Milestone 1:

Ask Micah

Assembly Language Specifications:

This is an accumulator-based process with two registers. The first register, named $t, is used to save temporary values. The second register, named $s, is used to save values that will be needed in the future. We store instructions in address begin with 00000. Instead we store arguments begin with first 5bits of PC.

The instructions are as follows:

R-Type

Addition (with overflow): Put the value from register and value from specified address into ALU and then do addition. After that, send the result back into the register.

And: Put the value from register and value from specified address into ALU and then do “and” operation. After that, send the result back into the register.

Or: Put the value from register and value from specified address into ALU and then “or” operation. After that, send the result back into the register.

Subtract (with overflow): Put the difference of the register and specified address into the register.

Set Less Than: Set register to 1 if less than specified address, and to 0 otherwise.

Jump: Unconditionally jump to the instruction at the specified address.

Load Word: Load the 16 bit word at address into the register.

Store Word: Store the word from the register at address.

|  |  |  |  |
| --- | --- | --- | --- |
| Op  4 | Extra Space  3 | Address  8 | Register  1 |

I-Type

Addition Immediate (with overflow): Put the sum of the register and the sign-extended immediate into register.

And Immediate: Put the logical AND of register and sign-extended immediate into register.

Or Immediate: Put the logical OR of register and sign-extended immediate into register.

Load Upper Immediate: Loads the upper 8 bits of the immediate into the register.

Load Lower Immediate: Loads the lower 8 bits of the immediate into the register.

|  |  |  |
| --- | --- | --- |
| Op  4 | Immediate  11 | Register  1 |

B-Type

Branch on Equal: Conditionally branch the number of instructions specified by the function if the register equals the immediate.

Branch on Not Equal: Conditionally branch the number of instructions specified by the function if the register is not equal to the immediate.

|  |  |  |  |
| --- | --- | --- | --- |
| Op  4 | Immediate  3 | Address  8 | Register  1 |

Register Conventions:

We have 2 registers in our CPU design: one register is used to save temporary value named “$t”. The other is used to save value that need to be used in the future named “$s”

Machine Language Format: We have exactly 16 instructions so that we reserve 4-bits for operation code. When the value of “Register” equals 0, we use register $t, and when the value of “Register” equals 1, we use register $s.

R-type:

General instruction format:

|  |  |  |  |
| --- | --- | --- | --- |
| Op  4 | Extra Space  3 | Address  8 | Register  1 |

Add: add Addr reg. add the value in an address of a memory to the value in register, and then save the calculated value in register. Opcode: 0

And: and Addr reg. do “and” operation between the value in an address of a memory and the value in register, and then save the calculated value in register. Opcode: 1

Or: or Addr reg. do “or” operation between the value in an address of a memory and the value in register, and then save the calculated value in register. Opcode: 2

Sub: sub Addr reg. subtract the value in an address of a memory from the value in register, and then save the calculated value in register. Opcode: 3

Slt: slt Addr reg. Compare the value in an address of a memory with the value in register, and if the value in register is smaller, set the value of register to 1, otherwise, set it to 0

Opcode: 4

Jr: jr Addr reg. unconditionally jump to the instruction whose address is in the register. Opcode: 5

Sw: sw Addr reg. Store the 16 bit word of the register into at an address of memory the register. Opcode: 6

Jal: jal Addr reg. Unconditionally jump to the instruction at the specified address. Save the address of the next instruction in the register. Opcode: 7

Lw: lw Addr reg. Load the 16 bit word at an address of memory into the register.

Opcode: A

I-type:

|  |  |  |
| --- | --- | --- |
| Op  4 | Immediate  11 | Register  1 |

Addi: addi Imm reg. add the immediate value to the value in register, and then save the calculated value in register. Opcode: B

Li: li imm reg Load the immediate number to the register Opcode:F

B-Type

|  |  |  |  |
| --- | --- | --- | --- |
| Op  4 | BranchImmediate/Null  3 | Address/UpperImm  8 | Register  1 |

Beq: beq imm reg. DO Compare the value in an address of a memory with the value in register, if they are equal, branch. Opcode: 8

Lui: lua Imm reg. Load the first 8 bits of 16 bits address at an address of memory into the register. Opcode: E

Bne: bne Addr reg. DO Compare the value in an address of a memory with the value in register, if they are not equal, branch. Opcode: 9

N-Type

|  |  |  |  |
| --- | --- | --- | --- |
| Op  4 | BranchImmediate  3(001) | Address/UpperImm  8(00000000) | Register  1(1) |

Return: return . Return to the address that is stored in the $s(put that into PC), and puts the data from the address pointed by 0x0300 to $s Opcode: D (D201)

