# Design Document

King, Brendan

Ouyang, Aurora

Tomlinson, Geoffrey

Zhang, Chet

*CSSE232 Computer Architecture I*

*Rose-Hulman Institute of Technology*

# Contents

## Project Description

for our project, we will be using Microsoft teams. A link to our team is included below
https://teams.microsoft.com/_#/school/files/General?threadId=19%3Afaa1373762b04083842b90f200d622f2%40thread.skype&ctx=channel

Our CPU of choice is an accumulator. We choose this because of the freedom that we will have while designing the processor. Two of our members also have experience using accumulator style microcontrollers and were especially interested in learning more about how they work.

## Instruction Type Description

The following describes the instruction type used for all instructions on the processor.

| **Instruction Format** | | |
|---|---|---|
| A-Type | Opcode(8) | Immediate(8) |

We use an 8-bit Opcode for our only instruction format purely to show the instruction type.

● The A-Type instruction format takes an 8-bit opcode followed by the 8-bit immediate. The opcode is used to determine the operation being performed by the instruction.

## List of Registers

The following lists all important registers in the processor.

| Reg | Purpose |
|---|---|
| ACC | Accumulator registers. Nearly all instructions interact with acc. |
| PC | Program Counter. Stores the instruction of currently executing instructions. |
| SP | Stack Pointer. Stores the current memory address of where stack pointer is pointing to. |
| IN | Contains the value currently on the input port. |
| OUT | Contains the value currently on the output port. |

All registers cannot serve as operands. They can only be manipulated and modified by hardware.

## List of Pseudo-Instructions

The following lists all pseudo-instructions supported by the assembler. The assembler can be found in assembler.java.

| Instruction | Description | Syntax expansion |
|---|---|---|
| la label | Finds the address of a label and stores than address in acc. | lui msb(address)<br>ori lsb(address) |
| li imm | Stores an immediate value in acc. | lui msb(imm)<br>ori lsb(imm) |

## List of Instructions

The following lists all of the instructions supported by the processor.

| NAME | OPCODE | DESCRIPTION | SEMANTIC |
|---|---|---|---|
| Addi | 0 | Adds ACC to Sign extended Immediate then stores result into ACC | Addi imm |
| Ori | 1 | Ors ACC to zero extended Immediate then stores result into ACC | Ori imm |
| Andi | 2 | Ands ACC to zero extended Immediate then stores result into ACC | Andi imm |
| Lui | 3 | Loads the immediate into the most significant byte of ACC, clears the least significant byte of ACC | Lui imm |
| Sli | 4 | Shifts the bits in ACC to the left logical by the amount in the Immediate filling with zeroes, where 0 <= Immediate <= 15 | Sli imm |
| Sri | 5 | Shifts the bits in ACC to the right logical by the amount in the Immediate filling with zeroes, where 0 <= Immediate <= 15 | Sri imm |
| Srai | 6 | Shifts the bits in ACC to the right arithmetically by the amount in the Immediate filling with zeroes, where 0 <= Immediate <= 15 | Srai imm |
| lw | 7 | Loads the value at memory address sp + (Imm. << 1) into ACC (Base offset addressing) | Lw imm |
| sw | 8 | Stores the value of ACC into memory address sp + (Imm. << 1) (Base offset addressing) | Sw imm |
| Add | 73 | Adds ACC to the value at memory address (sp + (Imm. << 1)) and stores result into ACC (Base offset addressing) | Add imm |
| Sub | 74 | Computes ACC minus the value at memory address (sp + (Imm. << 1)) and stores result into ACC (Base offset addressing) | Sub imm |
| Or | 75 | Ors ACC to the value at memory address (sp + (Imm. << 1)) and stores result into ACC (Base offset addressing) | Or imm |
| And | 76 | Ands ACC to the value at memory address (sp + (Imm. << 1)) and stores result into ACC (Base offset addressing) | And imm |
| Jal | 13 | Sets the value at memory address sp to PC + 2 then sets PC to the address stored in ACC | Jal |
| J | 14 | Sets PC to the address stored in ACC | J |
| Bin | 143 | Branches to the location PC + 2 + Sign Extended Imm << 1 if ACC is negative (PC relative) | Bin Label |
| Bifz | 144 | Branches to the location PC + 2 + Sign Extended Imm << 1 if ACC is zero (PC relative) | Biz Label |
| Binz | 145 | Branches to the location PC + 2 + Sign Extended Imm << 1 if ACC is not zero (PC relative) | Binz Label |
| Bip | 146 | Branches to the location PC + 2 + Sign Extended Imm << 1 if ACC is positive (PC relative) | Bip Label |
| In | 19 | Sets ACC to the value on the input port | In |
| Out | 20 | Sets the value on the output port to the value of ACC | Out |
| spi | 21 | Increments sp by the sign extended immediate left shifted by 1 | Spi imm |
| spc | 22 | Increments sp by the value in memory at location (sp + (Imm. << 1)) | Spc imm |

