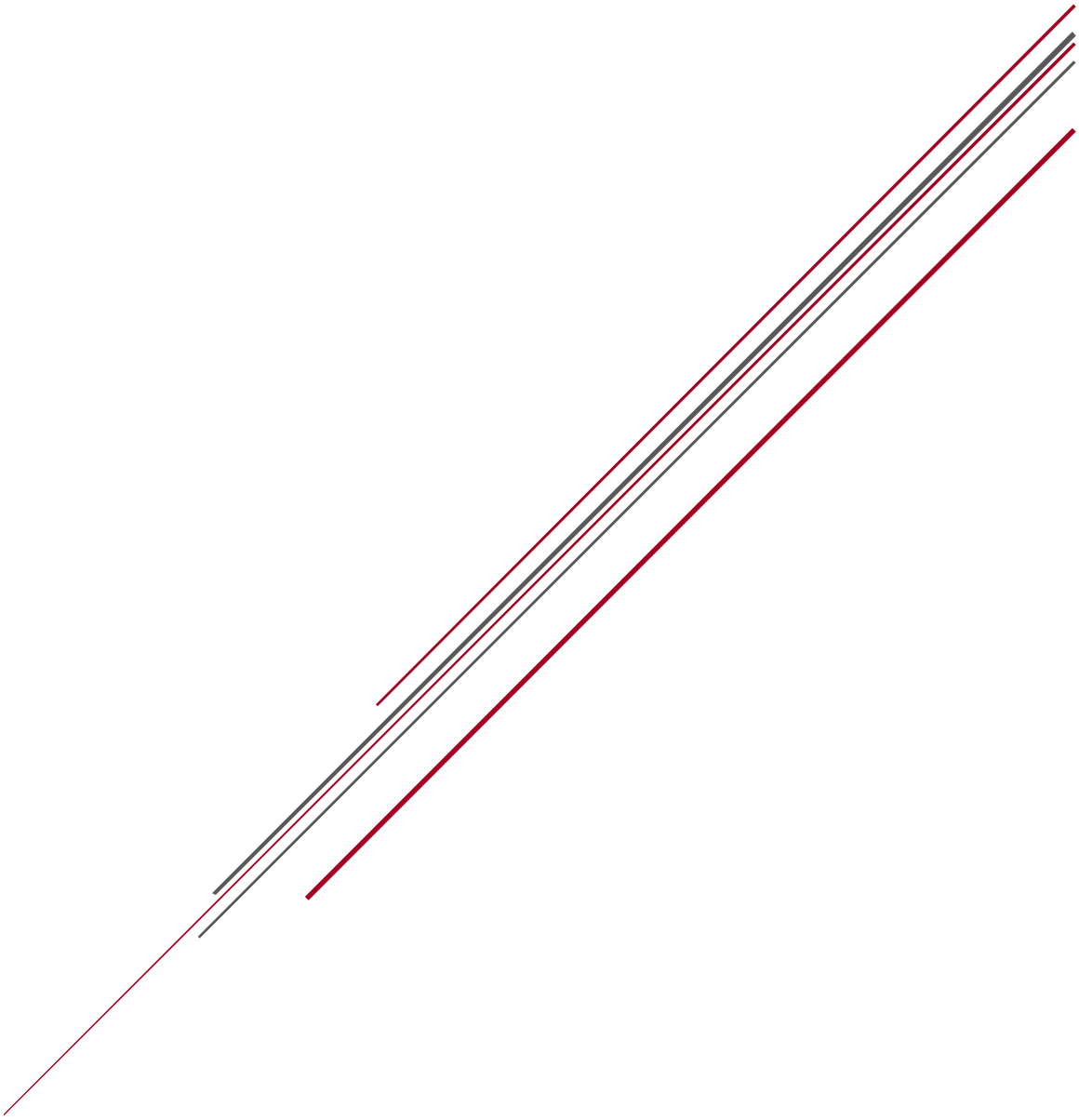
BAEJ Architecture v0.1

Team 3B - Alex Dripchak, Joshua Eckels, Bailey Morgan, Eric Tu



Rose-Hulman Institute of Technology

CSSE232 Computer Architecture – Dr. Micah Taylor

Contents

1. Project Scope

The BAEJ v0.1 processor is a 16-bit, dual-port memory, load-store inspired and accumulator-influenced, multi-cycle architecture. The purpose of the design is to allow for a simple arithmetic and memory operation instruction set. The capabilities of the processor include function calling and returning, recursive calls, basic arithmetic operations, memory caching, data storage and retrieval, and basic register-mapped input and output. The performance of the processor was evaluated based on clock speed, execution time, and the cycles per instruction required to execute a relative prime program with a given input.

2. Design Overview

2.1 Instruction Set

BAEJ architecture supports two instruction types: I-types and G-types. I-types are 2-word instructions, with the first 16-bit word specifying the op-code and register addresses, and the second 16-bit word representing an immediate value to be used with the instruction. G-type instructions are general-purpose, single-word instructions that follow the same format as I-types, but do not require an immediate value.

BAEJ architecture currently supports 15 instructions:

Category	Type	Mnemonic	Instruction
Arithmetic	G	add	add (+)
	G	sub	subtract (-)
	G	and	bitwise and (&)
	G	orr	bitwise or (|)
	G	slt	set on less than
	I	sft	bit shift
	G	cop	copy
Memory operation	I	ldi	load immediate
	I	lda	load address
	I	str	store
Program Control	I	bop	bop (branching)
	I	beq	bop on equal
	I	bne	bop on not equal
	I	cal	function call
	G	ret	function return

Table 2.1.1: BAEJ Instruction Set

See the Instruction section in Appendix A for more detail on the instruction set architecture.

2.2 Implementation

The goal of this section is to clearly identify each hardware component of the processor and provide a brief overview of the datapath. Table 2.2.1 shows a list of all hardware components used in the processor. See the Hardware section in Appendix A for a more comprehensive list.

Component	Quantity	Description
Adder	2	Takes in two 16-bit inputs and outputs the 16-bit addition result
1-bit multiplexer	8	1 selector bit for two 16-bit inputs and one 16-bit output
2-bit multiplexer	1	2 selector bits for four 16-bit inputs and one 16-bit output
ALU	1	Performs all basic arithmetic operations
Memory Unit	1	Dual-port memory 16x1024 (width x depth)
Register File	1	Read and write dual-port register file 16x64 (width x depth)
Register	10	16-bit rising-edge flip-flop
Fcache	1	Memory cache 256x1024 (width x depth)
Comparator	1	Performs equal (=) and not equal (!=) comparisons
Control Unit	1	Implemented as a state machine (combinational logic)

