

CSSE 232 COMP ARCH 1
DESIGN DOCUMENT

MISCV TEAM ORANGE:

Drew Kilner, Yash Anand, Julian
Ferrer Rodriguez, Eli Granade,
Ziyu Xie

Implementation plan:

Make Register.

Test: When reset high, output always 0. When reset low, input nonzero value. After a clock cycle, it should stay outputting the inputted value.

Make Register File.

Test:

Make Imm Gen.

Test: Input is 16 bit instruction. Output is 16 bit immediate. Sign extend according to the opcode.

Make IF/ID register. Test:

Make ID/EX register. Test:

Make ALU. Test:

Make EX/MEM register. Test:

Make MEM/WB. Test:

Make Hazard unit. Test:

Make forwarding unit. Test:

Make branch predictor. Test:

Make memory. Test:

Make control. Test:

Potentially integrate while we implement. Like:

Make Register. Test:

Make Register File out of Registers. Test:

Make Imm Gen. Test:

Make IF/ID register. Test:

Connect all of the above. Test:

Make the rest of the components in the IF/ID steps. Test:

Make ID/EX register. Test:

Connect ID parts to ID/EX. Test:

Make ALU. Test:

Finish EX step. Test:

Make EX/MEM register. Test:

Connect all of the above. Test:

Make Data Memory. Test:

Connect EX/MEM to Data Memory. Test:

Make MEM/WB. Test:

Connect all of the above. Test:

Make Hazard unit. Test:

Make forwarding unit. Test:

Connect all parts. Test:

Load and store design mimicking RISC-V at first to ensure full functionality first with the same design philosophies.

- ☐ **Bottom-up approach with a solid foundation eliminates repeated useless work.**
- ☐ **Before starting work, have a clear definition of the task at hand and the desired outcomes first.**
- ☐ **Hardware components defined by Verilog should be vigorously and holistically tested and debugged before assembly software implementation.**

After full functionality is achieved, optimize for performance (defined below) of the benchmark version of Euclid's Algorithm. New instructions specifically optimized for this task seems the most direct. This can later result in the elimination of other instructions (given processor still is general purpose), which satisfies the simplicity rule. Combination with accumulator is a stretch goal, if with proven contribution to performance, or with extra time.

Performance Measure:

Time it takes to run Euclid's Algorithm for the same numbers.

Euclid's algorithm in MISC assembly:

```
relPrime:  sp +_ -8, sp
           ra -> sp+0
           s0 -> sp+2
           s1 -> sp+4
           x0 +_ 2, s0
           x0 +_ 1, s1
           a0 -> sp+6
loop1:     s0 +_ 0, a1
           a0 <- sp+6
           ra \ / gcd
           a0, s1 Y= done
           s0 +_ 1, s0
           x0 \ / loop1
done:      s0 +_ 0, a0
           ra <- sp+0
           s0 <- sp+2
           s1 <- sp+4
           ra /\ 4

gcd:       sp +_ -2, sp
           ra -> sp+0
           a0, x0 Y= returnb
loop2:     a1, x0 Y= returna
           a1, a0 Y< agreater
           a1 - a0, a1
           x0 \ / loop2
agreater: a0 - a1, a0
           x0 \ / loop2
returnb:   a1 +_ 0, a0
returna:   ra <- sp+0
           ra /\ 1
```