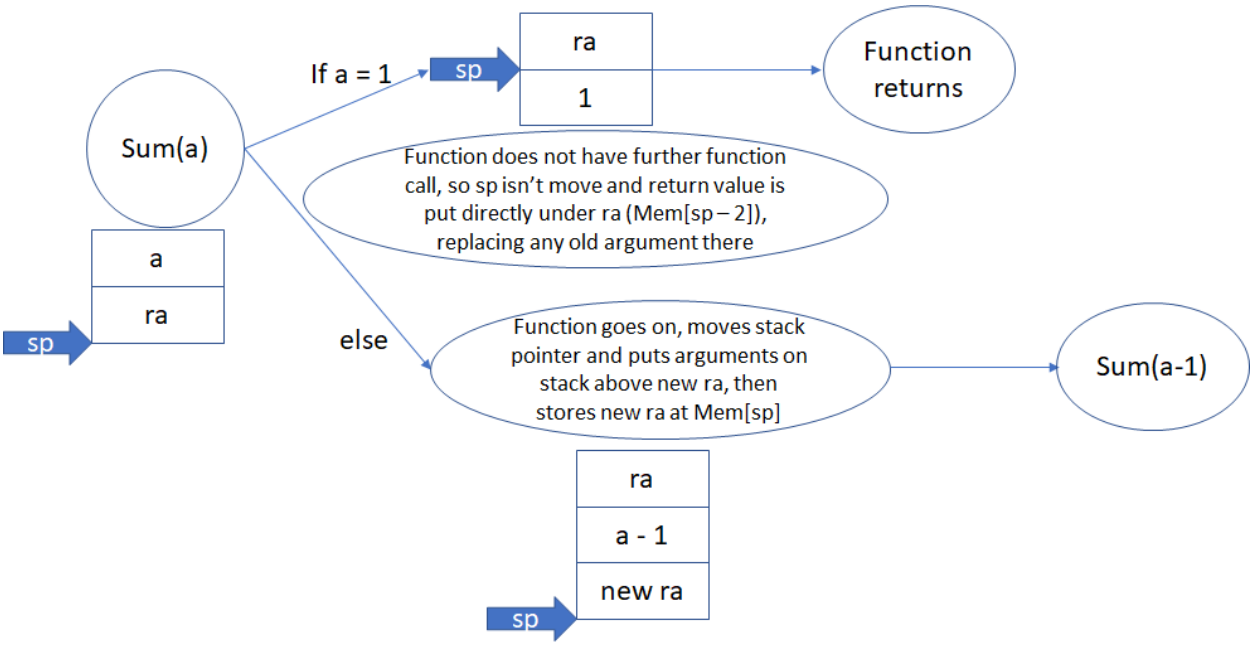
| | | | |
|---|---|---|---|
| lwa | 23 | Changes the value of ACC to be the value in memory at location (ACC + (Imm. << 1)) | Lwa imm |

## Programming Conventions

The following states the programming conventions regarding stack management and calling procedures that should be followed when writing assembly for the processor.

When managing the stack, the programmer should be aware of the following conventions:

- When calling a function, the caller should store all values it wants to preserve across the call on the stack. Then argument 0, argument 1, argument 2, etc. should be stored. The stack pointer should then point to the "empty" memory address that is two bytes beyond the address of the last argument. This is where the return address will be located once jal is called. Everything above the stack pointer will be preserved by the callee.
- When a function is returning a value, return value 0 should be stored at address sp - 2, return value 1 should be stored at address sp - 4, etc., where sp is the value of the stack pointer when the function was originally called. The sp at the beginning of the execution of the function should be at the same location as the sp at the end of the function.




- As shown in the figure above, function sum(a) (a function that calculates the summation of every integer from 1 to a) is used to illustrate the programming convention. A programmer has to know whether the function needs to do a further function call in order to make correct stack pointer adjustments. When a is not 1, which means sum still has to recursively call itself, stack pointer must be moved and all necessary arguments and return address must be stored properly into the stack. Return address should always be stored at Mem[sp]. On returning, function should have its return value stored right below ra, then after it returns, return value can be retrieved at Mem[sp – 2].

# Assembly Code Programs

The following are example assembly programs written for the processor.

Machine code was assembled using an assembler built in Java.

## RelPrime and GCD Procedures

This program finds the smallest relatively prime number to an input.

| | | |
|---|---|---|
| RelPrime: | lw 1 | 0x16 \| 0b0000011100000001 lw 1 |
| | sw -1 #make a copy of n | 0x18 \| 0b0000100011111111 sw -1 |
| | li 2 | 0x1a \| 0b0000001100000000 lui 0 |
| | | 0x1c \| 0b0000000100000010 ori 2 |
| | sw -2 #load 2 into m | 0x1e \| 0b0000100011111110 sw -2 |
| RelLoop: | spi -3 #prepare the sp | 0x20 \| 0b0001010111111101 spi -3 |
| | la GCD | 0x22 \| 0b0000001100000000 lui gcd |
| | | 0x24 \| 0b0000000101000100 ori gcd |
| | jal #call GCD | 0x26 \| 0b0000110100000000 jal |
| | spi 3 #move sp back | 0x28 \| 0b0001010100000011 spi 3 |
| | Lw -4 #get result | 0x2a \| 0b0000011111111100 lw -4 |
| | addi -1 | 0x2c \| 0b0000000011111111 addi -1 |
| | bifz RelDone #See if result != 1 | 0x2e \| 0b1001000000000110 bifz reldone |
| RelWhile: | lw -2 | 0x30 \| 0b0000011111111110 lw -2 |
| | addi 1 | 0x32 \| 0b0000000000000001 addi 1 |
| | sw -2 #increment m | 0x34 \| 0b0000100011111110 sw -2 |
| | la RelLoop #loop again | 0x36 \| 0b0000001100000000 lui relloop |
| | | 0x38 \| 0b0000000100100000 ori relloop |
| | j | 0x3a \| 0b0000111000000000 j |
| RelDone: | lw -2 | 0x3c \| 0b0000011111111110 lw -2 |
| | sw -1 #move m to be the return value | 0x3e \| 0b0000100011111111 sw -1 |
| | lw 0 | 0x40 \| 0b0000011100000000 lw 0 |
| | j #return | 0x42 \| 0b0000111000000000 j |
| GCD: | lw 1 | 0x44 \| 0b0000011100000001 lw 1 |
| | sw -2 #make a copy of b | 0x46 \| 0b0000100011111110 sw -2 |
| | lw 2 | 0x48 \| 0b0000011100000010 lw 2 |
| | sw -1 #make a copy of a | 0x4a \| 0b0000100011111111 sw -1 |
| | binz Loop #return if a ==0 | 0x4c \| 0b1001000100000100 binz loop |
| | lw -2 | 0x4e \| 0b0000011111111110 lw -2 |
| | sw -1 #move a to return value | 0x50 \| 0b0000100011111111 sw -1 |
| | lw 0 | 0x52 \| 0b0000011100000000 lw 0 |
| | j #return | 0x54 \| 0b0000111000000000 j |
| Loop: | lw -2 #load b | 0x56 \| 0b0000011111111110 lw -2 |
| | bifz Done #done if b==0 | 0x58 \| 0b1001000000001101 bifz done |
| While: | lw -2 #load b | 0x5a \| 0b0000011111111110 lw -2 |
| | sub -1 #acc = b - a | 0x5c \| 0b0100101011111111 sub -1 |
| | bin if #branch if b-a>0 | 0x5e \| 0b1000111100000100 bin if |
| | sw -2 #store b-a into b | 0x60 \| 0b0000100011111110 sw -2 |

| | | |
|---|---|---|
| | la Loop | 0x62 \| 0b0000001100000000 lui loop |
| | | 0x64 \| 0b0000000101010110 ori loop |
| | j #loop again | 0x66 \| 0b0000111000000000 j |
| If: | lw -1 #load a | 0x68 \| 0b0000011111111111 lw -1 |
| | sub -2 #acc = a - b | 0x6a \| 0b0100101011111110 sub -2 |
| | sw -1 #store a - b into a | 0x6c \| 0b0000100011111111 sw -1 |
| | la Loop | 0x6e \| 0b0000001100000000 lui loop |
| | | 0x70 \| 0b0000000101010110 ori loop |
| | j | 0x72 \| 0b0000111000000000 j |
| Done: | lw 0 #return a | 0x74 \| 0b0000011100000000 lw 0 |
| | j | 0x76 \| 0b0000111000000000 j |

## Example machine code conversions for each operation

(0bddddddddd represents how many operations to skip from branch operation to reach the label X)

| | |
|---|---|
| Addi 9 #add acc to 9 | 0x4000 \| 0b0000000000001001 addi 9 |
| Ori 8 #or acc with 8 | 0x4002 \| 0b0000000100001000 ori 8 |
| Andi 7 #and acc with 7 | 0x4004 \| 0b0000001000000111 andi 7 |
| Lui 6 #6 into upper byte of acc | 0x4006 \| 0b0000001100000110 lui 6 |
| Sli 5 #shift acc left by 5 | 0x4008 \| 0b0000010000000101 sli 5 |
| Sri 4 #shift acc right logically by 4 | 0x400a \| 0b0000010100000100 sri 4 |
| Srai 3 #shift acc right arithmetically by 3 | 0x400c \| 0b0000011000000011 srai 3 |
| lw 2 #load Mem[sp + 2] into acc | 0x400e \| 0b0000011100000010 lw 2 |
| sw 1 #store acc into Mem[sp + 1] | 0x4010 \| 0b0000100000000001 sw 1 |
| Add 0 #add acc to Mem[sp] | 0x4012 \| 0b0100100100000000 add 0 |
| Sub -1 #subtract Mem[sp-1] from acc | 0x4014 \| 0b0100101011111111 sub -1 |
| Or -2 #or acc and Mem[sp –2] | 0x4016 \| 0b0100101111111110 or -2 |
| And -3 #and acc and Mem[sp-3] | 0x4018 \| 0b0100110011111101 and -3 |
| Jal #jump and link | 0x401a \| 0b0000110100000000 jal |
| J #jump | 0x401c \| 0b0000111000000000 j |
| Bin X #branch to label X if acc<0 | 0x401e \| 0b10001111dddddddd bin x |
| Bifz X #branch to label X if acc==0 | 0x4020 \| 0b10010000dddddddd bifz x |
| Binz X #branch to label X if acc!=0 | 0x4022 \| 0b10010001dddddddd binz x |
| Bip X #branch to label X if acc>0 | 0x4024 \| 0b10010010dddddddd bip x |
| In #set acc to equal input | 0x4026 \| 0b0001001100000000 in |
| Out #set output to equal acc | 0x4028 \| 0b0001010000000000 out |
| spi 3 #set sp to sp + 3 | 0x402a \| 0b0001010100000011 spi 3 |
| spc -2 #set sp to sp + Mem[sp – 2] | 0x402c \| 0b0001011011111110 spc -2 |
| lwa –3 #set ACC to Mem[ACC – 3] | 0x402e \| 0b0001011111111101 spa -3 |

**Example recursive procedure**

This snippet finds the sum of all positive integers from 0 to x.

```
Int sum(x){
        if(x==0) {
                return 0;
        }
        return x + sum(x – 1);
}
```

| | | |
|---|---|---|
| SUM: | lw 1 #load x | 0x0 \| 0b0000011100000001 lw 1 |
| | binz A #stay if x ==0 | 0x2 \| 0b1001000100000011 binz a |
| | sw –1 #make return value 0 | 0x4 \| 0b0000100011111111 sw -1 |
| | lw 0 | 0x6 \| 0b0000011100000000 lw 0 |
| | J #return | 0x8 \| 0b0000111000000000 j |
| A: | addi –1 | 0xa \| 0b0000000011111111 addi -1 |
| | sw –1 #store x-1 as argument | 0xc \| 0b0000100011111111 sw -1 |
| | spi –2 #prepare sp | 0xe \| 0b0001010111111110 spi -2 |
| | lui SUM | 0x10 \| 0b0000001100000000 lui sum |
| | ori SUM | 0x12 \| 0b0000000100000000 ori sum |
| | Jal #call sum | 0x14 \| 0b0000110100000000 jal |
| | spi 2 #move sp back | 0x16 \| 0b0001010100000010 spi 2 |
| | lw –3 #load result | 0x18 \| 0b0000011111111101 lw -3 |
| | add 1 #acc = x + result | 0x1a \| 0b0100100100000001 add 1 |
| | sw –1 #set return value | 0x1c \| 0b0000100011111111 sw -1 |
| | lw 0 | 0x1e \| 0b0000011100000000 lw 0 |
| | J #return | 0x20 \| 0b0000111000000000 j |