J-Type

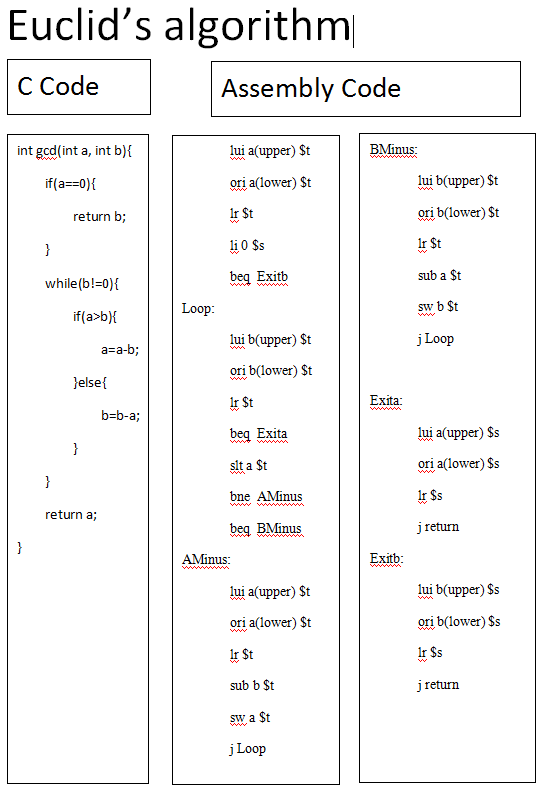
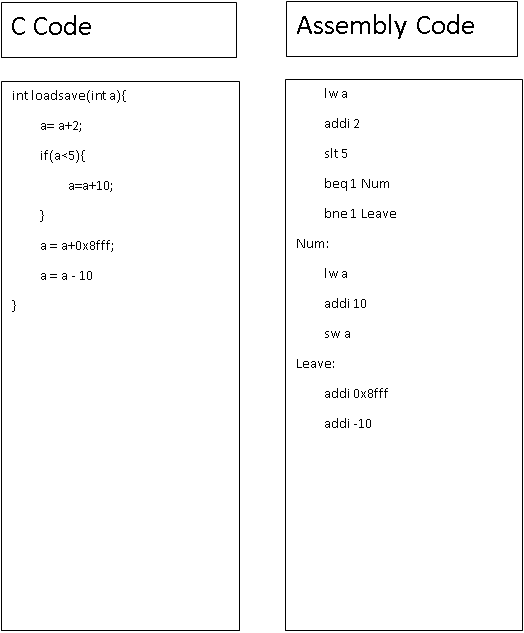
|  |  |  |
| --- | --- | --- |
| Op  4 | Null  2(00) | Address  10 |

J: andi Imm reg. do jump operation and jump to the address that you like. Opcode: C

Rule for Translating:

Will use binary as machine language, to make it easier to be translated to hexadecimal or decimal. Example Assembly Language Programs and Assembly Language Fragments:

Load/Store Word and Exception



Machine Code(a = 36; b = 8)

f048

6000

f010

6002

a000

8024

a002

801a

a002

4000

820c

c10c

a002

3000

6002

c106

a000

3002

6000

c106

a000

d201

Revised version of Euclid’s Algorithm

li (the value of a) $t x0100

sw 0x000 $t x0101

li (the value of b) $t x0102

sw 0x001 $t x0103

lw 0x000 $t x0104

beq 0 (shift bits to ExitB) $t x0105

Loop: lw 0x001 $t x0106

beq 0 (shift bits to ExitA) $t x0107

lw 0x001 $t x0108

slt 0x000 $t x0109

beq 1 (shift bits to AMinus) $t x010a

j BMinus x010b

B Minus: lw 0x001 $t x010c

sub 0x000 $t x010d

sw 0x001 $t x010e

j Loop x010f

AMinus: lw 0x000 $t x0110

sub 0x001 $t x0111

sw 0x000 $t x0112

j Loop x0113

ExitA:lw 0x000 $t x0114

return x0115

j End x0116

c11a

a002

d201

c11a

ExitB:lw 0x001 $t x0117

return x0118

j End x0119

End: x011a

Machine Code

e1ff

f004

6002

e026

b160

6000

721d

8208

a002

b002

6002

c106

a002

d201

a000

6004

a002

Algorithm for relPrime:

relPrime:li 2(the value of m) $t x0100

sw 0x001 $t x0101

lui 0x13 $t x0102

addi 0x0b0 $t x0103

sw 0x000 $t x0104

while: jal (the address of gcd) x0105

beq 1 (shift bits to return m) $t x0106

lw 0x001 $t x0107

addi 0x001 $t x0108

sw 0x001 $t x0109

j while x010a

return m: lw 0x001 $t x010b

return x010c

gcd: lw 0x000 $t x010d

sw 0x002 $t x010e

lw 0x001 $t x010f

sw 0x003 $t x0110

a002

6006

a004

8020

a005

8018

a006

4004

820a

c11a

a006

3004

6006

c114

a004

3006

6004

c114

a004

d201

a006

d201

lw 0x002 $t x0111

beq 0 (shift bits to ExitB) $t x0112

Loop: lw 0x003 $t x0113

beq 0 (shift bits to ExitA) $t x0114

lw 0x003 $t x0115

slt 0x002 $t x0116

beq 1 (shift bits to AMinus) $t x0117

j BMinus x0118

B Minus: lw 0x003 $t x0119

sub 0x002 $t x011a

sw 0x003 $t x011b

j Loop x011c

AMinus: lw 0x002 $t x011d

sub 0x003 $t x011e

sw 0x002 $t x011f

j Loop x0120

ExitA:lw 0x002 $t x0121

return x0122

ExitB:lw 0x003 $t x0123

return x0124

End: x0125

Load address:

lw a

Since the address is 16 bits while the instruction is 32 bits. There are 16 bits for the immediate which is enough to read any types of the data. “a” is a variable that stores a 16-bits address.

Iteration:

Loop:

“content of loop”

beq address Exit

j Loop

Exit:

“exit code”

This is a loop that makes the program goes over and over until the value in the register equals to the value in the specified memory address. When they equal, the program will go to the “Exit” address and finish the loop.

