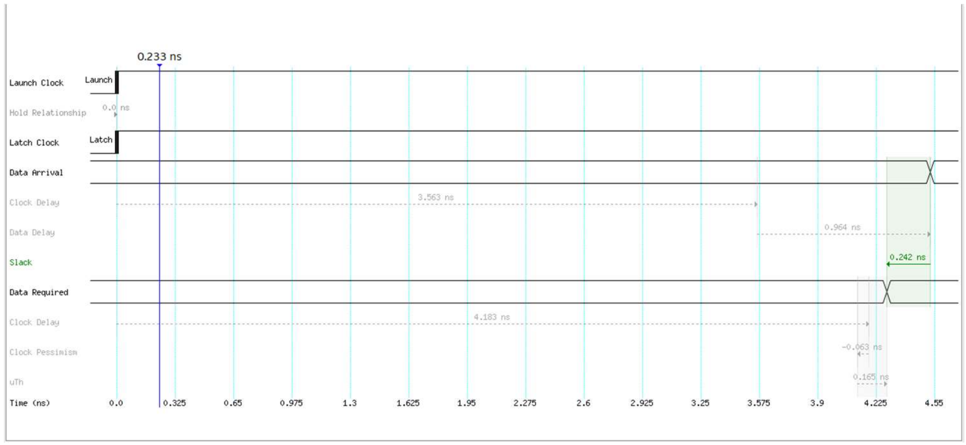

Figure 24. Pipelined Hold Slack Analysis

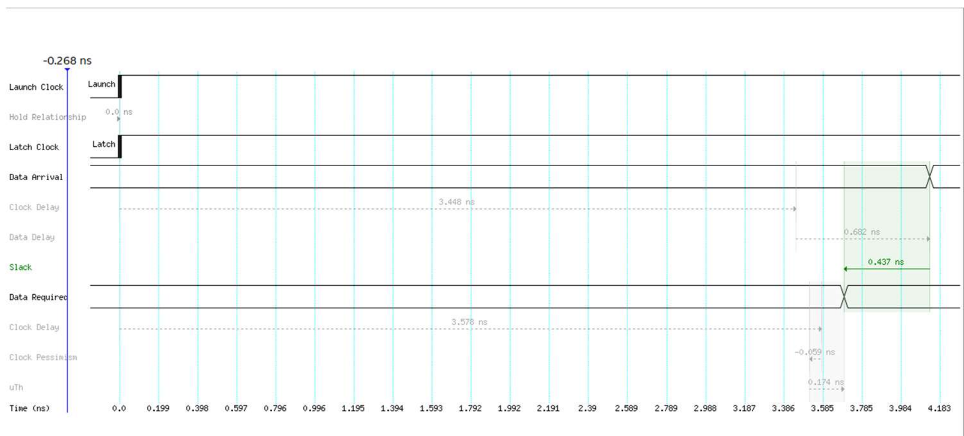

Figure 25. Bonus Hold Slack Analysis

Fmax Analysis:

Slow 1200mV 85C Model

	Fmax	Restricted Fmax	Clock Name
1	63.21 MHz	63.21 MHz	clk

Figure 26. Single-Cycle Fmax

	Fmax	Restricted Fmax	Clock Name
1	96.51 MHz	96.51 MHz	clk

Figure 27. Pipelined Fmax

Slow 1200mV 85C Model			
	Fmax	Restricted Fmax	Clock Name
1	78.72 MHz	78.72 MHz	clk

Figure 28. Bonus Fmax

Improvements from Pipelining

There is *no improvement* in latency from utilizing a pipelined design, and in fact it is worse under conditions where the pipeline is frequently flushed. However, the real potential benefit for a pipelined implementation comes from the increase in throughput of the system when the pipeline is kept full for much of the time.

Due to the decreased amount of logic between each set of launch and latch flipflops in the system, the clock frequency can be increased from **63.21 MHz** to **96.51 MHz** for the pipelined implementation, yielding an improvement in the Fmax metric of **~52.68%**. With full-forwarding and infrequent branch operations, the increase in performance is significant.

Hardware overhead related to pipelining

Table 1. Hardware Requirements for Single cycle vs. Pipelined Implementations

	Single Cycle	**Pipelined**
Logic Gates	1,862 gates	2,001 gates
Memory	16,384 Mem Elements	16,384 Mem Elements
Total Registers	1,054 registers	1,380 registers

The pipelined implementation requires a 7% increase in logic gates, and a 30.92% increase in total registers relative to the single-cycle implementation. The number of memory elements required remains the same for both implementations.

Performance Penalty of Hazard Detection and Forwarding Units

Table 2. Performance Penalty of Hazard Detection/Forwarding

	Single Cycle	**Pipelined**	**Pipelined W/Hazard & Forwarding Units**
Fmax	63.21 MHz	96.51 MHz	78.72 MHz

Implementing the hazard detection & forwarding units results in an 18.43% reduction in performance. However, even with the hazard and forwarding units installed, the processor is still 24.53% faster than the single cycle implementation. Therefore, it can potentially provide a boost in throughput performance under circumstances where the pipeline remains filled for most of the execution time of a given program.

Hardware Overhead of Hazard Detection and Forwarding Units

Table 2. Hardware Requirements for all 3 Implementations

	Single Cycle	**Pipelined**	**Pipelined W/Hazard & Forwarding Units**
Logic Gates	1,862 gates	2,001 gates	2,081 gates
Memory	16,384 Mem Elements	16,384 Mem Elements	16,384 Mem Elements
Total Registers	1,054 registers	1,380 registers	1,385 registers

For our design, implementing hazard detection and forwarding requires a 11.76% increase in logic gates relative to the single cycle implementation and a 3% increase in logic gates relative to the pipelined implementation. It requires a 31.4% increase in total registers relative to the single cycle implementation and a 0.36% increase in total registers relative to the pipelined implementation.