# RTL

The following specifies the RTL for our processor.

| Instruction | Inst. Fetch | Decoding | Execution/Memory Access | Load into ACC |
|---|---|---|---|---|
| Add/Or/Sub/And | IR = [Mem [PC]]<br>PC = PC + 2 | ALUout = SP + (SE[IR[7:0]] << 1)* | Memout = Mem[Aluout] | ACC = ACC op Memout |
| lw | | | ACC = Mem[Aluout] | N.A. |
| lwa | | | Aluout = ACC + (SE[IR[7:0]] << 1) | ACC = Mem[Aluout] |
| sw | | | Mem[Aluout] = Acc | N.A. |
| J | | | PC = ACC | N.A. |
| Jal | | | PC <= ACC<br>Mem[SP] <= PC | N.A. |
| out | | | Out = ACC | N.A. |
| in | | | ACC = In | N.A. |
| Andi/Ori/Lui/Sli/Sri/Srai | | | ACC = ACC op ZE(IR[7:0]) | N.A. |
| Addi | | | ACC = ACC op SE(IR[7:0]) | N.A. |
| spi | | | SP = Aluout | N.A. |
| spc | | | Memout = Mem[Aluout] | SP = SP + Memout |
| Bip/Bin/Bifz/Binz | | ALUout = PC + (SE[IR[7:0]] << 1)* | If (Condition) PC = ALUout | N.A. |

(*)one bit from the instruction opcode will be wired directly to a mux to decide whether sp or pc is used.

## Double Checking the RTL

To Double check out RTL we took sample inputs and ran them through our RTL to make sure that when given the machine language instruction that the RTL would execute the instructions properly. We chose the below sample inputs because they give a wide range of instructions and should effectively show that our RTL is functional in its current state.

## Sample Inputs

All the sample inputs below are executed using the above RTL and all of them perform their instruction properly.

**0b0000000011111111: addi -1**

This instruction should add the immediate value of -1 to the accumulator.
IR = Mem[PC]
PC = PC + 2
Acc = Acc + (-1)

**0b1000001100000010: Bip x**

This instruction should increment the PC by the destination specified in the immediate field.
IR = Mem[PC]
PC = PC + 2
ALUout = PC + 4
If (Acc > 0) PC = ALUout;

**0b0000011111111101: lw -3**

This instruction loads the value in memory at PC (– 3<<1) into the Accumulator
IR = Mem[PC]
PC = PC + 2
ALUout = SP + (-6)
Acc = Mem[SP – 6]

**0b0000110100000000: Jal**

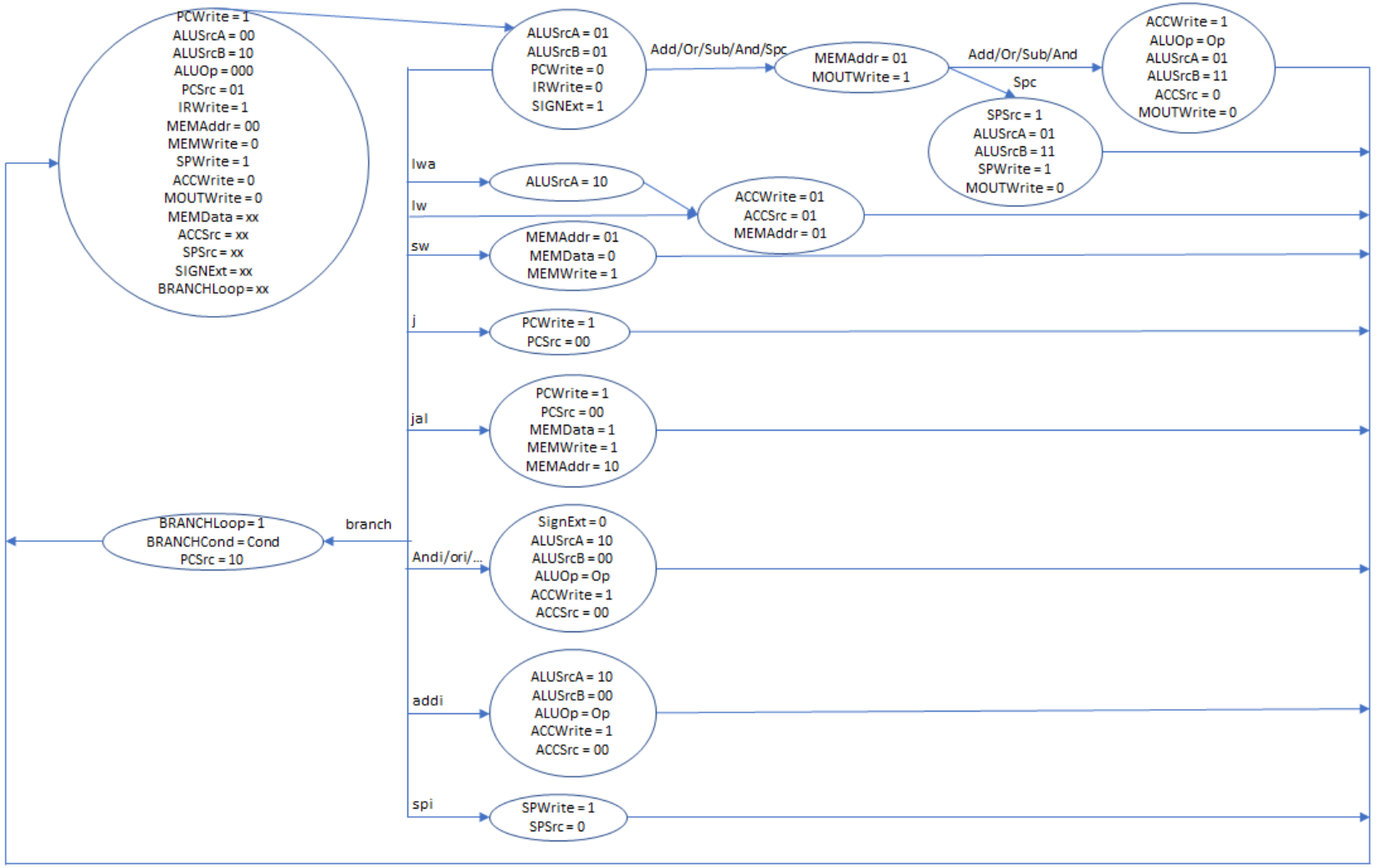
This instruction should Load the value of PC + 2 into the Memory at SP. It will also set PC to the value in the accumulator.
IR = Mem[PC}
PC = PC + 2
ALUout = SP + 0 (Jal does not use the immediate field)
PC = Acc
Mem[SP} = PC

**0b0001011011111110: spc -2**

This instruction should increment SP by the value stored in memory at SP + immediate
IR = Mem[PC]
PC = PC + 2
ALUout = SP + (-4)
Memout = Mem[ALUout]
SP = SP + Memout

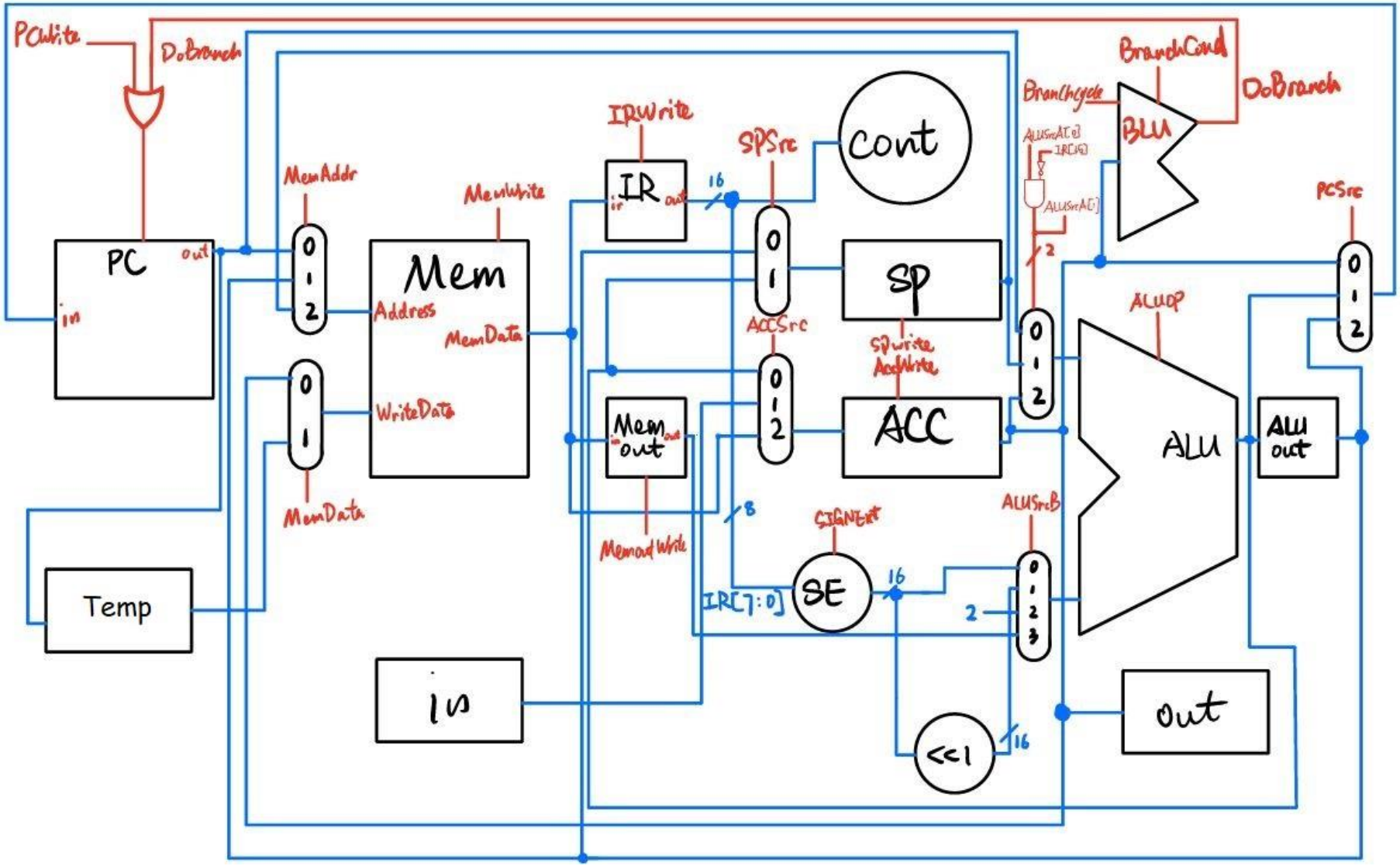
## State Diagram

The following specifies the state diagram for the state machine within the control.


PCWrite = 1
ALUSrcA = 00
ALUSrcB = 10
ALUOp = 000
PCSrc = 01
IRWrite = 1
MEMAddr = 00
MEMWrite = 0
SPWrite = 1
ACCWrite = 0
MOUTWrite = 0
MEMData = xx
ACCSrc = xx
SPSrc = xx
SIGNExt = xx
BRANCHLoop = xx
ALUSrcA = 01
ALUSrcB = 01
PCWrite = 0
IRWrite = 0
SIGNExt = 1
Add/Or/Sub/And/Spc
MEMAddr = 01
MOUTWrite = 1
Add/Or/Sub/And
Spc
ACCWrite = 1
ALUOp = Op
ALUSrcA = 01
ALUSrcB = 11
ACCSrc = 0
MOUTWrite = 0
SPSrc = 1
ALUSrcA = 01
ALUSrcB = 11
SPWrite = 1
MOUTWrite = 0
lwa
ALUSrcA = 10
lw
ACCWrite = 01
ACCSrc = 01
MEMAddr = 01
sw
MEMAddr = 01
MEMData = 0
MEMWrite = 1
j
PCWrite = 1
PCSrc = 00
jal
PCWrite = 1
PCSrc = 00
MEMData = 1
MEMWrite = 1
MEMAddr = 10
branch
BRANCHLoop = 1
BRANCHCond = Cond
PCSrc = 10
Andi/ori/...
SignExt = 0
ALUSrcA = 10
ALUSrcB = 00
ALUOp = Op
ACCWrite = 1
ACCSrc = 00
addi
ALUSrcA = 10
ALUSrcB = 00
ALUOp = Op
ACCWrite = 1
ACCSrc = 00
spi
SPWrite = 1
SPSrc = 0

## DataPath

The main thing that makes our datapath unique is that we do not utilize a large register file. We instead only have ACC and SP to be used and changed. We also only make use of a single ALU. We chose to do this because we are implementing a multicycle style processor which allows us to use the ALU multiple times per instruction if we only use it a maximum of once per cycle.


PCwrite
DoBranch
BranchCond
BranchCycle
BLU
DoBranch
IRWrite
SPSrc
cont
MemAddr
MemWrite
IR
PCSrc
PC
out
in
Mem
Address
MemData
WriteData
SP
ACCSrc
SPwrite
AccWrite
ALUOP
ACC
ALU
ALU out
MemData
Mem out
MemoryWrite
SIGNEXT
ALUSrcB
Temp
IR[7:0]
SE
16
2
8
in
<<1
out

## Hardware components

Below is a list of all the hardware components that will be implemented in our design and all their inputs, outputs, and control bits. All of these components will be implemented using Verilog.

**ALU**

A - 16 bit in

B - 16 bit in

C - 16 bit out

zero - 1 bit out

neg - 1 bit out

ALUop - 3 bit in control

The ALU will take in A and B, preform an arithmetic operation using those two values, then output the result to C.

zero will be on if and only if the result of the operation is zero.

neg will be on if and only if the result is negative.

ALUop determines which operation takes place in the ALU.

This component implements op and + in our rtl.

Implementation plan: We will build and test a single 1 bit ALU that can add, subtract, AND and OR. Once that works, we will place 16 1 bit ALUs in a row with necessary modifications to make the other operations (like the shifts) work. Lastly, some gates be added to determine the value of zero and neg.

Testing plan: We will test the ALU by doing at least one normal operation of each ALUop with both positive and negative values of A and B. Extra tests will done to see if the ALU will perform edge cases correctly (For example, shifting by the maximum amount).

**Register**

In - 16 bit in

Out - 16 bit out

Clock

“Write” - 1 bit in control

The register will take In and change its Output to match in on the rising clock edge if Control is on.

This component implements IR,ACC, SP, PC, ALUout, Memout, In and Out.

Implementation plan: We will likely use the example register provided in the resources section as a base. The write bit may be used for some registers (like the IR), while for others it may be connected to VCC.

Testing plan: will test the ALU by doing at least one normal operation of each ALUop with both positive numbers, negative numbers and zero as values of A and B. Extra tests will done to see if the ALU will perform edge cases correctly (max and min values).

**Memory**

Address - 10 bit

WriteData - 16 bit

MemData - 16 bit

MemRead - 1 bit Control

MemWrite - 1 bit Control

Clock

If MemWrite is on, during the clock cycle, the data at memory address "Address" will be overwritten with WriteData.

If MemRead is on, during the clock cycle, the data at memory address "Address" will be output to MemData.

This implements Mem[x].

Implementation plan: The memory will be implemented similarly to lab7, where the memory is auto-generated using the specifications we provide. We may have to decrease the address size if the memory depth we choose is too high. We will use distributed memory because it is faster.

Testing plan: First, the tests will read from memory in a few locations to see if the data initialized in the coe file can be read correctly. The tests will include writing to memory in a variety of locations including the first and last address of memory. Then, the tests will see if those written locations have changed values. A test will be performed to make sure a memory value does not change if both control bits are off.

**Extender**

In - 8 bit in

Out - 16 bit out

SignEx – 1 bit in control

Each of the most significant 8 bits of the output will be equal to the MSB of the input if SignEx. Otherwise, these bits will be zero.

The Least significant 8 bits of the output are the same as the input.

This implements SE and ZE.

Implementation plan: The 9 MSB's of output will come from an and gate with the inputs of the gate being SignEx and the MSB of input. All other bits stay the same.

Testing plan: Inputs of 0 and –1 with SignEx both on and off will be input to see if the sign or zero extends correctly.

**Single left shift**
In - 16 bit in

Out - 16 bit out

The output is the input shifted left by 1 bit.

This implements "<< 1".

Implementation plan: To implement the device, each of the wires in the bus will shift over by one. The MSB of input will not connect anywhere. The LSB of output will be grounded.

Testing plan: Inputs of all bits on, all bits off, and a few bits on will be tested to see if they shift left by one.

**2x1 Multiplexer**
A – an N bit bus in
B - an N bit bus in
C – an N bit bus out
"switch" - a 1 bit control

If switch is off, C will output A, otherwise C will output B.

Implementation plan: We will use what we learned in class to implement the device using verilog.

Testing plan: We can exhaustively test that the multiplexer passes the correct value for both values of switch.

**4x2 Multiplexer**
A – an N bit bus in
B - an N bit bus in
C– an N bit bus in
D - an N bit bus in
O– an N bit bus out
“switch” - a 2 bit control

If switch is 00, O will output A. If switch is 01, O will output B. If switch is 10, O will output C. If switch is 11, O will output D.

Implementation plan: We will use 3 2x1 muxes wired together to form this multiplexer.

Testing plan: We can exhaustively test that the multiplexer passes the correct value for all values of switch.

**Branch Logic Unit (BLU)**
In – 16 bit in
BranchCond – 2 bit in control
BranchCycle – 1 bit in control
DoBranch – 1 bit out

Finds whether the input is zero and whether it is negative. Then uses the BranchCond to determine if the PC should branch or not and sends that result on DoBranch. DoBranch is zeroed out if BranchCycle is zero.

Implementation plan: Only a line of and gates are needed to see if the input is zero, and the MSB determines if the input is negative. From there, it is a simple case statement to see if DoBranch should be positive.

Testing plan: At a minimum, a test should be performed with each of the 4 branch condition with an input that should cause a branch and one that should not cause a branch and check the corresponding output on DoBranch.

**Control**
Opcode - 8 bit in

Control bits – 23 bits out (listed below)

The control will take in the opcode and set the control bits so that the necessary operation can occur in the DataPath. The control acts as a state machine, outputting control bits based on the current operation and cycle.

Implementation plan: We will implement this as a state machine in verilog, similar to the state machine example on the course website.

Testing plan: We will input one opcode of each major type of instruction and make sure that control outputs the correct control bits for each of the following clock cycles.

## Control Bits

Below is a list of all the control signals that are used by our design. There is a short description of each signal as well as what each value of the signal corresponds to.

**MemOutWrite** – This bit determines if the data from memory is written to MemOut.

0 – Not writing
1 – Writing enabled

**MemWrite** – This bit determines whether the 16-bit value at MemData is wrote to the address.

0 – Not writing
1 – Writing enabled

**ACCWrite** – Determines if data can be written to the ACC.

0 – Writing to the ACC is disabled
1 – Writing to the Acc is enabled

**SPWrite –** Determines if data can be written to SP.

0 – Writing to SP is disabled
1 – Writing to SP is enabled

**SignExt** – Determines if the immediate value needs to be sign extended or zero extended.

0 – The immediate will be zero extended
1 – The immediate will be sign extended

**AluSrcA** – determines what value is input into the ALU at input A. The least significant bit (AluSrcA[0]) is hardwired to the most significant bit of the opcode to switch between SP and PC, this is done to accommodate the Branch instructions in our RTL

00 – PC
01 – SP
10 – Acc
11 – Unused

**AluSrcB** – Determines what value is input into the ALU at input B.

00 – Extended Immediate
01 – Extended Immediate Left Shifted by 1
10 – 2 (used for PC = PC + 2)
11 – MemOut

**PCSrc** – Determines what value is stored back into the PC. This is necessary for the Jump and Branch instructions as the PC will need to be different from PC + 2. Two of the inputs use the ALUout value but one of them uses the stored value in the ALUout register. This is because our branch instructions require that we store the ALUout value to be used later, whereas incrementing PC by 2 can store directly back into PC.

00 – Acc
01 – ALUout value before being stored in the ALUout register
10 – ALUout value after being stored in the ALUout register
11 – Unused

**PCWrite** – Determines if the PC can be written to or not.

0 – PC can not be written
1 – Writing to PC is enabled

**AluOp** – These control bits are used by the ALU to determine what operation is being done by the ALU

000 – Add
001 – Sub
010 – Or
011 – And
100 – Shift Left
101 – Shift Right
110 – Shift Right Arithmetic
111 – No Op (used for branch condition checking)

**IRWrite** – This determines if the Instruction Register can be written to.

0 – Writing to IR is disable
1 – Writing to IR is enabled

**AccSrc** – This bit determines the data being input into the Acc

00 – Input to the ACC is the output of the ALU
01 – Input to the ACC is the value in the IN register
10 – The output of memory

**SpSrc** – This bit determines what data is input into SP

0 – The value of the ALUout register
1 – The value of the ALU before being stored in the ALUout register

**BranchCond** – This control signal determines what condition is necessary for a branch instruction and is used by the BLU

00 – Branch if Neg
01 – Branch if Zero
10 – Branch if Not Zero
11 – Branch if Positive

**BranchCycle** - This control signal zeroes out the output of the BLU if the cycle is not the execution phase of a branch instruction.

0 – zero out DoBranch

1 – DoBranch remains the same

**MemAddr** – Determines what address the memory block uses to read from or write to

00 – PC

01 – ALUout

10 – SP

11 – Unused

**MemData** – Determines the data that will be written to the address in memory given that the appropriate control bits are active.

