**Group 1-B**

Bart Miskowiec

Ryan Benshoof

CJ Miller

Luke Miller

Table of Contents

Executive Summary Page 3

Introduction Page 3

Design Page 3

Implementation Page 3

Xilinx Model Page 3

Testing Methodology Page 4

Results and Performance Page

Conclusion Page

Appendix A – Register Descriptions Page

Appendix B – Instruction Type Formats Page

Appendix C – Instruction Descriptions Page

Appendix D – Example Instruction Formats Page

Appendix E - RTL Page

Appendix F – Instruction Tests Page

Appendix G – Datapath Diagram Page

Appendix H – Component Descriptions Page

Appendix I – Testing Plan Page

Appendix J – Control State Diagram Page

Appendix K – Euclid’s Algorithm Instructions Page

Appendix L – Design Process Journal Page

Appendix M – Performance Results Page

**Executive Summary:**

The goal of our processor was to implement a design that was easy to use, while still maintaining fast performance. These design considerations led us to create a processor that used a single memory unit and ALU surrounded by a multicycle datapath. The processor uses five instruction types to comprise a versatile instruction set that makes coding a program easier and more accurate in the process. We also implemented an assembler and compiler so that translating any program to machine code is done automatically. All of these design considerations created incredible speed and ease of use to the user. This speed is shown in our computation of Euclid’s algorithm that ran in X milliseconds. The cycle time for our processor was XX nanoseconds which translates to XX MHZ.

**Introduction:**

The goal of this project was to create a fast and efficient processor that added some benefits otherwise unused in a typical MIPS processor. We used 4-bit opcodes, thereby allowing us to use 16 instructions. We also implemented a register file with sixteen 16-bit registers, each with a specific purpose. Throughout this report, we will provide information about the design and our thought process for the entirety of the project.

**Design:**

Our design implemented register to register computations using sixteen registers and four different instruction types. We also decided to use a multicycle datapath as opposed to a singlecycle or pipelined datapath to create a good balance between efficiency, speed, and cost of hardware. The multicycle datapath also allowed us to reduce the cycle time and improve the overall runtime of programs. This is shown in our RTL and Control diagrams (Appendices E and J) where most of the instructions only take four cycles to complete, and lw/sw takes five cycles. When deciding the specifications of our instruction layout, we discussed the tradeoffs that would have to be made with a 16-bit register instead of a 32-bit register. We settled on using sixteen 16-bit registers, since any less would not allow us enough space to run programs effectively, but more registers would have been unnecessary. A complete list of our registers and their uses is shown in Appendix A. While using only sixteen registers may make multiple register to register computations difficult, we believe the tradeoff for simplicity is better in the long run.

**Implementation:**

Our instruction set, shown in Appendix C, is comprised of 16 instructions with which multiple algorithms can be run, particularly Euler’s algorithm. We used R-Type, I-Type, J-Type, A-Type, and JR-Type instructions, which all have differing formats shown in Appendix B. Each instruction contains a 4-bit op code to denote one of the sixteen instructions, and then a combination of registers and immediate values to accomplish the variety of instructions.

**Xilinx Model:**

Our Xilinx design took advantage of multiple schematics integrated into a single datapath as shown in Appendix G. Each of these schematics was created in a separate project and then copied into a final datapath project to be wired together. We implemented the control unit using Verilog, and based it off of the state machine in Appendix J.

**Testing Methodology:**

Our testing process involved testing each of the components required for each cycle individually and then integrated the components together within each cycle. For example, in the first cycle, we tested the PC, ALUSrcA, ALUSrcB, the ALU, PCSrc, and PCWrite. We ensured that when the required controls were set, the PC would be incremented by 2. Next, we tested the Memory unit to ensure that it properly read and wrote data correctly. Our next test involved checking that the correct value was written into the instruction register by testing the PC, Memory, IRWrite, and IorD control signal. We then tested the individual registers to ensure that they would input, store, and output data correctly. Finally, we tested the full datapath with each type of instruction to ensure that they produced the expected results. This involved setting the control signals to the correct value, loading the instruction into memory, and ensuring that the correct value is outputted from the ALU and then written into the correct registers.

**Results and Performance:**

Performance

**Conclusion:**

Overall, we were impressed by our processor speed which accomplished its task of running Euclid’s algorithm in XX milliseconds. Although Xilinx proved troublesome and we faced a variety of challenges along the way, we were able to build a processor that is capable of running most general computations and met our overall expectations.

