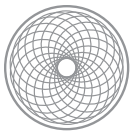


Nicholas Lippis

EC410 Multicycle CPU Project

Nicholas Lippis 33 Harry Aggains Way Boston, Massachusetts 02215 nlippis@bu.edu

Table of Contents

CPU Design

Datapath

The Datapath design is based on the MIPS Datapath, with changes due to instruction format and memory structure differences.

MIPS Instruction Format {R-Type}

Empty	RD	RS	RT	Shamt	Function

EC413 Instruction Format {R-Type}

Opcode bits [31:26]	R1 bits [25:21]	R2 bits [20:16]	R3 bits [15:0]	~~

Due to this change the path between the *Instruction Register* and the *Register File* has been changed. For logical I-Type instructions, there is a *zero extender* because *sign extension* will change the immediate's value while being converted from 16 to 32 bits. Secondly a *Shift* module was added, which is used for LUI (Load Upper Immediate), this will shift the *zero extended* value into the upper sixteen bits so that it is really a upper value.

In MIPS the instruction and data memory are one unit, however for the EC413 project they are two separate entities. To compensate for this the memwrite control line and the output of Read Data 1 from the *Instruction Register* was routed to the Data Memory (*DMem*). The MUX for the input to Memory Address was removed and now the program count value from the *PC* register is directly connected to the Instruction Memory (*IMem*). The Address for the *DMem* is now taken from the last 16 bits of the instruction, Zero Extended (explained later) and then inputed into the address of the *DMem*.

Elements

- Sequential Logic
  - **Registers**
    - PC (Program Counter) - Stores the current instruction reference [All instructions]
    - MDR (Memory Data Register) - Stores output of the *Data Memory* for one clock cycle so data is received in the register file on the next clock cycle, which synchronizes it with the write back stage of the controller. [Used for LWI]

- A (Read Data 1 Output) - Stores output of Read Data 1 from *Register File* so data arrives in the *ALU* one clock cycle later, which synchronizes it with the execution stage of the controller. [Used for R-Type instructions, I-Type instructions, LI & LUI]
- B (Read Data 2 Output) - Stores output of Read Data 2 from *Register File* so data arrives in the *ALU* one clock cycle later, which synchronizes it with the execution stage of the controller. [Used for R-Type instructions]
- ALUREG (ALU Output) - Stores output of ALU so data arrives at *Register File* one clock cycle later, which synchronizes it with write back stage of the controller. Secondly during branch target calculation the target is stored for one clock cycle so that if the instruction is a branch instruction, the CPU can decide whether to take the branch or not. [Used for R-Type instructions, I-Type instructions and Branch instructions]
- Register File - Stores values that are to be operated on, in a per program basis

- **Memory**
  - IMem (Instruction Memory) - Holds the instructions with numerical references (0,1,...N)
  - DMem (Data Memory) - Holds data values with with numerical references (0,1,...N)

- Combinational Logic
  - SE (Sign Extend) - Converts a 16 bit value into a 32 bit value, used for arithmetic functions where signed values need to be preserved. If most significant bit is set then ones are prepended onto the original 16 bits to create a 32 bit signed word. If most significant bit is not set then zeros are prepended onto the original 16 bits to create a 32 bit unsigned word. [Used for arithmetic I-Type instructions]
  - ZE (Zero Extend) - Converts a 16 bit value into a 32 bit value, used for logical and loading functions where (perceived) signed values do not need to be preserved. Zeros are prepended onto the original 16 bits to create a 32 bit unsigned word. [Used for Logical I-Type instructions, LI & LUI]
  - Shift - Converts a 32 bit zero extended value into a bottom zero extended 32 bit value. Used when upper half of a value is required by a function. Shift is implemented by shifting value by 16 bits, resulting in a value with the original 16 bits in the top half and zeros in the bottom half.
    [Used for LUI]
  - MUX (Multiplexor) - Receives a variable input and based on select value, outputs one of them
    - Reg1det (Register 1 Determine) - Receives R1 and R2. If select is 0, R1 is the output, if select is 1, R2 is output
    - Memtoreg (Memory to Register) - Receives the output of *ALUREG* and *MDR*. If select is 0, *ALUREG* is the output, if select is 1, *MDR* is the output
    - Edet (Extension Determine) - Receives the output of *Shift* and *Zero Extend*. If select is 0, *Shift* is the output, if select is 1, *Zero Extend* is the output.
    - Alusrcamux (ALU Source A MUX) - Receives the output of *PC* (Program Counter), register *A*, and Read Data 1 from the *Register File*. If select is 00, *PC* is the output, if select is 01, *A* is the output, lastly if select is 10, then Read Data 1 is the output. [*PC* is used for Branch Target Calculation. *A* is used for all R-Type, I-Type, Branch, & LI, LUI. Read Data 1 is used for Li, LUI, and Branch instructions]
    - Alusrcbmux (ALU Source B MUX) - Receives the output of *B*, one bit binary 1, *Sign Extend*, and *Edet*. If select is 00, *B* is the output, if select is 01, one bit binary 1 is the output, if select is 10, *Sign Extend* is the output, lastly if select is 11, *Edet* is the output. [*B* is used for R-Type instructions. One bit binary 1 is used for incrementing the program counter (dictated by

controller). *Sign Extend* is used for Arithmetic I-Type instructions, *Edet* is used for Logical I-Type and LI, LUI instructions]

- PCSource (Program Counter Source) - Receives output of *ALU,* Concatenation of PC [31:26] and Instruction [25:0], and *ALUREG. If select is 00,* ALU is the output, if select is 01, the Concatenation is the output, lastly if select is 10, *ALUREG* is the output. [*ALU* is used when the *PC* is being incremented. The Concatenation is used during Jump instructions. *ALUREG* is used during Branch calculation.]

- ALU (Arithmetic Logic Unit) - Receives two 32 bit words (R2 & R3) and performs an operation { listed below } based on a opcode supplied by the controller, and outputs a transformed value (R1).

ALU Operations

Function	Binary Encoding / Aluop (5 Bit)	Operation
MOV	10000	R1 = R2
NOT	10001	R1 = ~R2
ADD	10010	R1 = R2 + R3
SUB	10011	R1 = R2 - R3
OR	10100	R1 = R2 | R3
AND	10101	R1 = R2 & R3
XOR	10110	R1 = R2 ^ R3
SLT (Set Less Than)	10111	R1 = 1 If R2 < R3, else 0 the less than or equal to function does not work when either R2 or R3 is a signed value. So if the most significant bit of either R2 or R3 is set then the less than is flipped to a greater than which will then handle the operation in the correct way.
BEQZ (Branch Equal to Zero)	0	If R1 == 0 then branch = 1, else branch = 0
BNEZ (Branch Not Equal to Zero)	1	If R1 != 0 then branch = 1, else branch = 0
LI (Load Immediate)	11001	Concatenate R2[31:16] and R3[15:0]
LUI (Load Upper Immediate)	11100	Concatenate R3[31:16] and R2[15:0]

Instruction Path [Instructions are traced from *Instruction Register to Write Back]*

- J (Jump) - Instruction Bits [25:0] are concatenated with PC [31:16] to create a usable 32 bit instruction. It is then passed into PC Source MUX, the select value 01 is given which allows it to pass to the *PC*. PCWrite is then set to high which writes the new pc value into the *PC*.
- R-Type Instruction
  - Data Fetch & Transformation
    - Reg1det MUX is set to 1 to allow the R2 value to pass into the *Register File* along with R3. The values of Read Data 1 & 2 are passed into registers *A & B* respectively. The output of register *A* is passed to *Alusrca MUX* with select 01. The output of register *B is passed to Alusrcb MUX* with select 00.
  - Compute
    - These values from Alusrc a&b MUX are passed into the *ALU* with the instructions respective ALUOP.
  - Write Back
    - The output of *ALU* is passed to *ALUREG* in order to delay its arrival to the *Register File*. The output of ALUREG is then passed to Memtoreg MUX with select 0 in order to pass the value into the Write Data input of *Register File*. The R1 value from the instruction is also passed into the Write Reg input of the *Register File*, and lastly the regfile bit is set, which allows the value to be written to the address specified in the R1 field. [Possible instructions MOV, NOT, ADD, SUB, OR, AND, XOR, SLT]
- Arithmetic I-Type Instruction (Immediate)
  - Data Fetch & Transformation
    - Reg1det MUX is set to 1 to allow the R2 value to pass into the *Register File*. The value of Read Data 1 is the passed into register *A*, which is then passed into Alusrca with select 01. Because the second argument for the instruction is an immediate it must take a different path. The value of Instruction [15:0] is passed into the *Sign Extend* module in order to transform it into a 32 bit word (the only size that the *ALU* will accept). The sign extended value is then passed into Alusrcb MUX with select 10.
  - Compute
    - The values from Alusrc a&b are then passed into the ALU along with the respective ALUOP of the instruction.
  - Write Back
    - The transformed value is then passed to *ALUREG*, which is then passed to Memtoreg MUX along with select 0. The value is then wrote into the *Register File* with the R1 field from the instruction passed to the Write Reg input of *Register File*, the output of Memtoreg MUX is passed to the Write Data input of *Register File* and lastly the regfile bit is set which allows the value to be written to *Register File*. [Possible instructions ADDI, SUBI]
- Logical I-Type Instruction (Immediate)
  - Data Fetch & Transformation
    - Reg1det MUX is set to 1 to allow the R2 value to pass into the *Register File*. The value of Read Data 1 is the passed into register *A*, which is then passed into Alusrca with select 01. Because the second argument for the instruction is an immediate it must take a different path. The value

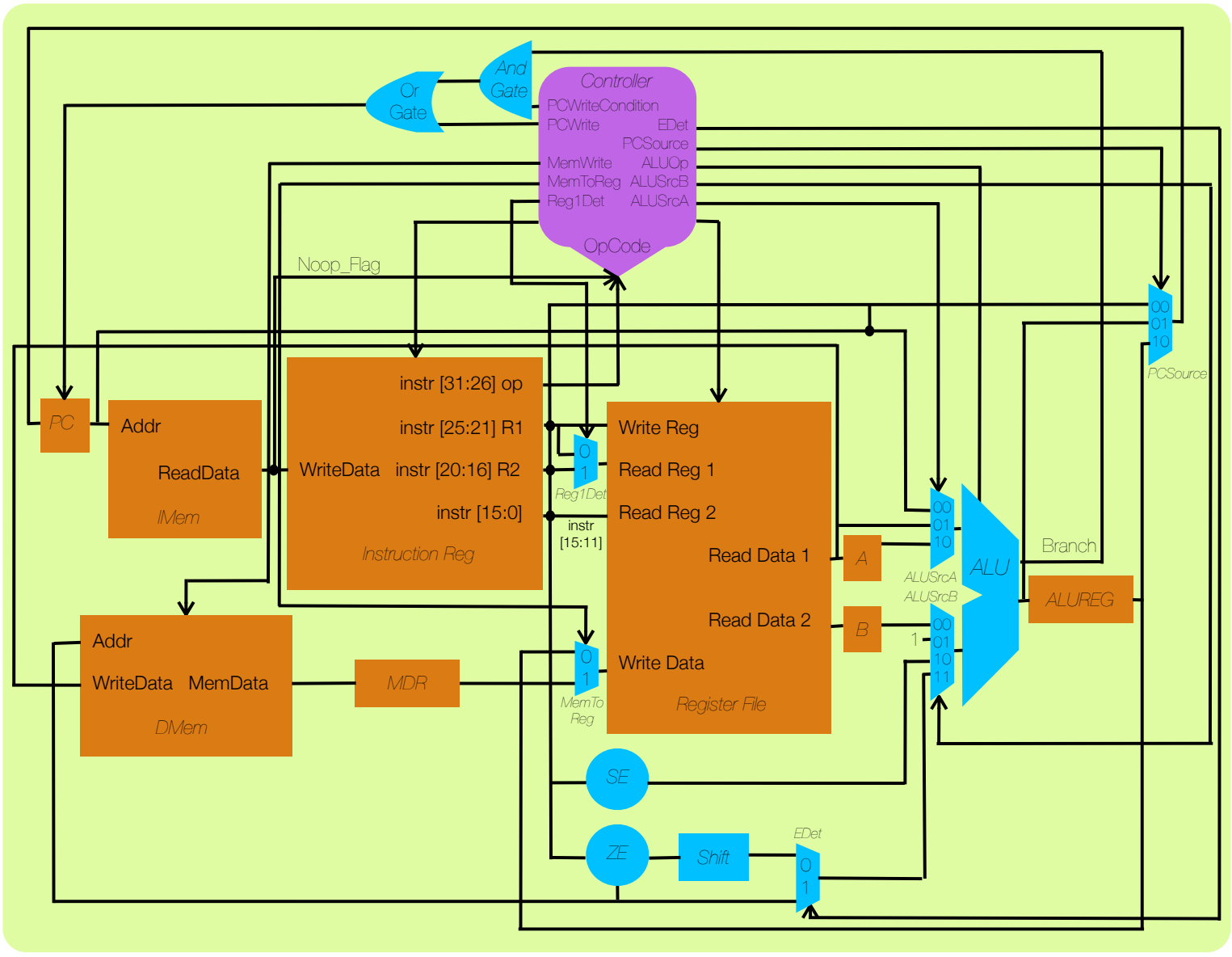
of Instruction [15:0] is passed into the *Zero Extend* module in order to transform it into a 32 bit word (the only size that the *ALU* will accept). The zero extended value is then passed into Edet MUX with select 1. The edet value is then passed into Alusrcb MUX with select 11.