| address | assembly | Machine code | comments |
|---------|--------------------|---------------------|---------------------|
| 0x00 | sp += -8, sp | 0011 1000 1001 0001 | //begin relprime |
| 0x01 | ra -> sp+0 | 0000 0000 1000 1011 | |
| 0x02 | s0 -> sp+2 | 0000 0100 1011 0011 | |
| 0x03 | s1 -> sp+4 | 0000 1000 1011 1011 | |
| 0x04 | x0 += 2, s0 | 0000 0100 0011 0001 | |
| 0x05 | x0 += 1, s1 | 0000 0010 0011 1001 | |
| 0x06 | a0 -> sp+6 | 0000 1100 1010 0011 | |
| 0x07 | s0 += 0, a1 | 0000 0001 0111 0001 | //begin loop1 |
| 0x08 | a0 <- sp+6 | 0000 1100 1001 0010 | |
| 0x09 | ra \ / gcd | 0000 0010 0100 1110 | //imm = 9 |
| 0x0A | a0, s1 Y= done | 0000 1111 0001 1100 | //imm = 3 |
| 0x0B | s0 += 1, s0 | 0000 0011 1011 0001 | |
| 0x0C | x0 \ / loop | 1111 1110 1100 0110 | //imm = -5 |
| 0x0D | s0 += 0, a0 | 0000 0001 1010 0001 | //begin done |
| 0x0E | ra <- sp+0 | 0000 0000 1000 1010 | |
| 0x0F | s0 <- sp+2 | 0000 0100 1011 0010 | |
| 0x10 | s1 <- sp+4 | 0000 1000 1011 1010 | |
| 0x11 | ra /\ 4 | 0000 0001 0000 1111 | |
| 0x12 | sp += -2, sp | 0011 1100 1001 0001 | //begin gcd |
| 0x13 | ra -> sp+0 | 0000 0000 1000 1011 | |
| 0x14 | a0, x0 Y= returnb | 0000 0001 0011 1100 | //imm = 7 |
| 0x15 | a1, x0 Y= returna | 0000 0001 0111 1100 | //begin loop2;imm=7 |
| 0x16 | a1, a0 Y< agreater | 0000 1001 0101 1101 | //imm = 3 |
| 0x17 | a1 - a0, a1 | 0001 1001 0110 1000 | |
| 0x18 | x0 \ / loop | 1111 1111 0100 0110 | //imm = -3 |
| 0x19 | a0 - a1, a0 | 0001 1011 0010 1000 | //begin agreater |
| 0x1A | x0 \ / loop | 1111 1110 1100 0110 | //imm = -5 |
| 0x1B | a1 += 0, a0 | 0000 0001 0110 0001 | //begin returnb |
| 0x1C | ra <- sp+0 | 0000 0000 1000 1010 | //begin returna |
| 0x1D | ra /\ 1 | 0000 0000 0100 1111 | |

Base Instructions:

| inst | fmt | func | opcode | description |
|------|-----|------|--------|---|
| + | R | 0000 | 000 | $R[rd] = R[rs1] + R[rs2]$ |
| - | R | 0001 | 000 | $R[rd] = R[rs1] - R[rs2]$ |
| | R | 0010 | 000 | $R[rd] = R[rs1] \mid R[rs2]$ |
| & | R | 0011 | 000 | $R[rd] = R[rs1] \& R[rs2]$ |
| +_ | I | 00 | 001 | $R[rd] = R[rs1] + SE(imm)$ |
| <<_ | I | 01 | 001 | $R[rd] = R[rs1] \ll imm$ |
| >>_ | I | 10 | 001 | $R[rd] = R[rs1] \gg imm$ |
| X _ | I | 11 | 001 | $R[rd] = R[rs1] \wedge SE(imm)$ |
| <- | M | | 010 | $R[rd] = M[R[rs1] + SE(imm)]$ |
| -> | M | | 011 | $M[R[rs1] + SE(imm)] = R[rd]$ |
| Y= | Y | | 100 | If($rs1 == rs2$) $PC += SE(imm) \ll 1$ |
| Y< | Y | | 101 | If($rs1 < rs2$) $PC += SE(imm) \ll 1$ |
| \ | J | | 110 | $R[rd] = PC + 2$ $PC += SE(imm) \ll 1$ |
| /\ | J | | 111 | $PC = R[rd]$ $R[rd+1] += SE(imm) \ll 1$ |

| 15 | 14 | 13 | 12 | 11 | 9 | 8 | 6 | 5 | 3 | 2 | 0 | Type |
|-----------|-----------|----------|----------|----|---|-----|---|----------|---|----|---|------|
| func[3:2] | func[1:0] | | rs2 | | | rs1 | | rd | | op | | R |
| func | | imm[4:0] | | | | rs1 | | rd | | op | | I |
| | imm[6:3] | | rs2 | | | rs1 | | imm[2:0] | | op | | Y |
| | | imm[6:0] | | | | rs1 | | rd | | op | | M |
| | | | imm[9:0] | | | | | rd | | op | | J |

| Register | Name | Description | Saver |
|----------|------|--|--------|
| x0 | zero | This register is always zero | - |
| x1 | ra | This is the return address | caller |
| x2 | sp | This is the stack pointer | - |
| x3 | at | This is the assembler temporary | - |
| x4 | a0 | This is a temporary register that is used for function inputs and function return values | caller |
| x5 | a1 | | |
| x6 | s0 | These are usable saved registers | callee |
| x7 | s1 | | |

| Instruction | Definitions |
|-------------|---|
| + | Takes the two registers with the operand between, adds the values, then puts it into the third register. |
| - | Subtracts the register after the operand from the first, then puts it into the last. |
| | Takes the two registers with the operand between, bitwise or of the values, then puts it into the third register. |
| & | Takes the two registers with the operand between, bitwise and the values, then puts it into the third register. |
| +_ | Takes the first register and the immediate after the operand, adds them and puts them into second register. |
| <<_ | Takes the first register and shifts it left the immediate number of times, then puts it in the second register. |
| >>_ | Takes the first register and shifts it right the immediate number of times, then puts it in the second register. This operation does not sign extend the shifted bits. |
| X _ | Takes the first register and xoris it with the sign extended immediate, then puts it into the second register. |
| <- | Loads a 16 bit word out of the memory address equal to the second register plus the immediate, and puts it in the first register |
| -> | Takes a 16 bit word out of the first register and puts it in the memory address equal to the second register plus the immediate |
| Y= | If the first two registers are equal, then then bit shift the immediate left and add it to the program counter |
| Y< | If the first register is less than the second register, then then bit shift the immediate left and add it to the program counter |
| \ | Used to step into function. Put the current PC into the first register, add two to it, then bit shift the immediate left and add that to the PC |
| /\ | Used to step out of function. Put the first register into the PC. Adds sign extended and shifted immediate to rd+1.S |

Example Commands in both MISK-V assembly and machine code:

| assembly | comments |
|--|---------------------------|
| x6, x7 Y= skip x0, x0 Y= label skip: ...continue code without branch | //branch not equal x6, x7 |
| x6, x7 Y< skip x0, x0 Y= label skip: ...continue code without branch | //branch if x6 >= x7 |

Simple function implementation:

Simple if-else statements:

```
If (a <= b) {
a++;
else {
a = a + b;
}
```

| <u>Assembly (a = x6, b = x7):</u> | <u>Machine code:</u> |
|-----------------------------------|---------------------------------|
| x7, x6 Y< 3 | Y-Type 0000 110 1 11 01 1 101 |
| x6 += 1, x6 | I-Type 00 00 001 1 10 11 0 001 |
| x0 \ / 1 | J-Type 0000 0000 1 000 0 110 |
| x6 + x6, x7 | R-Type 00 00 111 1 10 11 0 000 |

Simple loop (adds all elements in array):

```
int[] a = {1, 3, 5, 7, 9};
int sum = 0;
for(int i = 0; i < a.length; i++){
sum = sum + a[i];
}
```