**Appendix A: Register Descriptions**

|  |  |  |
| --- | --- | --- |
| Register Name | Number | Usage |
| $0 | 0 | Constant zero register |
| $ra | 1 | Return address (used by function call) |
| $sp | 2 | Stack pointer |
| $v0 | 3 | Return value of a function |
| $a0 | 4 | Argument 1 for procedure call |
| $a1 | 5 | Argument 2 for procedure call |
| $cr | 6 | Reserved for compiler operations |
| $t0 | 7 | Temporary (not preserved across procedure call) |
| $t1 | 8 | Temporary (not preserved across procedure call) |
| $t2 | 9 | Temporary (not preserved across procedure call) |
| $t3 | 10 | Temporary (not preserved across procedure call) |
| $s0 | 11 | Saved (preserved across procedure call) |
| $s1 | 12 | Saved (preserved across procedure call) |
| $s2 | 13 | Saved (preserved across procedure call) |
| $s3 | 14 | Saved (preserved across procedure call) |
| $s4 | 15 | Saved (preserved across procedure call) |

**Appendix B: Instruction Type Formats**

* R-Type

|  |  |  |  |
| --- | --- | --- | --- |
| OP (4 bits) | rs (4 bits) | rt (4 bits) | rd (4 bits) |

* + R-Type instructions receive two source registers (rs and rt) and store the result in the destination register (rd)
* I-Type

|  |  |  |  |
| --- | --- | --- | --- |
| OP (4 bits) | rs (4 bits) | rd (4 bits) | Imm (4 bits) |

* + I-Type instructions receive one source register (rs) and one immediate value, and stores the result in the destination register (rd)
* J-Type

|  |  |
| --- | --- |
| OP (4 bits) | Imm (12 bit) |

* + J-Type instructions use the 12 bit immediate value as an address to jump to
* A-Type

|  |  |  |
| --- | --- | --- |
| OP (4 bits) | rd (4 bits) | Imm (8 bits) |

* + A-Type Instructions receive an 8 bit immediate value and stores the result in the destination register (rd). Sign extension of one 8 bit immediate then adding another 8 bit immediate value will yield a 16 bit value.
* JR-Type

|  |  |  |
| --- | --- | --- |
| OP (4 bits) | rd (4 bits) | Imm (8 bits) |

* + JR-types are similar to A-type, but the immediate is always 0. It them takes the memory at register value, and sets PC equal to it.

**Appendix C: Instruction Descriptions**

* ADD - Add rs, rt, rd (R-Type)
  + Adds rs and rt and stores the result in rd
* ADD IMMEDIATE - Addi rd, Imm (A-Type)
  + Adds rd and Imm and stores the result back in rd
* AND - And rs, rt, rd (R-Type)
  + Logically And’s rs and rt and stores the result in rd
* BRANCH EQUAL - Beq rs, rt, rd (R-Type)
  + If rs is equal to rt, then sets PC equal to the memory address in rd
* BRANCH NOT EQUAL - Bne rs, rt, rd (R-Type)
  + If rs is not equal to rt, then sets PC equal to the memory address in rd
* JUMP - J Imm (J-Type)
  + Jumps to address in Imm which has been shifted left once and concatenated the first three bits of the PC to the most significant bit.
* JUMP AND LINK - Jal Imm (J-Type)
  + Jump to address in Imm and sets $ra to the PC value
* JUMP REGISTER - Jr rd, 0 (JR-Type)
  + Jump to address in the rd register. The immediate is always 0
* LOAD WORD - Lw rs, rd, Imm (I-Type)
  + Loads the immediate at address rs into rd
* OR - Or rs, rt, rd (R-Type)
  + Logically Or’s rs and rt and stores the result in rd
* SET LESS THAN - Slt rs, rt, rd (R-Type)
  + If rs is less than rt, rd equals 1, otherwise rd equals 0
* SHIFT LEFT LOGICAL - Sll rs, rd, Imm (I-Type)
  + rs is shifted left by the Imm value, and is stored in rd
* SHIFT RIGHT LOGICAL - Srl rs, rd, Imm (I-Type)
  + rs is shifted right by the Imm value, and is stored in rd
* SHIFT RIGHT ARITHMETIC - Sra rs, rd, Imm (I-Type)
  + rs is shifted right arithmetically by the Imm value, and is stored in rd
* SUBTRACT - Sub rs, rt, rd (R-Type)
  + Subtract rt from rs and stores the result in rd
* STORE WORD - Sw rs, rd, Imm (I-Type)
  + Stores the value rs at the 4 bit Imm offset of rd