- Compute
  - The values from Alusrc a&b are then passed into the ALU along with the respective ALUOP of the instruction.
- Write Back
  - The transformed value is then passed to *ALUREG*, which is then passed to Memtoreg MUX along with select 0. The value is then wrote into the *Register File* with the R1 field from the instruction passed to the Write Reg input of *Register File*, the output of Memtoreg MUX is passed to the Write Data input of *Register File* and lastly the regfile bit is set which allows the value to be written to *Register File*. [Possible instructions ORI, ANDI, XORI, SLTI]

- Branch Instruction
  - Data Fetch & Transformation
    - Reg1det MUX is set to 1 to allow the R2 value to pass into the *Register File*. The value of Read Data 1 is the passed into register *A*, which is then passed into Alusrca with select 10.
  - Compute
    - The value from Alusrca is then passed into the ALU along with the respective ALUOP of the instruction.
  - Branch
    - Before any instruction takes place, a Branch Target is always calculated in case that a branch needs to be taken. If the conditions of the Branch Instruction are met, then the Branch bit will be set from the *ALU* which along with the PCWriteCondition bit being set, will cause the new value of the program counter to be written to the *PC.* [Possible instructions BEQZ, BNEZ]
- LI (Load Immediate)
  - Data Fetch & Transformation
    - Reg1det MUX is set to 0 to allow the R1 value to pass into the *Register File* (We wish to replace the lower 16 bits of the R1 value with the immediate). The value of Read Data 1 is the passed into register *A*, which is then passed into Alusrca with select 10. Because the second argument for the instruction is an immediate it must take a different path. The value of Instruction [15:0] is passed into the *Zero Extend* module in order to transform it into a 32 bit word (the only size that the *ALU* will accept). The zero extended value is then passed to Edet MUX with select 1. The edet value is then passed into Alusrcb MUX with select 11.
  - Compute
    - The values from Alusrc a&b are then passed into the ALU along with the ALUOP 11001 (LI).
  - Write Back
    - The transformed value is then passed to *ALUREG*, which is then passed to Memtoreg MUX along with select 0. The value is then wrote into the *Register File* with the R1 field from the instruction passed to the Write Reg input of *Register File*, the output of Memtoreg MUX is passed to the Write Data input of *Register File* and lastly the regfile bit is set which allows the value to be written to *Register File*. [Possible instructions ORI, ANDI, XORI, SLTI]

- LUI (Load Upper Immediate)
  - Data Fetch & Transformation
    - Reg1det MUX is set to 0 to allow the R1 value to pass into the *Register File* (We wish to replace the upper 16 bits of the R1 value with the immediate). The value of Read Data 1 is the passed into register *A*, which is then passed into Alusrca with select 10. Because the second argument for the instruction is an immediate it must take a different path. The value of Instruction [15:0] is passed into the *Zero Extend* module in order to transform it into a 32 bit word (the only size that the *ALU* will accept). The zero extended value is then passed to *Shift* in order to move the lower 16 bits into the upper 16. The shifter value is then passed to the Edet MUX with select 0. The edet value is then passed into Alusrcb MUX with select 11.
  - Compute
    - The values from Alusrc a&b are then passed into the ALU along with the ALUOP 11010 (LUI).
  - Write Back
    - The transformed value is then passed to *ALUREG*, which is then passed to Memtoreg MUX along with select 0. The value is then wrote into the *Register File* with the R1 field from the instruction passed to the Write Reg input of *Register File*, the output of Memtoreg MUX is passed to the Write Data input of *Register File* and lastly the regfile bit is set which allows the value to be written to *Register File*.
- LWI (Load Word Immediate)
  - Data Fetch & Transformation
    - Instruction [15:0] is passed into the *Zero Extend* module in order to transform it into a 32 bit word (the only size that can be used to reference a address in the *DMem*). The zero extended value is then passed into the Address input of the *DMem*. The value stored at the previously referenced address is then passed to the input of the *MDR*. The value is then passed to the Memtoreg MUX with select 1.
  - Write Back
    - The transformed value is then passed to *ALUREG*, which is then passed to Memtoreg MUX along with select 1. The value is then wrote into the *Register File* with the R1 field from the instruction passed to the Write Reg input of *Register File*, the output of Memtoreg MUX is passed to the Write Data input of *Register File* and lastly the regfile bit is set which allows the value to be written to *Register File*.
- SWI (Store Word Immediate)
  - Data Fetch, Transformation & Memory Write
    - Reg1det MUX is set to 0 to allow the R1 value to pass into the *Register File*. The value of Read Data 1 is the passed into the Write Data input of *DMem.* Instruction [15:0] is passed into the *Zero Extend* module in order to transform it into a 32 bit word (the only size that can be used to reference a address in the *DMem*). The zero extended value is then passed into the Address input of the *DMem*. The memwrite bit is then set in order to write the value into the *DMem*.

Datapath Diagram



Controller

States

- State 0 - Increment Program Counter + 1 to access next instruction
  - IrWrite = 1
  - Reg1Det = 1
  - ALUSrcA = 00
  - ALUSrcB = 01
  - ALUOP = 10010 (ADD)
  - PCSrc = 00
  - PCWrite = 1
  - Go to State 1
- State 1 - Compute Branch Target & Instruction Determine
  - Reg1Det = 1
  - ALUSrcA = 00
  - ALUSrcB = 10
  - ALUOP = 10010 (ADD)
  - Next State
    - if NOOP go to State 0
    - If R-Type go to State 2
    - If Arithmetic I-Type go to State 3
    - If Logical I-Type go to State 4
    - If Branch Instruction go to State 5
    - If Jump Instruction go to State 6
    - If Load Immediate go to State 7
    - If Load Word Immediate go to State 8
    - If Load Upper Immediate go to State 9

    - If Store Word Immediate go to State 10

- State 2 - R-Type [Data Fetch, Transformation, & Compute]
    - RegDet = 1
    - ALUSrcA = 01
    - ALUSrcB = 00
    - ALUOP = OPCODE [4:0]
    - Go to State 11
- State 3 - Arithmetic I-Type [Data Fetch, Transformation, & Compute]
    - RegDet = 1
    - ALUSrcA = 01
    - ALUSrcB = 10
    - ALUOP = OPCODE [4:0]
    - Go to State 11
- State 4 - Logical I-Type [Data Fetch, Transformation, & Compute]
    - RegDet = 1
    - EDet = 1
    - ALUSrcA = 01
    - ALUSrcB = 11
    - ALUOP = OPCODE [4:0]
    - Go to State 11
- State 5 - Branch Instruction [Data Fetch, Transformation, Compute, & Branch]
    - RegDet = 0
    - ALUSrcA = 10
    - ALUOP = OPCODE [4:0]
    - PCSrc = 10
    - PCWriteCond = 1
    - Go to State 0

- State 6 - Jump [Write Back]
  - PCWrite = 1
  - PCSource = 01
  - Go to State 0
- State 7 - Load Immediate [Data Fetch, Transformation, & Compute]
  - RegDet = 0
  - EDet = 1
  - ALUSrcA = 01
  - ALUSrcB = 11
  - ALUOP = OPCODE [4:0]
  - Go to State 11
- State 8 - Load Word Immediate [Data Fetch, Transformation, & Compute]
  - MemToReg = 1
  - Go to State 0
- State 9 - Load Upper Immediate [Data Fetch, Transformation, & Compute]
  - RegDet = 0
  - EDet = 0
  - ALUSrcA = 01
  - ALUSrcB = 11
  - ALUOP = OPCODE [4:0]
  - Go to State 11
- State 10 - Store Word Immediate [Data Fetch, Transformation, & Write Back]
  - RegDet = 0
  - MemToReg = 1
  - Go to State 0
- State 11 - Write Back

- RegFile = 1
- MemToReg = 0
- Go to State 0

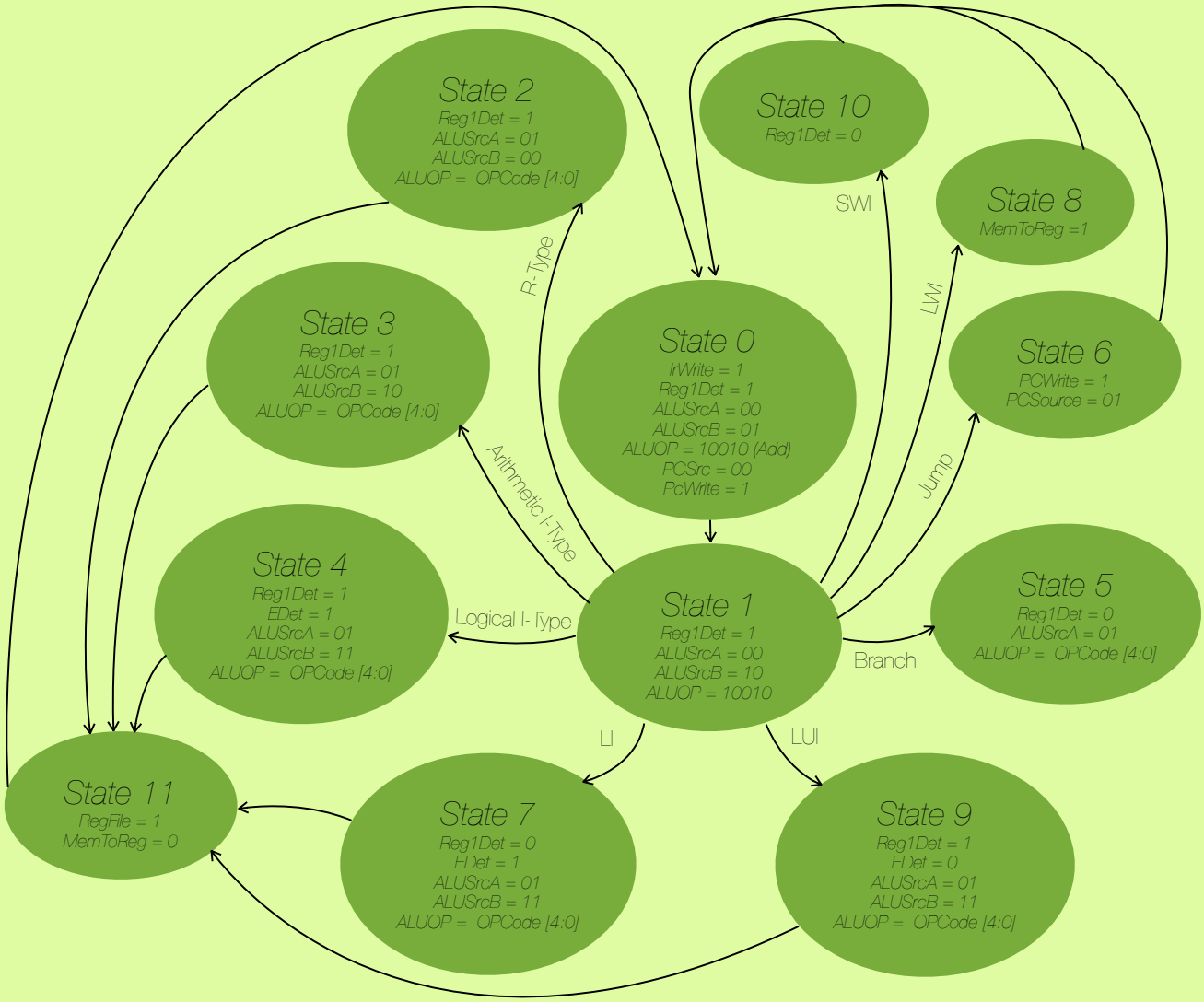
Instruction Path [Instructions are traced to and from State 0]

