CSSE -232 Computer Architecture I

Design Documents and Journal

Group 1-B

Michael Bell, Lindsay Klaetsch, David McGinnis, and Samuel Newman

Contents

[Executive Summary 4](#_Toc254728170)

[Introduction 5](#_Toc254728171)

[Instruction Design 5](#_Toc254728172)

[Hardware Implementation 5](#_Toc254728173)

[Xilinx Model 6](#_Toc254728174)

[Testing Methodology 6](#_Toc254728175)

[Final Results & Data 6](#_Toc254728176)

[Conclusion 7](#_Toc254728177)

[Design Documents 8](#_Toc254728178)

[Registers 8](#_Toc254728179)

[Additional Registers 8](#_Toc254728180)

[Exceptions 8](#_Toc254728181)

[Instruction Types 9](#_Toc254728182)

[R-Type 9](#_Toc254728183)

[I-Type 9](#_Toc254728184)

[J-Type 9](#_Toc254728185)

[M-Type (Move Type) 9](#_Toc254728186)

[Instruction Listing 9](#_Toc254728187)

[Pseudo-Instructions 12](#_Toc254728188)

[Assembly-to-Machine Language Translation Rules 13](#_Toc254728189)

[Example Assembly Language Program (Euclid’s Algorithm) 13](#_Toc254728190)

[Example Assembly Language Fragments 15](#_Toc254728191)

[RTL Instruction Descriptions 15](#_Toc254728192)

[Register Transfer Statements 17](#_Toc254728193)

[Components & Control Signals 17](#_Toc254728194)

[Control Table 20](#_Toc254728195)

[Datapath Block Diagram 21](#_Toc254728196)

[System Testing 22](#_Toc254728197)

[Performance Data Summary 22](#_Toc254728198)

[Design Process Journal 23](#_Toc254728199)

[January 11th-12th 23](#_Toc254728200)

[January 18th 23](#_Toc254728201)

[January 19th 23](#_Toc254728202)

[January 25th 23](#_Toc254728203)

[February 2nd 23](#_Toc254728204)

[February 9th 23](#_Toc254728205)

[February 14th 24](#_Toc254728206)

[February 19th - 21st 24](#_Toc254728207)

# Executive Summary

This document's purpose is to describe the architecture and use of the 16-Bit Epicness processor, a 16-bit performance-enhanced pipelined processor. The introduction describes the purpose and thought process in making the processor, and the key ideas applied to the design of this system. The next section then details how the system was built, including testing procedures and the process with which the system was implemented and tested. After this, we detail the actual design of the processor, listing the registers, along with their uses in a standard program for the processor, as well as detailing each instruction included for the instruction. After this, we detail how these instructions are translated into machine code to be executed in the processor. After this, we show some example code, as well as the machine code translation of this code. We then describe the RTL of the system and the control signals which control the processor. The datapath is then shown, in order to tie everything together. Finally, a diary taken during development is displayed, to show the trials and successes we achieved during development.

# Introduction

The purpose of this project is to design and implement a fully-functional 16-bit processor for an FPGA board, capable of viewing and processing input and output on an LCD display, running programs, and handling interrupts. The processor also must have enough functionality to run Euclid’s Algorithm. This processor has a focus on size and performance. In exchange for this things, we have made the processor much more complex, handling things such as multiple opcode sizes and double instructions (instructions which use 32 bits instead of the normal 16 bits). In exchange, however, we have created a pipelined processor which has small instruction sizes, while keeping its focus on performance. We have also implemented interrupts in order to better connect with the circuit board and the outside world, and it is easy to modify in order to add more interrupt types.

# Instruction Design

The processor has a set of 23 instructions, with a combination of 4- and 5-bit opcodes. The 4-bit opcodes are indicated with a leading “1”, where the 5-bit opcodes are indicated with a leading “0”. With this implementation, 8 additional opcodes could be added, opposed to the 16 opcodes that 4-bits would allow.

There are also four types of instruction formats: R, I, J, and M-type. R-type instructions have a 4-bit opcode, 4-bit destination register, 4-bit source register, and a 4-bit transfer register. These instructions are generally used for branching, and loading/storing words. I-type instructions have a 5-bit opcode, 4-bit destination register, and a 7-bit immediate register. These instructions are used for performing operations where a value is needed in the immediate register. J-types also have a 5-bit opcode, but an 11-bit immediate register. These instructions are used for “jump” instructions because no information is actually needed to be stored/loaded into other registers. The last type, M-type has a 5-bit opcode, 4-bit destination register, 4-bit source register, and a 3-bit function code. These instructions are used like R-type, except no values are needed to be transferred and different functions are called using a unique function code.

# Hardware Implementation

Our processor is similar to a typical pipelined processor, but with some unique features and optimizations. We focused on improving the performance of the design as much as possible.

The fundamental components basic to most processors are also found in our processor. The Program Counter, Register File, ALU, Control Unit, and Memory being the most apparent. We also included an Exception Handler as well an I/O Coprocessor in order to support input/output and exceptions.

One difference in our design is the length of the opcode. For some instructions the length is 4 bits long, while it is 5 bits long for others. This forced us to make our own custom multiplexers in order to select the right bit width. As for our ALU, it behaves just like a typical ALU, performing a specific operation of two inputs based on the given opcode. It returns the result as a single output.

The Main Control Unit is the main module of the processor. It sets control signals for the Register File, Memory, ALU, as well as other items. Instead of having several multiplexers outside the Control Unit, we decided to have the Control Unit perform the output selection itself. We did this to allow for much simpler testing procedures.

# Xilinx Model

Our processor was primarily implemented schematically in Xilinx. We did write some multiplexers in Verilog, as we found out it was much simpler than doing it schematically. The memory was a pre-made CoreGen controller made automatically for us by Xilinx.

Our Main Control Unit was probably one of the hardest modules to implement in Xilinx. We took each individual bit and hooked it up to either Vcc or Gnd. In hindsight, it would have been much simpler to write the control in Verilog. All that would be needed is a block of conditional statements. We also put our components for Data Forwarding in the Control Unit. The theory behind this is the Main Control is already getting the bits from all of the previous registers, so it would be efficient to put them through our forwarding unit as well.

# Testing Methodology

Testing the processor was done in three stages. First with individual components, next with groups of components, or subsystems, and finally with the entire processor on the Spartan-3E board.

Our testing began at the component level, with each module being tested for errors before being connected with others. We found small bugs on most components, but were easily fixed. One problem that consistently came up was differences in bit widths. We would more often than not end up creating new components that had the correct width in order for the module to work.

The next step was to test different subsystems. Our plan was to add each component individually to the processor as a whole, but found that most needed other components in order to function correctly. For the most part, the testing was slow and tedious. This was due to naming issues and different bit widths.

# Final Results & Data

This processor design has met the design requirements and performs adequately. Additionally, a compiler and assembler were made so that this processor could not only use “MIPS-like” code, but traditional C code as well.

While the system was not functional in the end, we believe with a working testbench, it could be completed in a 6 hour time frame at most. The lack of such a testbench was the largest barrier to the system being completed in time for this report.

# Conclusion

Our team’s objectives for this project were to develop a fully-functional 16-bit processor, which includes input/output capabilities as well as its own assembly language. It was expected to handle exceptions and to run programs efficiently at an acceptable level of performance. The processor was implemented with the Spartan-3E FPGA as the intended hardware platform, which would use the pushbuttons and dial for inputs, and the LCD display as output.

The first thing to consider was register conventions and the assembly language. Next, RTL-level descriptions of each instruction were determined. After that, each component was designed and implemented schematically in Xilinx. Once all of the components were completed, each one was tested individually then as bigger subsystems. Finally, the entire processor was tested with test bench waveforms.

While working on this project, the team improved their knowledge in several areas. These included programming with assembly language as well as hardware design. We also faced several choices for the design of a processor, weighing the pros and cons for each decision we made. It was a good skill to develop seeing how the overall functioning of the architecture was affected by these decisions.

# Design Documents

## Registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Number | Use | Preserved? | Other Information |
| $zero | 0 | The Constant Value 0 | N/A | Always returns a value of 0 |
| $at | 1 | Assembler Temporary | No | Used for pseudo-instructions |
| $v | 2 | Value for Function Results and Expression Evaluation | No | Must be assigned in order to return a value from a function call. |
| $a0-$a2 | 3-5 | Arguments | No | Arguments to function calls; must be saved to memory if more than 3 exist |
| $s0-$s2 | 6-7 | Saved Temporaries | Yes | General use and must be saved across function calls |
| $t0 - $t1 | 8-9 | Unsaved Temporaries | No | General use. It is not guaranteed to be saved across function calls. Mainly for setting up stack so that saved temporaries can be used. |
| $k0-$k2 | 10-12 | Reserved for OS Kernel | No | Used by coprocessor 0 for interrupts |
| $sp | 13 | Stack Pointer | Yes | Points to current top of the stack; must be set when allocating memory. |
| $rr | 14 | Return | No | This is where any M-type arithmetic information is obtained from. |
| $ra | 15 | Return Address | Yes | Holds address for return statements after functions have finished. |

### Additional Registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Number | Use | Preserved? | Other Information |
| $dr | N/A | Display Register | No | Holds the local copy of display |
| $ie | 0 | Interrupt Enable | No | Allows interrupts to occur |
| $it | 1 | Interrupt Type | No | Designates which type of interrupt occurs |
| $epc | 2 | EPC Register | No | Holds address of instruction after the one that caused the interrupt |

## Exceptions

When an interrupt occurs:

1. All control bits become 0.
2. The current instruction executes and the next instruction’s address is stored in the EPC Register.
3. The PC is then set to a specific region.

00 Exception Code: An event occurred with the first input button.

01 Exception Code: An event occurred with the second input button.

## Instruction Types

### R-Type

|  |  |  |  |
| --- | --- | --- | --- |
| 0x? | rd | rs | rt |
| 15 12  Opcode (4 bits) | 11 8  Destination register (4 bits) | 7 4  Source register (4 bits) | 3 0  Transfer register (4 bits) |

R-type instruction codes have a 4-bit opcode and three 4-bit registers.

### I-Type

|  |  |  |
| --- | --- | --- |
| 0x? | rd | imm |
| 15 11  Opcode (5 bits) | 10 7  Destination register (4 bits) | 6 0  Immediate (7 bits) |

I-type instruction codes have a 5-bit opcode, a 4-bit register, and a 7-bit immediate value.

### J-Type

|  |  |
| --- | --- |
| 0x? | imm |
| 15 11  Opcode (5 bits) | 10 0  Immediate (11 bits) |

J-type instruction codes have a 5-bit opcode and an 11-bit immediate value.

### M-Type (Move Type)

|  |  |  |  |
| --- | --- | --- | --- |
| 0x? | rd | rs | func |
| 15 11  Opcode (5 bits) | 10 7  Destination register (4 bits) | 6 3  Source register (4 bits) | 2 0  Function (3 bits) |

M-type instruction codes have a 5-bit opcode, two 4-bit registers,and 3 bits that are unused.

## Instruction Listing

**add:** add rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x0E – 01110 | rd | rs | 0x0 |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Func (3bits) |

Adds the values in rd and rs and stores the result in return register.

**addi:** addi rd imm

|  |  |  |
| --- | --- | --- |
| 0x0C - 01100 | rd | imm |
| Opcode (5 bits) | Destination register (4 bits) | Immediate (7 bits) |

Adds the value in rd to the immediate value, and stores the result in return register.

**and:** and rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x0E - 01110 | rd | rs | 0x1 |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Func (3 bits) |

Ands rd and rs together, and then places the result in return register.

**beq:** beq rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0xA – 1010 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Jumps to the given link at rt if the 2 registers, rd and rs, are equal.

**bgt:** bgt rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0x8 – 1000 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Goes to the address at a given register, rt, if rd is greater than rs.

**blt:** blt rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0x9 – 1001 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Goes to the address at a given register, rt, if rd is less than rs.

**bne:** bne rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0xB – 1011 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Jumps to address in rt if rd and rs are not equal.

**j:** j target

|  |  |
| --- | --- |
| 0x00 - 00000 | Target |
| Opcode (5 bits) | Target address (11 bits) |

Jumps to address represented by target.

**jal:** jal target

|  |  |
| --- | --- |
| 0x01 - 00001 | Target |
| Opcode (5 bits) | Target address (11 bits) |

Jumps to target address represented by a given label and changes the return address to instruction after called. Should only be called when jumping to another function.

**jr:** jr rd

|  |  |  |  |
| --- | --- | --- | --- |
| 0x02 - 00010 | rd | XXXX | XXX |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Jumps to the address in rd. rs is unused for this instruction.

**ldi:** li rd imm

|  |  |  |
| --- | --- | --- |
| 0x05 - 00101 | rd | xxxxxxx |
| Opcode (5 bits) | Destination register (4 bits) | Immediate (7 bits) |

Stores the value in the next word into the register in rd.

**lwn:** lwn rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0xE – 1110 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Load the word stored at rs with an offset corresponding to the value of rt and stores the value at that place in memory to rd.

**mfc0:** mfc0 rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x08 - 01000 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Moves data to the register rs from the coprocessor register represented by rd.

**move:**  move rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x04 - 00100 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Func (3bits) |

Moves the value in rs to rd.

**mtc0:** mtc0 rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x09 - 01001 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Moves data from the register rs to the coprocessor register represented by rd.

**neg:** neg rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x06 - 00110 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Negates the value of rs and places it into rd.

nop

|  |  |  |  |
| --- | --- | --- | --- |
| 0x03 - 00011 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Does nothing. Used to avoid hazards during testing, and also to idle while waiting for a response.

**not:** not rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x07 - 00111 | rd | rs | (null) |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Applies not to the value of rs and stores the value in rd.

**or:** or rd rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x0E - 01110 | rd | rs | 0x2 |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Func (3 bits) |

Ors rs and rd together, and then places the result in rr.

**ori:** ori rd rs rt

|  |  |  |
| --- | --- | --- |
| 0x0D - 01101 | rd | imm |
| Opcode (5 bits) | Destination register (4 bits) | Immediate (7 bits) |

Ors rd with the immediate value, and then places the result in rr.

**rdr:** rdr rd

|  |  |  |  |
| --- | --- | --- | --- |
| 0x0A - 01010 | rd | rs | XXX |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Reads the data from the display register and writes it to rdr.

**rp:** rp rd

|  |  |  |  |
| --- | --- | --- | --- |
| 0xC – 1100 | rd | XXXX | XXXX |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Reads the 4-bit input port and writes it to rd.

**sll:** sll rd imm

|  |  |  |
| --- | --- | --- |
| 0x0F – 01111 | rd | imm |
| Opcode (5 bits) | Destination register (4 bits) | Immediate (7 bits) |

Takes an immediate value imm, and shifts rt by imm, placing the result in rr. If imm is negative, will shift right logical rs and place the result in rd.

**swn:** swn rd rs rt

|  |  |  |  |
| --- | --- | --- | --- |
| 0xF – 1111 | rd | rs | rt |
| Opcode (4 bits) | Destination register (4 bits) | Source register (4 bits) | Transfer register (4 bits) |

Stores the value of rd into the memory location at rs with an offset corresponding to the value of rt.

**wdr:** wdr r­d rs

|  |  |  |  |
| --- | --- | --- | --- |
| 0x0B - 01011 | rd | XXXX | XXX |
| Opcode (5 bits) | Destination register (4 bits) | Source register (4 bits) | Unused (3 bits) |

Writes the value of $rd into the display register, which is then written to the output port to be displayed on the LCD screen.

### Pseudo-Instructions

**la:** la rd Label

This instruction places the address of Label into the register rd. This instruction is equivalent to the following code:

ldi

sll rd 7

ldi rd Label

## Assembly-to-Machine Language Translation Rules

Each assembly instruction is reduced to a four-digit hexadecimal number when translated. We will use the instruction “or $s0 $s1 2” as an example.

1. Each instruction is first split up into separate sections by white space. For our example, the separate parts would be, “or”, “$s0”, “$s1”, and “2”.
2. Next, each instruction is converted to hexadecimal based on the opcode. The opcodes vary between 4 and 5 bits long, so the opcode is either represented by one or two hexadecimal digits. For the or instruction, the opcode is 0x0E.
3. The next step is converting each register to the hexadecimal representations. Each register field is only 4 bits long, so only single hexadecimal digits are necessary. Register $s0 is labeled as register 6 and register $s1 is labeled as register 7. So, “$s0” converts to 0x6, and “$s1” to 0x7.
4. We now need to convert any offsets, immediate, addresses, and functions to the hexadecimal equivalents. For the or instruction, a function code is needed to distinguish the instruction from others and is 3 bits long. The function code representation for or is 0x2.
5. The last step is to arrange the converted hex digits according to the instruction format and concatenate them together. The equivalent digits are 0x0E, 0x6, 0x7, and 0x2. Since 0x0E is five bits long, we need to convert all the instructions to binary, and then back into the correct hexadecimal notation.

0x0E 🡪 01110

0x6 🡪 0010

0x07 🡪 0011

0x2 🡪 010

Splitting the bits into four bit chunks,

0111 🡪 0x7

0011 🡪 0x3

0011 🡪 0x3

1010 🡪 0xA

So the final machine language instruction is “0x733A”.

## Example Assembly Language Program (Euclid’s Algorithm)

relPrime: addi $sp, 16

swn $rr, $sp, $0

ldi $t0, 4

swn $s1, $sp, $t0

li $t0, 8

swn $s2, $sp, $t0

ldi $t0, 12

swn $ra, $sp, $t0

ldi $s0, 2

ldi $s1, 1

la $s2, end

loop: move $s0, $a1

jal gcd

beq $v, $s1, $s2

addi $s0, 1

j loop

end: move $s0, $v

ldi $t0, 12

lwn $ra, $sp, $t0

ldi $t0, 8

lwn $s2, $sp, $t0

ldi $t0, 4

lwn $s1, $sp, $t0

lwn $s0, $sp, $0

ldi $t0, -16

add $sp, $t0

jr $ra

gcd: addi $sp, 12

swn $s0, $sp, $0

ldi $t0, -4

swn $s1, $sp, $t0

ldi $t0, -8

swn $s2, $sp, $t0

la $s0, returnB

la $s1, returnA

la $s2, amore

beq $a0, $0, $s0

loop: beq $a1, $0, $s1

bgt $a0, $a1, $s2

neg $s0, $a0

add $a1, $s0

j loop

amore: neg $s0, $a1

add $a0, $s0

j loop

returnA: move $a0, $v

j return

returnB: move $a1, $v

return: ldi $t0, 8

lwn $s2, $sp, $t0

ldi $t0, 4

lwn $s1, $sp, $t0

lwn $s0, $sp, $0

addi $sp, -12

jr $ra

## Example Assembly Language Fragments

# Iteration

Loop: add $s2 $s1 # Add $s1 to $s2

addi $s0 1 # Increment i by 1

beq $s0 $t0 loop # Check if i = N, if so, continue loop

# Conditional Statement

bne $s0 $s1 label

…

…

label: ….

# Reading Data from the Input Port

rp $s0

## RTL Instruction Descriptions

**add:** $rr = Reg[rd] + Reg[rs]

**addi:** $rr = Reg[rd] + imm

**and:** $rr = Reg[rd] & Reg[rs]

**beq:** if (Reg[rd] == Reg[rs]) PC = Reg[rt]

**bgt:** if (Reg[rd] > Reg[rs]) PC = Reg[rt]

**blt:** if (Reg[rd] < Reg[rs]) PC = Reg[rt]

**bne:** if (Reg[rd] != Reg[rs]) PC = Reg[rt]

**j:** PC = PC[15:11] || imm

**jal:** $ra= PC

PC = PC[15:11] || imm

**jr:** PC = Reg[rd]

**ldi:** Reg[rd] = imm

**lwn:** Reg[rd] = Mem[Reg[rs]] + Reg[rt]

**neg:** Reg[rd] = -1×Reg[rs]

**not:** Reg[rd] = !Reg[rs]

**noOp**: Do nothing.

**or:** Reg[rd] = Reg[rs] OR Reg[rt]

**ori:** $rr = Reg[rd] OR imm

**rdr:** $rdr = Reg[rd]

**rp:** Reg[rd] = input

**sll:** if (imm > 0) $rr = Reg[rd] << imm

else $rr = Reg[rd] >> |imm|

**swn:** Mem[Reg[rs]] = Reg[rd] + Reg[rt]

**wdr:** output = $rdr

## Register Transfer Statements

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | j | jal | b\*\* | jr | | add/or/and/sll/not/neg | addi/ori |
| IF | IR = Mem[PC]  PC = PC + 2 | | | | | | |
| ID | X | | A = Reg[IR[4:7]]  B = Reg[IR[8:11]]  C = Reg[IR[12:15]] | | A = Reg[IR[5:8]]  B = Reg[IR[9:12]] | | |
|  | | IMM = S/E(IR[9:15]) |
| EX | X | | X | | PC = A | ALUOut = A op B | ALUOut = AopALUOut |
| MEM | X | | IF (B op C)  PC = A | | X | X | X |
| WB |  | Reg[$ra] = PC | X | | X | Reg[$rr] = ALUOut | Reg[$rr] = ALUOut |
| PC = PC[12:15]||(IR[5:15]<<1) | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | noOp | ldi | lwn | | swn | rp | wp |
| IF | IR = Mem[PC]  PC = PC + 2 | | | | | | |
| ID |  | A = IR[5:8]  ALUOut = IR[9:15]<<9 |  | A = Reg[IR[4:7]] | | DROut = $dr | A = IR[4:7] |
| B = Reg[IR[8:11]]  C = Reg[IR[12:15]] | | |
| EX |  | ALUOut = A or ALUOut | ALUOut = B + C | | | X | X |
| MEM |  | X | MemOut = Mem[ALUOut] | | Mem[ALUOut] = A | X | $dr = A |
| WB |  | Reg[IR[5:8]] = ALUOut | Reg[IR[4:7]] = MemOut | | X | Reg[IR[4:7]] = DROut | X |