Table 2.2.1: Hardware Components

2.3 Final Datapath

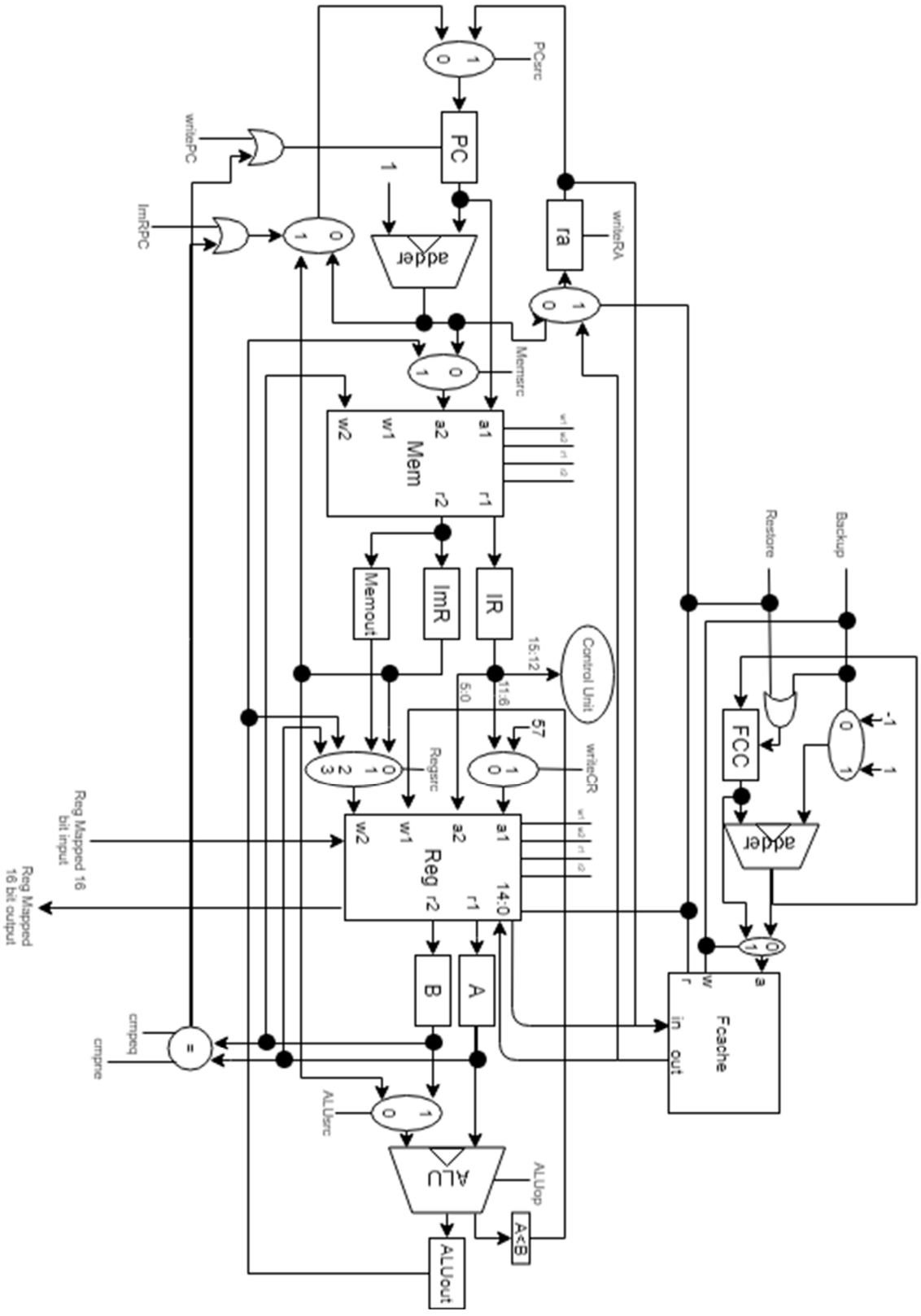
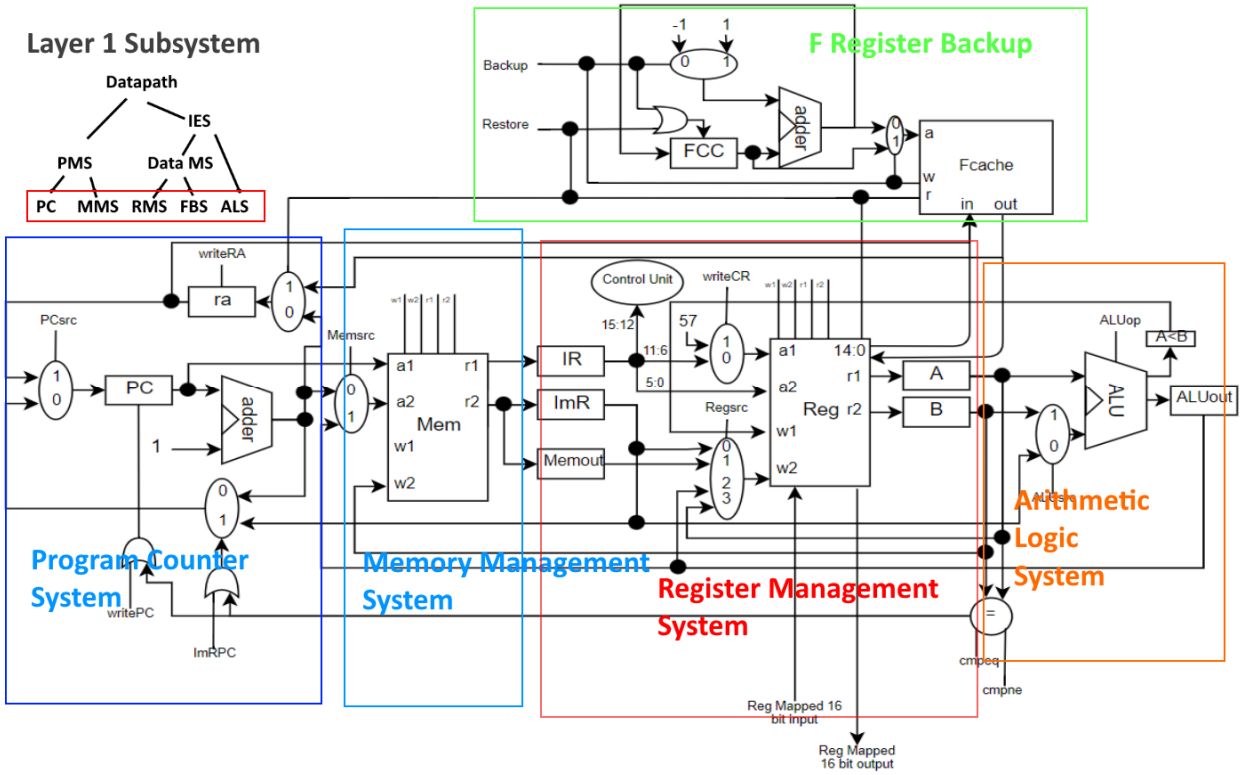


