Design Document
Team 1E
Last Updated: November 12, 2017

16-Bit MRAA

16-Bit Multi-Register Accumulator Architecture

By: Russell Johnson, Griffin Steffy,

Charley Beeman, Stella Park

Table of Contents

Table of Contents 2

Instructions 3

M-type 3

I-type 3

R-type 5

Registers 6

Procedure Call Conventions 7

Addressing Modes 7

Relative Prime & Euclidean Algorithm - Assembly & Machine Code 8

Assembly Language Fragments 10

RTL 11

General Components Specifications 11

Testing RTL 17

Instructions Tested 17

Datapath 18

Integration of Datapath Plan 19

Integration Plan Diagram 20

Control Unit Finite State Machine 21

Control Unit Testing 21

System Testing 22

System Testing Plan 22

Performance 23

Input/Output 24

Instructions

Throughout this section, $acc refers to the dedicated accumulator register, $reg refers to whatever register is being used for the instruction call, and imm represents the immediate value used for the instruction call.

M-type

These instructions are used for handling memory operations: load and store. It is also used for jr (jump register). The immediate is used as the offset, by adding it to the value in the register.

Opcode (4 bits)	Register (4 bits)	Immediate (8 bits)

Instruction	**Description**	**Syntax**	**Opcode**
load	$acc = Mem[$reg + SE(imm)]	load $reg, imm	0x0
store	Mem[$reg +SE(imm)] = $acc	store $reg, imm	0x1
jr	Jump to address in register ($reg + x)	jr $reg, imm	0x2

I-type

These instructions are used for handling immediates. This includes loading to and from memory, jumping to different addresses, along with doing some simple logical and arithmetic operations involving those immediates. The lui instruction loads the upper 4 bits of the immediate since an I-type can only use a 12-bit immediate; therefore, when combined you get a 16-bit immediate value.

.

Opcode (4 bits)	Immediate (12 bits)

Instruction	**Description**	**Syntax**	**Opcode**
jump	Jump to address	jump imm	0x3
jal	Jump and link	jal imm	0x4
jump1	Jumps if the MSB of acc is 1 (acc < 0)	jump1 imm	0x5
jump0	Jumps if the MSB of acc is 0 (acc >= 0)	jump0 imm	0x6

Link to: *Table of Contents*

shiftl	acc = acc<<imm	shiftl imm	0x7
shiftr	Acc = acc>>imm	shiftr imm	0x8
lui	Loads upper 4 bits of imm into the upper 4 bits of $acc	lui imm	0x9
ori	Ors imm with the lower 3 bytes of $acc, stores in $acc	ori imm	0xA
andi	Ands imm with $acc, stores in $acc	andi imm	0xB
loadi	Loads immediate into $acc	loadi imm	0xC
addi	Add immediate to $acc register	addi imm	0xD
X	X	X	0xE
R TYPE	This opcode is reserved for R-Type instructions		0xF

Ex. Machine code

loadi 0xFFF % $acc <- 0xFFF

1100 1111 1111 1111

R-type

These instructions are used for handling registers. This includes moving the contents of a register to the accumulator as well as from the accumulator to a register. Also, there are instructions for adding, subtracting, anding, and oring registers with the accumulator. The user has the ability to choose whether to store the result in the accumulator or the register based on the “to acc” bit value (1 = acc, 0 = reg). All R-types have an opcode of 0xF, and the func code is used to determine which function is being used in the ALU.

Opcode (4 bits)	Register (4 bits)	Func (3 bits)	XXX (4 bits)	to acc (1 bit)

Instruction	**Description**	**Syntax**	**Opcode**	**Func**
move	Moves from $acc to temp reg or from temp reg to $acc depending on the “to acc” bit (1 = acc, 0 = reg)	move $reg, 1 move $reg, 0	0xF	0x0
add	Adds value of the accumulator register to the value of another specified register, result is stored in accumulator or register based on “to acc” bit	add $reg, 1 add $reg, 0	0xF	0x1
sub	Subtracts the value from a register from the value of the accumulator “acc-t0”, and the result is stored in the accumulator or register based on “to acc” bit	sub $reg, 1 sub $reg, 0	0xF	0x2
subr	Subtracts the value from the accumulator from the value of the register “t0-acc”, and the result is stored in the accumulator or register based on the “to acc” bit	subr $reg, 1 subr $reg, 0	0xF	0x3
or	acc||reg is stored in acc or reg based on “to acc” bit	or $reg, 1 or $reg, 0	0xF	0x4
and	acc&reg is stored in acc or reg based on “to acc” bit	and $reg, 1 and $reg, 0	0xF	0x5

Ex. Machine code

```
add $t0 1      # $acc <- $acc + $t0
1111 1000 0001 0001

move $t0 0    # $t0 <- $acc
1111 1000 0000 0000
```

To $acc: The “to acc” bit determines where the result is stored for the operation. If the bit is set to 1, the result from the operation will be sent to the acc register. If the result is 0, it will be stored in the specified register.

Registers

Register	Description	Hex
$0	Holds a constant value of 0	0x0
$sp	Points to the last active position on the stack	0x1
$ra	Holds return address	0x2
$iszero	Holds a 1 in the MSB if the last ALU operation resulted in 0 $iszero = SE[iszero ALU flag]	0x3
$s0-$s3	Saved Temporary Registers	0x4-0x7
$t0-$t4	Temporary registers	0x8-0xC
$in	Used for accessing hardware input	0xD
$out	Used for outputting to hardware	0xE
$acc	Accumulator register	0xF

Procedure Call Conventions

- Arguments and return values are stored in temporary registers as defined in the procedure being called
- $ra represents the return address, and will need to be backed up on the stack before procedure calls are made.
- The saved temporary registers ($s0-$s3) must be backed up before being used and restored after
- Depending on the use of the program the user might also need to backup the temporary register(s) to the stack as well.

Addressing Modes

Relative-to-PC Addressing:
This will be used by jump0 and jump1 of the addressing I-type instructions. PC-relative addressing will sign extend the 12-bit immediate.The result of that operation will be added to PC (PC+1) and stored back into PC.

PC = (PC+1) + [SE(12-bit imm)]

Register+Offset Addressing:
This addressing mode will be used for M-type instructions. The address will be the offset immediate added to value stored in the in the register specified.

Address = $reg + SE(imm)

Pseudo-Direct Addressing:
This is used for the jump and jal instructions. It will set the value of PC to the immediate shifted left 1 appended to the top 4 bits of PC.

PC = $\text{PC}_{15\text{-}12}$ + (imm)[11:0]

Extended-Direct Addressing:
This mode will be used in logical and arithmetic instructions. Depending on the type of instruction the program will either zero extend or sign extend the 12-bit immediate of the I-Type instruction. If it is an arithmetic instruction it will be sign extended, and if it is a logical instruction it will be zero extended.

Relative Prime & Euclidean Algorithm - Assembly & Machine Code

Assembly		Machine Code
# Move Set Input as the argument		
move $in, 1	# $acc = $in	1111110100000001,
move $t0, 0	# $t0 = $acc	1111100000000000,
# During Execution		
# s0 → n		
# s1 → m		
relPrime:		
# Backup $ra, $s0, $s1		
move $sp, 1	# $acc = $sp	1111000100000001,
addi -3	# $acc = $acc - 3	1101111111111101,
move $sp, 0	# $sp = $acc	1111000100000000,
move $ra, 1	# $acc = $ra	1111001000000001,
store $sp, 0	# Mem[$sp + 0] = $acc	0001000100000000,
move $s0, 1	# $acc = $s0	1111010000000001,
store $sp, 1	# Mem[$sp+1] = $acc	0001000100000001,
move $s1, 1	# $acc = $s1	1111010100000001,
store $sp, 2	# Mem[$sp+2] = $acc	0001000100000010,
move $t0, 1	# $acc = $t0	1111100000000001,
move $s0, 0	# $s0 = $acc	1111010000000000,
loadi 2	# $acc = 2	1100000000000010,
move $t1, 0	# $t1 = $acc	1111100100000000,
move $s1, 0	# $s1 = $acc	1111010100000000,
loop:		
jal gcd	# goto gcd	0100000000100111,
addi -1	# $acc = $acc - 1	1101111111111111,
move $iszero, 1	# $acc = $iszero	1111001100000001,
jump1 exitloop	# goto exitloop if $acc[15] = 1	0101000000000111,
move $s1, 1	# $acc = $s1	1111010100000001,
addi 1	# $acc = $acc + 1	1101000000000001,
move $s1, 0	# $s1 = $acc	1111010100000000,
move $t1, 0	# $t1 = $acc	1111100100000000,
move $s0, 1	# $acc = $s0	1111010000000001,
move $t0, 0	# $t0 = $acc	1111100000000000,
jump loop	# goto loop	0011000000010000,
exitloop:		
# Put m into output reg		
move $s1, 1	# $acc = $s1	1111010100000001,
move $out, 0	# $out = $acc	1111111000000000,

```
        # Restore $ra, $s0, $s1
        load $sp, 2           # $acc = Mem[$sp+2]                  0000000100000010,
        move $s1, 0           # $s1 = $acc                         1111010100000000,
        load $sp, 1           # $acc = Mem[$sp+1]                  0000000100000001,
        move $s0, 0           # $s0 = $acc                         1111010000000000,
        load $sp, 0           # $acc = Mem[$sp+0]                  0000000100000000,
        move $ra, 0           # $ra = $acc                         1111001000000000,
        move $sp, 1           # $acc = $sp                         1111000100000001,
        addi 3                # $acc = $acc + 3                    1101000000000011,
        move $sp, 0           # $sp = $acc                         1111000100000000,
        jump finish           # goto end of program                0011000000111110,
gcd:
        move $t0, 1           # $acc = $t0                         1111100000000001,
        addi 0                # $acc = $acc + 0                    1101000000000000,
        move $iszero, 1       # $acc = $iszero                     1111001100000001,
        jump0 gcdloop         # goto gcdloop if $acc[15] = 0       0110000000000011,
        move $t1, 1           # $acc = $t1                         1111100100000001,
        move $t2, 0           # $t2 = $acc                         1111101000000000,
        jr $ra, 0             # goto address in $ra+0              0010001000000000,
gcdloop:
        move $t1, 1           # $acc = $t1                         1111100100000001,
        addi 0                # $acc = $acc + 0                    1101000000000000,
        move $iszero, 1       # $acc = $iszero                     1111001100000001,
        jump1 exitgcdloop     # goto exitgcdloop if $acc[15] = 1   0101000000001001,
        move $t1, 1           # $acc = $t1                         1111100100000001,
        sub $t0, 1            # $acc = $acc - $t0                  1111100001000001,
        jump0 else            # goto else if $acc[15] = 0          0110000000000011,
        move $t1, 1           # $acc = $t1                         1111100100000001,
        subr $t0, 0           # $t0 = $t0 - $acc                   1111100001100000,
        jump gcdloop          # goto gcdloop                       0011000000101110,
else:
        move $t0, 1           # $acc = $t0                         1111100000000001,
        subr $t1, 0           # $t1 = $t1 - $acc                   1111100101100000,
        jump gcdloop          # goto gcdloop                       0011000000101110,
exitgcdloop:                                                       1111100000000001,
        move $t0, 1           # $acc = $t0                         1111101000000000,
        move $t2, 0           # $t2 = $acc                         0010001000000000,
        jr $ra, 0             # goto address in $ra + 0
finish:
```

Assembly Language Fragments

Description	**Assembly**	**Machine Code**
Loading an address	lui 0x0A00 # $acc = 0xA000 ori 0x0BCD # $acc = 0xABCD jr $acc, 0 # jump to address in $acc	1001101000000000 1010101111001101 0010111100000000
Iteration	addi 1	1101000000000001
Cond:$t0 > $acc	sub $t0, 1 jump1 label	1111100000100001 0101xxxxxxxxxxxx
Cond: $t0 < $acc	subr $t0, 1 jump1 label	1111100000110001 0101xxxxxxxxxxxx
Cond: $t0 <= $acc	sub $t0, 1 jump0 label	1111100000100001 0110xxxxxxxxxxxx
Cond: $t0 >= $acc	subr $t0, 1 jump0 label	1111100000110001 0110xxxxxxxxxxxx
Cond: $t0 == $acc	sub $t0, 0 move $iszero, 1 jump1 label	1111100000100000 1111001100000001 0101xxxxxxxxxxxx
Cond: $t0 != $acc	sub $t0, 0 move $iszero, 1 jump0 label	1111100000100000 1111001100000001 0110xxxxxxxxxxxx
Read Data From Input	move $in, 1	1111110100000001
Write Data to Output	lui 0x0A00 ori 0x0BCD move $out, 0	1001101000000000 1010101111001101 1111111000000000

RTL

See attached document.

General Components Specifications

ALU ✔

Description:

Takes in 2 16-bit inputs and performs an operation. The 16-bit output goes to ALUout. Operations: pass_b (000), add (001), sub (010), subr (011), or (100), and (101), lsift (110), rshift (111)

It also outputs a 1 bit value that is determined by if the output was 0 or not (stored in $iszero register).

Input:

2 16 bit values

Output:

1 16 bit value

2 1 bit signals

Control Signals:

ALUctrl (3 bits)

RTL Symbols:

op

Building:

The ALU will consist of 2 16 bit input data busses, 2 bit input control signal, 1 16 bit output data bus, and 1 bit output control signal (iszero, overflow). We will use the control signal to decide which output of the operations to use via muxes. The operations that can be performed will be implemented through blocks of each type of operation. We will aslo

Testing:

We will test each function of our ALU individually. Since we know that each of those components will work we can specify edge cases for each type of operation and test those. We will also test each input for the ALU ctrl and ensure that it corresponds to the correct type of operation. We will test a case that results in 0.

Adder ✔

Description:

Takes in 2 16-bit inputs, adds them together creating a 16 bit output

Input:

2 16 bit values

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

+

Building:

It will have two 16 bit input data busses as well as one 16 bit output data bus. First we will design a 1 bit adder, then extend that block to become a 4 bit adder. We'll extend that 4 bit adder to a 16 bit adder that takes in our 2 16 bit data busses and outputs the 16 bit data on the bus.

Testing:

We will specify certain 16 bit values that could cause our adder to fail, and test them with our adder. Our edge case that we will test is when there is a carry on the MSB of the output.

Memory and Instruction Data ✔

Description:

Takes in a 16-bit address of a location in memory as well as 16-bits of data. If MemWrite=0, then the input address is read. If MemWrite=1, then the input data is written to the input address.

Input:

3 16 bit values

Clock

Output:

1 16 bit value

Control Signals:

MemWrite (1 bit)

RTL Symbols:

Mem[address]

Building:

This will have 3 16 bit input data busses (Read Address, Write Address, and Write Data) as well as 1 16 bit output data bus. The memory unit itself will be built using Xilinx's core generator. We will use distributed memory to optimize performance.

Testing:

We will test this by writing data to specific addresses in memory and reading it to verify the correct data was written.

<u>Reg File</u> ✔

Description:

Takes in 1 4-bit register address, the 4-bit address of the register to write to, and 16-bits of data to write. If RegWrite=1, then the data is written to the register specified to write to. The data in the register that is input to the Reg File as well as the data in the acc register are output (16 bits each).

Input:

2 4 bit values, 2 16 bit values

Output:

3 16 bit values

Control Signals:

RegWrite (1 bit)

RTL Symbols:

Reg[address]

Building:

We will have all of the registers connected with a 32 16-bit bus to a 16-bit bus output mux o select which is being read from, and another similar mux to determine which is being written to.

1 16 bit input data bus will be for the input (external)

1 16 bit output data bus will be for output (external)

Testing:

By inputting a 16-bit data signal and the address of the register we will write to a specific register and the we will select the same mux to read it to see if it properly stored the value. We will also check to ensure none of the other registers were modified.

<u>Sign Extenders</u> ✔

Description:

Takes a 12 bit or 1 bit input and sign extends it to 16-bits

Input:

1 12 bit value or 1 1 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

SE(value)

Building:

We will have a 12 bit or 1 bit data bus, and through verilog, we will extend the top bit to the

upper 4 bits of the 16 bit data bus

Testing:

By inputting a 12 bit or 1 bit value we will test that the 16 bit value has sign-extended properly, Ensuring that the decimal value of our input has not changed whether the input is a positive or negative number.

Zero Extenders ✔

Description:

Takes a 12-bit or 4 bit input and zero extends it to 16-bits

Input:

1 12 bit or 4 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

ZE(value)

Building:

We will have 4 or 12 bit data bus, and through verilog, we will insert 0s as the upper 4 or 12 bits of the 16 bit data bus

Testing:

By inputting a 12 bit value we will test that the 16 bit value has zero-extended properly by using. We will test if the 12-bit data bus's MSB is a 1 or 0 and ensure that it still extends only 0s to the upper 4 or 12 bits of the output.

Left Shift 12 ✔

Description:

Takes a 16-bit input and left shift it by 12 creating a 16-bit output

Input:

1 16 bit value

Output:

1 16 bit value

Control Signals:

N/A

RTL Symbols:

<<4

Building:

We will have 16 bit data bus and using verilog we will move the lower 4 bits of the input to the upper 4 bits of the output, and insert 0s in the lower 12 bits

Testing:

We will input a 16 bit value and after processing it through, we will see that it has been left shifted by 12 properly. We will test to make sure the last 12 bits ([11:0]) of the output are set to 0.

Mux 16 bit (Multiple Input) 1 Output ✔

Description:

Take in up to 6 inputs (in our datapath) and using a control signal chooses one of those inputs to become the output

Input:

2, 3, 4, and 6 16 bit values

Output:

1 16 bit value

Control Signals:

Depending on the mux it is either a 1, 2, or 3 bit signal. Each signal is specific to the mux. *Signals:* IorM, ItypeSel, Asel, Bsel, jcontrol, destData,

RTL Symbols:

N/A

Building:

It will have 2, 3, 4, or 6 16 bit input data buses and 1 16 bit output data bus. The 1, 2, or 3 bit control signal determines which input to choose.

Testing:

We will go through all possibilities of the control signal and make sure the output is what is expected for that specific control signal.

Mux 4 bit (3 Input) 1 Output ✔

Description:

Take in 3 4 bit inputs and using a control signal chooses one of those inputs to become the output

Input:

3 4 bit values

Output:

1 4 bit value

Control Signals:

1 2 bit control signal, destAdd

RTL Symbols:

N/A

Building:

It will have 3 4 bit input data busses and 1 4 bit output data bus. The 2 bit control signal determines which input to choose.

Testing:

We will test the mux with each value for the control signal and make sure that the output is what we expect.

<u>Control Unit</u> ✔

Description:

Takes in 2 4-bit signals, 1 1 bit value, and possibly the MSB of the Accumulator Register and determines the control signals for the rest of the datapath

Input:

1 4 bit value (opcode)
1 4 bit value (func code)
5 1 bit values (clock, reset, toacc (IR[0]), MSB bit of $acc register, noOp (If IR = 0, noOp = 1))

Output:

13 1 bit values
2 2 bit value

Control Signals:

N/A

RTL Symbols:

Control signals are listed in the **Datapath** *section*

Testing RTL

- In order to test the RTL that we created, we picked instructions out to test every scenario. The instructions chosen were load, jr, store, [sub, subr, & move with toacc = 1], [sub, subr, & move with toacc = 0], jump, ori, addi, jump0, jump1, and lui. We made a test example for each instruction shown in the table below. We then verified each one worked. After testing each instruction, we determined our RTL works as expected.

Instructions Tested

Test Code for Instruction	**Outcome**
load $t0, 4	Success!
jr $t0, 0	Success!
store $t0, 4	Success!
sub $t0, 1 sub $t0, 0	Success!
move $t0, 1 move $t0, 0	Success!
jump label	Success!
ori 0xABC	Success!
addi 0xABC	Success!
jump0 label	Success!
Jump1 label	Success!
lui 0xA00	Success!

Datapath

See attached document.

List of Control Signals

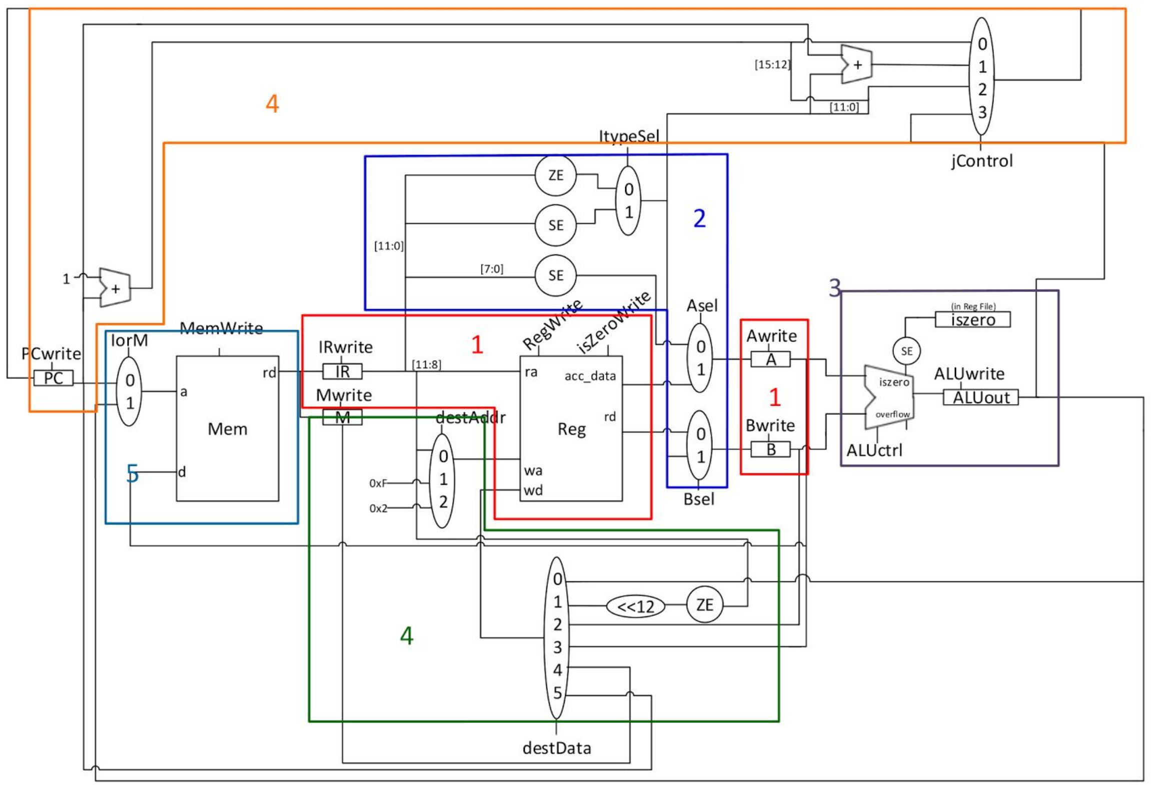
- PCwrite - Determines whether or not you write a new value to the PC
- IorM - Determines what you are fetching from memory (Instruction or other memory data)
- MemWrite - allows writes from memory
- IRwrite - Determines whether or not you write the instruction to IR
- Mwrite - determines whether or not to write to M
- destAddr - Comes from the LSB in instruction and if the instruction is a jal or not goes to $acc register or another register
- destData - Determines what data can be written the to Register File
- ItypeSel - Chooses between a ZE and SE immediate
- Asel - Chooses which value is stored in the A register
- Bsel - Chooses which value is stored in the B register
- Awrite - Enables the A register to be written to
- Bwrite - Enables the B register to be written to
- RegWrite - Allows the Register File to be written to
- isZeroWrite - Allows the ALU to write to the isZero register
- ALUctrl - Controls the operation for the ALU
- ALUwrite - determines whether or not to write to ALUOut
- jControl - Determines what the new PC Value will be

Integration of Datapath Plan

1. Add a RegFile, IR, A, & B
    - Test that the RegFile properly reads the register specified from IR and stores the ACC register data in A as well as the specified register's data in B
    - Test that the RegFile properly writes data to a register by setting the "wa" input to RegFile to acc's address and the "wd" input to the RegFile to a constant 16-bit value. Then read acc and verify that A holds the new value.
2. Add 8-bit SE, Asel mux, Bsel mux, 16-bit SE and ZE, Imm mux
    - Test that we can properly choose the value that gets stored in A and B correctly based off the immediates that come from the Instruction or the data being read from the register file
3. Add ALU, iszero SE, ALUOut register
    - Test that we can operate on the ALU with the values in the A and B registers
    - Test that we can detect the if the operation ended in 0 and store that in the iszero register.
    - Test that we can properly write the output of the ALU to ALUOut
4. Add muxes used to write to register and M register
    - Implement the M register, but it will not be connected to memory yet. It will just be set to a value.
    - Make sure we can properly select the correct write address and write data for the reg file from the two corresponding muxes
5. Add Jump Control & PC Control
    - Implement PC register, and jControl mux
    - Test PC = PC+1 with adder, jcontrol = 0, and PCWrite
    - Test PC = PC + ZE(12bit imm) with adder, jcontrol = 1, and PCWrite
    - Test PC = PC[15:12]+ZE(12bit imm)[11:0], jcontrol = 2, and PCWrite
    - Test PC = ALUOut with jcontrol = 3, and PCWrite
    - Test PC = PC with no PCWrite signal (for jal instructions)
6. Add Memory Unit
    - Test the ability to load instructions based off the value in the PC register and load the result in the IR register (Instruction should also be stored in the M register)
    - Test the ability to load values from memory at location of ALUOut, and store the result in the MDC Register
    - This step adds an option to write to the register file with data from memory. We will test this case in this step as well.
7. Add Control Unit
    - Add the control unit to implemented datapath
    - Run through as many different instructions as it takes in order to make sure that every state in from the finitie state machine has been tested and works properly
    - Make any modifications to the control unit that is needed
8. Go over the final schematic. Make sure it is as efficient as possible and that all units work properly.

9. Add input and output integration with the FPGA
    - Test different inputs and make sure that the correct output is displayed on the FPGA

Integration Plan Diagram



Control Unit Finite State Machine

See attached document.

Control Unit Testing

For testing the control units, we created a loop to iterate through each opcode. After printing out the waveform we verified that the correct control signals were triggering/not-triggering during the correct cycle for each opcode. Additional testing of each was done during the final stage of integration in Xilinx. We made sure that every state in the control unit worked as anticipated. Changes that were made are outlined in the Design Journal Document.

System Testing

System Testing Plan

We will begin testing our system by creating small fragments of code that test one type of instruction at a time. With this we will be able to verify that our entire datapath is working for each instruction. After testing the instructions we will move to a more complicated fragment of code, that could cause problems with timing, like jumps and memory. Once we have verified that the mini-program works on our processor correctly, we will test the euclidean formula program that we wrote at the beginning of the project for milestone 1 with a smaller value like 0xC. If we do not get the correct answer, we will use the waveforms to detect issues and resolve them accordingly. After we verify our system is working correctly with a small input value, we will test the desired value of 0x13B0 to ensure that we are working at full capacity.

Performance

- Total Number of Bytes Required:
  - Number of Bytes for Program: 128
  - Number of Bytes for Stack: 6
  - **Total for Program & Stack: 134**

- Total Number of Instructions executed when n = 0x13B0
  - **102111**

- Total Number of cycles required when n = 0x13B0
  - **408,458**

- Cycle time for design
  - **10.669ns**
  - **93.733 MHz**

- Total execution time when n = 0x13B0
  - $CPI = \frac{\# Cycles}{\# Inst} = \frac{405454}{102111} = 4.1$
  - $Exec.\ Time = CPI \times Cycle\ time \times \#\ instructions = 4.1 \times 10.669n \times 102111$

    = 4.358ms
  -
- Device utilization summary

  Selected Device : 3s500efg320-4

Number of Slices:	*475 out of*	*4656*	*10%*
Number of Slice Flip Flops:	*403 out of*	*9312*	*4%*
Number of 4 input LUTs:	*751 out of*	*9312*	*8%*
Number of IOs:	*35*		
Number of bonded IOBs:	*35 out of*	*232*	*15%*
Number of BRAMs:	*1 out of*	*20*	*5%*
Number of GCLKs:	*1 out of*	*24*	*4%*

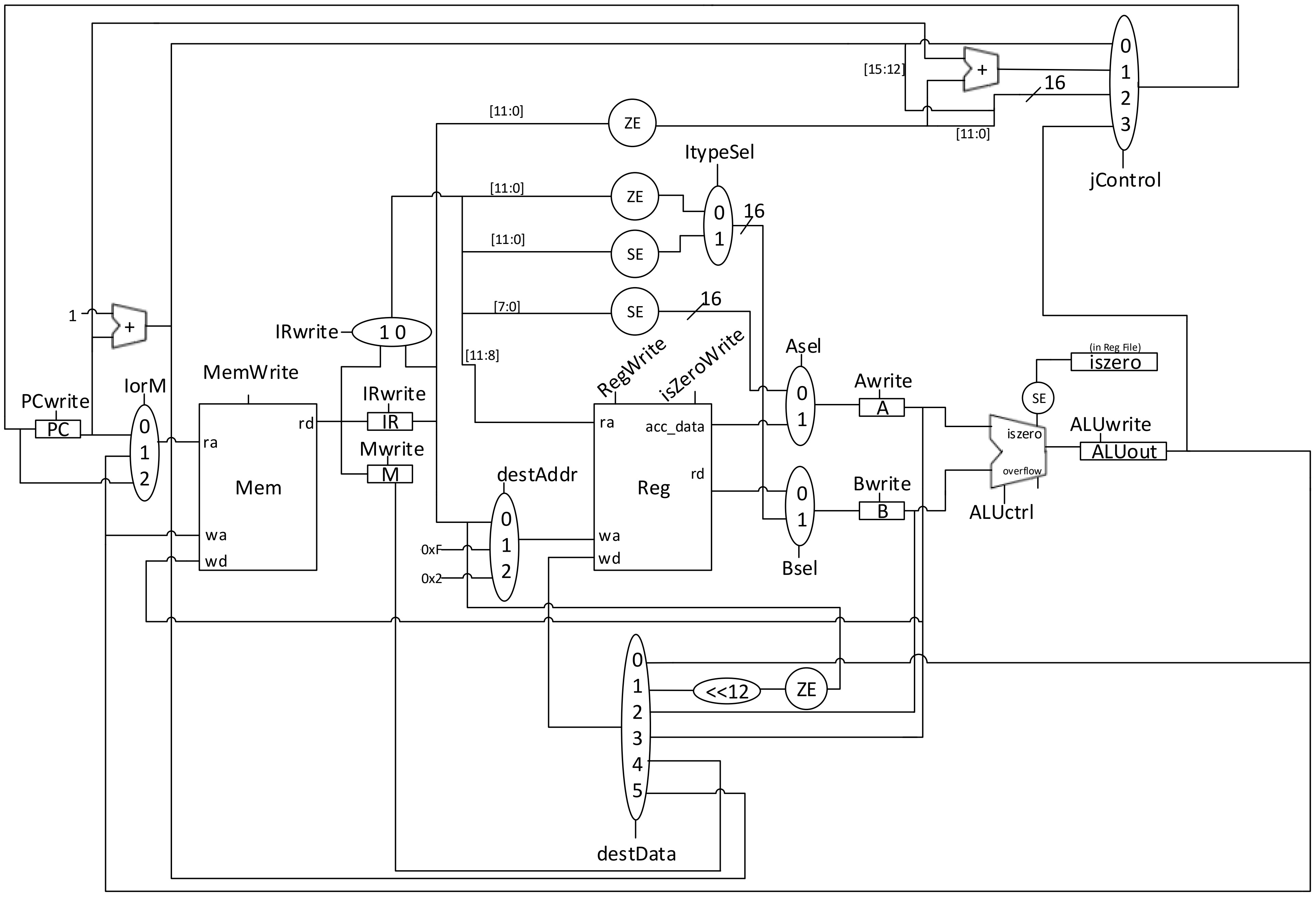
Input/Output

We decided to put our processor on the Spartan-3e FPGA board for one of our extra features. We set it up so that the input can be changed by using the rotary button, and both the input and output values are displayed on the lcd. The button directly below the rotary button is used as the reset button for the processor. The user must first move the rotary button until the desired input is displayed on the LCD and then hit the reset button in order to run the new input through the processor and have the correct corresponding output displayed on the LCD.

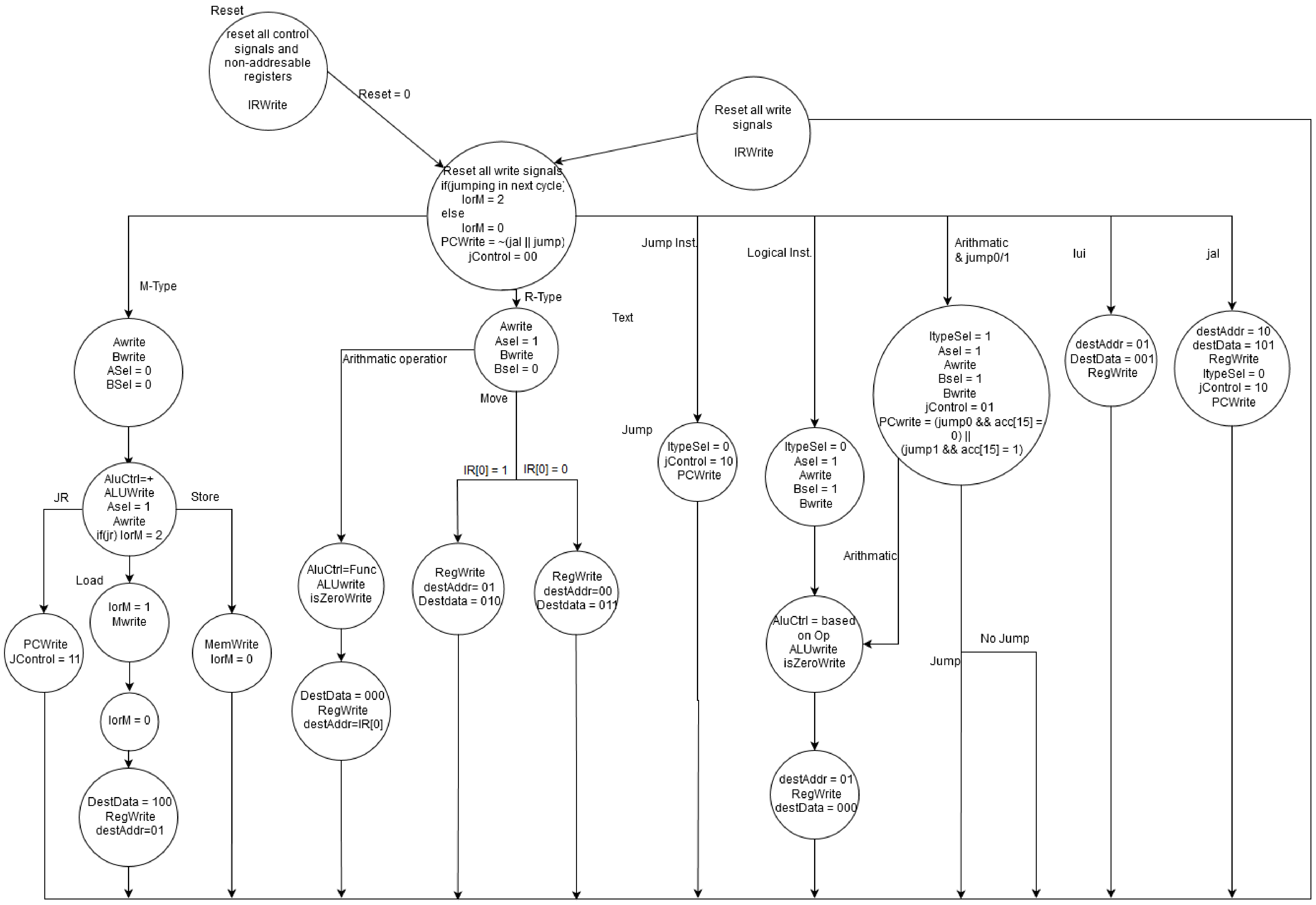
Link to: *Table of Contents*

Cycles	**Multicycle RTL**						
	M-Type			**R-Type**			
	JR	**Load**	**Store**	**add, sub, subr, or, and**		**move**	
				toacc = 1	**toacc = 0**	**toacc = 1**	**toacc = 0**
0 (reset)	IR = Mem[PC] (reset state)						
1	PC=PC+1						
2	A = SE[IR[7:0]] B= Reg[IR[11:8]]			A = Reg[0xF] (acc) B = Reg[IR[11:8]]			
3	ALUout = A +B A = Reg[0xF] (acc)			ALUout = A op B		Reg[0xF] (acc) = B	Reg[IR[11:8]] = A
4	PC=ALUout	M=Mem[ALUout]	Mem[ALUout] = A	Reg[0xF] (acc) = ALUout	Reg[IR[11:8]] = ALUout	IR = Mem[PC]	
5	*Jump Delay Cycle*	*Memory Delay Cycle*	IR = Mem[PC]				
6	IR = Mem[PC]	Reg[0xF] (acc)=M					
7		IR = Mem[PC]					

Cycles	Multicycle RTL					
	I-Type					
	Sign Extend		Zero Extend			
	Jump0, and Jump1	**Arithmetic Inst.**	**Logical Inst.**	**Jump Inst.**	**lui**	**jal**
0 (reset)	IR = Mem[PC] (reset state)					
1	PC=PC+1					
2	A = Reg[0xF] (acc) B = SE(IR[11:0]) if (IR[15:12] == 0x6 && acc[15] = 0) || (IR[15:12] == 0x5 && acc[15] = 1) then PC = PC + SE(IR[11:0])		B=ZE(IR(11:0)) A = Reg[0xF] (acc)	PC = PC[15:12]+ZE(IR[11:0])	Reg[0xF] (acc) = ZE(IR[11:8])<<12	Reg[0x2] = PC PC = PC[15:12]+ZE(IR[11:0])
3	*Jump Delay Cycle (If Branching)* *Otherwise Straight to Cycle 4*	ALUout = A op B		*Jump Delay Cycle*	IR = Mem[PC]	*Jump Delay Cycle*
4	IR = Mem[PC]	Reg[0xF] (acc) = ALUout				IR = Mem[PC]
5		IR = Mem[PC]				
6						
7						



PCwrite
PC
IorM
0
1
2
1
+
MemWrite
Mem
ra
wa
wd
rd
IRwrite
1 0
IRwrite
IR
Mwrite
M
[11:0]
ZE
[11:0]
ZE
[11:0]
SE
[7:0]
SE
16
[11:8]
ItypeSel
0
1
16
[15:12]
+
16
[11:0]
jControl
0
1
2
3
destAddr
0xF
0x2
0
1
2
RegWrite
isZeroWrite
Reg
ra
acc_data
rd
wa
wd
Asel
0
1
Awrite
A
Bsel
0
1
Bwrite
B
ALUctrl
iszero
overflow
SE
(in Reg File)
iszero
ALUwrite
ALUout
destData
0
1
2
3
4
5
<<12
ZE


Reset
reset all control signals and non-addresable registers
IRWrite
Reset = 0
Reset all write signals
IRWrite
Reset all write signals
if(jumping in next cycle)
lorM = 2
else
lorM = 0
PCWrite = ~(jal || jump)
jControl = 00
M-Type
R-Type
Jump Inst.
Logical Inst.
Arithmatic & jump0/1
lui
jal
Text
Awrite
Bwrite
ASel = 0
BSel = 0
Awrite
Asel = 1
Bwrite
Bsel = 0
Arithmatic operatior
Move
Jump
ItypeSel = 0
jControl = 10
PCWrite
ItypeSel = 0
Asel = 1
Awrite
Bsel = 1
Bwrite
ItypeSel = 1
Asel = 1
Awrite
Bsel = 1
Bwrite
jControl = 01
PCwrite = (jump0 && acc[15] = 0) || (jump1 && acc[15] = 1)
destAddr = 01
DestData = 001
RegWrite
destAddr = 10
destData = 101
RegWrite
ItypeSel = 0
jControl = 10
PCWrite
AluCtrl=+
ALUWrite
Asel = 1
Awrite
if(jr) lorM = 2
JR
Store
Load
IR[0] = 1
IR[0] = 0
PCWrite
JControl = 11
lorM = 1
Mwrite
MemWrite
lorM = 0
AluCtrl=Func
ALUwrite
isZeroWrite
RegWrite
destAddr= 01
Destdata = 010
RegWrite
destAddr=00
Destdata = 011
Arithmatic
AluCtrl = based on Op
ALUwrite
isZeroWrite
Jump
No Jump
lorM = 0
DestData = 000
RegWrite
destAddr=IR[0]
DestData = 100
RegWrite
destAddr=01
destAddr = 01
RegWrite
destData = 000