CSSE-232 Project

Team Avicenna

Ron Farrell and Zhen Yang

1. High-Level design description:

Our design will be mainly an accumulator type architecture.

Actually, users won't be able to directly access any of those registers.. Users can only use INSTRUCTIONs is manipulate those registers. So in this case we don't have a register code here.

Register	Name	Description	SAVER
acc	Accumulator	The accumulator register	N.A

2. Addressing mode:
    1. If no immediate entails at all: direct addressing. (not sure if we need this)
    2. If need immediate : immediate addressing for everything(include stackset, jal, branch, etc..)

3. Stack usage:

Red part is “convention”, every function has\ to follow this format. The yellow part user can do whatever they want here, like store temporary value etc. Note: input/output should be a fixed and specified place in memory.. But not stack..

0(sp)	ra - return address
1(sp)	v0 - return value
2(sp)	a0 - argument 0
3(sp)	a1 - argument 1
4(sp)	b0 - branch
5(sp)	whatever you want g0, s0, t1 …
6(sp)	whatever you want g0, s0, t1 …
…	…

4. An unambiguous English description of each machine language instruction format type and its semantics (see Section 2.5, pages 80 through 87 of your book).

N Type:(means no arguments) total 8 bit (16 bit for now)

3 bit unused	5 bit opcode

I Type: (for instruction that immediate) total 16 bit

11 bit immediate	5 bit opcode

5. English Description:
    1. see Excel Sheet: Instructions(excel file also in design folder)
6. Symbolic description of instructions:
    1. see Excel Sheet: Instructions(excel file also in design folder)
7. Rules for Translating Assembly language to machine language
    1. One to one correlation between opcode and instruction.
    2. see Excel Sheet: Instructions(excel file also in design folder)
8. Explanation of Procedure call convention
    1. Return address: before a function(f1) calls another function(f2), f1 should always save its caller(f0)'s return address register(ra) on the stack, and retrieve it from the stack after the function call
    2. Parsing argument: it's a little weird, for example like relprime now needs to call gcd, relprime will first put n, m on the specified places on the stack 4(sp) and 6(sp), and then jal gcd. So now gcd will first make space on the stack, let's assume it made 14 places on the stack like sp = sp - 14 in risc-v. And then gcd will need to access the n and m that is actually stored at relprime's stack.. So it needs to access 14+4(sp) for n and 14+6(sp) for m… we could have used a frame pointer for this.. But ..
    3. All functions should always start with making space on the stack. And right before recovering the stack, put the return value on the acc register, and then after recovering the stack, put the return value at 4(sp), (in this case, it is the caller's 4(sp))
    4. If a function is calling multiple sub-function, then it is the caller's job to store the v0 stack value in other places on the stack. Since all of its callee will try to save their return value on the same place of caller's stack.
    5. Only the first five places on the stack is fixed: 0(sp) for return address, 4(sp) for return value, 8(sp) for argument1, 12(sp) for argument 2, 16 for "branch comparing value"
    6. the branch instruction will always compare the value between the acc and the value store at the 16(sp).