Figure 2.3.1: BAEJ Architecture Datapath

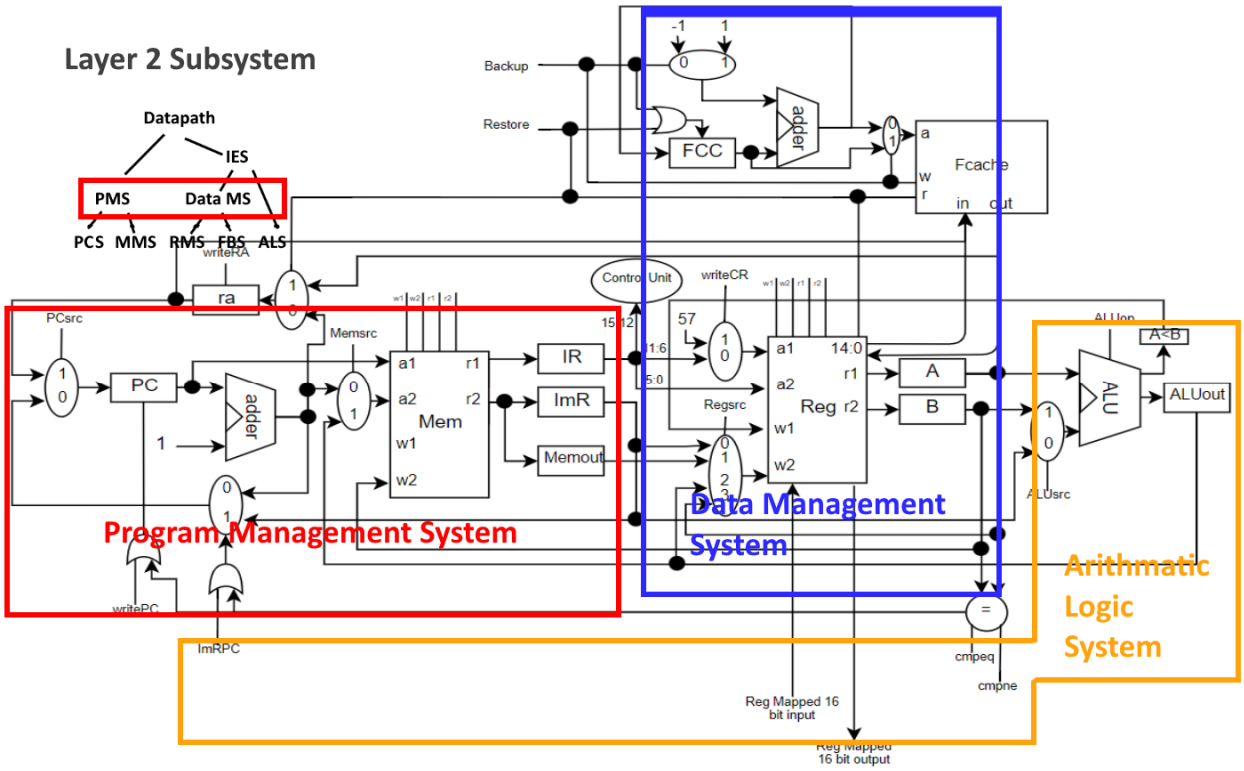
2.4 Integration and Testing

The datapath was assembled in three distinct integration layers, as summarized in the tables and figures below. Each integration subsystem was extensively tested by executing several permutations of inputs and control signals and verifying the correct outputs. See the Testing section of Appendix A for the detailed testing plan of each subsystem.



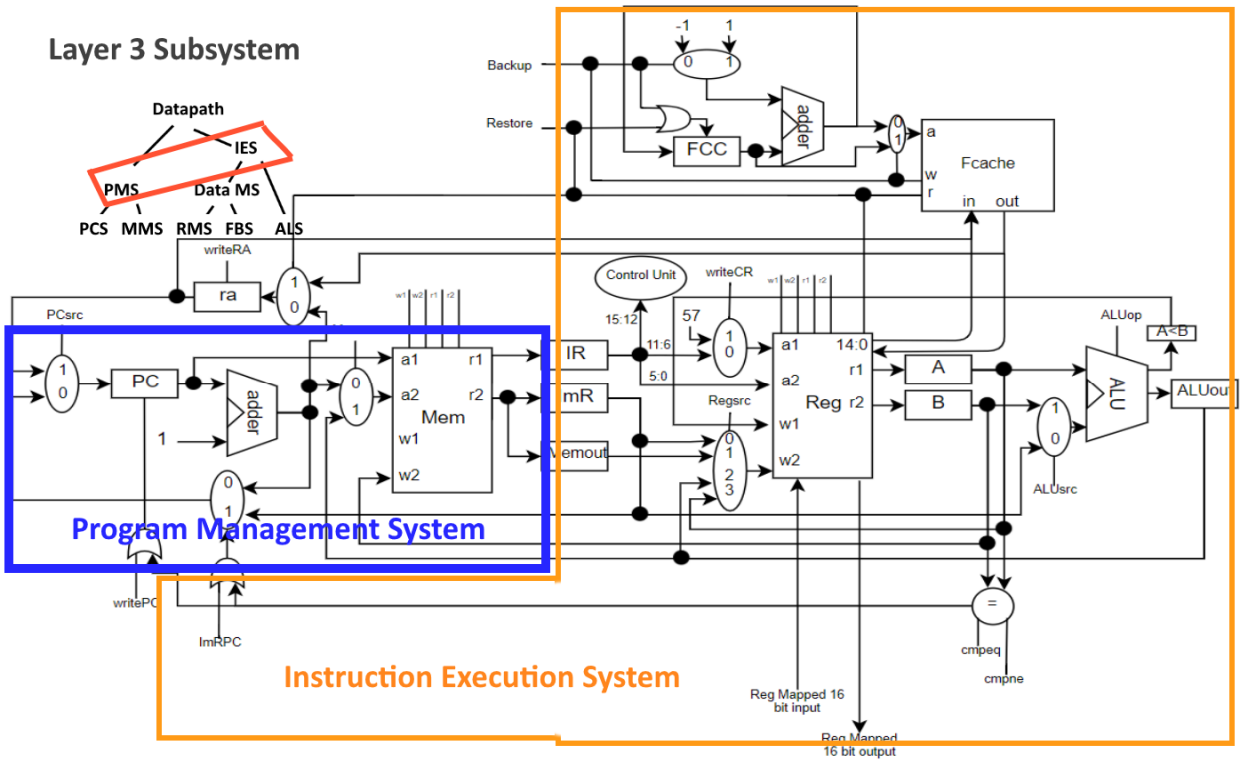
Subsystem	Description
Program Counter System (PCS)	Tracks and increments program counter (PC)
Memory Management System (MMS)	Manages input/output from memory unit
Register Management System (RMS)	Manages input/output from register file
Arithmetic Logic System (ALS)	Performs arithmetic operations
F Register Backup System (FBS)	Manages Fcache backup and restoring

Figure 2.4.1: Layer 1 Sub-systems



Subsystem	Description
Program Management System (PMS)	Obtains values from memory at PC
Data Management System (DMS)	Manages register file backing up and restoring
Arithmetic Logic System (ALS)	SEE LAYER 1

Figure 2.4.2: Layer 2 Sub-systems



Subsystem	Description
Program Management System (PMS)	SEE LAYER 2
Instruction Execution System (IES)	Performs arithmetic operations on values obtained from register file

Figure 2.4.3: Layer 3 Sub-systems

3. Features

3.1 Caching

Conventionally, in a load-store based architecture, the user is required to backup variables and the return address on the stack to preserve them across function calls. The BAEJ architecture eliminates this responsibility from the user, instead having all necessary backup protocols occur automatically within the hardware. Upon function calling, the hardware sends the data in the designated "f registers", along with the return address, up to the memory cache (Fcache), where it is safely stored in a 256-bit word of cache memory. Upon function returning, these values are safely returned to the user in the "f registers" and to the return address register. This backup and restore caching system can support multiple function calls at a time, (up to the designated Fcache memory depth of 1024), without any need of stack memory.