Conditional Statement:

beq(bne) address Target

j Else

Target:

“target content”

Else:

“else content”

This is an if statement. If the value of the register equals (not equal) to the data value in the specified address, then executes specified code in the “Target” address. Else, it will executes codes in the “Else” address.

Reading Data from Input Port: Undecided

Reading/Writing from Display Register: Undecided

Writing to Output Port: Undecided

Load/Store words & Interrupt

lui a(upper) $t

1111 000AAAAAAAA 0

ori a(lower) $t

1110 000AAAAAAAA 0

lr $t

0111 XXX XXXXXXXX 0

addi 2 $t

1100 00000000010 0

sw a $t

1000 XXX AAAAAAAA 0

li 0 $s

0110 0000000000 1

addi 5 $s

1100 00000000101 1

sw b $s

1000 XXX BBBBBBBB 1

li $s

0110 00000000000 1

slt b $t

0100 XXX BBBBBBBB 0

bne Num

1011 XXX NUMMMMMM X

beq Leave

1010 XXX LEAVEEEE X

Num:

lui a(upper) $t

1111 000AAAAAAAA 0

ori a(lower) $t

1110 000AAAAAAAA 0

lr $t

0111 XXX XXXXXXXX 0

addi 10 $t

1100 00000001010 0

sw a $t

1000 XXX AAAAAAAA 0

Leave:

lui a(upper) $t

1111 000AAAAAAAA 0

ori a(lower) $t

1110 000AAAAAAAA 0

lr $t

0111 XXX XXXXXXXX 0

addi 0x8fff $t

1100 errorrrrr 0

addi -10 $t

1100 11110110 0

sw a $t

1000 XXX AAAAAAAA 0

Milestone 2:

RTL Code:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Step | and, or, sub, add, slt | lw, sw | | j | Jr | beq/bne |
| IF | IR = Mem[PC]  PC = PC + 1 | | | | | |
| ID | R = Reg[IR[0:0]]  I = IR[11:1]  BI = IR[11:9]  UA = IR[8:1]  ALUOut = PC + (SE((IR[11:1]))) | | | | | |
| EX  beq/j done | MDR = Mem[00000(I)] | MDR = Mem[(00000(I))] | Mem[(00000(I))] = R | PC = (000000IR[9:0]) | PC=ZE(Reg[IR[0:0]]) | If (R== BI) /(R!=BI) |
| MEM  and, or, etc. done | Reg[IR[0:0]] = R op MDR | Reg[IR[0:0]] = MDR | |  |  | then PC = PC + R |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Step | Jal | Addi | lui | li | | return |
| IF | IR = Mem[PC]  PC = PC + 1 | | | | | |
| ID | R = Reg[IR[0:0]]  I = IR[11:1]  BI = IR[11:9]  UA = IR[8:1]  ALUOut = PC + (SE(IR[11:1])) | | | | | |
| EX  beq/j done | MDR = Mem[0x0300] | ALUOut(wire) = R + SE(I) | Reg[IR[0:0]] = SE(IR[8:1])<<8 | | Reg[IR[0:0]]=SE(I) | PC = $s |
| MEM  and, or, etc. done | Mem[MDR] = $s | Reg[IR[0:0]] = ALUOut(wire) |  | |  | 0x0300 =  Mem[0x0300] - 1 |
| lw done | 0x0300= Mem[0x0300] + 1 |  |  | |  | MDR = Mem[0x0300] |
| Procedure ALU | $s = PC |  |  | |  | $s = Mem[MDR] |
| Procedure calls done | PC = (00000(I)) |  |  | |  |  |

Components Needed:

* PC(program counter, which keeps tracking the address of current instruction)
* Memory(the place where instructions and memory stores)
* Instruction Register(the register where current instruction has been put)
* Register File(the place where deal with the load and store of registers)
* ALU(do simple operation with two inputs)
* Control Unit(translate the operation code to specified control signal bits)
* MDR(memory data register, which is used to store temporary value read from the memory.)
* I(for immediate)
* BI(for branch immediate)
* UA(for upper address use)
* R(for register)

Control Signals Needed:

* PCSource – to control the source of PC(0: R; 1:ALUout; 2:target address that going to jump after the j instruction)
* ALUsrc1 – to control the input of ALU(0:immediate; 1:branch immediate 2: 1)
* ALUsrc2 – to control the input of ALU(0:from register file; 1:MDR 2:PC)
* PCWrite – enable or disable writing to the PC (0: disable; 1:enable)
* MemRead – enable or disable reading from Memory (0: disable; 1:enable)
* IRWrite – enable or disable writing to Instruction Register (0: disable; 1:enable)
* RegWrite – enable or disable writing to Register File (0: disable; 1:enable)
* RegSrc – determines what goes to register (0: sign extended immediate; 1:MDR; 2:UA shifts left by 8; 3:PC; 4:ALUout)
* ALUoutSrc – determines what output should be in the ALUout (0: address begin with 5bits from PC; 1:result from ALU 2:sign extended immediate)
* ALUOp – determines what operation is used in the ALU
* IorD – to control which part should be write back.

Test Cases:(haven’t done yet)

Add: when an add instruction have been read from the instruction memory, for example:

“Add 0xfe $s”. It represents to add data from 0x10fe to register $s. We assume that $s stores a value like “5” and the value of 0x10fe to be “2”. Then, it should come out an answer of 7 in the register $s. We just need to check whether the Opcode is 0, the MDR should be 2 and ALUout should be 7.(Sub/or/and/slt are similar)

Lw:example: “lw b $t” We assume the address of b is 0x102a and the data in 0x102a should be 78. Thus, we should check MDR which should be 78 and Opcode should be A. ALUout should be the full address which is 0x102a.