## Components & Control Signals

**PC**

* Inputs: input address (16-bit)
* Outputs: output address (16 bit)

The program counter keeps track of the current instruction in memory.

**Instruction Memory**

* inputs: address (16-bit)
* outputs: value (16-bit)

The instruction memory outputs the value of the instruction stored at the address.

**Data Memory**

* inputs: address (16-bit), input data (16-bit)
* outputs: output data (16-bit)
* controls: writeEnable (1-bit)

The data memory stores the input data at the given address. The output data is written when writeEnable is high.

**Register File**

* inputs: read1 (4-bit), read2 (4-bit), read3 (4-bit), writeRegister (4-bit), data (16-bit)
* outputs: out1 (16-bit), out2 (16-bit), out3 (16-bit)
* controls: writeEnable (1-bit)

The register file reads in data values from the registers specified by read1, read2, and read3. The data is sent out on out1, out2, and out3. The control signal writeEnable enables data to be stored according to the writeRegister input.

**ALU**

* inputs: A (16-bit), B (16-bit)
* outputs: zero, R (16-bit)
* controls: opcode (3-bit)

The arithmetic & logic unit performs a given operation (designated with the opcode) with A and B and returns the result R, or zero.

**IF/ID Register**

* inputs: IR (16-bit), PC (16-bit)
* outputs: IR (16-bit), PC(16-bit)

**ID/EX Register**

* inputs: PC (16-bit), EX (12-bit), Mem (7-bit), WB (5-bit), A (16-bit), B (16-bit), C (16-bit), RD (4-bit), Imm (16-bit), JTarget (11-bit), DR (16-bit)
* outputs: PC (16-bit), EX (12-bit), Mem (7-bit), WB (5-bit), A (16-bit), B (16-bit), C (16-bit), RD (4-bit), Imm (16-bit), JTarget (11-bit), DR (16-bit)
* controls: Clear (1-bit)

**EX/MEM Register**

* inputs: PC (16-bit), Mem (7-bit), WB (5-bit), A (16-bit), RD (4-bit), Result (16-bit), zero
* outputs: PC (16-bit), Mem (7-bit), WB (5-bit), A (16-bit), RD (4-bit), Result (16-bit), zero
* controls: Clear (1-bit)

**MEM/WB Register**

* inputs: PC (16-bit), WB (5-bit), RD (4-bit), Result (16-bit)
* outputs: PC (16-bit), WB (5-bit), RD (4-bit), Result (16-bit)
* controls: Clear (1-bit)

IF stage: The instruction is read from memory using the address in the PC and then placed inIF/ID register. PC is incremented by 4 then written back to PC and read on the following clock cycle.

ID stage: The IF/ID register supplies the immediate field, which is then sign extended to 16 bits. The value from the registers and tehh sign extended value are stored in the ID/EX register.

EX stage: The values from the IF/ID register are added by the ALU and the result is stored in the EX/Mem register.

Mem stage: The result from the ALU operation is now stored in the EX/MEM register.

WB stage: The result and rd register are sent to the inputs of the register component. Data is written to the rd register.

**Control**

* inputs: Opcode (5-bit), FunctionCode (2-bit), RD (4-bit), RT (4-bit), RS (4-bit)
* outputs: ALUOp (3-bit), ResultSrc (3-bit), WriteSrc (2-bit), PCSource (2-bit), BranchType (2-bit), WriteTar (2-bit), RSCont (2-bit), RTCont (2-bit), ALUSrcA (1-bit), ALUSrcB (1-bit), MemWrite (1-bit), DRWrite (1-bit), CP0Write (1-bit), RegWrite (1-bit)

Control determines which signals are to be high, based on the opcode, functionCode, and the values in rd, rt, and rs. The Control is responsible for selecting the correct inputs, and specifying the correct operation for the ALU to perform.

**CoProcessor0**

* inputs: enableData (16-bit), codeData (16-bit), returnPCData (16-bit)
* outputs: output (16-bit)
* controls: enableWrite (1-bit), codeWrite (1-bit), PCWrite (1-bit)

CoProcessor0 stores values needed while the processor is handling an interrupt. These values include the return PC once the interrupt is handled, the bit enabling interrupts, and the source from where the interrupt was originated.

**Branch Handler**

* inputs: Result (16-bit), zero
* outputs: newPCSrc (2-bit), clear (1-bit)
* controls: branchType (2-bit), PCSrc (2-bit)

The Branch Handler determines whether or not a branch should occur, based on the input result.

**Instruction Memory**

Memory was simply made with the CoreGen memory controller module built in Xilinx. We specified how big it should be and generated the file.

### Control Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Op | Op Code | Function Code | ALU SrcA | ALU SrcB | ALU Op | Result Src | Mem Write | Write Src | PC Src | Branch Type | DR Write | Write Target | CPO Write | Reg Write |
|  |  |  | [0] | [2][1] | [5][4][3] | [8][7] [6] | [9] | [11] [10] | [13][12] | [15][14] | [16] | [18] [17] | [19] | [20] |
| nop | 00000 | xxx | 0 | 0 | 000 | 000 | 0 | 00 | 00 | 00 | 0 | 00 | 0 | 0 |
| j | 00001 | xxx | x | x | xxx | xxx | 0 | xx | 01 | xx | 0 | xx | 0 | 0 |
| jal | 00010 | xxx | x | x | xxx | xxx | 0 | 11 | 01 | xx | 0 | 00 | 0 | 1 |
| jr | 00011 | xxx | x | x | xxx | xxx | 0 | xx |  |  | 0 | xx | 0 | 0 |
|  | 00100 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ldi | 00101 | xxx | x | x | xxx | 011 | 0 | 00 | 00 | xx | 0 | 00 | 0 | 1 |
| neg | 00110 | xxx | x | 0 | 100 | 000 | 0 | 00 | 00 | xx | 0 | 00 | 0 | 1 |
| not | 00111 | xxx | x | 0 | 101 | 000 | 0 | 00 | 00 | xx | 0 | 00 | 0 | 1 |
| mfc0 | 01000 | xxx | x | x | xxx | 010 | 0 | 00 | 00 | xx | 0 | 00 | 0 | 1 |
| mtc0 | 01001 | xxx | x | x | xxx | xxx | 0 | 01 | 00 | xx | 0 | 11 | 1 | 0 |
| rdr | 01010 | xxx | x | x | xxx | 001 | 0 | xx | 00 | xx | 0 | xx | 0 | 1 |
| wdr | 01011 | xxx | 0 | 0 | xxx | xxx | 0 | 00 | 00 | xx | 1 | 00 | 0 | 0 |
| addi | 01100 | xxx | 0 | 1 | 010 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
| ori | 01101 | xxx | 0 | 1 | 000 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
| add | 01110 | 000 | 0 | 0 | 000 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
| and | 01110 | 001 | 0 | 0 | 001 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
| or | 01110 | 010 | 0 | 0 | 010 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
| sll | 01110 | 011 | 0 | 0 | 011 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
|  | 01110 | 100 |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 01110 | 101 |  |  |  |  |  |  |  |  |  |  |  |  |
| sub | 01110 | 110 | 0 | 0 | 110 | 000 | 0 | 01 | 00 | xx | 0 | 10 | 0 | 1 |
|  | 01110 | 111 |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 01111 | xxx |  |  |  |  |  |  |  |  |  |  |  |  |
| bgt | 1000x | xxx | 1 | 0 | 110 | 000 | 0 | 00 | 10 | 00 | 0 | xx | 0 | 0 |
| blt | 1001x | xxx | 1 | 0 | 110 | 000 | 0 | 00 | 10 | 01 | 0 | xx | 0 | 0 |
| beq | 1010x | xxx | 1 | 0 | 110 | 000 | 0 | 00 | 10 | 10 | 0 | xx | 0 | 0 |
| bne | 1011x | xxx | 1 | 0 | 110 | 000 | 0 | 00 | 10 | 11 | 0 | xx | 0 | 0 |
| rp | 1100x | xxx | x | x | xxx | 100 | 0 | 00 | 00 | xx | 0 | 00 | 0 | 1 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| lwn | 1110x | xxx | 1 | 0 | 010 | 000 | 0 | 10 | 00 | xx | 0 | 00 | 0 | 1 |
| swn | 1111x | xxx | 1 | 0 | 010 | 000 | 1 | xx | 00 | xx | 0 | xx | 0 | 0 |