3.2 BAEJ Assembler

The BAEJ assembler included is written in C++ and converts a baej source file to machine code. The assembler makes two passes over the source code. In the first pass, the source is read from the file provided. It looks up all the symbols in the source and maps them into a symbol table. In the second pass, the symbol table is used to convert the source code to machine code which is then written to an output file, baej.out. The current implementation of the assembler outputs the machine code as a text file rather than binary so that it can easily be used with the verilog test bench and the simulator.

3.3 Performance Simulation

The BAEJ simulator is written in C++ as well and it does the job of simulating algorithms implemented in the BAEJ language. It must be given a machine code file, such as the output from

the assembler, and an input to the system. The simulator will then simulate the given program and return the output of the program (corresponding to the given input), as well as performance feedback. Performance metrics returned by the simulator include bytes transferred to and from memory, program size in memory, number of instructions and cycles executed, average CPI, and execution time of the program. On top of providing these performance metrics, the simulator is a means of testing the language itself, as well as programs and algorithms written in the BAEJ language. Where Xilinx test benches test the hardware implementation of the datapath, the BAEJ simulator tests the correctness of the code being ran.

4. Conclusion

The BAEJ architecture provides simple and robust functionality for the execution of algorithms such as relative prime and recursive summation. The architecture was designed for ease of use on the part of the user in its straightforward and intuitive interface. Its versatility is evidenced by its use of caching, its dual-port memory functionality, and its compact instruction set.

The BAEJ architecture comes equipped with an assembler, as well as a simulator for its instruction set. The high performance of the architecture is evidenced by low execution times and correct results provided by the simulator.

On top of its current excellent performance, BAEJ architecture has the potential for further growth and development. Opportunities presented by exploring a larger instruction set, a larger memory size, and increased performance optimizations leave BAEJ with much potential in continued development.

Appendices

Appendix A: Design Documentation

BAEJ

BAEJ is a Reduced Instruction Set Computer Architecture which implements a load store architecture

Registers

Registers	Address	Use
.f0 - .f14	0-14	General purpose “function registers” where data is not lost after a function call
.ip	15	Register file mapped 16 bit input port
.op	16	Register file mapped 16 bit output port
.t0 - .t27	17-44	General purpose “temporary registers” where data may be overwritten during a function call
.a0 - .a5	45-50	Argument registers for function calls
.m0 - .m5	51-56	Accumulator registers on which default mathematical operations are committed
.cr	57	Compiler register (used for slt)
.v0 - .v3	58-61	Return value registers from a function call
.sp	62	Stack pointer register
.z0	63	Register always holding the value 0

Function Registers (.f0 - .f14)

Function registers serve as registers which can be safely used during any function without backing up on a stack. In order to reduce the requirements imposed on the user, backing up and restoring the first 15 registers in our register file (.f0- .f14) happens automatically to an internal memory unit, called the

Fcache, upon function calls and returns (i.e. cal and ret). For the user to make a function call, they need to move all of their values that they expect to be saved into the F registers. Upon return from the function call, the user's values will be safely returned to the F registers via the Fcache.

Instruction Formats

I Types

|15 OPCODE 12|11 RS 6|5 RD 0| (1st word)

|15 IMMEDIATE 0| (2nd word)

> I type instructions use the format above. They are multi-word instructions with the first word consisting of a 4 bit op code followed by two 6 bit register addresses. The second word will be the 16 bit immediate value used in the instruction.

G Types

|15 OPCODE 12|11 RS 6|5 RD 0| (1st word)

> G type instructions use the same format as I type as described above. They do not, however, have an immediate, and only have one word in their machine code format.

Instructions

**IM stands for immediate*

**Optional argument of .rm specifies an accumulator register to operate on (defaults to .m0)*

Instr	Type	OP	Usage	Description	Rtl
`lda`	I	`0000`	`lda .rs[IM] .rd`	Loads a value from memory to rd	`rd = Mem[rs+IM]`
`ldi`	I	`0001`	`ldi .rd IM`	Loads an immediate to rd	`rd = IM`
`str`	I	`0010`	`str .rs[IM] .rd`	Stores value in rd to memory	`Mem[rs+IM] = rd`
`bop`	I	`0011`	`bop IM`	Changes pc to immediate	`pc = IM`

Instr	Type	OP	Usage	Description	Rtl
cal	I	0100	cal IM	Changes pc to immediate and sets a return address; backs up f registers (.f0-.f14)	ra = pc+2pc = IMFCC += 1
beq	I	0101	beq .rs .rd IM	Changes pc to immediate if rs and rd are equal	if rs==rd pc=IM;
bne	I	0110	bne .rs .rd IM	Changes pc to immediate if rs and rd aren't equal	if rs!=rd pc = IM;
sft	I	0111	sft .rs .rd IM	Shifts value in rs to rd by immediate. Positive shifts left, negative shifts right	rd=rs<<IM
cop	G	1000	cop .rs .rd	Copies the value of rs to rd while retaining the original value of rs	rd = rs
slt	G	1010	slt .rs .rd	Sets cr to 1 if rs is less than rd; 0 otherwise	cr=rs<rd?1:0
ret	G	1011	ret	Sets pc to the value in ra; restores f registers (.f0-.f14)	pc=raFCC-=1
add	G	1100	add .rs [.rm]	Adds rs into the accumulator*	[rm] += rs
sub	G	1101	sub .rs [.rm]	Subtracts rs from the accumulator*	[rm] -= rs
and	G	1110	and .rs [.rm]	Ands rs with the accumulator*	[rm] &= rs

Instr	Type	OP	Usage	Description	Rtl
`orr`	G	1111	`orr .rs [.rm]`	Ors rs with the accumulator*	`[rm] |= rs`

Function Calls

When calling a function, the programmer places the arguments in registers .a0 - .a5 and uses the command cal (i.e. `cal myFunc`). The instruction will jump the program counter to the address of the function while also putting the incremented previous program counter value into the return address register (.ra). The function will then return (ret) to the address in the ra register. The programmer can expect their data in f registers to be retained; they should not expect data in any other register to be retained. After a function returns, returned values will be in the v registers. When writing a function, the user is responsible for returning from the function using `ret`.