Sw: example: “sw b $t”. We assume the address of b is 0x102a. We assume the value of $t would be 6. We should check ALUouy be the full address of b, which is 0x102a. Also, we should check the value of register $t after that. It should have a value of 6.

J: example: “j 0x08fe” This one should check whether PC is 0x08fe after the instruction has been done.

Jr: example: “jr $t” We assume that the address stores in the $t is 0x063a. After we finish the execution of the instruction, the value of PC should be 0x063a.

Beq/bne: example: “beq $t 4 A” We assume that the value stores in $t is 4. We assume A has the address 0x10b8 and PC is 0x10a5 Thus, we should check whether the PC after the execution is 0x10b8. Also, we should check the value of register A is 4 and register B is 4.

(bne is similar)

Jal: example: “jal 0xfe $t” This one should check whether PC is 0x10fe or not. We assume the address of “jal 0xfe $t” is 0x1050 0x Also, we should check whether $t contains current address of “jal 0xfe $t”, which should be 0x1052 then.

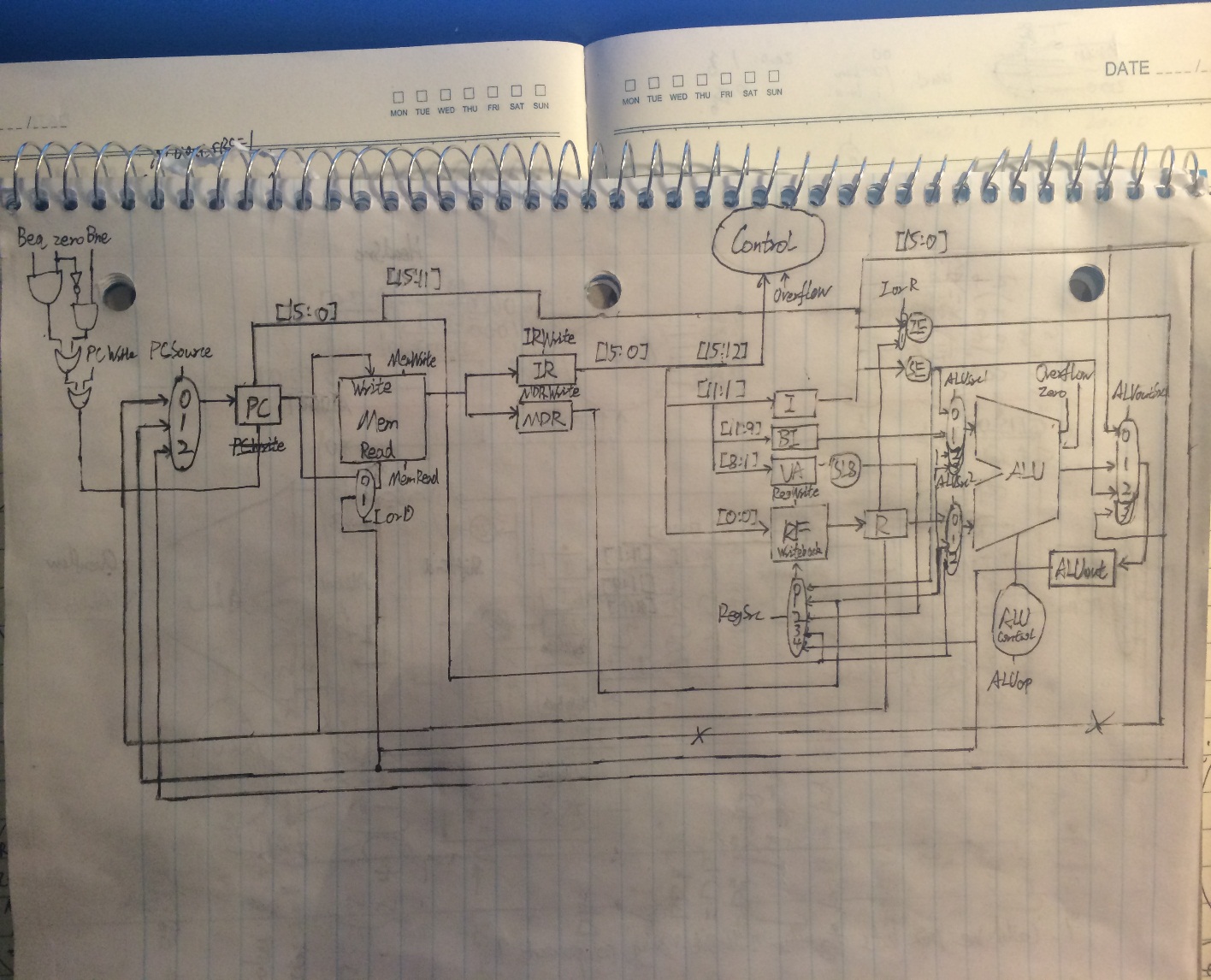
Addi/ori/andi: “addi 0xa2 $t” We assume that register t has a value of 8. Then, after the execution of this instruction, the value of $t should be 170.

Lui: example: “lui 0xea $t” After executing the instruction, we should check whether the value of $t is 0xea00 or not.

Li: example: “li 0x34 $t” After executing the instruction, we should check whether the value of $t is 0x34 or not.

DataPath:

df



Necessary Testing

Unit Testing:

To correctly test this data path, multiple testbenches must be implemented to verify the implementation. There must a separate test for the memory, control, ALU, IR, and MDR. But also another overall test for the overall schematic. The test for memory, must be provided addresses and all inputs, outputs, and registers initialized. The test for the control, will the most difficult it will require the most attention to detail, but after given the necessary inputs it should be able to run autonomously. The test for the ALU can be broken down to the different operations. The test for the IR and MDR can be similar as they are both registers. In the schematic test bench, there must be a file loaded into created memory block that provides the processor with all of the necessary opcodes for the instructions that need to be executed. All of these test should run smoothly if the datapath is implemented correctly and if the path is not correct can point the group in the appropriate area that needs correction.

Integration Testing:

The integration testing will be implement by splitting the datapath up into smaller chunks and tested until satisfaction. Then it will be built bottom up and written in the fashion of our RTL code. Right now, we are thinking we could began our tests by testing in procedures as follow:

1. ALU tests
2. Bne/Zero/Beq/PCWrite + PC part tests
3. Memory block + Instruction register + Memory Data register tests
4. Part 2 + part 3 tests
5. Instruction register + immediate register + branch register + upper address register

tests

1. Register file + R register tests
2. Mux tests
3. Part 5 + part 6 tests
4. Part 4 + part 8 tests
5. Part 1 + part 10 tests
6. Part 10 + control unit tests + mux tests (which is the system tests then)

How to implement each part in Xilinx:

PC/IR/MDR/ALUout: It’s a register that has 16 bits, we could use flip-flop to implement that.

Memory: It’s a block memory that could implement by using the example memory block. (Or using block memory generator to create one)

Mux: We could use symbols from the Xilinx.

I: It’s a register that has 11 bits, we could use flip-flop to implement that.

BI: It’s a register that has 3 bits, we could use flip-flop to implement that.

UA: It’s a register that has 8 bits, we could use flip-flop to implement that.

R: It’s a one bit register, we could use the basic symbol in Xilinx to implement that.

Register file: Use example in Final Project Resources as reference.

Shift left 8: could use Xilinx code to implement that.

Zero Extend: Use Verilog to implement that.

Sign Extend: Use Xilinx wire graph to implement that.

ALU: finished in lab6, but need to change it to 16-bits.

Control: could use Xilinx code to implement that(use cases to implement).

Test RTL

Test of state 1(Instruction Fetch)

Check to see the value assigned into state 1 had plus 1 or not.

Example:

PC=0x8000

Add 0x7a6 $t

PC expect: 8001

Test of state2 (Inst Decode Register Fetch)

After state 2. Just check the register R, I, BI, UA and ALUOut store the right value from Reg[IR[0:0]], IR[11:1], IR[11:9], IR[8:1], and PC + (SE((IR[11:1]) <<1)) respectively.

Example:

PC=0x8000/ $t =5; 0x0ef6 = 12

Add 0x77b $t

R expect: 5; I expect: 0x77b; BI expect: 0x7; UA expect: 0x7b; ALUout expect: 17

Test of state3

1.Add

Make ALU do the add operation and check if the number will be overflow

Example:

$t =5; 0x0ef6 = 12

Add 0x77b $t

Reg[IR[0-0]] = R = ALUout expect: 17

Example:

$t = 10; 0x0ef6 = 0xfffe

Add 0x77b $t

Overflow expect: 1

2.sub

3.or

4.add

5.Slt

6.lw

7.sw

8.j

9.jr

10.beq

11.bne

12.jal

13.addi

14.andi

15.li

16.lui

Test of state4

1. and, or, sub, add, slt

Check that Reg[IR[0:0]] contains the value stores in ALUout.

Example:

$t =5; 0x0ef6 = 12

Add 0x77b $t

Reg[IR[0-0]]:R expect: 17

2. sw

Check the value of specified address is the value of register R

Example:

$t =0x002a; 0x0ef6 = 12

Sw 0x77b , $t

Expect: MEM[ALUout]=12 = R

1. lw

Check the value of the MDR is the same value in the specified address.

Example:

0x0ef6 = 12

Lw 0x77b, $t

Expect: $t = Mem[ALUout]

Test of state5

1. lw

Check if the Reg[IR[0:0]] contains the value stores in MDR.

Example:

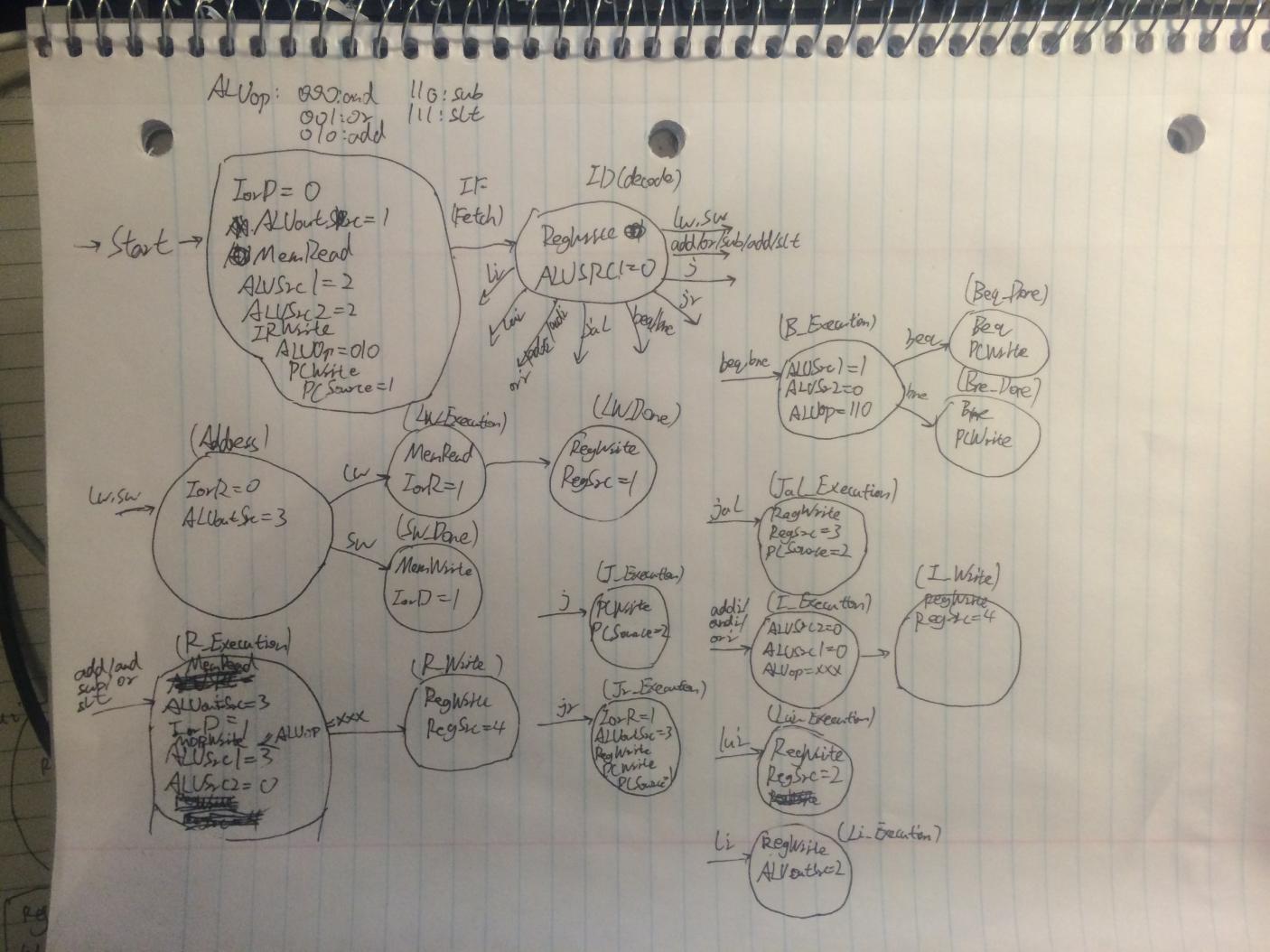
0x0ef6 = 12

Lw 0x77b, $t

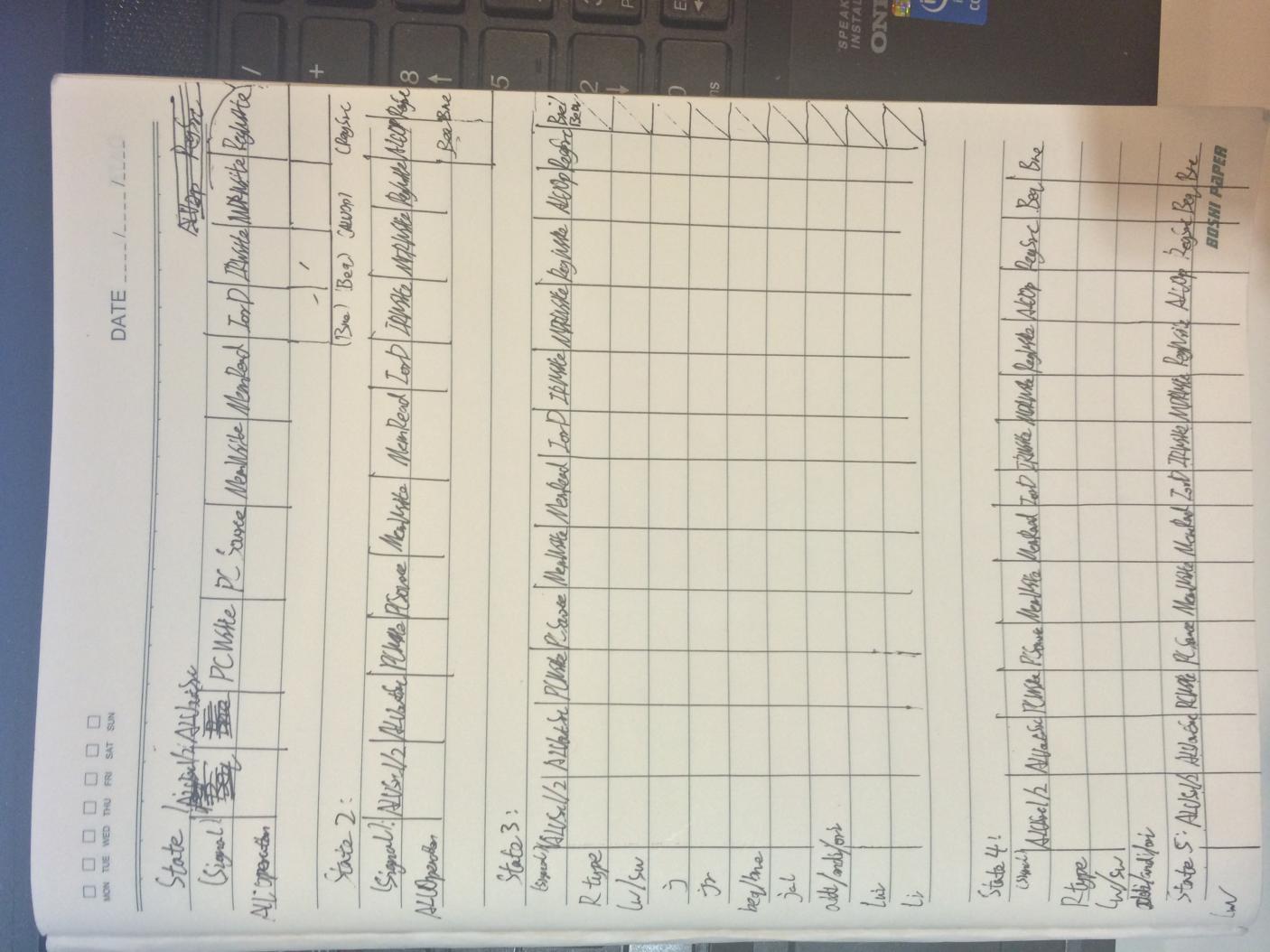
Expect: Reg[IR[0:0]]=$t

We use FSM to implement control unit.

State transition diagram/table:



Test necessary to verify the correct implementation of the control unit:



In order to make sure that every types of instructions run well in different states, we need to make a table that records the bits of signal that we get from the test bench. Then, we need to compare this table with the state transition diagram and try to find those in-correspond values. Those in-correspond values could make us find out which signal and what state that our control unit goes wrong.

Changes to the signals and RTL:

There is no longer MemWrite Signal or IorR signal. Instead, there is one more port with the number of two in the MemRead mux and a new port with number of three in the ALUSrc2. (which is the merge of the first 8 bits of PC and 8 bits of immediate ).The newest version of the datapath is implemented in Xilinx. Also, we delete addi and ori to add the return instruction and the J instruction. (We want to do the procedure call; addi and ori are not commonly used) Cause, we need to keep the number of instructions to 16. (Jal is not designed correctly, already fixed that in the RTL, need to create a stack that stores return value, we assigned the 0x3000 to be the stack pointer of the stack begin from 0x2000. what we are considering is to have $s to keep the current return address.) However, due to the changes that we made to the RTL, we need to change the control unit and the datapath which is not fully implemented yet.

System Test Plan:

For the integration tests, we should test the whole datapath with all of the instructions that we have. In order to test all of them, we could write short programs and translate them to machine code to put them into the memory. The machine code could be finished according to the opcode and the translation rules that we already given in Milestone 1 and 2. Then, we just need to check whether the value of each components correspond to what we expected. Also, we could run the Euclid algorithm by giving the machine code to the memory. (In order to test the procedure call)

R-Type:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

$s = 2

Add 0x001 $t

Sub 0x001 $s

And 0x000 $s

Or 0x001 $t

Slt 0x000 $t

Expected:

$t = 1

$s = 10

Lw and Sw:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

$s = 2

Lw 0x000 $t

Add 0x001 $t

Sw 0x000 $t

Expected:

$t = 14

Mem[0x0000] = 14

Jr:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

$s = 0x1005

0x1000 Add 0x001 $t

0x1001 Jr $s

0x1002 Add 0x001 $t

0x1003 Add 0x000 $t

0x1004 Sub 0x000 $t

0x1005 Sub 0x001 $t

Expected:

$t = 10

Beq & Bne:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 2

$s = 0x1005

0x1000 Add 0x001 $t

0x1001 Beq 4 0x 05 $t

0x1002 Add 0x001 $t

0x1003 Add 0x000 $t

0x1004 Sub 0x000 $t

0x1005 Sub 0x001 $t

Expected:

$t = 2

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 2

$s = 0x1005

0x1000 Add 0x001 $t

0x1001 Bne 0 0x 05 $t

0x1002 Add 0x001 $t

0x1003 Add 0x000 $t

0x1004 Sub 0x000 $t

0x1005 Sub 0x001 $t

Expected:

$t = 2

Jal & Jr:

PC = 0x1000

Mem[0x0000] = 1

Mem[0x0001] = -1

$t = 0

$s = 2

0x1000 Add 0x001 $t

0x1001 Jal 0x004 $t

0x1002 Add 0x001 $t

0x1003 li 0x000 $t

0x1004 beq 0 0x00 $t.