**Appendix D: Example Instruction Formats**

* add $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | 0111 | 1000 | 1001 |

* addi $t0, 0000 0001

|  |  |  |
| --- | --- | --- |
| 0001 | 0111 | 0000 0001 |

* and $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0010 | 0111 | 1000 | 1001 |

* beq $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0011 | 0111 | 1000 | 1001 |

* bne $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0100 | 0111 | 1000 | 1001 |

* j 0000 0000 0001

|  |  |
| --- | --- |
| 0101 | 0000 0000 0001 |

* jal 0000 0000 0001

|  |  |
| --- | --- |
| 0110 | 0000 0000 0001 |

* jr $t0, 0000 0000

|  |  |  |
| --- | --- | --- |
| 0111 | 0111 | 0000 0000 |

* lw $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 | 0111 | 1000 | 0001 |

* or $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1001 | 0111 | 1000 | 1001 |

* slt $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1010 | 0111 | 1000 | 1001 |

* sll $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1011 | 0111 | 1000 | 0001 |

* srl $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1100 | 0111 | 1000 | 0001 |

* sra $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1101 | 0111 | 1000 | 0001 |

* sub $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1110 | 0111 | 1000 | 1001 |

* sw $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1111 | 0111 | 1000 | 0001 |

**Appendix E: RTL**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | A-Type | R-Type | lw/sw | Beq/bne | j/jal | I-Type | jr |
| Inst Fetch | IR[PC] | | | | | | |
| Inst Decode  Reg Fetch | PC=PC+1  A = Reg[IR[11-8]]  B = Reg[IR[7-4]]  C = Reg[IR[3-0]] | | | | | | |
| Execution Address Comparison  Beq/bne/j done | ALUout = A op IR[7-0] | ALUout = A op B | ALUout = A + IR[3-0] | ALUOut = PC +C | PC = PC[15-13] || IR[11-0] << 1  $ra = PC | ALUOut = A op IR[3-0] | ALUout=A op IR[7-0] |
| Mem access  I,A,R,jr Done | Reg[IR[11-8]] = ALUout | Reg[IR[3-0]] = ALUout | MDR = Mem[ALUout]  Mem[ALUout] = A | If ( A==B)  PC= ALUout |  | Reg[IR[7-4]] = ALUOut | PC=Mem[ALUout] |
| Lw done |  |  | Reg[IR[7-4]] = MDR |  |  |  |  |

**Appendix F: Instruction Tests**

1. ADD
   1. Test Basic Cases
      1. 1 + 1 = 2
      2. 1 + -1 = 0
      3. -1 + -1 = -2
   2. Test Overflow
      1. 0xFFFF + 0x1 =
2. ADDI
   1. Test Basic Case
      1. 1 + 1 = 2
   2. Test Overflow
      1. 0xFFFF + 0x1 = 0
3. AND
   1. Test Cases
      1. 0xFFFF & 0x0000 = 0x0000
      2. 0x1111 & 0x8888 = 0x0000
      3. 0x3333 & 0x1111 = 0x1111
4. BRANCH EQUAL
   1. Should Jump
      1. 1 == 1 Goto 0
      2. 0 == 0 Goto 1
   2. Should not Jump
      1. 1 == 0 Goto 0
      2. 0xFFFF == 0x1111 Goto 0
5. BRANCH NOT EQUAL
   1. Should Not Jump
      1. 1 != 1 Goto 0
      2. 0 != 0 Goto 1
   2. Should Jump
      1. 1 != 0 Goto 0
      2. 0xFFFF != 0x1111 Goto 0
6. JUMP
   1. Check value is written into PC
      1. 0x0000
      2. 0xFFFF
      3. 0x1111
   2. Examples
      1. 0x0000
      2. 0xFFFF
7. JUMP AND LINK
   1. Check PC is in $ra after command executes
8. JUMP REGISTER
   1. Check that value in PC is same that is in the defined register
9. LOAD WORD
   1. Test good addresses
      1. Make sure returned data value is correct and put into the correct register
   2. Test Bad addresses
      1. Exception should be raised
10. OR
    1. Test Cases
       1. 0x0000 | 0xFFFF = 0xFFFF
       2. 0x9999 | 0x6666 = 0xFFFF
11. SET LESS THAN
    * 1. 1 < 1 = 0
      2. 1 < 2 = 1
      3. -1 < 0 = 1
      4. 4 < 1 = 0
12. SHIFT LEFT LOGICAL
    1. Test Cases
       1. 8 << 2 = 32
       2. 7 << 2 = 28
       3. -2 << 1 = -1
13. SHIFT RIGHT LOGICAL
    1. Test Cases
       1. 8 >> 2 = 2
       2. 0xFFFF >> 4 = 0x0FFF
14. SHIFT RIGHT ARITHMETIC
    1. Test Cases
       1. 8 >> 2 = 2
       2. 0xFFFF >> 4 = 0xFFFF
15. SUBTRACT
    1. Test Case
       1. 1 – 2 = -1
       2. 2 – 1 = 1
       3. 2 - -1 = 3
16. STORE WORD
    1. Good Addresses
       1. Check to see data is in memory and value from register
    2. Bad Addresses
       1. Raise Exception