Examples

Common Assembly/Machine Language Fragments

Loading an address into a register (.f1)

BAEJ Code

```
ldi .f0 addr
lda .f0[0] .f1
```

Machine Code Translation (assuming the value stored in addr is 280)

```
0x00            0001 000000 000000
0x01            0000 000100 011000
0x02            0000 000000 000001
0x03            0000 000000 000000
```

Sum Values from x (.a0) to y (.a1) assuming x < y

BAEJ Code

```
        cop .a0 .m0
        cop .a0 .m1
        ldi .f0 1
loop:   add .f0 .m1
        add .m1
        slt .m1 .a1
        bne .z0 .cr loop
```

Machine Code Translation (Assuming the address of loop is 0x8)

```
0x00            1000 101101 110011
0x01            1000 101101 110100
0x02            0001 000000 000000
0x03            0000 000000 000001
0x04    loop:   1100 000000 110100
0x05            1100 110100 110011
0x06            1010 110100 101110
0x07            0110 111111 111001
0x08            0000 000000 000100
```

Modulus

BAEJ Code

```
loop:   add .a1
        slt .a0 .m0
        bne .z0 .cr loop
        sub .a1
        cop .a0 .m1
        sub .m0 .m1
```

Machine Language Translation (Assuming the address of loop is at 0x00)

```
0x00    loop:   1100 101110 110011
0x01            1010 101101 110011
0x02            0110 111111 111001
0x03            0000 000000 000000
0x04            1101 101110 110011
```

```
0x05            1000 101101 110011
0x06            1101 110011 110100
```

Euclid's Algorithm

C Code

```c
// Find m that is relatively prime to n.
int
relPrime(int n)
{
   int m;

   m = 2;

   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }

   return m;
}

// The following method determines the Greatest Common Divisor of a and b
// using Euclid's algorithm.
int
gcd(int a, int b)
{
  if (a == 0) {
    return b;
  }

  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }
```

```
  return a;
}
```

BAEJ and Machine Code Translation

```
relP:   0x00    0001 000000 010000  ldi .op 2
        0x01    0000 000000 000010
        0x02    0001 000000 000001  ldi .f1 1
        0x03    0000 000000 000001
loop:   0x04    1000 001111 101101  cop .ip .a0
        0x05    1000 010000 101110  cop .op .a1
        0x06    0100 000000 000000  cal gcd
        0x07    0000 000000 001111
        0x08    0101 111010 000001  beq .v0 .f1 end
        0x09    0000 000000 001101
        0x0A    1100 000001 010000  add .f1 .op
        0x0B    0011 000000 000000  bop loop
        0x0C    0000 000000 000100
end:    0x0D    0011 000000 000000  bop 32
        0x0E    0000 000000 100000
gcd:    0x0F    0110 101101 111111  bne .a0 .z0 cont
        0x10    0000 000000 010011
        0x11    1000 101110 111010  cop .a1 .v0
        0x12    1011 000000 000000  ret
cont:   0x13    0101 101110 101101  beq .a1 .a0 done
        0x14    0000 000000 011110
        0x15    1010 101110 101101  slt .a1 .a0
        0x16    0101 111001 111111  beq .cr .z0 else
        0x17    0000 000000 011011
        0x18    1101 101110 101101  sub .a1 .a0
        0x19    0011 000000 000000  bop cont
        0x1A    0000 000000 010011
else:   0x1B    1101 101101 101110  sub .a0 .a1
        0x1C    0011 000000 000000  bop cont
        0x1D    0000 000000 010011
done:   0x1E    1000 101101 111010  cop .a0 .v0
        0x1F    1011 000000 000000  ret
```

RTL

I Types

`lda`	`str`	`ldi`	`beq/bne`	`sft`	`bop`	`cal`
IR = Mem[PC] ImR = Mem[Wire(PC+1)] PC += 1						
PC += 1 A = Reg[IR[11:6]] B = Reg[IR[5:0]]					PC = ImR	ra = PC + 1 PC = ImR Fcache[FCC] = {RA, Reg[14:0]} FCC += 1
ALUout = A + ImR		Reg[IR[5:0]] = ImR	if(A==B) PC = ImR	ALUout = A << ImR	Cycle Delay	
Memout = Mem[ALUout]	Mem[ALUout] = B			Reg[IR[5:0]] = ALUout		
Reg[IR[5:0]] = Memout						

G Types

`cop`	`slt`	`Other G Types`	`ret`
IR = Mem[PC] PC += 1			
A = Reg[IR[11:6]] B = Reg[IR[5:0]]			PC = ra FCC -= 1 Reg[14:0]=Fcache[FCC][239:0] RA=Fcache[FCC][255:240]
Reg[IR[5:0]] = A	AlessThanB = A < B ? 1 : 0	ALUout = A op B	
	cr = ALessThanB	Reg[IR[5:0]] = ALUout	

Testing the RTL

Code Tracing

In order to verify the RTL, a code tracing exercise was used. The RTL was verified by tracing the logic and values through the RTL as if a Java program were executing it. If it gave unexpected results, the RTL was modified accordingly.

Simulation

Java will be used to simulate the RTL, such that variables will represent registers, a Java Map will represent the register and storage files (with addresses mapped to their respective values), and multiplication will represent bit shifting. For example, an implementation of lda, according to our RTL, will be implemented using the following Java code.

```
int IR;
int IMR;
int PC;
int ALUout;
int Memout;
int A;
int B;

HashMap<Integer, Integer> Mem;
HashMap<Integer, Integer> Reg;

public void lda () {
    IR = Mem.get(PC);
    IMR = Mem.get(PC + 1);
    PC += 1;
    // ----------------------
    PC += 1;
    A = Reg.get(IR >> 6 & 0b00111111); // IR[11:6];
    B = Reg.get(IR >> 0 & 0b00111111); // IR[5:0];
    // ----------------------
    ALUout = A + IMR;
    // ----------------------
    Memout = Mem.get(ALUout);
    Reg.put(Memout, Reg.get(IR & 0b00111111)); // IR[5:0];
}
```

A similar implementation was done for other instructions.

Hardware Components

Adder

Items	Descriptions
Inputs	A[15:0], B[15:0]
Outputs	R[15:0]
Control Signals	None
Functionality	Outputs A+B onto R
Hardware Implementation	In Verilog, assign A+B to the output R
Unit Tests	Input all permutations of two integers from -20 to 20 and verifies the output is correct

Single bit Multiplexer

Items	Descriptions
Inputs	A[15:0], B[15:0]
Outputs	R[15:0]
Control Signals	S
Functionality	Will constantly put the value of A or B specified by the S control bit onto R
Hardware Implementation	Verilog switch case, which assigns A to R given a low signal for S, otherwise, assigns B to R
Unit Tests	Put all permutations of -10 to 10 on A and B and attempts to select A then B while testing that the correct output is on R

Two bit Multiplexer

Items	Descriptions
Inputs	A[15:0], B[15:0], C[15:0], D[15:0]
Outputs	R[15:0]

Items	Descriptions
Control Signals	S[1:0]
Functionality	Will constantly put the value of A, B, C, D specified by the S control bit onto R
Hardware Implementation	Verilog switch case, which given 0, 1, 2, or 3, assigns A, B, C, or D to R respectively.
Unit Tests	Put all permutations of 4 numbers each from -10 to 10 on A, B, C, and D and attempts to select A, then B, then C, and finally D while testing that the correct output is on R

ALU

Items	Descriptions	ALU op	Operation
Inputs	A[15:0], B[15:0]	000	AND
Outputs	A<B (AltB), R[15:0]	001	OR
Control Signals	ALUop[2:0]	010	ADD
Functionality	Takes the mathematical operation specified by Operation and preforms in on operand A and B, puts result on A<B or R depending on operation	011	SUBTRACT
Hardware Implementation	Verilog switch case that assigns the result of the appropriate operation on A and B to R based off of the op code	100	SHIFT
Unit Tests	A loop in Verilog for each op code which inputs all permutations of two inputs from -10 to 10 and verifies with the output that the operation was performed correctly on the inputs	101	SET LESS THAN

Comparator

Items	Descriptions
Inputs	A[15:0], B[15:0]

Items	Descriptions
Outputs	R
Control Signals	cmpeq, cmpne
Functionality	Whenever the cmpeq signal is high, outputs a 1 on R if A == B, when cmpne is high, outputs a 1 on R if A != B, otherwise a 0 is output on R.
Hardware Implementation	Verilog module which assigns A==B if cmpeq is high, A!=B if cmpne is high, otherwise 0 to R
Unit Tests	A loop in verilog which inputs a range of 0-32 on to A and B and checks that the output for each control signal is correct

Fcache

Items	Descriptions
Inputs	wData[255:0], addr[15:0], clk
Outputs	rData[255:0]
Control Signals	write
Functionality	When the Write signal is high, takes the value on wData and stores it in address addr. The Fcache always puts the value at addr on rData.
Hardware Implementation	Static storage implemented using a register-file like structure. Use the verilog register file provided on the course website, altering it to 256 bit words. In verilog, write a module which wraps the register file to allow for a bus serving as both input and output.
Unit Tests	A loop in verilog which goes through a large range of addresses and writes many different 256 bit values while reading them each iteration to ensure they are correct.