0x1005 Sub 0x001 $t

0x1006 Jr $t

0x1007 Sw $t

Expected:

$t = 0

I-Type:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

$s = 2

Addi 0xff5 $t

Ori 0x001 $t

andi 0x001 $t

Expected:

$t = 0x0fff

Lui:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

$s = 2

Lui 0x01 $t

Expected:

$t = 0x0100

Li:

PC = 0x1000

Mem[0x0000] = 10

Mem[0x0001] = 4

$t = 10

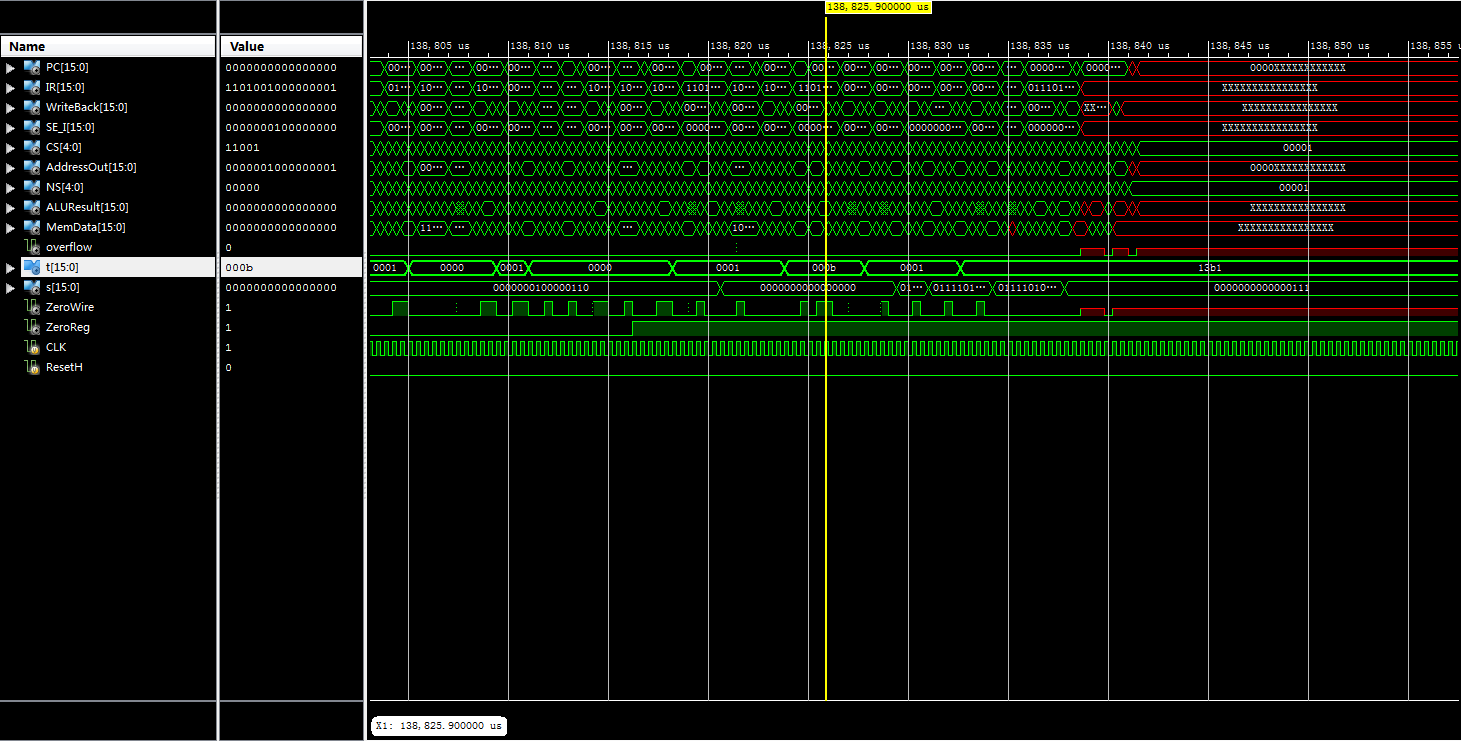
$s = 2

Li 0x001 $s

Expected:

$s = 4

Final result for relPrime:



About the design:

104 Bytes needed to implement the design

112261 Instructions/367506 cycles involve in relPrime

3.273cycles/instruction in average

77.472ns for one single cycle in average

138.988ms for the whole relPrime to finish

=========================================================================

\* Final Report \*

=========================================================================

Clock Information:

------------------

----------------------------------------------------------+------------------------------+-------+

Clock Signal | Clock buffer(FF name) | Load |

----------------------------------------------------------+------------------------------+-------+

CLK | BUFGP | 608 |

XLXI\_52/next\_state\_not0001(XLXI\_52/next\_state\_not000127:O)| NONE(\*)(XLXI\_52/next\_state\_4)| 10 |

----------------------------------------------------------+------------------------------+-------+

(\*) This 1 clock signal(s) are generated by combinatorial logic,

and XST is not able to identify which are the primary clock signals.

Please use the CLOCK\_SIGNAL constraint to specify the clock signal(s) generated by combinatorial logic.

INFO:Xst:2169 - HDL ADVISOR - Some clock signals were not automatically buffered by XST with BUFG/BUFR resources. Please use the buffer\_type constraint in order to insert these buffers to the clock signals to help prevent skew problems.

Asynchronous Control Signals Information:

----------------------------------------

-----------------------------------+--------------------------+-------+

Control Signal | Buffer(FF name) | Load |

-----------------------------------+--------------------------+-------+

XLXI\_51/G(XLXI\_51/XLXI\_2:G) | NONE(XLXI\_12/XLXI\_2/I\_Q0)| 49 |

ResetH | IBUF | 15 |

-----------------------------------+--------------------------+-------+

Timing Summary:

---------------

Speed Grade: -4

Minimum period: 73.977ns (Maximum Frequency: 13.518MHz)

Minimum input arrival time before clock: No path found

Maximum output required time after clock: 77.472ns

Maximum combinational path delay: No path found

=========================================================================