**Appendix G: Datapath Diagram**

**Appendix H: Component Descriptions**

All considered to use 16 bits and transfer in 16 bit segments unless stated otherwise.

1. Program Counter Register
   1. Stores the Current Instruction Address.
   2. Inputs – Next Address
   3. Outputs – Current Address
   4. Controls – PCWrite, Zero, PCWriteCond
2. Memory
   1. RAM that contains the instructions and data.
   2. Inputs – Address, Data to Write
   3. Outputs – Data / Instruction
   4. Controls – MemRead, MemWrite
3. Instruction Register
   1. Stores the Current Instruction
   2. Inputs – Instruction from Memory
   3. Outputs – Current Instruction
   4. Controls – IRWrite
4. Register File
   1. Contains 16, 16 Bit Registers
   2. Inputs – Read Register 1,2 Read/Write Register 3, Write Data
   3. Outputs, Read Data 1,2,3
   4. Controls – RegWrite
5. A Register
   1. Stores the Data from Read Data 1 of the Register File
   2. Inputs – Read Data 1
   3. Outputs – Data in Register
   4. Controls - None
6. B Register
   1. Stores the Data from Read Data 2 of the Register File
   2. Inputs – Read Data 2
   3. Outputs – Data in Register
   4. Controls - None
7. C Register
   1. Stores the Data from Read Data 3 of the Register File
   2. Inputs – Read Data 3
   3. Outputs – Data in Register
   4. Controls - None
8. ALU
   1. Performs arithmetic and logical operations
   2. Inputs – ALUsrcA, ALUsrcB
   3. Outputs – zero, overflow, ALUout
   4. Controls – ALU Control
9. ALUout Register
   1. Stores the value from the ALU
   2. Inputs – Data from ALU
   3. Outputs – Data in register
   4. Controls - None
10. Memory Data Register (MDR)
    1. Stores the value from memory
    2. Inputs – Data from Memory
    3. Outputs – Data in register
    4. Controls – None
11. ALU Control
    1. Determines the ALU Control Code from the ALUop Code
    2. Inputs – Instruction[15-13]
    3. Outputs – ALUop Code
    4. Controls – AluOutControl
12. MemtoReg Mux
    1. Switches between the ALUout and MDR Registers
    2. Inputs - MDR, ALUout
    3. Outputs – Data to Write to register
    4. Controls MemToReg
13. IorD Mux
    1. Determines the Address piped into memory
    2. Inputs – PC, ALUout
    3. Outputs – Address
    4. Controls – IorD
14. ALUSrcA Mux
    1. Determines the data to pipe into ALU Data 1
    2. Inputs - A, PC
    3. Outputs – One of the Inputs
    4. Controls – ALUSrcA
15. ALUSrcB Mux
    1. Determines the data to pipe into ALU Data 2
    2. Inputs B, Sign Extended IR[3-0], Sign Extended Shifted Left 1 IR[3-0]
    3. Outputs – One of the Inputs into ALU Data 2
    4. Control – ALUSrcB
16. PCSource Mux
    1. Determines the Address to be put into the PC
    2. Inputs, ALU Data Out, ALUOut, PC[15-13] || IR[11-0] << 1
    3. Outputs – The address to be put into the PC
    4. Control – PCSource