9. Assembly program + machine code for relPrime:

Relprime:

```
spInit    14              // sp = sp - 14 (make room for 7 values)
accgetra                  // acc = ra
```

```
        stackset 0              //0(sp) = acc = ra  //save ra on the stack

        // put m and n on to "general register places" on the stack
        accseti  2              //acc = m = 2
        stackset 10             // 10(sp) = acc = 2

        stackget 18             //acc = n          // actually accessing caller's stack: 14 + 4(ra,v0) = 18
        stackset 12             //12(sp) = acc = n


Whileloop:
        //put n, m into a0, a1
        stackget 12             // acc = 12(sp) = n
        stackset 4              // 4(sp) = acc = n
        stackget 10             // acc = 10(sp) = m
        stackset 6              // 6(sp) = acc= m

        jal      gcd            // go to gcd

        stackget 2              // acc = 2(sp) = gcd's return value
        stackset 8              // 8(sp) = acc = gcd's return value

        accseti 1               // acc = 1

        beq      returnM        // branch if acc == 8(sp), to returnM

        //if not branched continue below code:
        stackget 10             // acc = 10(sp) = m
        addi     1              // acc = acc + 1 = m+1
        stackset 10             //10(sp) = m+1 (m = m + 1)
        jal      whileloop      // continue to the next iteration



returnM:
        stackget 0              // acc = 0(sp) = ra
        ragetacc                // ra = acc
        stackget 10             // acc = 10(sp) = m
        sprelease 14            // sp = sp - 14
        stackset 2              // 2(sp) = acc = m
        jra                     // go back to caller

—---------------------------------------------------------------------------------------------------
Gcd:
        spInit   14             // sp = sp - 14                                  00000001110 01010
        accgetra                // acc = ra                                      00000000000 01101
```

```
        stackset 0              //0(sp) = acc = ra                                  00000000000 00000

        //get a0, a1 from previous function's stack and put it to current functions: g0 g1

        stackget 18             //acc = 18(sp) = a          // 18 = 14 + 4         00000010010 00001
        stackset 10             //10(sp) = acc = a                                  00000001010 00000

        stackget 20             //acc = 20(sp) = b          // 20 = 14 + 6         00000010100 00001
        stackset 12             // 12(sp) = acc = b

        accseti 0               // acc = 0                                          00000000000 00010
        stackset 8              // 8(sp) = acc = 0           //8(sp) is branch value 00000001000 00000

        stackget 10             // acc = 10(sp) = a                                 00000001010 00001
        bne     whileloop       // if acc != 0, go to whileloop                     ? 00110

        //otherwise continue below: ( which is return b)

        stackget 0              // acc = 0(sp) = ra                                 00000000000 00001
        ragetacc                // ra = acc                                         ? 01110
        stackget 12             //acc = 12(sp) = b                                  00000001100 00001
        spRelease 14            //  sp = sp + 14                                    00000001110 01011
        stackset 2              // sp(2) = acc = b                                  00000000010 00000
        jra                     //jump to return address                           00000000000 01100


Whileloop:
        stackget 12             // acc = 12(sp) = b                                 00000001100 00001
        stackset 8              // 8(sp) = acc = b(using for branch comparing)      00000001000 00000
        accseti 0               // acc = 0                                          00000000000 00010
        beq     returnA         // if 0 == b go to returnA                          ? 00111

        //if didn't branch away continue:
        stackget 12             // acc = 12(sp) = b                                 00000001100 00001
        stackset 8              // 8(sp) = acc = b                                  00000001000 00000

        stackget 10             // acc = 10(sp) = a                                 00000001010 00001
        ble     ELSE            //if acc <= 8(sp) go to else                       ? 01000
                                //if a <=b go to else
        //otherwise continue below: a = a - b
        stackget 10             // acc =a                                           00000001010 00001
        stacksub 12             // acc = acc - 12(sp)                               00000001100 00101
        stackset 10             // 10(sp) = acc                                     00000001010 00000
        jal whileloop                                                               ? 01001

ELSE:// b = b-a
```

```
        stackget 12         // acc =b                          00000001100 00001
        stacksub 10         // acc = acc - 10(sp) = acc - a    00000001010 00101
        stackset 12         // 12(sp) = acc                    00000001100 00000
        jal whileloop                                         ? 01001

returnA:
        stackget 0          // acc = 0(sp) = ra                00000001100 00001
        ragetacc            // ra = acc                        ? 01110
        Stackget 10         // acc = 10(sp) = a                00000001010 00001
        spRelease 14        // sp = sp + 14                    00000001110 01011
        Stackset 2          // 2(sp) = a                       00000000010 00000
        jra                 //                                00000000000 01100
```

10. Assembly program fragments:
    1. Reading from a register and save the value to accumulator:

       stackget 4 *// acc = 4(sp)*

    2. Write the value of accumulator to the stack:

       stackset 10 *// 10(sp) = acc value*

    3. Loading an address into the accumulator

       stackget 0 *// acc = value at ra*

    4. Iteration & Conditional

C code	**Machine Language**	**Instruction**
int a = 5	accseti 5 stackset 8 accseti 0	000000000101 0010 000000001000 0000 000000000000 0010

for (int i = 0; i < a; i++){ }	loop: beq break addi 1 jal loop	xxxxxxxxxxxx 0111 000000000001 0100 xxxxxxxxxxxx 1001