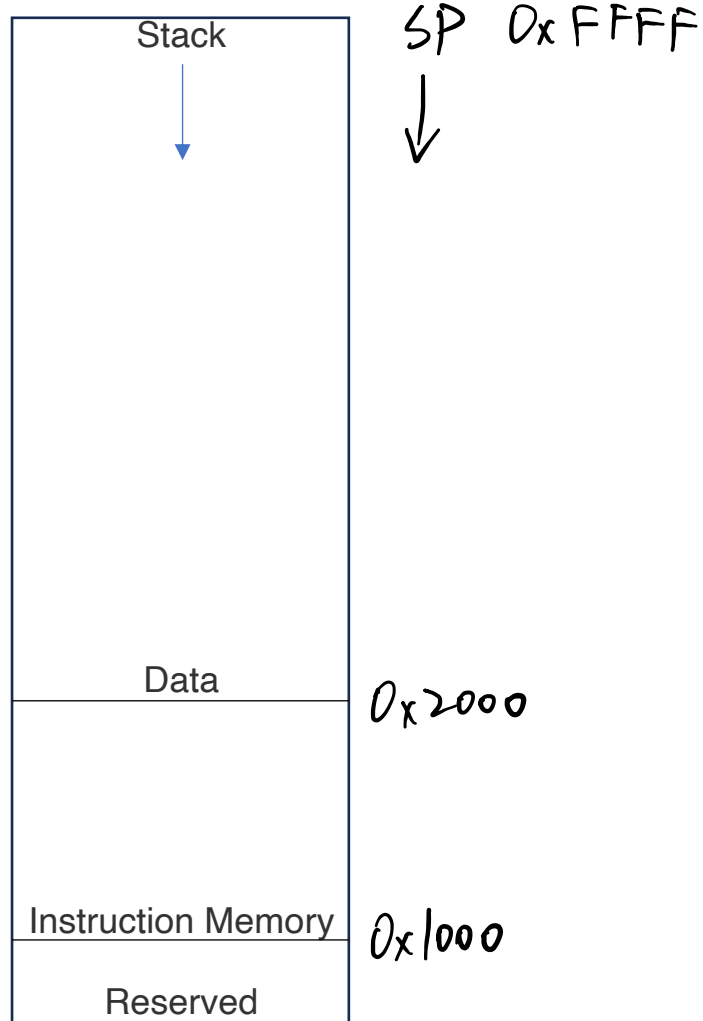
| <u>Assembly (address of array a in mem = x4, array length = x5):</u> | <u>Machine code:</u> |
|--|---------------------------------|
| x6 + x4, zero | R-Type 00 00 000 1 00 11 0 000 |
| x7 + x5, zero | R-Type 00 00 000 1 00 11 1 000 |
| x5, zero Y= 5 | Y-Type 0000 110 1 01 10 1 100 |
| x5 <- x6+0 | M-Type 0000 000 1 10 10 1 010 |
| x6 +_ 2, x6 | I-Type 00 00 010 1 10 11 0 001 |
| x5 +_ -1, x5 | I-Type 00 11 111 1 10 10 1 001 |
| x0 \/-4 | J-Type 1111 1111 00 00 0 110 |

Simple while loop:

```
While(a < b) {
a = a<<;
}
```

| <u>Assembly(a = x6, b = x7):</u> | <u>Machine code:</u> |
|----------------------------------|--------------------------------|
| x7, x6 Y< 3 | Y-Type 0000 110 1 11 01 1 101 |
| x6 <<_ 1 | I-Type 01 00 011 1 10 11 1 001 |
| x0 \/-2 | J-Type 1111 1111 10 00 0 110 |

Memory Map



(Can break data into dynamic, static, stack later if needed, but for now it's just a stack that grows from top to bottom)