**Appendix I: Testing Plan**

1. Program Counter Register (PC) | Instruction Register (IR) | ALUout Register
   1. Set input value to 0x0000 and turn on the write control.
   2. Check that output is 0x0000.
   3. Change input to 0xFFFF.
   4. Check that output is 0xFFFF.
   5. Turn off write control and change input to 0x0000.
   6. Check that output is still 0xFFFF
2. Memory
   1. Turn on MemRead and turn MemWrite to off. Set Addr to 0x0000.
   2. Check that Dataout is 0xFFFF
   3. Change Addr to 0x0002
   4. Check that Dataout is 0x0000
   5. Flip MemRead and MemWrite. Set Addr to 0x000. Set WriteData to 0x0000
   6. Flip MemRead and MemWrite. Check Dataout is 0x0000
3. Register File
   1. Set Regwrite to on. Set Instruction wire to 0x0123. Set WriteData to 0xFFFF
   2. Check that C Register is 0xFFFF
   3. Change Instruction wire to 0x0321. Set WriteData to 0x1111
   4. Check that A Register is 0xFFFF and C Register is 0x1111
   5. Change Instruction wire to 0x0330. Set WriteData to 0xFFFF
   6. Check that A,B Registers are 0xFFFF and C Register is 0x0000
   7. Change Instruction wire to 0x0333. Set WriteData to 0x0000. Set Regwrite to off.
   8. Check that A,B,C Registers are 0x FFFF.
4. A, B, C, MDR Registers

ALU Control Input / Output Values

* 1. Set input to 0xFFFF
  2. Check output is 0xFFFF
  3. Set input to 0x0000
  4. Check output is 0x0000

1. ALU

Refer to the table to the side for ALUControl Out codes which are

the ALU op Codes

* 1. Add
     + 0x0001 + 0x0001 = 0x0002
     + 0x0001 + 0xFFFF = 0x0000
  2. Sub
     + 0x0001 – 0x0001 = 0x0000
     + 0x0001 – 0x0002 = 0xFFFF
  3. And
     + 0x0000 & 0xFFFF = 0x0000
     + 0xCCCC & 0xCCCC = 0xCCCC
     + 0x8888 & 0xFFFF = 0x8888
  4. Or
     + 0x0000 & 0xFFFF = 0xFFFF
     + 0x1111& 0xEEEE = 0xFFFF
  5. Slt
     + 0x0000 < 0x0001 = 0x0001
     + 0x0000 < 0xFFFF = 0x0000
  6. Sll
     + 0x0001 << 1 = 0x0002
     + 0xFFFF << 4 = 0xFFF0
  7. Srl
     + 0xFFFF >> 4 = 0x0FFF
     + 0x0001 >> 1 = 0x0000
  8. Sra
     + 0xFFFF >> 4 = 0xFFFF
     + 0x0001 >> 1 = 0x0000

ALU Op codes

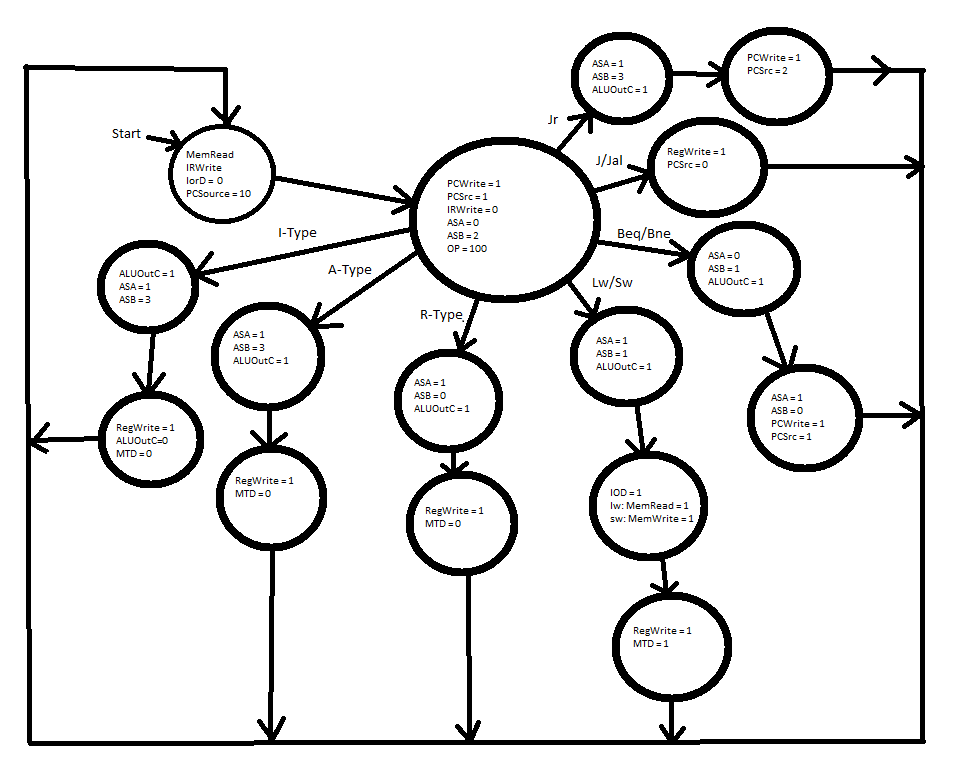
|  |  |
| --- | --- |
| 000 | And |
| 001 | And |
| 010 | Or |
| 011 | Or |
| 100 | Add |
| 101 | Sub |
| 110 | N/A |
| 111 | Slt |