- Jump
  - State 1 Compute branch target in case of Branch Instruction
  - State 6 Concatenate PC [31:26] and Instr [25:0], Write value to *PC* register
  - State 0 Increment program counter
- R-Type
  - State 1 Compute branch target in case of Branch Instruction
  - State 2 Fetch Data from *Instruction Register*, Compute
  - State 11 Write back to *Instruction Register*
  - State 0 Increment program counter
- Arithmetic I-Type
  - State 1 Compute branch target in case of Branch Instruction
  - State 3 Fetch Data from *Instruction Register*, convert immediate to 32 bit sign extended, Compute
  - State 11 Write back to *Instruction Register*
  - State 0 Increment program counter
- Logical I-Type
  - State 1 Compute branch target in case of Branch Instruction
  - State 4 Fetch Data from *Instruction Register*, convert immediate to 32 bit zero extended, Compute
  - State 11 Write back to *Instruction Register*
  - State 0 Increment program counter
- Branch Instruction
  - State 1 Compute branch target in case of Branch Instruction
  - State 5 Fetch Data from *Instruction Register,* Compute if Branch should be taken, Set *PC* to write back
  - State 0 Increment program counter
- Load Immediate
  - State 1 Compute branch target in case of Branch Instruction
  - State 7 Fetch Data from *Instruction Register*, convert immediate to 32 bit zero extended, Compute

- State 11 Write back to *Instruction Register*
- State 0 Increment program counter

- Load Word Immediate
  - State 1 Compute branch target in case of Branch Instruction
  - State 8 Convert immediate to 32 bit zero extended,
  - State 0 Increment program counter
- Load Upper Immediate
  - State 1 Compute branch target in case of Branch Instruction
  - State 9 Fetch Data from *Instruction Register*, convert immediate to 32 bit zero extended & shifted, Compute
  - State 11 Write back to *Instruction Register*
  - State 0 Increment program counter
- Store Word Immediate
  - State 1 Compute branch target in case of Branch Instruction
  - State 10 Fetch Data from *Instruction* Register*,* convert immediate to 32 bit zero extended, Write to memory
  - State 0 Increment program counter

Controller Diagram



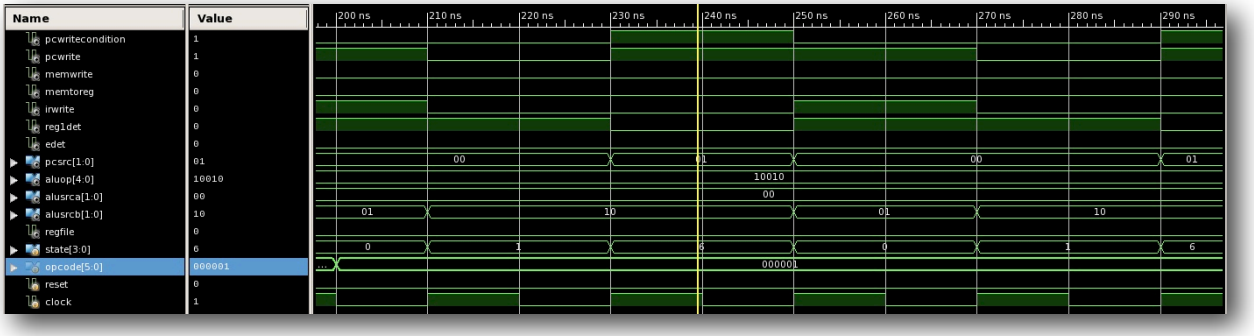
Controller Implementation

Waveforms

The following Waveforms, demonstrate that based on an opcode, the controller will be able to follow the state diagram and output the correct control line bits for each state.

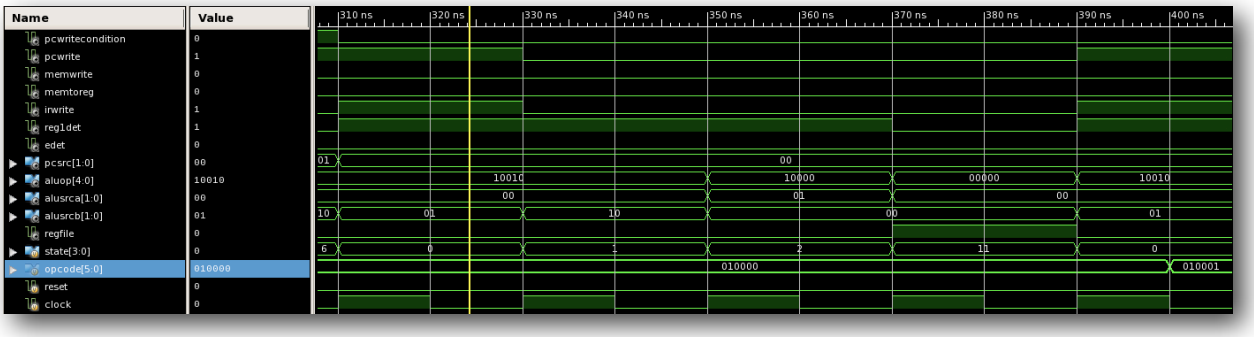
Jump Instruction - Opcode 000001

State Path [0 1 6 0]



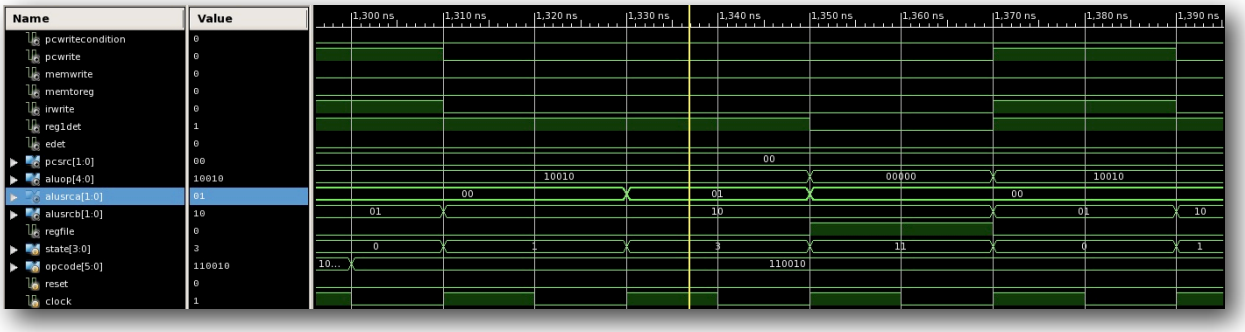
R-Type Instruction - Opcodes {010000 through 010111}

State Path [0 1 2 11 0]



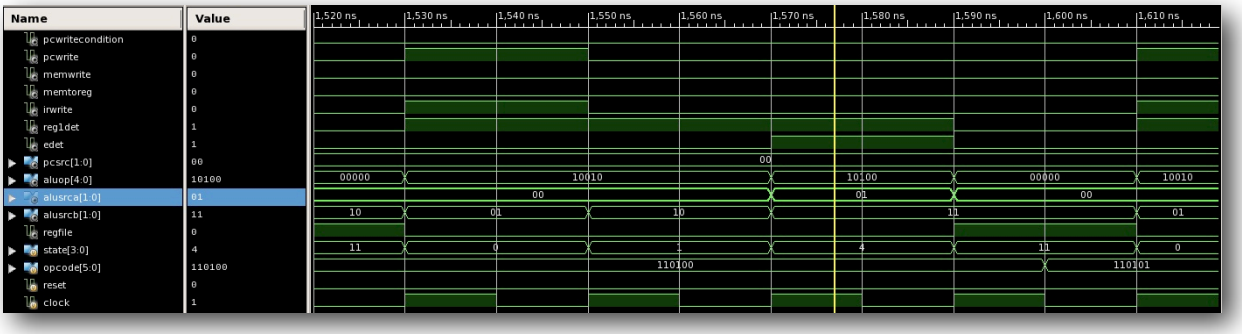
Arithmetic I-Type Instruction - Opcodes {110010 through 110100 }

State Path [0 1 3 11 0]



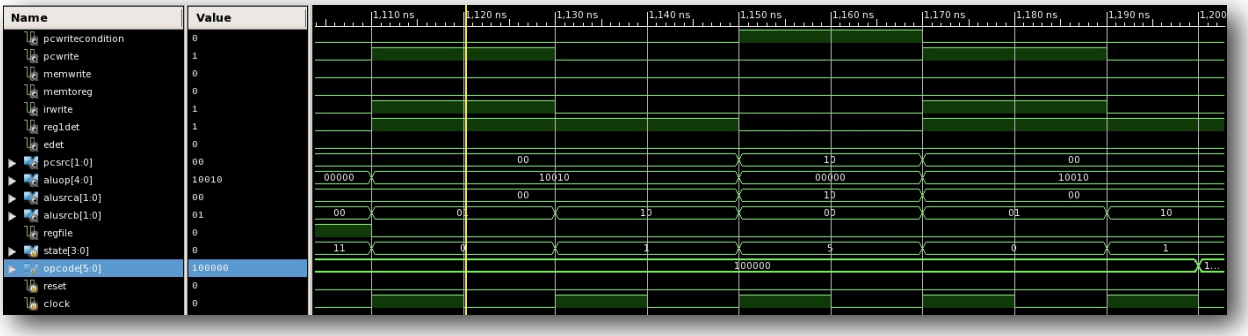
Logical I-Type Instruction - Opcodes {110100 through 110111 }

State Path [0 1 3 11 0]



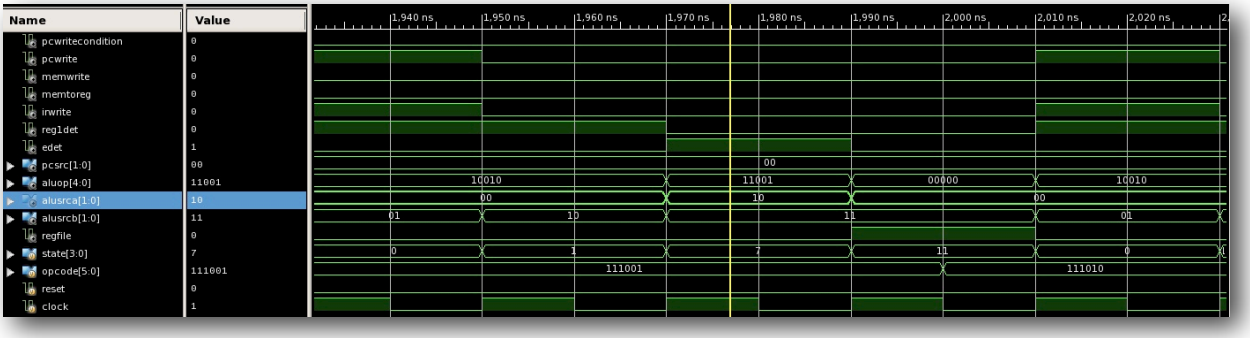
Branch Instruction - Opcodes {100000 through 100001}