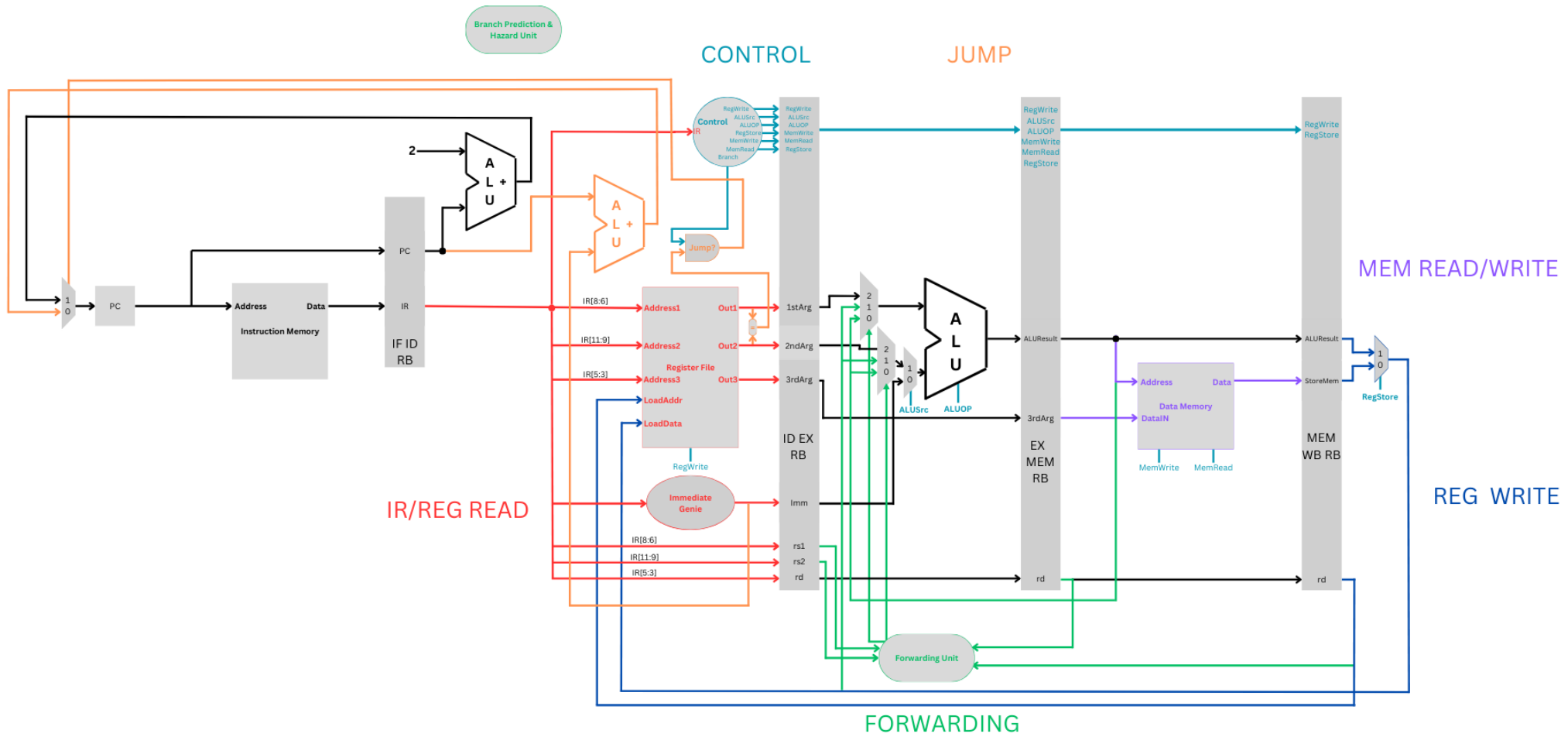
Procedure Call Convention

Caller saves ra, t, and a registers as needed. Moves sp. Do \/.

Callee saves s registers, moves sp. Puts return values in a registers and restores all s registers. Move sp back. Do /\.

Caller restores ra, t, and a registers.

| | R-type | I-type | M-type | Y-Type | J-type |
|---------|---|-------------------------------|--|--|---|
| Fetch | IR <= InstructionMem[PC] | | | | |
| Decode | PC <= PC + 2 1stArg <= Reg[IR[8:6]] 2ndArg <= Reg[IR[11:9]] 3rdArg <= Reg[IR[5:3]] Imm <= ImmGen[Reg[IR]] | | | | |
| Execute | ALUResult <= 1stArg op 2ndArg | ALUResult <= 1stArg op imm | ALUResult <= 1stArg + SE[15:9] | JumpTo <= PC + SE(2*immediate) Y=: If(1stArg == 2ndArg) Y<: If(1stArg < 2ndArg) | V: Reg[IR[5:3]] <= PC A: PC <= 3rdArg |
| Mem | | | ->: DataMem[ALUResult] <= 3rdArg <-: StoreMem <= DataMem[ALUResult] | If true: PC <= ALUResult | V: PC <= ALUResult A: Reg[IR[5:3] + 1] <= ALUResult |
| Write | Reg[IR[5:3]] <= ALUResult | Reg[IR[5:3]] <= ALUResult | <-: Reg[IR[5:3]] <= StoreMem | | |



| Component | Inputs | Outputs | Behavior | RTL Symbols |
|-------------------------|--|---|---|-------------|
| Register | RegWrite[0:0], CLK [0:0], Reset [0:0], RegInput[15:0] | RegOutput 1[15:0] | On the rising edge of the clock (CLK) the register reads the RegWrite port and outputs the new value. If the reset signal is high on the rising clock edge the register wipes its data and outputs 0x0000 instead. | PC |
| Register Block IF/ID | IPC[15:0], IIR[15:0], RegWrite[0:0], CLK [0:0], Reset [0:0] | OPC[15:0], OIR[15:0] | On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out. | IF ID RB |
| Register Block ID/EX | IRegWrite[0:0], IALUSrc[0:0], IALUOP[2:0], IBranch[0:0], IMemWrite[0:0], IMemRead[0:0], IRegStore[0:0], I1stArg[15:0], I2ndArg[15:0], I3rdArg[15:0], IImm[15:0], | ORegWrite[0:0], OALUSrc[0:0], OALUOP[2:0], OBranch[0:0], OMemWrite[0:0], OMemRead[0:0], ORegStore[0:0], O1stArg[15:0], O2ndArg[15:0], O3rdArg[15:0], OImm[15:0], ORs1[15:0], ORs2[15:0], ORd[15:0] | On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out. | ID EX RB |

| | | | | |
|-----------------------------|---|--|---|-----------|
| | IRs1[15:0], IRs2[15:0], IRd[15:0], RegWrite[0:0] CLK [0:0], Reset [0:0] | | | |
| Register Block EX/MEM | IRegWrite[0:0]], IALUSrc[0:0], IALUOP[2:0], IMemWrite[0:0]], IMemRead[0:0]], IRegStore[0:0]], IALUResult[15: :0], I3rdArg[15:0]], IRD[15:0], RegWrite[0:0] CLK [0:0], Reset [0:0] | ORegWrite[0:0]], OALUSrc[0:0], OALUOP[2:0], OMemWrite[0:0]], OMemRead[0:0], ORegStore[0:0]], OALUResult[15: :0], O3rdArg[15:0], ORD[15:0] | On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out. | EX MEM RB |
| Register Block MEM/WB | IRegWrite[0:0]], IRegStore[0:0]], IALUResult[15: :0], IStoreMem[15: :0], IRD[15:0], | ORegWrite[0:0]], ORegStore[0:0]], OALUResult[15: :0], OStoreMem[15: :0], ORD[15:0] | On the rising clock edge (CLK) if RegWrite is on, then it takes in all I inputs and stores them, outputting them for the next cycle on the matching O outputs. When Reset is on, then all registers reset to zero, and blank out. | MEM WB RB |

| | | | | |
|--------------------|---|---|--|--------------------|
| | RegWrite[0:0] , CLK [0:0], Reset [0:0] | | | |
| Register File | Address1[15:0]]; Address2[15:0]]; Address3[15:0]]; LoadAddr[15:0]]; LoadData[15:0]]; RegWrite[0:0] , CLK [0:0], Reset [0:0] | Out1[15:0], Out2[15:0], Out3[15:0], | On the rising edge of the clock (CLK), if RegWrite is on the Register file will take the LoadData input and load it into the register numbered by LoadAddr. On the falling edge of the clock (CLK), the register file reads the registers numbered by the Address 1, 2, and 3 inputs and then outputs them on the Out 1, 2, and 3 outputs. If reset is on then all registers reset to zero. | ID EX RB |
| ALU | 1stInput[15:0] , 2ndInput[15:0] , ALUOp[2:0], CLK[0:0] | ALUres[15:0], Zero[0:0] | Performs some kind of operation on the 2 input data arguments based on the [1:0] ALUOp code. Outputs the result and a [0:0]zero detection. | Main, Simple adder |
| Data Memory | InputAddress [15:0], Data [15:0] MemWrite [0:0], MemRead [0:0], CLK [0:0] | OutputData [15:0] | Holds data in addressable segments as bytes. Has a read and a write control to determine operation. Only outputs data when it is reading | Data, Instruction |
| Instruction Memory | InputAddress [15:0] | OutputData[15:0] | Gets the instruction from memory using PC as the input. Outputs the instruction to IR. | Data, Instruction |

| | | | | |
|-------------------------|--|---|---|--|
| Mux for k, n-bit inputs | Input0[n-1:0],..., InputK[n-1:0]; Select[ceil(k log2):0] | Output[n-1:0] | Uses a selector input to choose between k different inputs to put on the output wire. | |
| Forwarding Unit | Rs1[15:0] Rs2[15:0] Rd[15:0] | Output[0:0] Output[0:0] | Takes input from Rs1 and Rs2 from the ID/EX registers and the Rd from the EX/MEM and MEM/WB register. If any Rd outputs are the same as the Rs1 or Rs2 registers, the value is forwarded from the ALUResult to the ALU input by changing the mux select input. | |
| Control Unit | IR[15:0] | ALUop[2:0] ALUsrc[0:0] RegStore[0:0] RegWrite[0:0] MemRead[0:0] MemWrite[0:0] Branch[0:0] | Takes in the IR and decodes the instruction by setting all the control inputs accordingly to what instruction is being executed. | |
| Immediate Genie | IR[15:0] | Immediate[15:0]] | Reads the instruction and outputs an immediate based on the type of instruction. Will take the bits dependent on the type of instruction for the immediate and sign extend them for immediate type instructions. Will take the bits dependent on the type and shift them left 1 and sign extend for branch and jump instructions. | |

Reg

Naming Conventions

Our naming conventions for components are either a descriptive acronym done in all caps, or a descriptive word/group of words in all caps that represents the component.

Our naming conventions for inputs/outputs is Upper camel case, unless there is a input and output with the same name, in which case the input starts with an extra I, and the output starts with a capital O.

RTL Symbol descriptions:

PC - Program counter

IR - Instruction Register, holds the current instruction bits

1stArg - First register argument (rs1)

2ndArg - Second register argument (rs2)

3rdArg - Third register argument, load register and store data (rd)

ALUResult - Register holding result of our ALU

Zero - Register holding a 1 or 0 if ALUResult is 0

StoreMem - Data to store in rd from the memory

Op - opcode

Zero - is zero register to hold zero output of alu

Reg - Our main addressable register file

MainALU - alu that performs all functions

SimpleAdderALU - alu that is just an adder (for pc relative operations)

DataMem - section of memory that is for addressable and changeable memory

InstructionMem - section of memory that is non addressable and holds instructions

To double check this RTL, we have had each team member go over and ask questions to ensure that no major errors will impede our work.

CHANGES:

We updated all instructions to be written in a more compact and a more legible style. Each instruction has been converted to some set of symbols that reflect the instruction. In addition, the order of all instructions have been converted to RS1 Operation RS2, RD for any command with three registers. Most others take a form that they can be read from left to right, as applicable.

Control Signals

| Control Signal | Description |
|----------------|---|
| ALUop | 3-bit select input that goes into the main ALU to select the operation. |
| ALUsrc | Select input for the mux choosing the B input for the ALU. Selects between the immediate and the 2ndArg register. |
| RegStore | Select input for the mux choosing the load data for the register file. Chooses between ALUResult and StoreMem. |
| RegWrite | The input for the Register File enabling writing to the Register File. |
| MemRead | Input for data memory enabling reading from memory. |
| MemWrite | Input for data memory enabling writing to memory. |
| Branch | Input for AND gate to decide whether we will branch. |

Milestone 3 plan:

Broke up milestone 3 into datapath design and unit testing. Drew and Eli are focusing on datapath, Ziyu and Julian are focusing on units tests, and Yash is going to make state machine diagram for opcodes.

Checklist

Turning in M3

For this milestone, submit the following:

1. An updated design document that includes the following:
 - A complete but uncluttered block diagram of the datapath. Neat, hand-drawn and scanned diagrams are acceptable.
 - An unambiguous English description of each control signal.
 - An updated list of components, based on any parts needed to implement your datapath. Since you now have the datapath design, your updated component list should include the basic specs for your control unit.
 - For each component (aside from control), a brief description of the unit tests to verify the implementation is correct.

- Integration plan for iteratively combining parts into a complete datapath.
 - Tests for each step in your integration plan. Be specific enough so that each stage will be tested thoroughly.
 - Changes made to the RTL descriptions.
 - Be sure you are keeping things you added earlier up to date if you make changes.
2. A partial implementation in your `Implementation` directory:
- Complete implementations of some components, built according to your spec in Milestone 2.
 - Tests for some components, designed according to your spec in this Milestone.
3. Updated design journals. For this milestone, you might include things like:
- A discussion of your strategy for creating tests.
 - If you found any errors while testing, indicate what they were and how you fixed them here.
 - Describe how your choice in architecture affected your datapath design and component specification.
 - You should be keeping a log of how you deal with issues you run into while using Quartus and ModelSim. This is so you can go look at your own notes when you run into a similar problem again.
4. An updated individual itemized log of each member's work for the week and an estimated work time for each item. Each member is responsible for their own log.

Your (updated) design document and (updated) design process journals should be committed to git in the `Design` directory of your team's repository.

Your implementation files should be placed in the `Implementation` directory of your team's repository. You should have most of the individual components completed for this milestone, and should be beginning to unit test as you design integration tests.

Milestone 4 plan:

Drew and Ziyu – Implementing Forwarding unit, branch prediction, branching logic, and immediate genie

Yash – Control

Eli and Julian – Registers (file and pipeline), Alu, and memory

Milestone 4

1. Control Unit Design

The next step in the design process is to specify the control of your processor. The details of this step will depend on the design and complexity of your datapath. You may need several control units of different types to control your datapath easily.

For any simple combinational control units, you should provide a truth table relating the inputs to the outputs. For any FSMs, you should provide a state transition diagram or table.

However you choose to implement the control units, you should also specify how you plan to test their correct implementations. The control unit may be designed using Verilog.

Tip (seriously, read this): Add a reset control to initialize your processor to the appropriate values. (For multicycle, this will be a state node in the state diagram)

You should have all your parts implemented and tested according to your design document. You should also begin executing your integration plan. Good testing is key and will save you countless hours later!

You may modify your register transfer language specification during the control design phase, but not your assembly language or machine language specifications (unless you obtain instructor approval).

Remember to consider how your project will be evaluated. Maintaining good documentation is an important consideration, but creativity at this stage can still influence performance, area, and especially interestingness.

2. Turning in M4

For this milestone, submit the following:

1. An updated design document that includes the following

- ☐ The specifications of the control units, as described above.
- ☐ Descriptions of the tests necessary to verify the correct implementation of your control units.
- ☐ An updated list of the control signals.
- ☐ Changes made to the RTL descriptions.
- ☐ Changes to the integration plan.

2. Partial implementation:

- ☐ Electronic version of your current models for all components (except the control unit) must be available in your team's `Implementation` directory.
- ☐ All the test-benches necessary to implement your unit test plans.

- ☐ Partially executed integration plan: you should have at least some of your parts integrated.
- ☐ Most of the tests for your integration plan.
- 3. An updated design process journal.
- 4. An updated individual itemized log of each member's work for the week and an estimated work time for each item. Each member is responsible for their own log.
- 5. Your team should demo the Memory lab around the time this milestone is due. (But not during milestone meetings.)

Your (updated) design document and (updated) design process journals should be placed in the `Design` directory of your team's repository.

Your implementation files should be placed in the `Implementation` directory of your team's repository.

The names of your design document and design process journal should not change.