1. ALU Control
   1. For each Instruction[15:12] check to see the output code matches the table.
2. MemtoReg | IoR | ALUSrcA (1 bit Mux)
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
3. ALUSrcB Mux
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF, Wire 2 to 0x0002, Wire 3 to 0x1111.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
   5. Set control to 2.
   6. Check Output is 0x0002
   7. Set control to 3.
   8. Check Output is 0x1111
4. PCSource Mux
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF, Wire 2 to 0x000.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
   5. Set control to 2.
   6. Check Output is 0x0002

Integration Testing

1. PC + ALU (Any Instruction)
   1. Test to see that PC is incremented by 2.
2. PC + IR + Memory (Any Instruction)
   1. See that Instruction that the value of PC is pulled into IR.
3. PC + IR + Memory + Reg File + ALU (0x0123)
   1. Test that a loaded command outputs the correct values to A(0x1), B(0x2), C(0x3) Registers and then the ALU adds them together and stores back in Read 3(0x3) Register.

**Appendix J: Control State Diagram**

****

Control Signals:

All Control signals are 1-bit unless otherwise specified

1. Regwrite:

This control signal turns on writing for the registers, A, B, C. 1 is write, 0 is not write.

1. Memread:

This control signal turns on direct memory reading. 1 is read, 0 is don’t read.

1. Memwrite:  
   This control signal turns on direct memory writing. 1 is write, 0 is don’t write.
2. ASA:  
   This control signal determines the source for the first ALU input. 0 is PC, 1 is register A
3. ASB:  
   This 2-bit control signal determines the source for the second ALU input. 00 is register B, 01 is register C, 10 is the immediate value, 2, and 11 is sign-extended address.
4. MTD:  
   this control signal determines the input for the register file from either ALUOut or MDR
5. PCwrite:  
   This control signal controls if PC is being written to. 1 is write, 0 is don’t write.
6. IOD:  
   This control signal changes the input for the address in memory access. 0 is from the PC, 1 is from the ALUout
7. PCsrc:  
   This 2-bit control signal determines what source the PC is being written from. 00 is the branch command’s source, 01 is from the ALU result, 10 is from ALUout
8. IRwrite:  
   This control signal turns on and off writing to the instruction register from memory. 1 is write, 0 is don’t write.
9. Branch:  
   This control signal determines if a branch is occurring. 1 is branch, 0 is don’t branch.
10. ALUoutC  
    This control signal controls the ALUout writing. 1 is write, 0 is don’t write.

**Appendix K: Euclid’s Algorithm**

# v0 = n

addi $sp -12

sw $s0 $sp 0

sw $s1 $sp 1

sw $ra $sp 2

and $s0 $0 $0 #Load Blank

addi $s0 2 #put value of 2 in (m)

or $s0 $0 $k0

or $s1 $0 $a0 #move to saved register

Loop:

or $a0 $s1 $0 #n

or $a1 $s0 $0 #m

jal gcd #call gcd

addi $v0 -1 #Subtract by one. Don’t care about #under(over)flow due to just want to see if the #value had been 1. Ends up as gcd(n,m) - 1 != 0

beq $v0 $0 $k0 #go past the jump

addi $s0 1 #Increment by 1

j Loop

EndLoop:

lw $s0 $sp 0

lw $s1 $sp 1

lw $ra $sp 2

addi $sp 12

jr $ra 0

gcd:

and $t0 $0 $0 #Force Zero

addi $t0 2 #Load 2, Used as value of commands to skip

bne $v0 $0 $t0 #skip not equal (don’t return)

or $v0 $0 $a1 #Move b to return

jr $ra 0 #Return

addi $t0 4 #Currently has 2 so skip end lines minus 2, Used #as commands to skip for while loop (total 6)

and $t2 $0 $0 #Set t2 to zero

addi $t2 2 #Amount to skip to for if inside the while

begin:

beq $a1 $0 $t0 #If b != 0 jump t0 forward

slt $a0 $a1 $t1 #Set t1 to 1 if a less then b

bne $t1 $0 $t2 #Jump to else (forward 2)

sub $a0 $a0 $a1

j begin #End of If, Start of Else

sub $a1 $a1 $a0

j begin #Back up to look check

or $v0 $0 $a0 #Move a to return

jr $ra 0 #Return

**Appendix L: Design Process Journal**

We have decided to make a load-store architectural design for the sake of simplicity. We decided on four bit OP codes, which limits the number of possible instructions to 16, but allows us more bits in the instructions for registers and immediate values.

**Instruction Formats:**

We implemented four instruction types: R-Type, I-Type, J-Type, and A-Type. R-Type instructions receive two source registers (rs and rt) and store the result in a destination register (rd). I-Type instructions receive one source register (rs) and one immediate value, and stores the result in a destination register (rd). J-Type instructions use a 12 bit immediate value as an address to jump to. A-Type instructions receive an 8 bit immediate value and store the result in the destination register (rd).

**Choosing the Registers:**

We first decided which registers were absolutely required for this project. We determined that we would need a constant zero ($0), a return register ($v0), a stack pointer ($sp), an OS kernel register ($k0), two argument registers ($a0, $a1), and a return address register ($ra).

Since we plan on supporting 16 registers, there are nine registers remaining to allocate between s- and t-registers. We decided to allocate five registers as s-registers and four as t-registers. We thought it would be best to have an even balance of s- and t-registers, but chose to have more s-registers since it would be better to be able to save more values if needed.

**Instructions:**

We selected a 4 bit OP code for the instruction types, which limits us to 16 instructions. We selected the basic instructions that are required for Euclid’s algorithm, and added additional instructions that are required for loads, stores, and jumps. We decided on the following instructions for each instruction type:

R-Type: OP(4), rs(4), rt(4), rd(4)

* Add, And, Beq, Bne, Or, Slt, Subtract

I-Type: OP(4), rs(4), rd(4), Imm(4)

* Lw, Sll, Srl, Sra, Sw

J-Type: OP(4), Imm(12)

* J, Jal

A-Type: OP(4), rd(4), Imm(8)

* Addi, Jr

We created an RTL specification for our processor. Along with this, we have updated the component and signals lists and all of this was added to the design document. We've made tests to validate our RTL specification, which seems to work well right now. Soon, we will be able to actually test our design, as we start implementing it in Xilinx. We included a C register in the RTL to comply with our instruction type spec, which works our really well in the RTL document.

Our strategy for creating test cases was to test in a manner that went for the extreme values and a normal case. By doing this all ranges of values should work. In the case of the ALU Control lookup just test all values as there is only 16 different ways.

Our datapath revolves around our choice of register to register architecture. We also chose to only have one memory and one ALU. This caused us to have to have several muxs to allow to use the same components multiple times in the same command.

Our architecture caused a few issues with our datapath designed that caused us to have to add extra control signals and make the instruction take longer. The instruction was for branches as we have the [3:0] of the instruction for which register’s value we should add to the program counter. We first run the PC+Register store it into ALUOut then turn off ALUOut write and do the comparison between register 1 and 2. Also for this had to add another input into PCSource from ALUOut.

**Appendix M: Performance Results**

Total number of bytes required to store Euclid's algorithm and relPrime as well as any memory variables or constants is 72 bytes.

Total number of instructions executed when relPrime is called with 0x13B0 is 51,060.

Total number of cycles required to execute relPrime when 0x13B0 is called is 51,060.

Average cycles per instruction for relPrime 1.

Cycle time for our design is 94.803 Nanoseconds.

Total execution time for relPrime when 0x13B0 is called is .00484064118 seconds

Gate count for our design is 7966.  
Device Utilization Summary: