CSSE 232 COMP ARCH 1
DESIGN DOCUMENT

TEAM:

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

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:

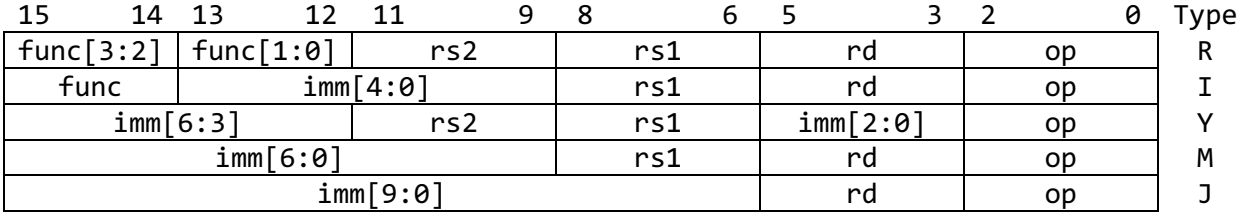
```
relPrime:   sp +_ -8, sp
            ra -> sp+0
            s0 -> sp+2
            s1 -> sp+4
            x0 +_ 2, s0
            x0 +_ 1, s1
            a0 -> sp+6
loop1:      s0 +_ 0, a1
            a0 <- sp+6
            ra \/ gcd
            a0, s1 Y= done
            s0 +_ 1, s0
            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] | 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] << imm
>>_	I	10	001	R[rd] = R[rs1] >> imm
X|_	I	11	001	R[rd] = R[rs1] ^ 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) << 1
Y<	Y		101	If(rs1<rs2) PC += SE(imm) << 1
\/	J		110	R[rd] = PC+2 PC += SE(imm) << 1
/\	J		111	PC = R[rd] R[rd+1] += SE(imm) << 1



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;

}
```

Assembly (a = x6, b = x7):	Machine code:
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];
}
```

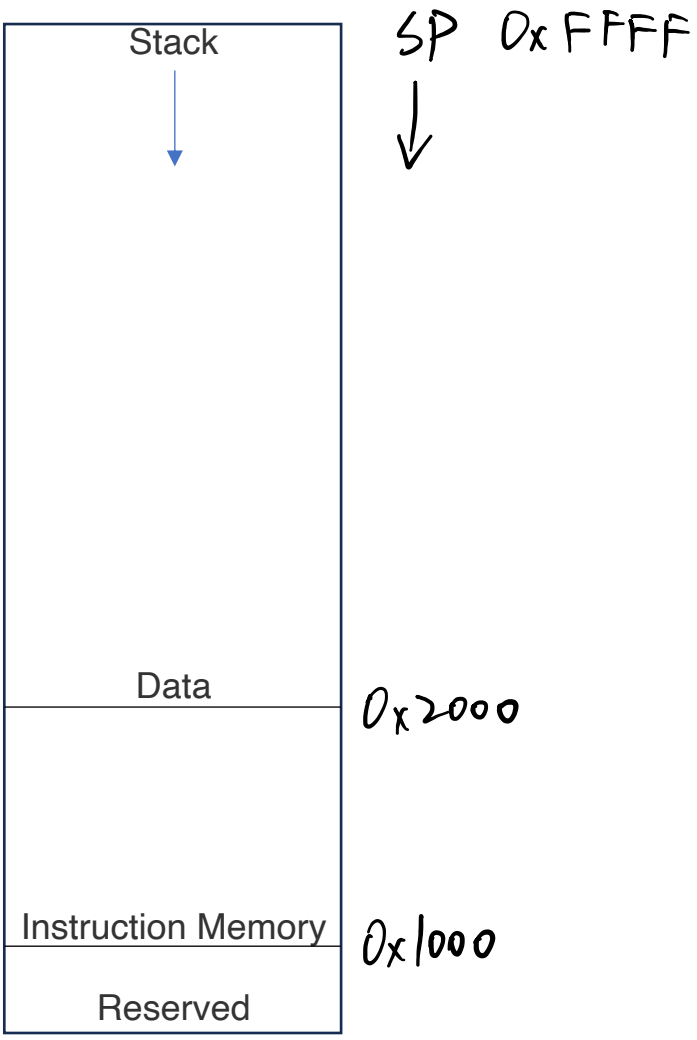
Assembly (address of array a in mem = x4, array length = x5):	Machine code:
x6 + x4, zero	R-Type 00|00 |000|1 00|11 0|000
x7 + x5, zero	R-Type 00|00 |000|1 00|11 1|000
x5, zero Y= 5	Y-Type 0000 |110|1 01|10 1|100
x5 <- x6+0	M-Type 0000 000|1 10|10 1|010
x6 +_ 2, x6	I-Type 00|00 010|1 10|11 0|001
x5 +_ -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<<;
}
```

Assembly(a = x6, b = x7):	Machine code:
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]] ALUResult <= PC + SE(2*immediate)				
Execute	ALUResult <= 1stArg op 2ndArg	ALUResult <= 1stArg op imm	ALUResult <= 1stArg + SE[15:9]	Y=: If(1stArg == 2ndArg) Y<: If(1stArg < 2ndArg) PC <= ALUResult	\/: PC <= ALUResult \/: Reg[IR[5:3]] <= PC /\: Reg[IR[5:3] + 1] <= ALUResult /\: PC <= 3rdArg
Mem			->: DataMem[ALUResult] <= 3rdArg <-: StoreMem <= DataMem[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]	Reg_output 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, IR, 1stArg, 2ndArg, 3rdArg, ALUResult, StoreMem, Op, Zero
Register File	Reg_address1[2:0] Reg_address2[2:0] Reg_address3[2:0] Reg_input_data[15 :0] CLK[0:0] Reg_Read Reg_write	Reg_output 1[15:0] Reg_output 2[15:0] Reg_output 3[15:0]	On the rising edge of the clock (CLK) the register file reads from Reg_Address 1, 2, and 3, outputting them as appropriate if Reg_Read is on. In addition, it writes Reg_input_data to reg_address3 when Reg_Read is on.	Reg
ALU	1stInput[15:0], 2ndInput[15:0], ALUOp[1:0], 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
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

Reg

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.

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.