0 – ACC

1 – PC

**OutWrite** - Determines if Out should be written to in this clock cycle

0 - Do not write to Out

1 – Write to Out

## Input/Output

Our plan to implement an I/O aspect to our project is to use the FPGA board switches and LCD. We will use the switches and one of the buttons as an immediate input to our accumulator, and we will use the LCD as an output to display the value in the accumulator. To create these peripherals, we used some files from Lab0 and the interrupt lab. We also created some of our own schematics and edited others.

The LCD will display a 16 bit hex value that is equal to whatever is stored in the accumulator at that time. The switches will be used to set the 4bit value which will be zero extended and loaded into the accumulator upon pressing the west button.

## Steps for integrating parts together:

We plan to integrate our parts together in the same order that they are used in the RTL so that an instruction can be traced through the datapath from when it is retrieved from memory. Minor intermediate testing will be performed after adding each component. Comprehensive testing will happen after adding each cycle/instruction.

- Step 1: First, we will connect the PC to the memory, the memory to the IR and IR to control. To test, we will use the PC to load and read data into memory and then see if the control will output the correct control bits based on the instruction read from memory.
- Step 2: We will connect alu to its input muxes and ALUOut. If ALU was testes exhaustively, few tests might need to be done on this step.
- Step 3: The hardware from parts 1 and 2 and those registers and muxes left will be wired together to form a somewhat complete datapath. We will test if basic arithmetic instructions such as addi and ori will work.
- Step 4: Other pieces of hardware from lesser used instructions will be added incrementally, and tests will be added for each of those instructions to see if their corresponding hardware was integrated correctly.

## Test Locations and Purposes

This is a list of the tests that we wrote for our pieces of hardware as well as tests for our integration steps. If a testbench requires a certain coe file to be initialized in the memory, the coe file name will be stated in the header of the testbench.

| Test Object | Test File Name | Test File Location (Directory) | Test Purpose |
|---|---|---|---|
| alu16 | alu16b_tb.v | implementation\alu16 | alu16 |
| BLU | BLUt.v | implementation\BLU | BLU |
| control | ControlTest.v | implementation\integration1 | control |
| Extender | Extender_Test.v | implementation\Extender | Extender |
| integration1 | IntegrateStep1Test.v | implementation\integration1 | control + memory |
| integration2 | implementationStep2_tb.v | implementation\IntegrationStep2 | Tests about ALU and muxes |
| Left shifter | Left_shift_1bt.v | implementation\Left_shift_1b | Left shift one bit |
| memory | Memory_tb.v | implementation\integration1 | memory |
| mux2x1 | Mux_2x1t.v | implementation\Mux_2x1 | 2x1 mux |
| mux4x2 | Mux_4x2t.v | implementation\Mux_4x2 | 4x2 mux |
| processor | acc_tests_a_tb.v | implementation\integration3 | Tests if and, or, sub, add operations work in processor |
| processor | acc_instructions_test.v | implementation\integration3 | Tests if and, or, sub, add operations work in processor |
| processor | load_store_tb.v | implementation\integration3 | Tests load and store in processor |
| processor | cpu_tb.v | implementation\integration3 | Tests immediate instructions |
| processor | inouttb.v | implementation\integration3 | Tests input and output of processor (no periphreals) |
| processor | SumTest.v | implementation\integration3 | Tests recursive sum program seen earlier in design document |
| processor | gcdtest.v | implementation\integration3 | Tests GCD program |
| processor | RelPrimeTest.v | implementation\integration3 | Tests RelPrime program |
| Register16 | test.v | implementation\Register16 | Register16 |

## Design Performance:

The following lists our processor's performance when running relprime.

- Bytes required to run relprime: 108 Bytes
    - Bytes required to store relprime and gcd: 98 Bytes
    - Bytes required to store variables: 10 Bytes
- Total number of instructions executed when the input is 0x13B0: 112233
- Total number of clock cycles required to execute relprime when the input is 0x13B0: 357166
- Average number of cycles per instruction: 3.18
- Cycle time for design: 35.2 Mhz frequency (28.388ns period)
- Total execution time to execute relprime when the input is 0x13B0: 10.71498 ms
- Device Utilization Summary:

| Device Utilization Summary | | | | [-] |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** | **Note(s)** |
| Number of Slice Flip Flops | 159 | 9,312 | 1% | |
| Number of 4 input LUTs | 1,824 | 9,312 | 19% | |
| Number of occupied Slices | 964 | 4,656 | 20% | |
| Number of Slices containing only related logic | 964 | 964 | 100% | |
| Number of Slices containing unrelated logic | 0 | 964 | 0% | |
| Total Number of 4 input LUTs | 1,832 | 9,312 | 19% | |
| Number used as logic | 800 | | | |
| Number used as a route-thru | 8 | | | |
| Number used for 32x1 RAMs | 1,024 | | | |
| Number of bonded IOBs | 35 | 232 | 15% | |
| Number of BUFGMUXs | 1 | 24 | 4% | |
| Average Fanout of Non-Clock Nets | 6.57 | | | |

## Signifigant Changes

**Milestone 3:**

- Added the Lwa instruction
- Added RTL tests
- **Added detailed description about programming convention**

**Milestone 4:**

- Added BranchCycle control bit to limit which cycles the BLU operates on
- Changed the logic for the write bit of the PC register to contain an or gate instead of an and gate.
- Registers now write on the falling edge of the clock cycle
- The integration plan is now more specific on which components will be integrated in what order

**Milestone 5:**

- Updated missing control signals in control for ALUSrc muxes during lw and sw cycles

**Milestone 6:**

- Added a register (“Temp”) used only in the jal instruction to temporarily hold the value of PC to be stored in memory as a return address for a half clock cycle while PC changes to the jump address.