Register File

Items	Descriptions
Inputs	a1[15:0], a2[15:0], w1[15:0], w2[15:0], fcIn[239:0], clk, ioIn[15:0]
Outputs	r1[15:0], r2[15:0], fcOut[239:0], ioOut[15:0]
Control Signals	RegW1, RegW2, RegR1, RegR2, restore
Functionality	With a write control signal high (RegW1 or RegW2), takes the respective value (w1 or w2) and stores it in the register specified by the respective address (a1 or a2). With a read signal high (RegR1 or RegR2), takes the value at the respective address and puts it onto the respective output (r1 or r2). The register file always puts the values in registers 0 to 14 on fcOut. When restore is high, stores the values on fCin into registers 0 to 14.
Hardware Implementation	Static storage implemented using a series of 64 registers. Use the verilog register file provided on the course website and alter as needed to enable dual port functionality (Multiple inputs and outputs).
Unit Tests	A loop in verilog which goes through a large range of addresses and writes many 16 bit values while reading them each iteration to ensure they are correct.

Memory Unit

Items	Descriptions
Inputs	A1[15:0], A2[15:0], W1[15:0], W2[15:0]
Outputs	R1[15:0], R2[15:0]
Control Signals	MemW1, MemW2, MemR1, MemR2
Functionality	With a write signal high (MemW1 or MemW2), takes the respective value (W1 or W2) and stores it in the respective address (A1 or A2). With a Read signal high (MemR1 or MemR2), takes the value at the respective address and puts it onto the respective output (R1 or R2).
Hardware Implementation	Implemented in verilog using the Memory unit provided on the course website, altering it as needed to enable dual port functionality (Multiple inputs and outputs).

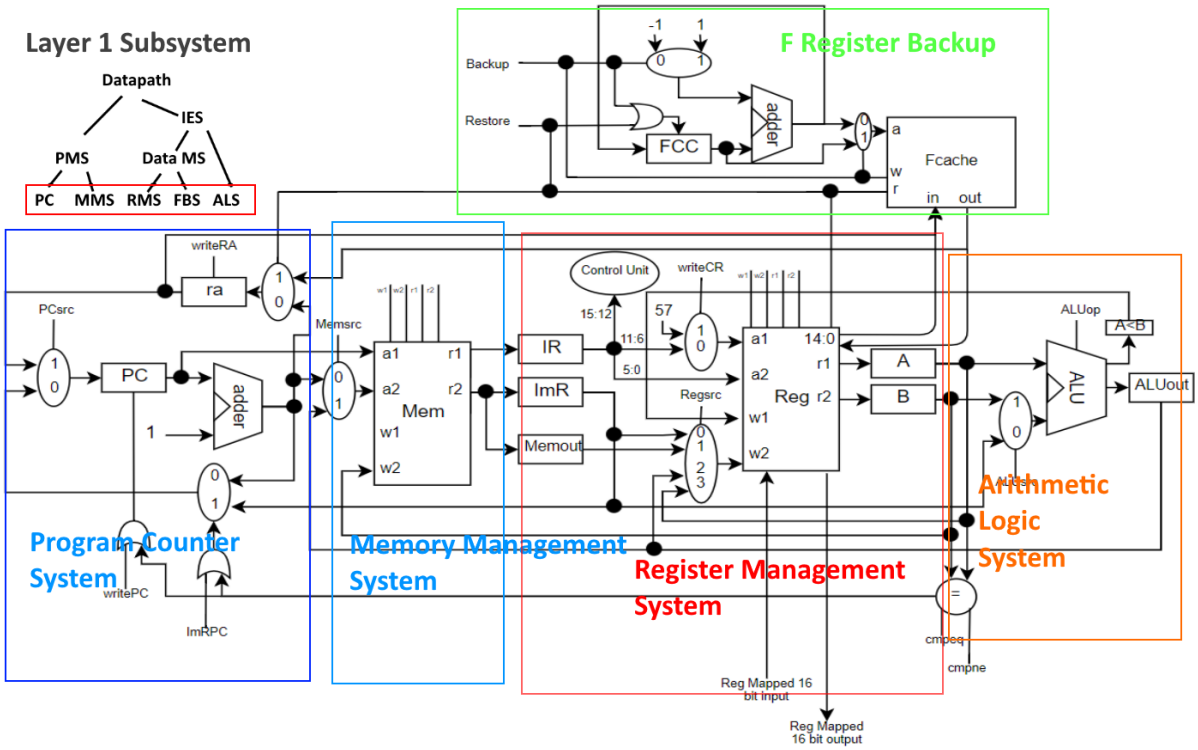
Items	Descriptions
Unit Tests	A loop in verilog which goes through a large range of addresses and writes many 16 bit values while reading them each iteration to ensure they are correct.

Integrating and Testing the Components

Integration Plan

Subsystem	Composition
Program Counting System	Register (x2), Single bit Multiplexer (x2), Adder, Or-gate
Memory Management System	Memory Unit, Single bit Multiplexer, Register (x3)
Register Management System	Register File, Single bit Multiplexer, Two bit Multiplexer, Register (x2)
Fcache Backup System	Fcache, Single bit Multiplexer (x2), Adder, Register, Or-gate
Arithmetic and Logic System	ALU, Single bit Multiplexer, Register (x2)
Program Management System	Program Counting System, Memory Management System
Data Management System	Register Management System, Fcache Backup System
Instruction Execution System	Data Management System, Arithmetic and Logic System
Datapath	Program Management System, Instruction Execution System

Datapath Block Diagram with Subsystems



Test Plans

Subsystem	Test Plan
Program Counting System (PCS)	Run the system through a few clock cycles to test that it correctly increments by one each time. Also ensure that we can write pc + 1 to ra. Once this is verified, inject addresses from a set of addresses, and from register ra, to test branching functionality.
Memory Management System (MMS)	Input values into a sequential block of memory then read from the same block, verifying that each read gives the output registers the correct values that were written.