11. List of generic components specifications:

1. **Memory:**
    a. Input:
        1. MemWrite: if we want to write memory, 0 for no, 1 for yes.
        2. MemRead: if we want to read memory, 0 for no, 1 for yes.
        3. WriteData: what data we want to write into the memory
        4. PC
        5. ALUOut
    b. Output:
        1. MemData: like the data on the stack
    c. Description:
        1. ..just main memory..like instruction file, stack, etc all here..
    d. Implementation:
        1. No idea…
2. **PC:**
    a. Input:
        1. PCWrite: like if we want to write on PC, 0 for no, 1 for yes.
        2. PC+2 ? like because pc = pc+2?
    b. Output:
        1. The new PC value
    c. Description:
        1. An independent PC register
    d. Implementation:
3. **sp: stack pointer register (assume made it independent):**
    a. Input:
        1. ALUOut?: not sure check RTL summary chart..
        2. Does sp need like spWrite or spRead?
        3. spWrite
        4. spRead
        5. Sp +- imme? How to represent this?

b. Output:
    1. The value stored at sp
c. Description:
    1. An independent stack pointer register
d. Implementation:

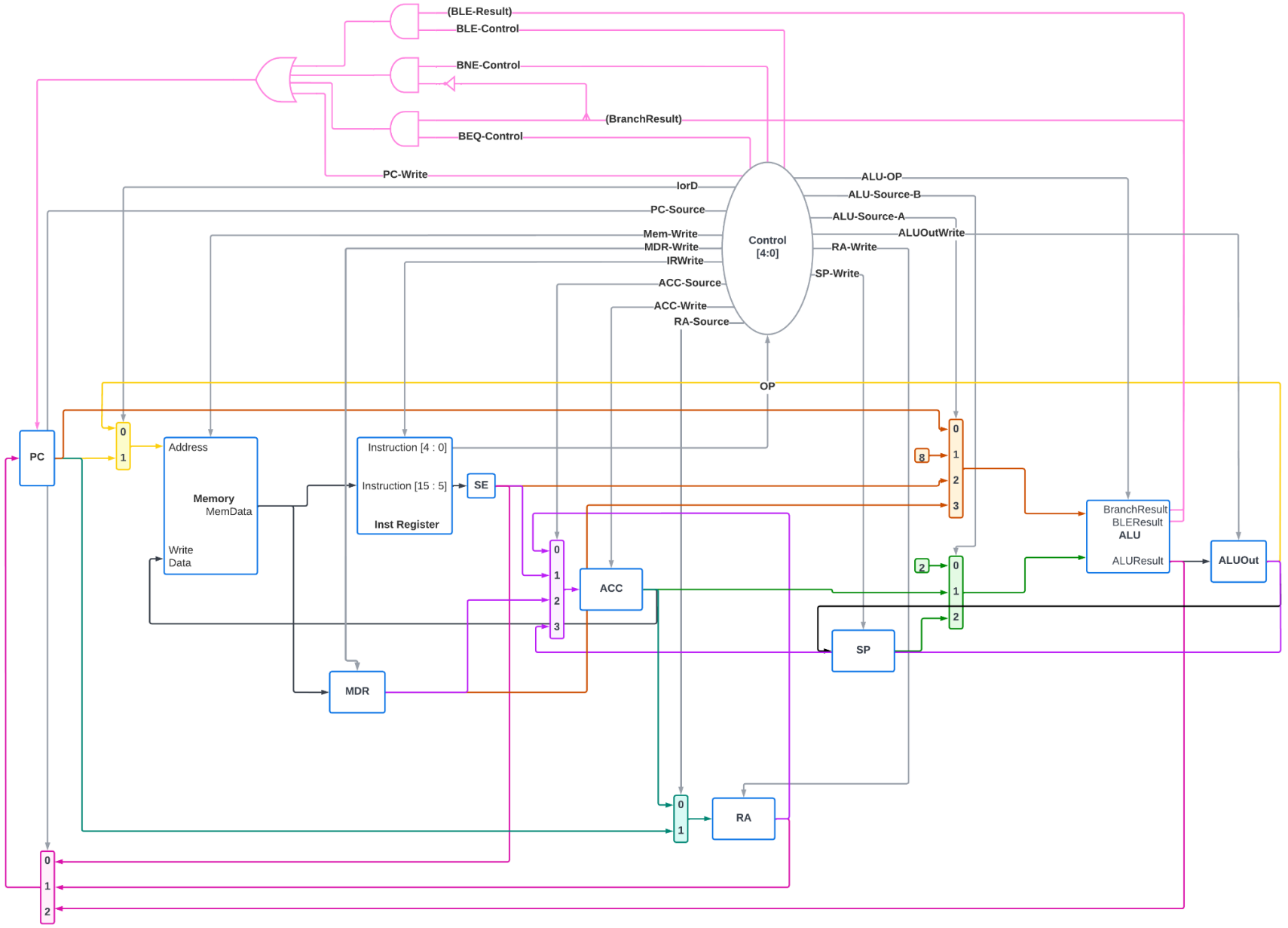
4. **ra: return address register (assume made it independent):**
    a. Input:
        1. Acc
        2. Does ra need like raWrite or raRead?
        3. raWrite
        4. raRead
    b. Output:
        1. The value stored at itself
    c. Description:
        1. An independent return address register
    d. Implementation:
5. **ALU:**
    a. Input:
        1. PC: needs to do like pc +1, pc + 2
        2. Sp: need to add sign extended immediate to sp to access value on the stack
        3. Sign extended immediate value
        4. acc: for addi
        5. ALUControl? Like telling ALU to do an add or sub
    b. Output:
        1. ALUOut:
    c. Description:
    d. Implementation:
6. **ALUControl:**
    a. We could have used only plus in our design. But since functionality are also part of grading criteria, we are still going to have a ALUControl?...
7. **Sign Extender:**
    a. Input:
        1. Immediate value
    b. Output:
        1. Sign extended immediate value
    c. Description:
        1. Just sign extend stuff.
    d. Implementation:

1. Since we got like 12 bits length immediate, we gonna wire the 11th bit to the “12 to 15th”bit….

8. **Accumulator register:**
    a. Input:
        1. AccWrite
        2. AccRead
        3. ra register: the value stored in ra
        4. Sign extended Immediate? Like the result outputted by the sign extender
    b. Output:
        1. The value stored in acc
    c. Description:
        1. An independent register.
    d. Implementation:
9. **Control Unit:**
    a. Input:
        1. OpCode: definitely need this..
    b. Output:
        1. MemWrite
        2. MemRead
        3. PCWrite
        4. ALUOpCode
        5. AccWrite
        6. AccRead
        7. SpWrite
        8. SpRead
        9. RaWrite
        10. RaRead
    c. Description:
        1. Control everything…
    d. Implementation:
10. **ALUOut:**
    a. Input:
        1. The calculated result from ALU
    b. Output:
        1. The same calculated result from ALU
    c. Description:
        1. It's a register
    d. Implementation:

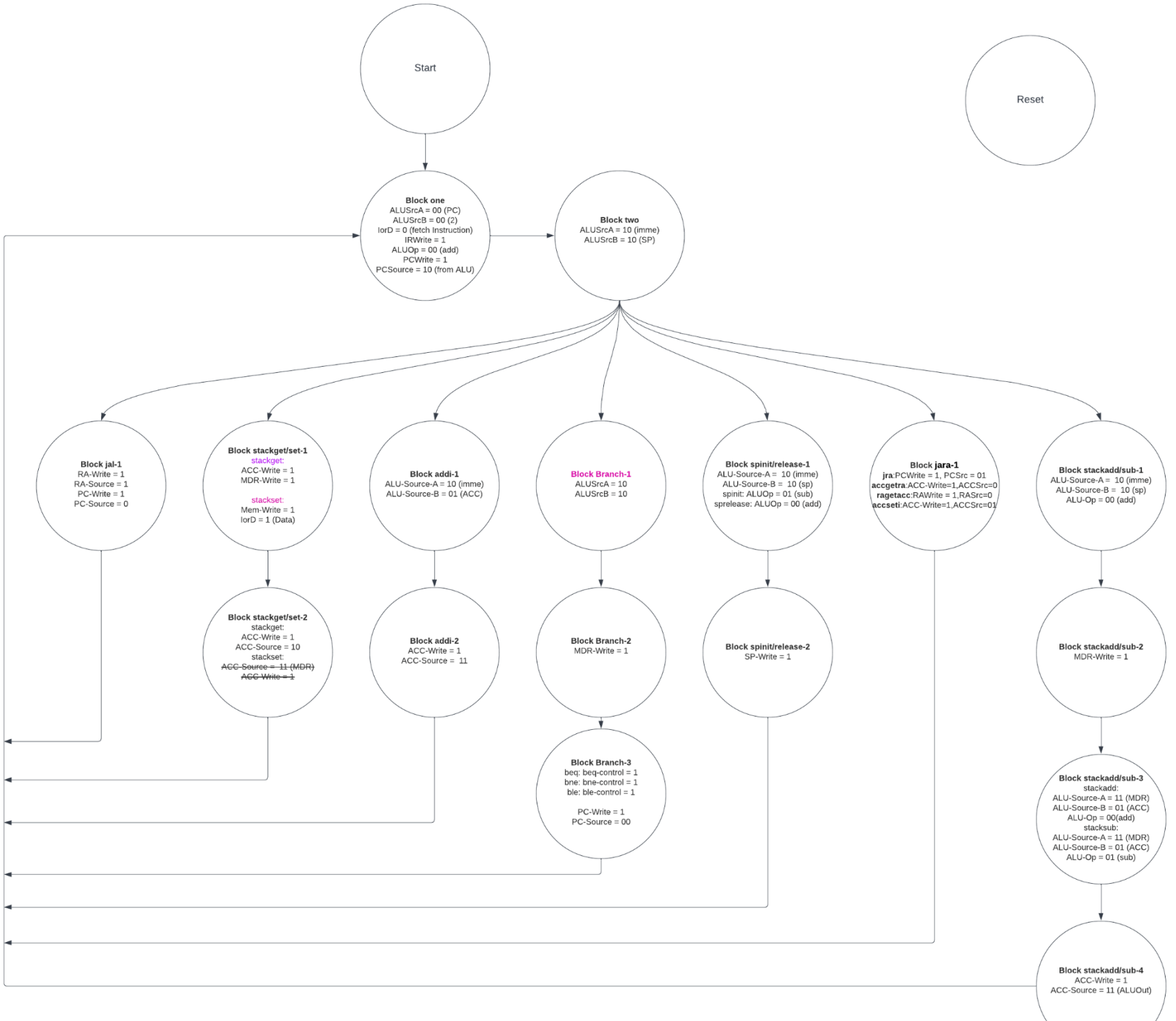
12. Overview of process for double-checking the RTL for errors:
    1. Don’t have any idea yet..
13. RTL Summary Chart: Please refer to this Summary Chart as the main source of RTL, there could be some minor mistakes at the RTLs in the instruction excel file

jal	stackset stackget	addi	Branch beq/bne/ble	spinit	sprelease	stackadd stacksub	jra accgetra ragetacc accseti
PC = PC + 1 Inst = Mem[PC]							
Imme = SE(inst[15 : 5]) ALUOut = sp + imme							
ra = PC PC=imme	stackset: Mem[ALUOut]= acc stackget: MDR=Mem[ALU Out]	ALUOut = acc + imme	ALUOut = sp+8	ALUOut=sp - imme	sp = ALUOut	MDR = Mem[ALU Out]	jra: PC=ra accgetra: acc = ra ragetacc: ra = acc accseti: acc=imme
	stackget: Acc = MDR	Acc = ALUOut	MDR = Mem[ALUOut]	sp = ALUOut		stackadd: ALUOut = acc + MDR stacksub: ALUOut = acc - MDR	
			if(acc op MDR) PC = Imme			acc = ALUOut	



14. [Datapath Design:]

15. [State Diagram]: (.)


Start
Reset
Block one
ALUSrcA = 00 (PC)
ALUSrcB = 00 (2)
IorD = 0 (fetch Instruction)
IRWrite = 1
ALUOp = 00 (add)
PCWrite = 1
PCSource = 10 (from ALU)
Block two
ALUSrcA = 10 (imme)
ALUSrcB = 10 (SP)
Block jal-1
RA-Write = 1
RA-Source = 1
PC-Write = 1
PC-Source = 0
Block stackget/set-1
stackget:
ACC-Write = 1
MDR-Write = 1
stackset:
Mem-Write = 1
IorD = 1 (Data)
Block addi-1
ALU-Source-A = 10 (imme)
ALU-Source-B = 01 (ACC)
Block Branch-1
ALUSrcA = 10
ALUSrcB = 10
Block spinit/release-1
ALU-Source-A = 10 (imme)
ALU-Source-B = 10 (sp)
spinit: ALUOp = 01 (sub)
sprelease: ALUOp = 00 (add)
Block jara-1
jra:PCWrite = 1, PCSrc = 01
accgetra:ACC-Write=1,ACCSrc=0
ragetacc:RAWrite = 1,RASrc=0
accseti:ACC-Write=1,ACCSrc=01
Block stackadd/sub-1
ALU-Source-A = 10 (imme)
ALU-Source-B = 10 (sp)
ALU-Op = 00 (add)
Block stackget/set-2
stackget:
ACC-Write = 1
ACC-Source = 10
stackset:
ACC-Source = 11 (MDR)
ACC-Write = 1
Block addi-2
ACC-Write = 1
ACC-Source = 11
Block Branch-2
MDR-Write = 1
Block spinit/release-2
SP-Write = 1
Block stackadd/sub-2
MDR-Write = 1
Block Branch-3
beq: beq-control = 1
bne: bne-control = 1
ble: ble-control = 1
PC-Write = 1
PC-Source = 00
Block stackadd/sub-3
stackadd:
ALU-Source-A = 11 (MDR)
ALU-Source-B = 01 (ACC)
ALU-Op = 00(add)
stacksub:
ALU-Source-A = 11 (MDR)
ALU-Source-B = 01 (ACC)
ALU-Op = 01 (sub)
Block stackadd/sub-4
ACC-Write = 1
ACC-Source = 11 (ALUOut)

Testing

(Currently have a 5 to 1 mux and an ALU implemented in verilog with testbench in the implement folder)

Unit testing

To test the control input all the different opcodes and make sure the outputs (IRWrite, Mem-write, etc..) are all expected

To test the ALU put in different inputs to test equal/not equal, and different states, check the branch result to = 0 when the desired operation is used (beq, bne, etc.)

To test ACC input different ACC-source and testing once with ACC-write on and off to make sure it only writes and changes when expected

To Test SP same as ACC but with ALU out and different SP-write

To test RA test with different PC's and once with RA-write on and off again to make sure it only changes when expected

To test MDR have different memory outs to make sure it changes.

To test Inst Register change the input on the IR write and memory data and the input on the IRwrite to make sure it only changes when supposed to, at the same time check the SE to make sure it sign extends.

Integration plan and testing

To test inst Register, ALU, ALU out, ACC, Memory, SE, MDR, input instruction that input an instruction that is a beq, bne, etc. for times when the comparison is true and false, and make sure everything is as expected, such as SE

and ALU out, Branch result = 0 when true and =1 when not, and the value is ultimately moved into the acc,

To test RA, SP test jal and jalr to make sure the PC address is saved correctly and saved correctly when the comparison is used in beq and others.

ACC is tested in all these by making sure the values are saved there at the end of every instruction the acc should be changed or updated.

Performance:

1. For testing 0x13B0 (5040) for relprime
    a. Total cycles: 939,664
    b. Execution time: 46,983,150 ns
    c. Instruction size: 65 lines in memory, 130 bytes
    d. CPI: 4.014

Process:

1.