State Path [0 1 5 0]



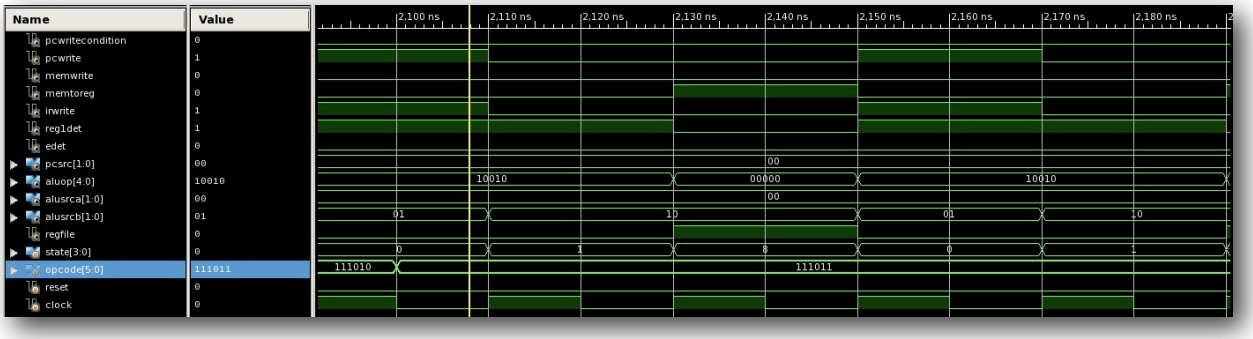
Load Immediate - Opcode 111001

State Path [0 1 7 11 0]



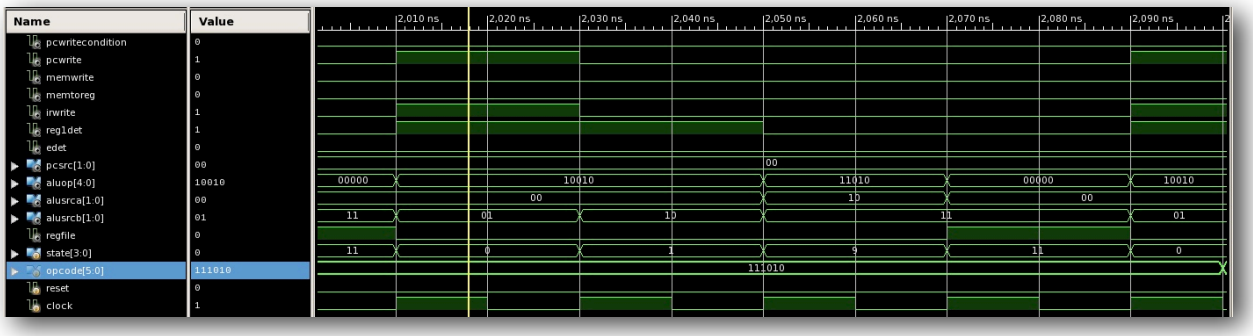
Load Word Immediate - Opcode 111011

State Path [0 1 8 0]



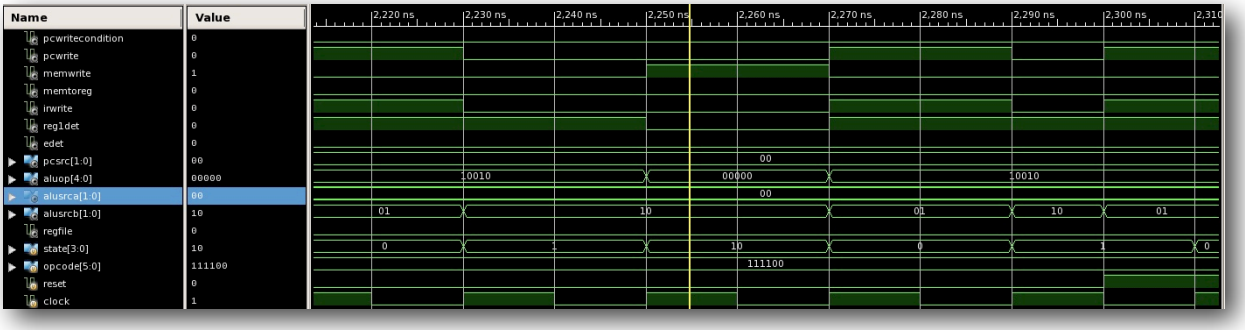
Load Upper Immediate - Opcode 111010

State Path [0 1 9 11 0]



Store Word Immediate - Opcode 111100

State Path [0 1 10 0]

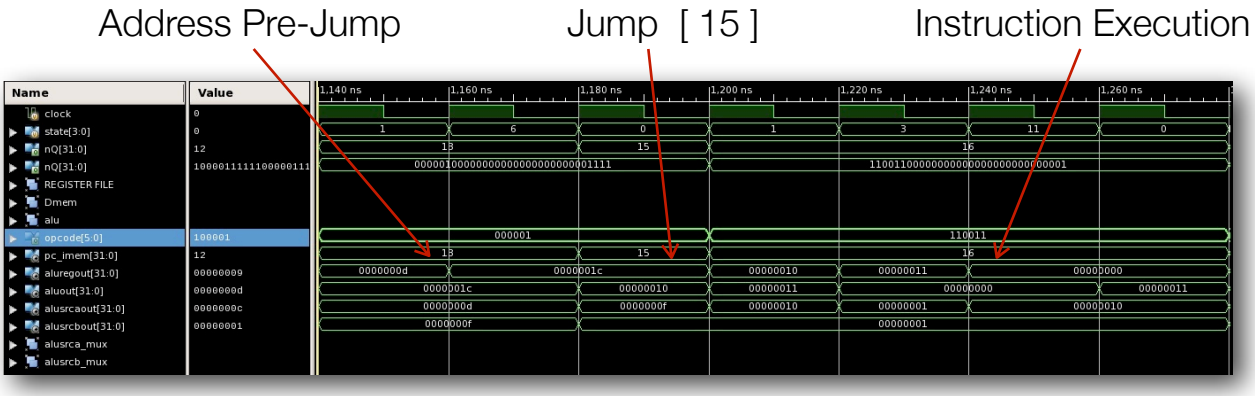


CPU Implementation

Waveforms

For all Arithmetic and Logical functions [R-Type, Arithmetic I-Type, Logical I-Type, LI, LUI, LWI] the result is stored in the Write Signal input of the Register File, shown in the waveforms below at state 11. For all Jump and Branch instructions you can see the value of pc_imem jump from its current value then to its target for a clock cycle, and then to the next instruction (the instructions are biased by one, so if pc_imem is executing instuction 15, it is actually executing 14 the Instruction Memory. Lastly for Store Word Immediate the result is stored in Write Data of the Data Memory.