Subsystem	Test Plan
Register Management System (RMS)	Input values into registers from all permutations of the input ports, then read from registers with known values verifying that each read gives the output registers the correct values. Input values on fcIn and check restore/backup functionality. Input values on ioIn and check input/output functionality. Compare register values with comparator and verify result.
Fcache Backup System (FBS)	Conduct multiple backups of known values to a sequential block in the Fcache memory, then using multiple restores, read back the same block verifying the output is what was written.
Arithmetic and Logic System (ALS)	Conduct all possible ALU operations on a wide range of input values using all possible input methods (i.e. different ALUsrc signals to the multiplexer). Test each operation for correct output values.
Program Management System (PMS)	Hard-code values into a sequential block of memory then allow the program counter to increment through memory and verify that the correct values which were written to memory are written to the output registers.
Data Management System (DMS)	Repeatedly write values to registers 0 - 14 using many permutations of input methods. Each time all 15 registers are filled, send a backup control signal. Do this many times then conduct the same number of restores, ensuring values are correct along the way.
Instruction Execution System (IES)	Give this system the control signals needed for basic instructions which don't require memory such as arithmetic operations and moving values around in the register file. Include many different input values with each set of control signals and verify expected output.
Datapath	SEE SYSTEM TESTING

System Testing

The following algorithms will be coded into memory.

	Test Algorithm	Expected Result
add (int a)	64 + a	
sub (int a)	64 - a	
and (int a)	64 & a	
orr (int a)	64 or a	
slt (int a)	(a < 64) ? 1:0	
sft (int a)	a << 2 ; a >> 2	
summation (int a)	Sum of all integers between 0 and a	
memory (int a)	“a” stored in memory and then retrieved to output	
relativePrime (int a)	The first relative prime number of a	

Control

Control Unit

Items	Descriptions
Inputs	op[3:0], clk
Outputs	B[23:0] (22 unique control signals, 24 bits in all)
Control Signals	Reset
Functionality	Given an op-code (or address), the unit outputs the necessary control signals to the corresponding instruction and state
Hardware Implementation	Implemented as a state machine in Verilog that sets the current state and control signals depending on the op code
Unit Tests	A loop in Verilog which sets every permutation of the 4-bit op-code then verifies the expected output control signals.

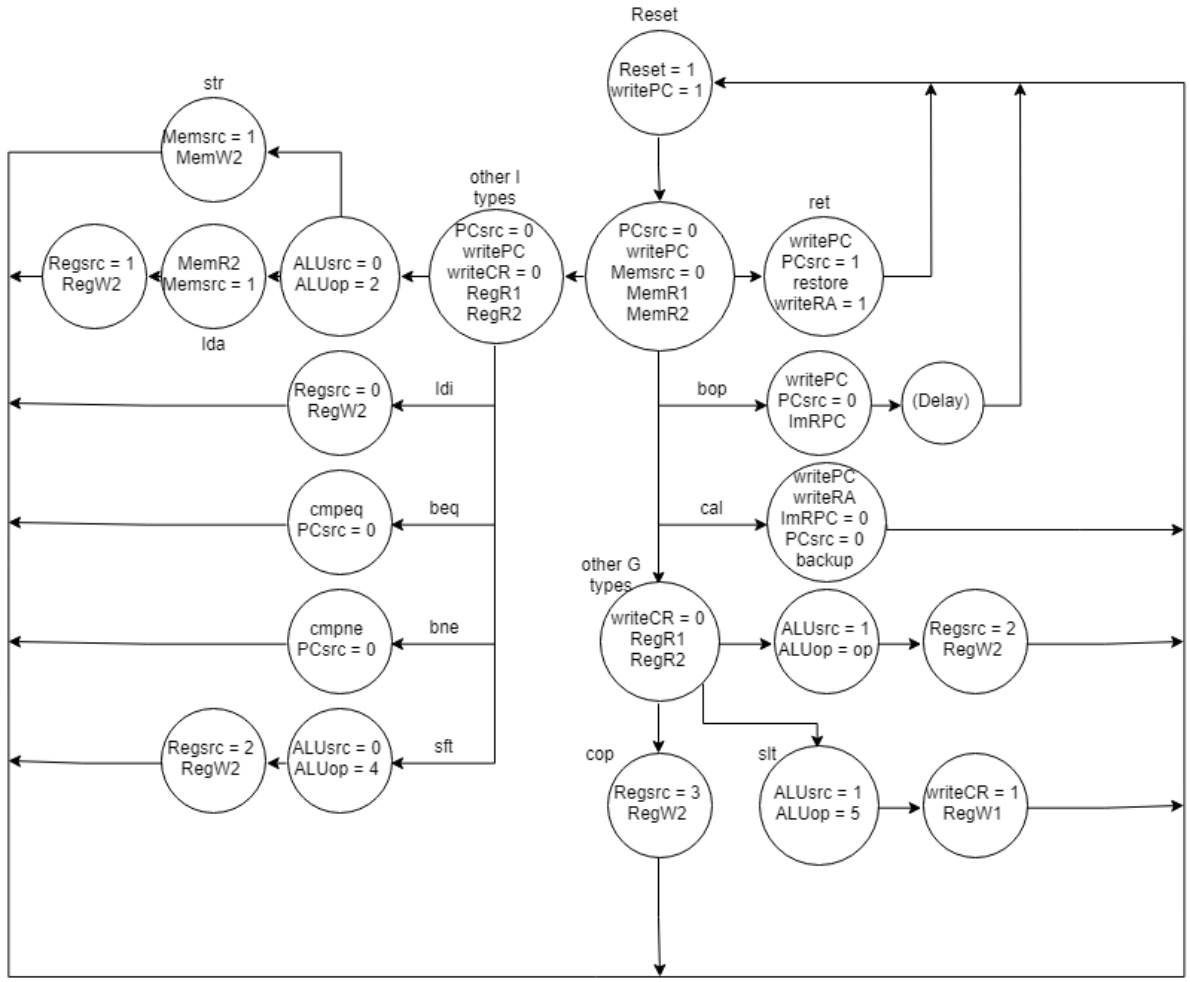
Control Signals

Signal Name	Bits	Effect when deasserted (0)	Effect when asserted (1)
PCsrc	1	PC is set to default value (PC+1) or ImR	PC is set to the value of RA
writePC	1	Nothing	PC gets the value chosen by PCsrc mux
writeRA	1	Nothing	RA gets the value of PC + 1
ImRPC	1	ImRPC mux chooses PC+1	ImRPC mux chooses immediate value (only when comparator is enabled and outputs a high signal)
Memsrc	1	Address 1 in Mem is pulled from PC + 1	Address 1 in Mem is pulled from ALUout
MemW1	1	Nothing	The value at port w1 is written to the address specified by a1
MemW2	1	Nothing	The value at port w2 is written to the address specified by a2
MemR1	1	Nothing	The value at the address specified by a1 is read to port r1
MemR2	1	Nothing	The value at the address specified by a2 is read to port r2
writeCR	1	The reg number specified at reg file port a1 is IR[11:6] (default)	The reg number specified at reg file port a1 is 57 (for compiler register)
Regsrc	2	0: Value at reg file port w2 comes from ImR; 1: Value at port w2 comes from Memout	2: Value at port w2 comes from ALUout; 3: Value at port w2 comes from reg A