## Datapath Block Diagram



## System Testing

**Control:** Testing the control was fairly straightforward. We simply selected each opcode and then ran a testbench waveform with it and looked at the resulting values for the control signals. We determined what each signal should before the individual opcodes, so it was as simply as going line by line and looking at the resulting bits.

**Datapath:** The datapath sytem testing was much more difficult. Our original intention was to load values into registers, and test one instruction at a time. Upon finding out it is impossible to load values into registers, we decided to start with ldi, and work up through all of the other instructions. Once the ldi instruction was ensured, however, the simulation stopped working correctly for the most part, limiting our ability to test the rest of the system. We believe it to be mostly functional, but cannot prove it.

# Performance Data Summary

1. Total bytes required for Euclid’s Algorithm, relPrime, memory variables, and constants: N/A
2. Total instructions executed when relPrime is called (with 0x13B0): 54
3. Total cycles required to execute relPrime: N/A
4. Average cycles per instruction: 5
5. Cycle time for the design: N/A
6. Total execution time for relPrime: N/A
7. Gate count for design: 144,189

Device utilization summary: N/A

# Design Process Journal

### January 11th-12th

All four members met and decided on the processor’s instruction set. We had to make a design decision of having the op-codes have 4 or 5 bits. Realizing the importance of saving the other bits for immediate value and registers, we made a compromise of 4-bit opcodes for the R-type instruction, and 5-bits for the other type, therefore allowing us to have more than 16 instructions. We also made a list of what instructions would be particularly useful to us and found ways to combine some, such as “sll” where a negative input value would shift right, thus leaving more space for another instruction. David was particularly helpful in writing the initial code for Euclid’s Algorithm. Ideas of how to handle exceptions and interrupts were discussed and are in the process of planning. There are still a few things that need to be figured out, such as how to deal with too many R-Type Instructions, and any ways instructions can be cut when using immediate values.

### January 18th

Sam, Michael and David got together to get started on the RTL. Since we hadn’t yet talked about pipelining in class, we decided to do a multi-cycle RTL, and convert it to a pipelined RTL later. We created the RTL for all instructions that could fit on David’s whiteboard, which was the entire instruction set except for lwn, swn, rp and wp. Once we had an acceptable RTL set up, we decided to break until later when we had more information about exactly what a pipelined RTL looks like.

### January 19th

All four members met and we went over the current RTL, making changes in order to make the system pipelined instead of multi-cycled. We then added the last four instructions (lwn, swn, rp, wp) and verified the rest of the RTL. We then entered the RTL into the design document for safe keeping. As this was being done, we ensured that all addressing was done with 16 bits, and that it was consistent across all instructions. Once the RTL was in the design document, Sam looked up a datapath of a pipelined processor in the book, and we used that to list the components and signals we thought we would need for the system. Although this is liable to change later, we at least have a start on it now.

### January 25th

All four members met and we designed the datapath based on the RTL from the previous week and modeling off of the one on the book. We all discussed it on the whiteboard while Michael typed it up on the computer. Samuel helped with steps along the way as Dave acted as the leader and did the physical writing. Lindsay made changes to the RTL diagram and reviewed and updated previous work done.

### February 2nd

Due to scheduling conflicts between all four group members, we decided to start dividing tasks up. David finished building the ALU, and Michael and Sam made the Register File (including all the associated MUX’s) and Memory. Lindsay followed the design process and updated the design documents.

### February 9th

All four members met and continued work on the implementation. Michael and Dave worked on wiring up the previously made components and Sam and Lindsay made smaller registers.

### February 14th

All four members met and continued work on the implementation. David did the majority of the work with Xilinx while Michael and Sam assisted. Lindsay got all design documents up to date so that implementation can be easier.

### February 19th - 21st

All four members met and continued work on the implementation. David continued work on getting the compiler and assembler working. He and Michael worked on debugging the system, after Sam finished implementing some of the components. Lindsay continued work on the design documents and prepared a PowerPoint presentation for the final exam.