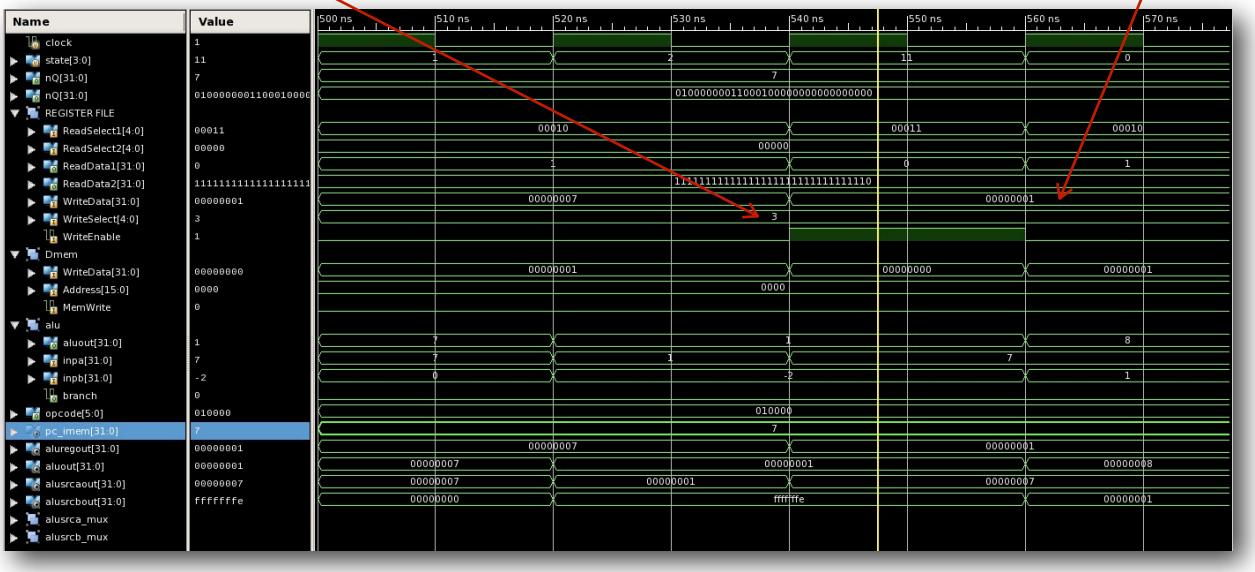
Jump Instruction | J 15 |



MOV Instruction | MOV $R3, $R2 |

R2 = 0x00000001

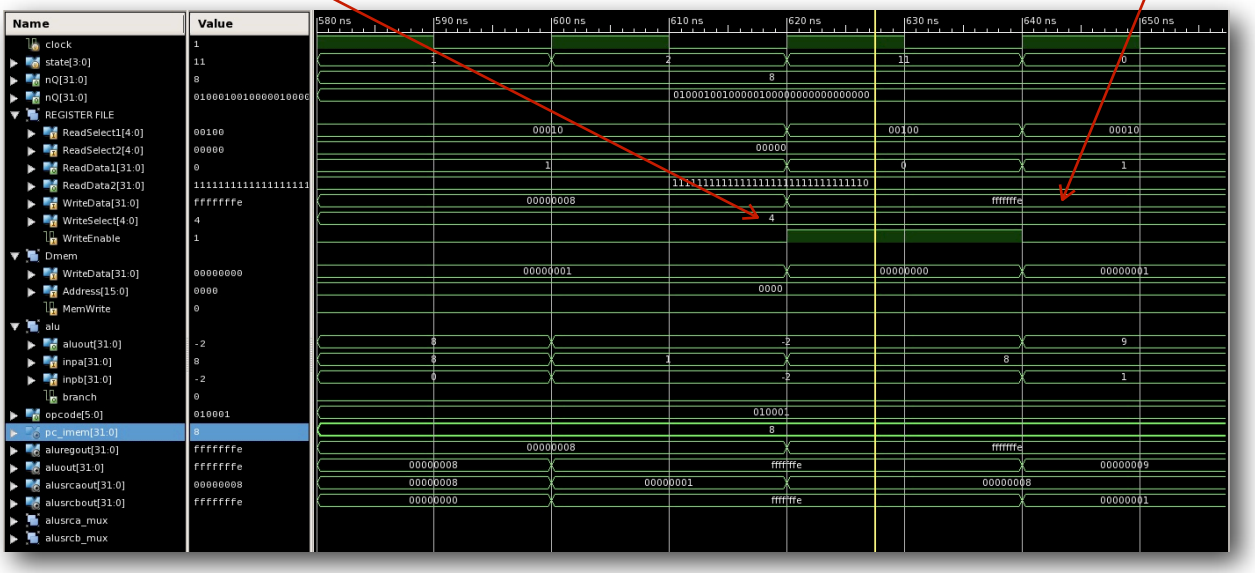
Address = 3 Write Data = 0x00000001



NOT Instruction | NOT $R4, $R2 |

R2 = 0x0000001

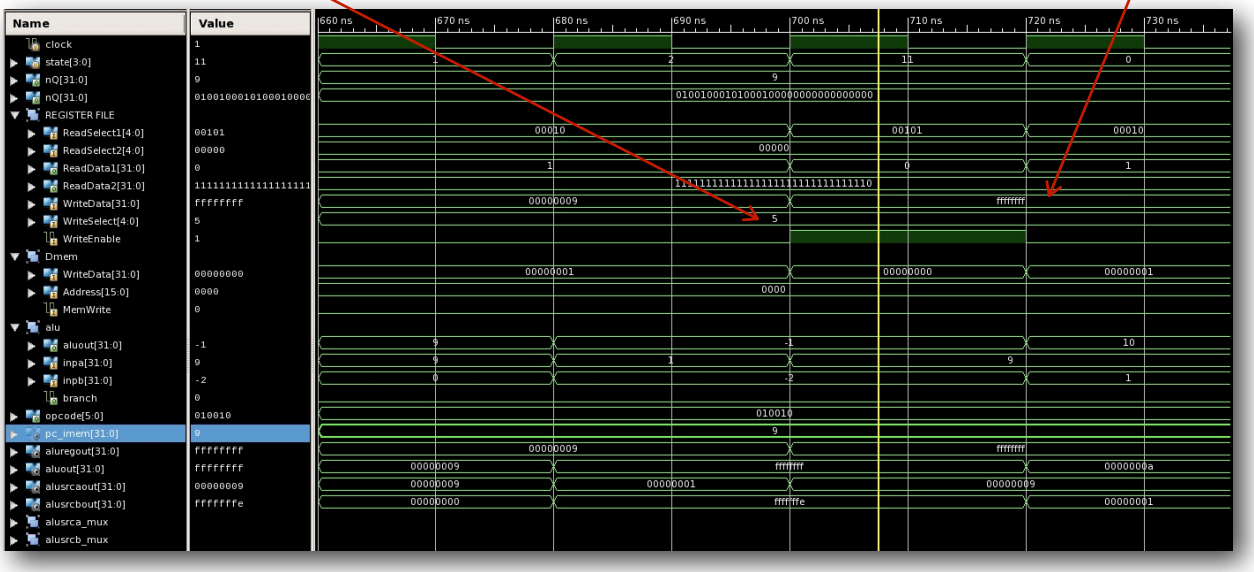
Address = 4 Write Data = 0xFFFFFFFE



ADD Instruction | ADD $R5, $R2, $R0 |

R2 = 0x00000001 R0 = 0xFFFFFFFE

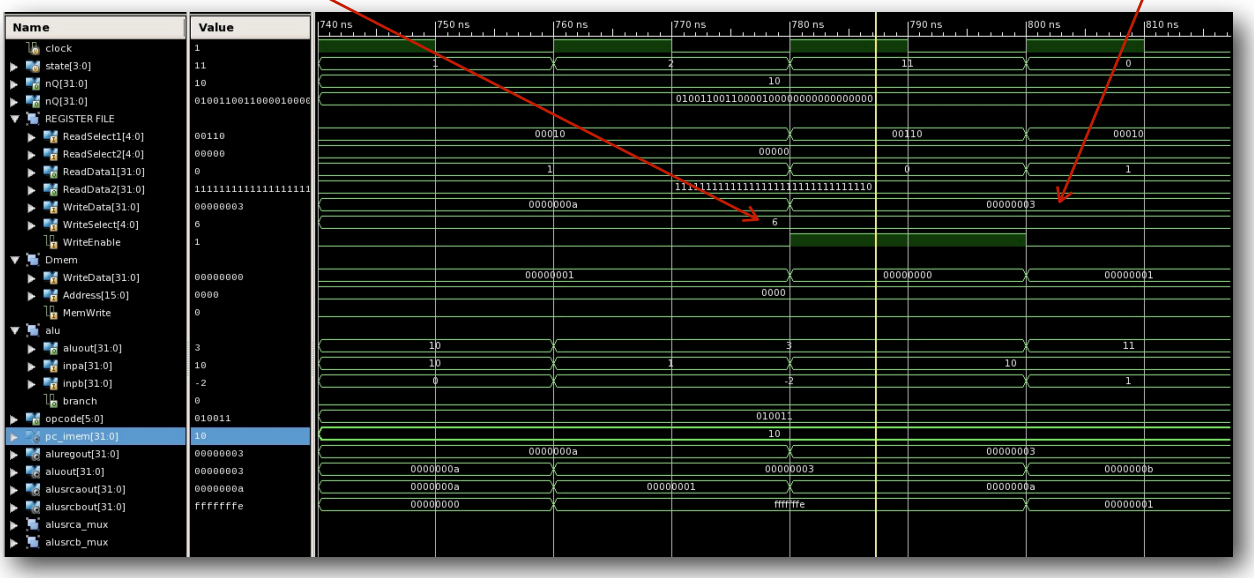
Address = 5 Write Data = 0xFFFFFFFF



SUB Instruction | SUB $R6, $R2, $R0 |

R2 = 0x00000001 R0 = 0xFFFFFFFE

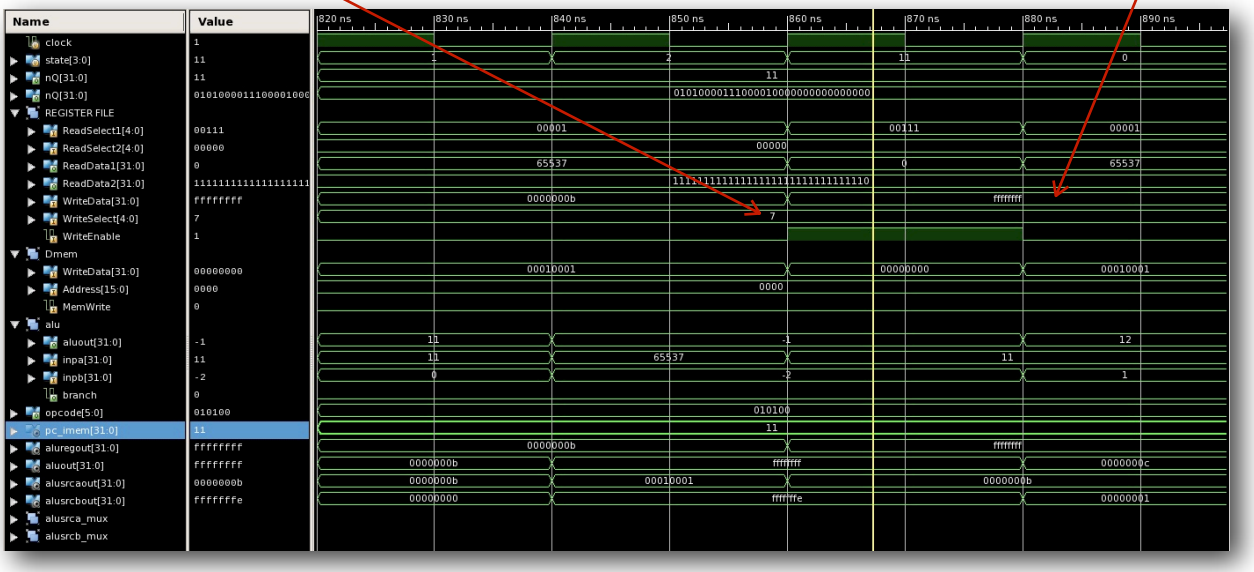
Address = 6 Write Data = 0x00000003



OR Instruction | OR $R7, $R1, $R0 |

R1 = 0x00010001 R0 = 0xFFFFFFFE

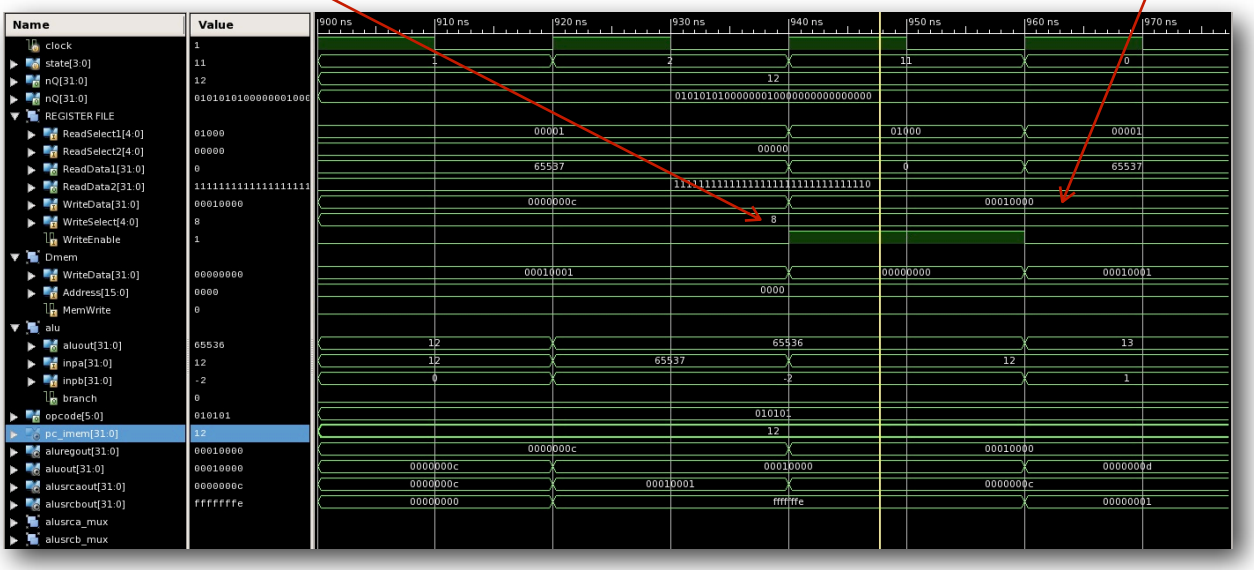
Address = 7 Write Data = 0xFFFFFFFF



AND Instruction | AND $R8, $R1, $R0 |

R1 = 0x00010001 R0 = 0xFFFFFFFE

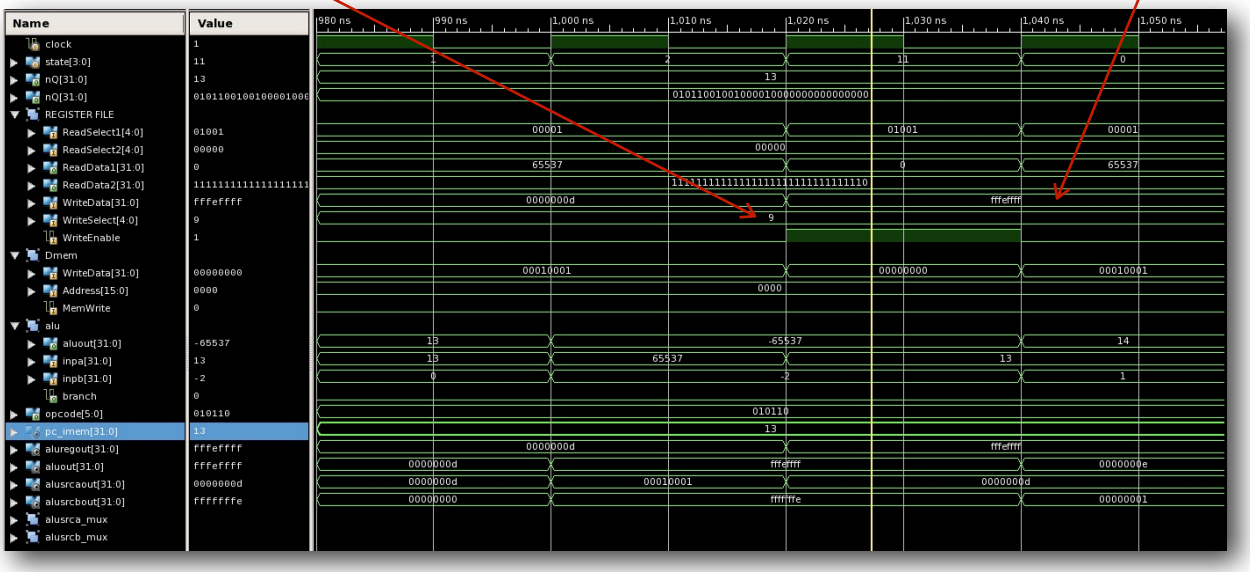
Address = 8 Write Data = 0x00010000



XOR Instruction | XOR $R9, $R1, $R0 |

R1 = 0x00010001 R0 = 0xFFFFFFFE

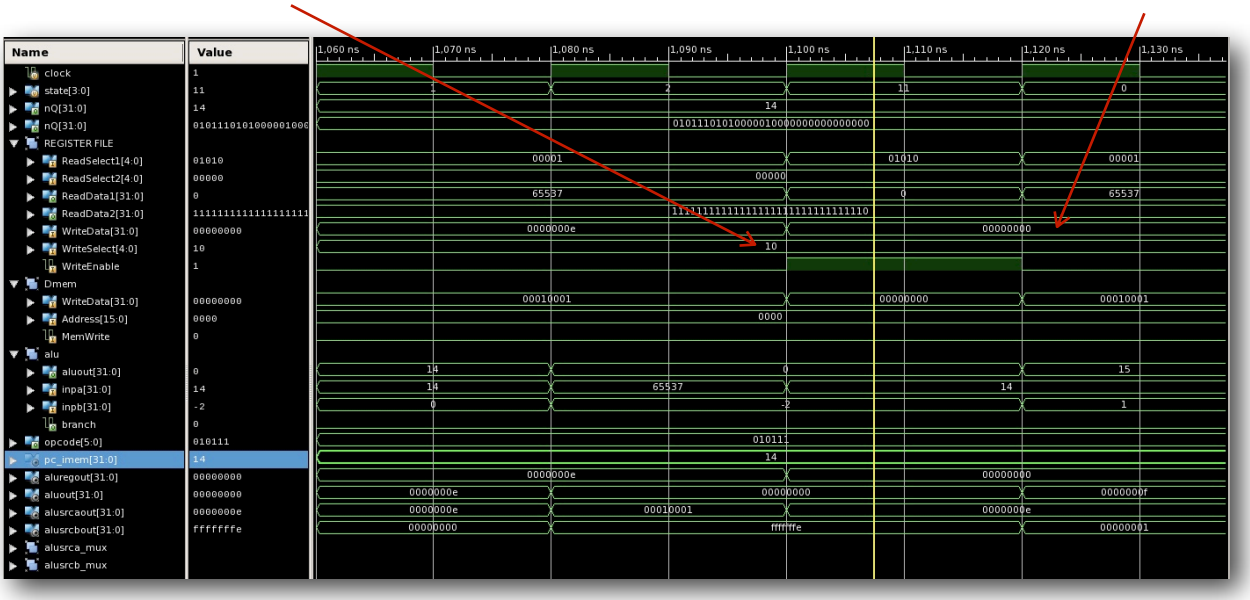
Address = 9 Write Data = 0xFFFEFFFF



SLT Instruction | SLT $R10, $R1, $R0 |

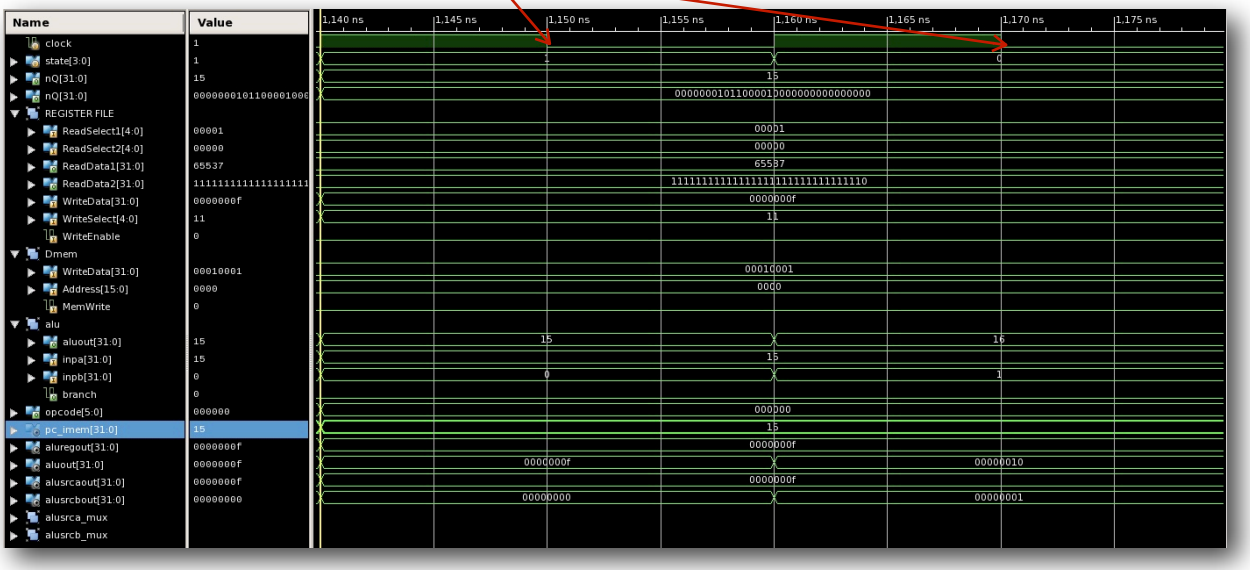
R1 = 0x00010001 R0 = 0xFFFFFFFE

Address = 10 Write Data = 0x00000000



NOOP Instruction

Takes Two Clock Cycles

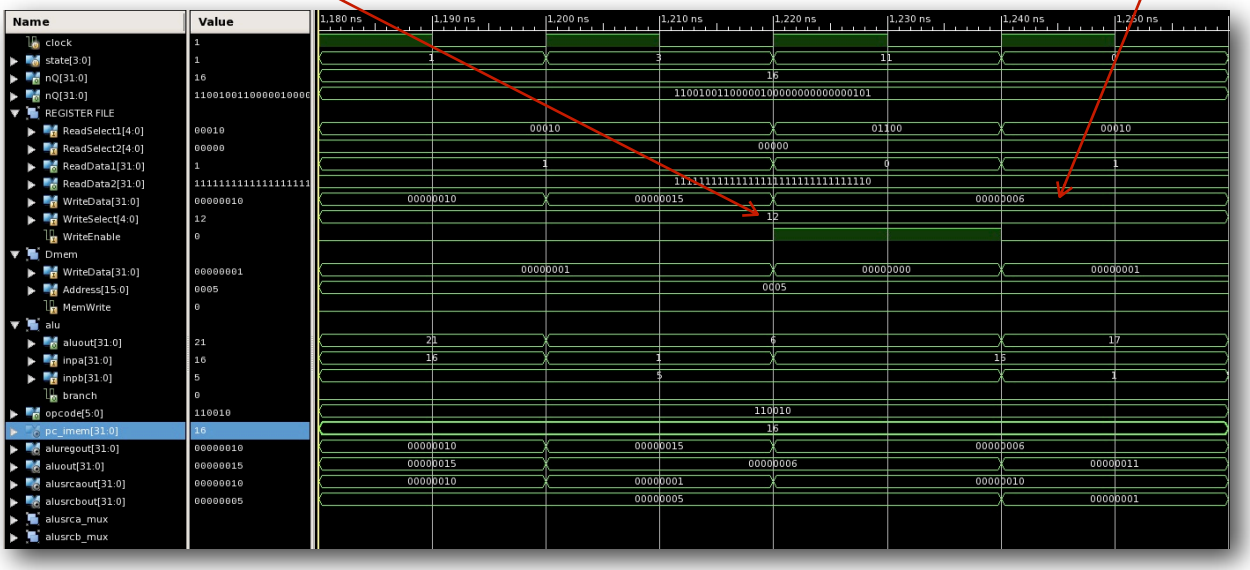


ADDI Instruction | ADDI $R12, $R2, 5 |

R2 = 0x00000001

Address = 12

Write Data = 6

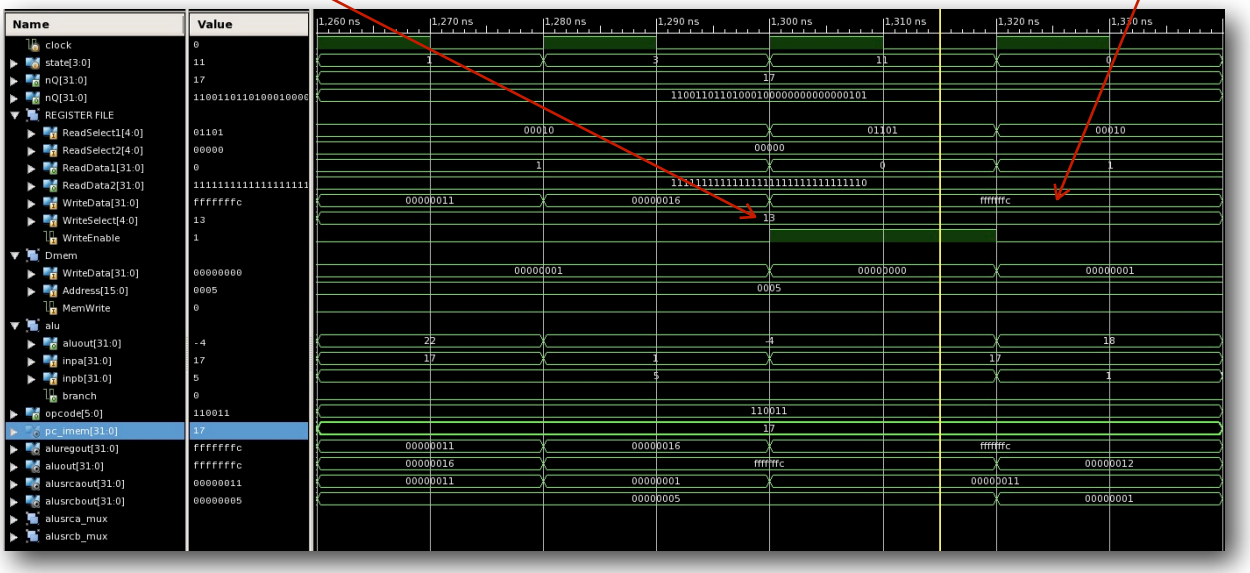


SUBI Instruction | SUBI $R13, $R2, 5 |

R2 = 0x00000001

Address = 13

Write Data = -4

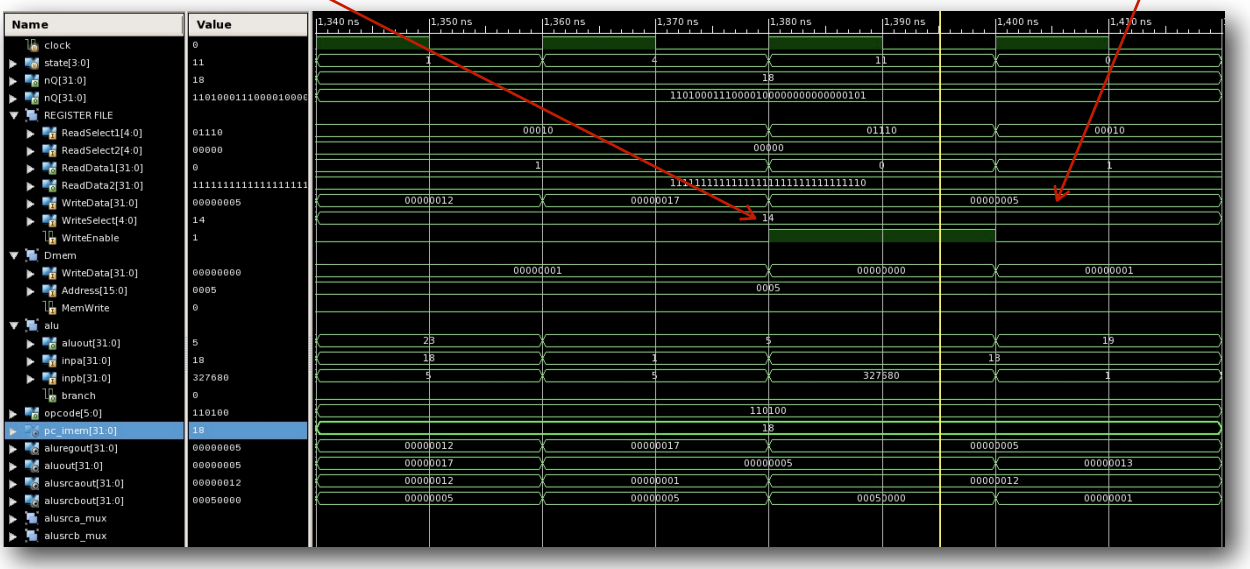


ORI Instruction | ORI $R14, $R2, 5 |

R2 = 0x00000001

Address = 14

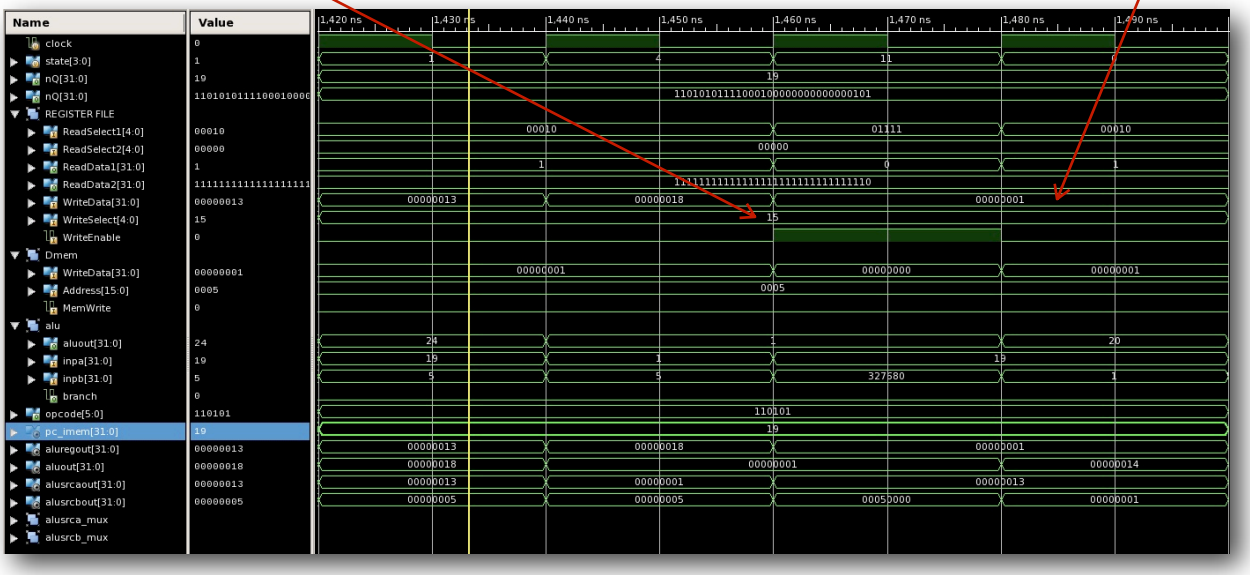
Write Data = 0x00000005



ANDI Instruction | ANDI $R15, $R2, 5 |

R2 = 0x00000001

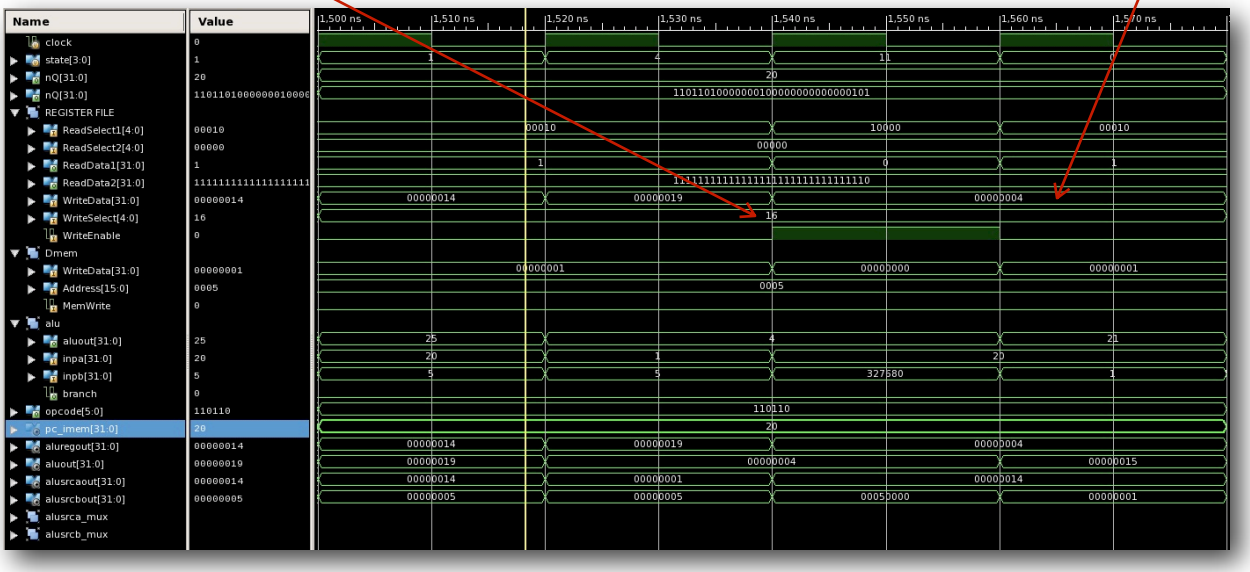
Address = 15 Write Data = 0x00000001



XORI Instruction | XORI $R16, $R2, 5 |

R2 = 0x00000001

Address = 16 Write Data = 0x00000004

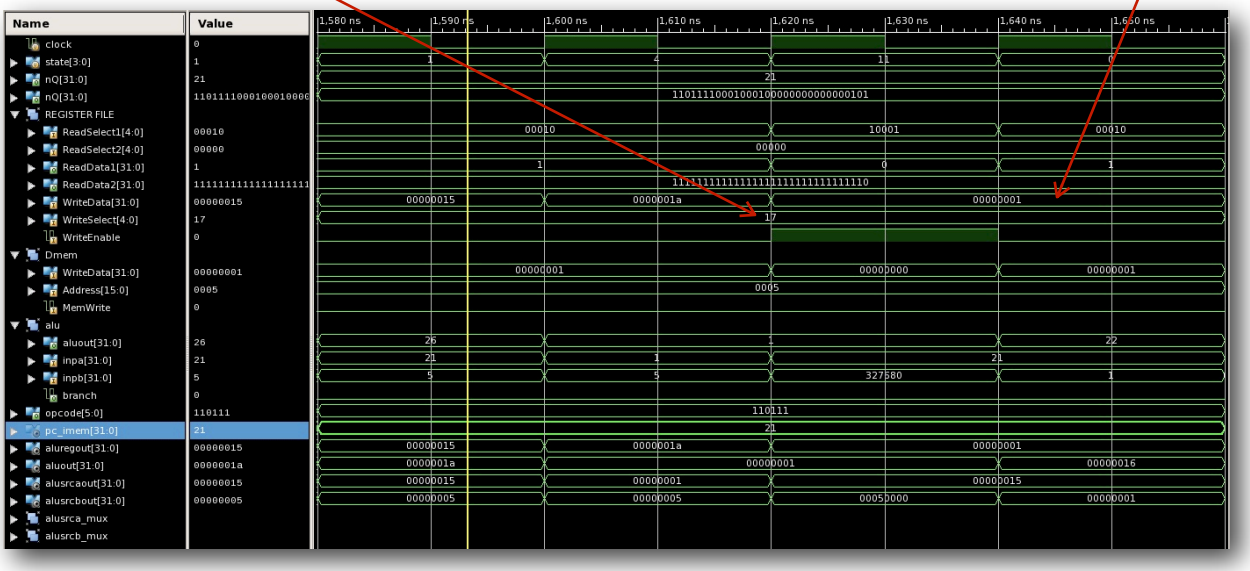


SLTI Instruction | SLTI $R17, $R2, 5 |

R2 = 0x00000001

Address = 17

Write Data = 0x00000001

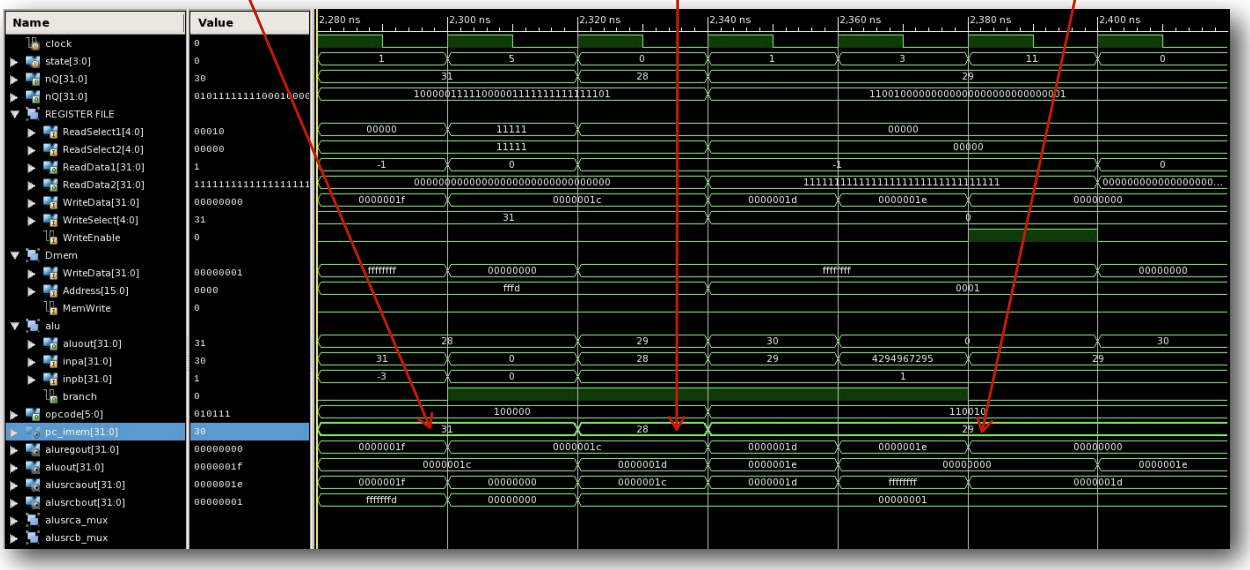


BEQZ Instruction | BEQZ $R31, 0xFFFD|

Address Pre-Branch

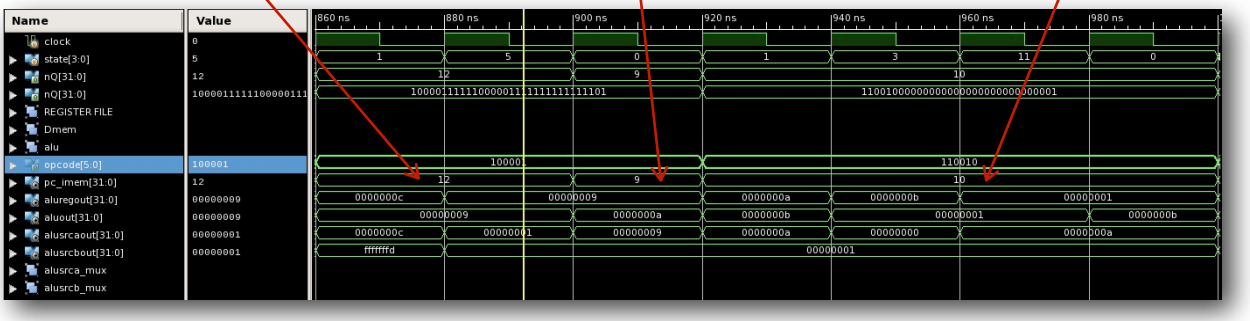
Branch [28]

Instruction Execution



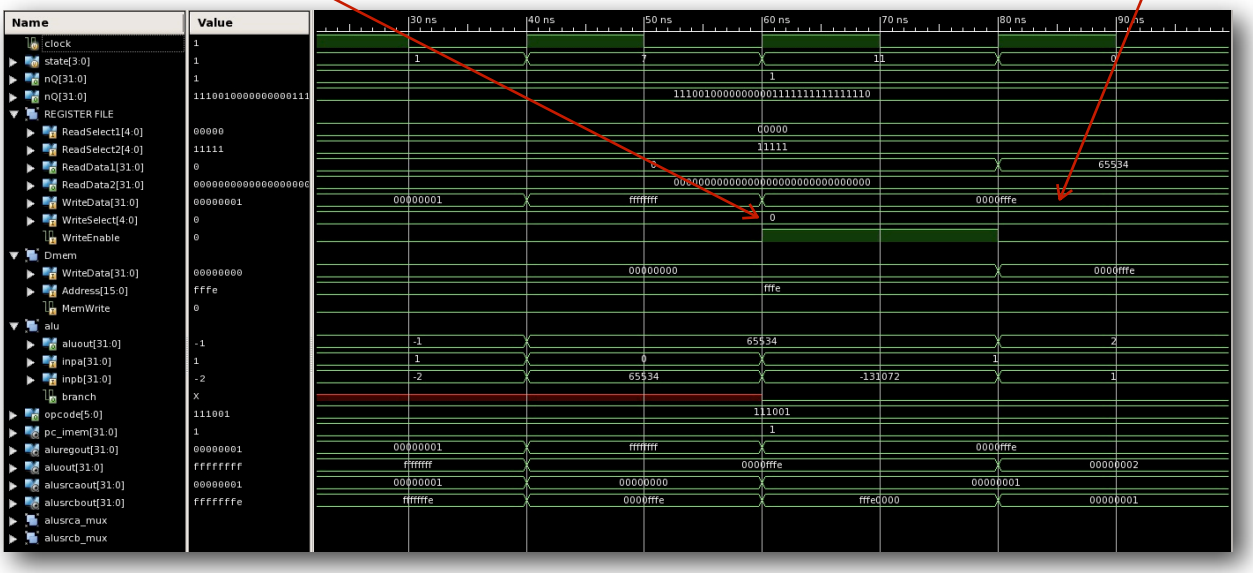
BNEZ Instruction | BNEZ $R31, 0xFFFD |

Address Pre-Branch Branch [28] Instruction Execution



LI Instruction | LI $R0, 0xFFFE |

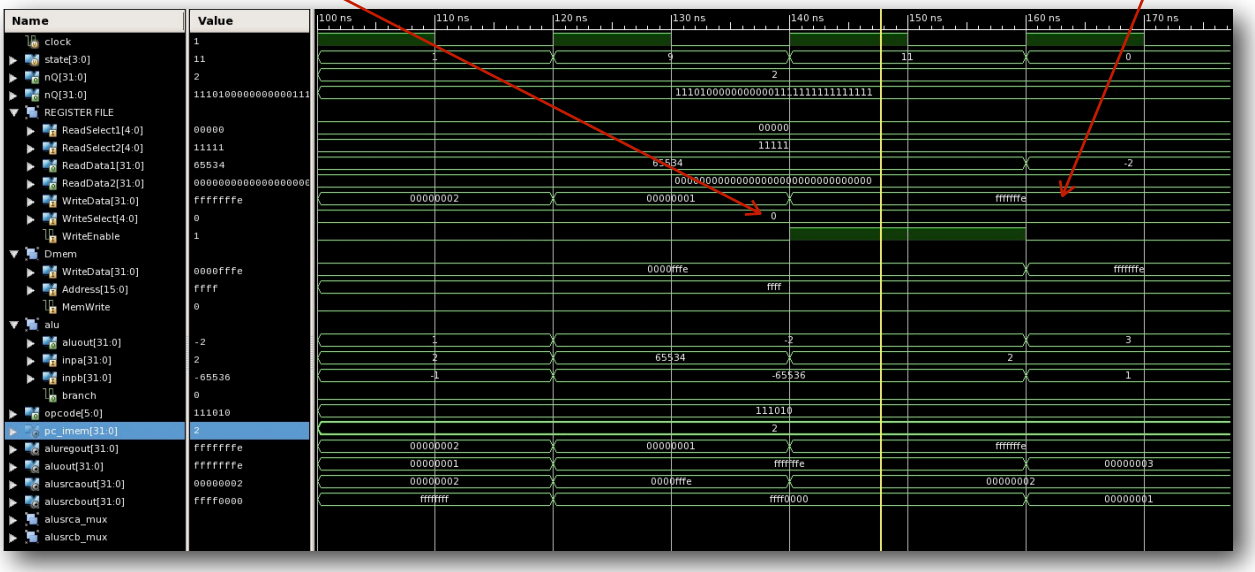
Address = 0 Write Data = 0x0000FFFE



LUI Instruction | LUI $R0, 0xFFFF |

Address = 0

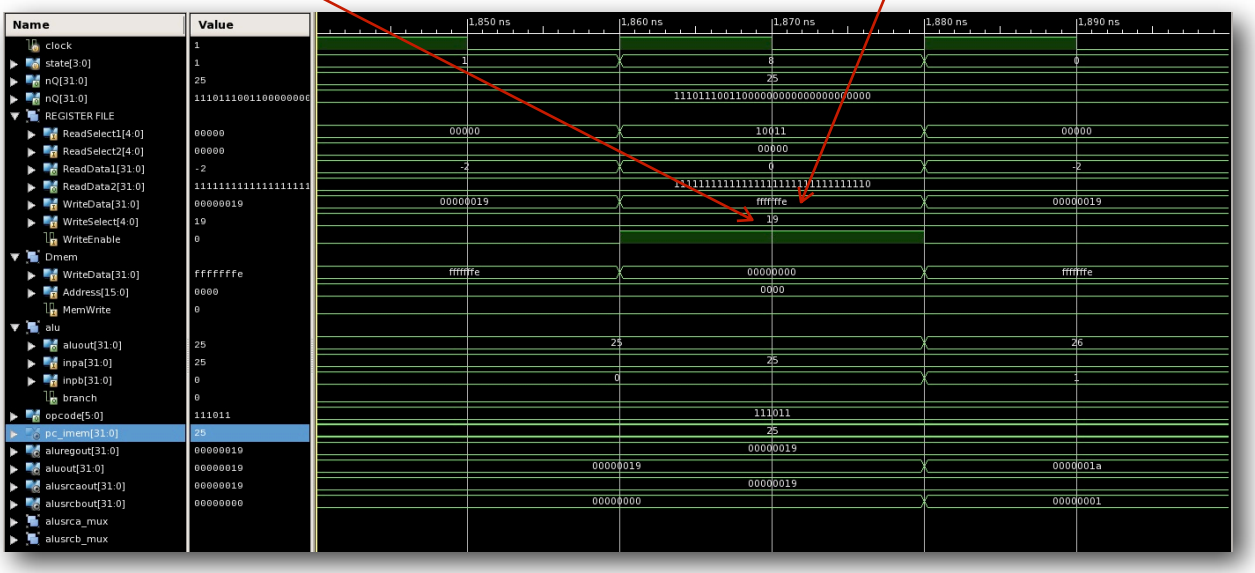
Write Data = 0xFFFFFFFE



LWI Instruction | LWI $R19, 0x0000 |

Address = 19

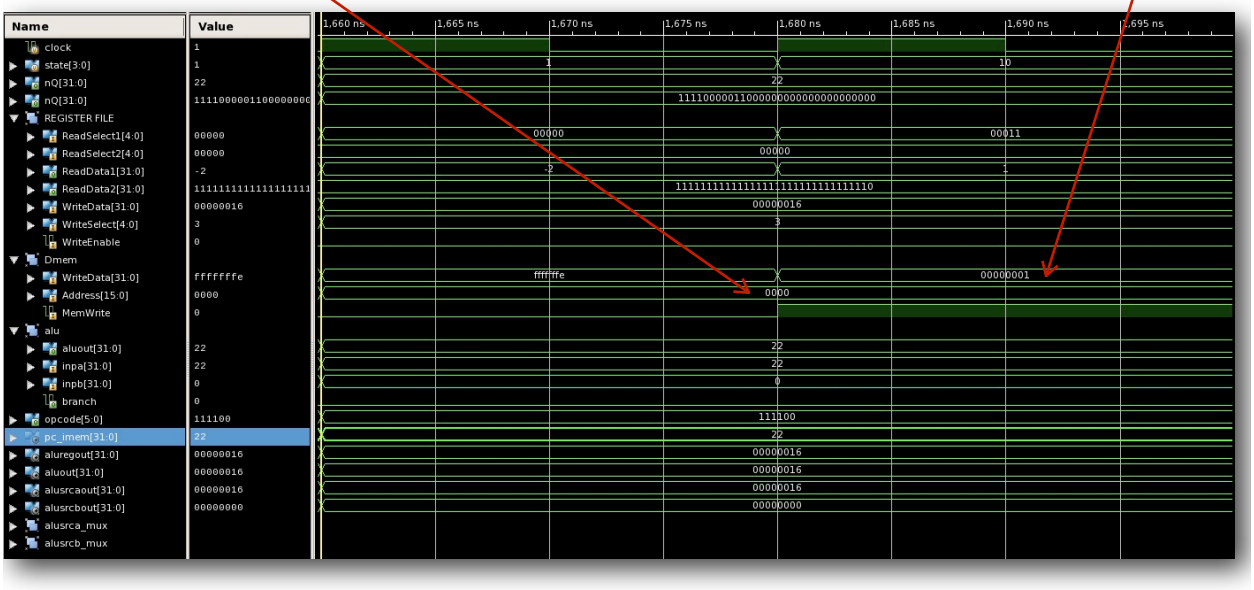
Write Data = 0xFFFFFFFE



SWI Instruction | SWI $R3, 0x0000 |

Address = 0

Write Data = 0x00000001



Debug Process

Controller

DESIGN

The controller was originally designed to handle more instructions that in the final version. Examples are LW, SW, and JAL. However when implementing the designs it was decided that it would be best to have base implementation running and then if there was extra time to put the extra functions on top. Unfortunately there was no extra time in the end so the controller stayed at its base implementation.

CODE

The first version of the controller had a system that the new state was written to a register called next state and then on the next clock cycle the state was changed. This was used because it had worked fine in previous projects; however when debugging the controller, it was noticed that the control signals would not change until a clock cycle after they were supposed to. So in order to mitigate this problem the state register was set to the new state on the spot. Secondly all of the control lines and next state logic was written within the same case statement. This seemed to be causing problems as well for the timing; so a second always block was added which responded to changed state or the signal reset. This always block held the control signal logic, and the first block held the next state logic.

Datapath

DESIGN

As with the controller, the original design of the datapath handled more instructions than the final implementation. Examples are LW, SW, and JAL. In an effort to streamline the implementation these added functions were dropped for base implementation.

While testing LI, LUI, BEQZ, BNEZ functions, the value going into alusrca mux was arriving one clock cycle later than the value of alusrcb mux. This allowed the value of alusrcb mux to be operated on with a dirty value from alusrca mux; thus the alu outputted an incorrect value. In order to fix this problem, the value of read data 1 was added as an input into alusrca mux. Now the values arrive at alusrc a&b mux at the same time, and a clean alu operation takes place.

While testing the NOOP instruction, the instruction would only hold for one clock cycle instead of the two that are needed; thus destroying the timing for all of the other instructions. This effect was due to the fact that the opcode is passed to the controller after the instruction register. Delaying the noop execution for one clock cycle. In order to fix this problem, the output of IMem is passed to the controller along with the opcode after the register file. So if a noop instruction is coming into the system it is acted on before all other instructions, giving us a two clock cycle wait time.