Signal Name	Bits	Effect when deasserted (0)	Effect when asserted (1)
backup	1	Nothing	Registers 14:0 (240 bits) from the reg file and RA (16 bits) are written to the Fcache at the address specified by “a”; FCC is incremented by 1
restore	1	Nothing	The 256 bit value at the address specified by “a” in the Fcache is written to registers 14:0 in the reg file and RA; FCC is decremented by 1
RegW1	1	Nothing	The value at port w1 is written to the reg address specified by a1
RegW2	1	Nothing	The value at port w2 is written to the reg address specified by a2
RegR1	1	Nothing	The value at the reg address specified by a1 is read to port r1
RegR2	1	Nothing	The value at the reg address specified by a2 is read to port r2
ALUsrc	1	2nd ALU operand comes from ImR	2nd ALU operand comes from reg B
ALUop	3	SEE ALU IN COMPONENTS	SEE ALU IN COMPONENTS

Signal Name	Bits	Effect when deasserted (0)	Effect when asserted (1)
cmpeq	1	Nothing	The result of the comparison A=B is sent to the ImRPC mux
cmpne	1	Nothing	The result of the comparison A!=B is sent to the ImRPC mux

FSM Diagram



Appendix B: Design Journals

Team Design Journal

TEAM JOURNAL - CSSE232-03

MEMBERS: Alex Dripchak, Joshua Eckels, Bailey Morgan, Eric Tu

Milestone 1

MEETING 1.0 - 01/07/2019 - 12:40 pm [60 min]

Members Present: Alex, Josh, Bailey

(Eric not present due to miscommunication on members present's parts)

We decided to use a load/store architecture to build our processor. Part of the reason was it would be the easiest architecture to use in order to have 16-bit immediates. We could also use an accumulator style for math instructions using a dedicated register.

MEETING 1.1 - 01/08/2019 - 2:30 pm [50 min]

All members present & on time

Discussed how to construct instructions (ie how to divide bits to opcodes, regs, etc), registers. Decided that we would use more registers in favor of speed as opposed to using the stack. To account for this, we would have 128 function registers that would be represented by the first 16 "f" regs that could be all accessed by using (FCC * 16) + regs.

MEETING 1.2 - 01/08/2019 - 8:30 pm [90 min]

All members present

Began working on translating Euclid's Algorithm, relPrime, and other small algorithms to demonstrate operations. Also translated our instructions to machine code.

MEETING 1.3 - 01/09/2019 - 1:30 pm [120 min]

All members present

Finished Euclid's algorithm and relPrime. Translated the assembly language snippets to machine code. Polished the document. Completed Milestone 1. Pushed everything to git log in Design branch.

Milestone 2

MEETING 2.0 - 01/14/2019 - 5:10 pm [60 min]

Members Present: Josh, Bailey, Eric

Put the skeleton of the RTL in the table and mapped out what kind of registers we need for our RTL and what they do

MEETING 2.1 - 01/15/2019 - 5:30 pm [90 min]

All members present

Finished the RTL write-up for all instructions. Polished them.

Recorded a list of components that will be needed for the data path.

NOTE: REMEMBER TO REMOVE THE "MOV" INSTR!!!

MEETING 2.2 - 01/16/2019 - 1:30pm [120 min]

All members present

Polished up and translate RTL over to our design document. Developed shopping list for all hardware in our RTL.

Started describing our components. Added a comparator to the list, ALU now has 3 bit opcodes as we had 5 operations and needed 3 control wires to accomodate this.

Developed a "code-tracing thought experiment" in order to test our RTL, easier and quicker to develop than a simluation and less reliant on others than a survery from other groups.

Milestone 3

MEETING 3.1 - 01/20/2019 - 2:30pm [150 min]

All members present (Alex facetime)

Completed rough draft of datapath after walking through RTL for each instruction. Made changes to RTL based on our datapath.

We are still in need of documentation for each component and subsystem. Worked on defining and documenting control signals.

Will use a ROM for control unit. Meeting tomorrow to move a lot of the work from today into the design document.

MEETING 3.2 - 01/21/2019 - 5:10pm [120 min]:

All members present

Defining and describing every control signal. Working on ways to perform unit and integration testing on

our datapath. Started looking at Verilog and how to implement some components. Working on documention on Design document.

MEETING 3.3 - 01/22/2019 - 7:00pm [310 min]:

Updated datapath to reflect control unit, subsystems, and integration plan. Wrote/planned all unit and integrations tests. Wrote verilog modules and test benches for comparator, adder, and mux.

Updated list of hardware components, and reformatted into tables. Updated RTL, including plan for a simulation to verify RTL. Updated design document.

Milestone 4

MEETING 4.1 - 01/27/2019 - 6:00 pm [180 min]:

Members present: Bailey, Alex, Eric

Completed the finite state machine for the datapath control signals. Described different

testing methods for the finite state machine.

MEETING 4.2 - 01/28/2019 - 6:00 pm [300 min]:

All members present

Implemented most of the components in VHDL, except for memory, which still has some

reading problems. Planned to meet tomorrow to write up integration plans.

MEETING 4.3 - 01/30/2019 - 3:00 pm [300 min]:

All members present

Finished implementing register file and memory unit, along with their unit tests. Implemented some integration between the components

and finalized the multicycle FSM.

Milestone 5

MEETING 5.1 - 02/03/1029 - 1:00 pm [90 min]:

All members present

Formulated and discussed our plans for the week. Discussed a few issues with the data path that needed to be fixed and what solutions we

wanted to go with. Divided integration work up and assign parts to everyone to have done by Wednesday. Alex whooped Eric in smash.

MEETING 5.2 - 02/04/2019 - 10:50 pm [50 min]

Members present: Josh, Alex

Created base code for the control unit.

MEETING 5.3 - 02/05/2019 - 6:00 pm [150 min]:

Members present: Josh (60 min), Alex, Bailey, Eric (late)

Worked on integrating our systems. Control unit finished along with PMS and ALS. Still need FBS and RMS. Updated the design document

to reflect changes. Base code for the F Register Backup System has been created.

MEETING 5.4 - 02/06/2019 - 7:00 pm [60 min]:

All members present.

F Register Backup System has been tested. Realized there were some clock timing issues

that had to be noted whene using the system.

ALU System has been successfully implemented.

MEETING 5.5 - 02/07/2019 - 10:00 pm [180 minutes]

All members present

ALU System has been tested.

The Register Management System has been implemented. Fixed some bugs that

were in the original control unit and updated the design document accordingly

MEETING 5.6 - 02/08/2019 - 7:00 pm [180 minutes]

All members present

The Instruction Execution System and datapath has been implemented. The Instruction

Execution System is also tested.

MEETING 5.7 - 02/10/2019 - 4:00 pm [240 minutes]

All members present

The datapath has been tested and the design document has been updated according to the changes from the current milestone. Updates to the team journal and individual journals were made.

Milestone 6

MEETING 6.1 - 02/15/2019 - 11:00 am [60 minutes]

All members present

Demonstration day. Signed off on completed processor. Started work on presentation and final documentation

MEETING 6.2 - 02/16/2019 - 8:00 pm [180 minutes]

All members present

Finished and practiced presentation. Final documentation complete.

Alex Dripchak

Work log = Alex Dripchak

Milestone 1

01/07/2019 - Met with team to start M1, Started team journal [50 mins]

01/08/2019 - Met with team, discussed instructions, registers, wrote in team journal, helped develop small algorithms, exposed + fixed flaws in current model [3.5 hours]

Listened to Josh complain about git [30 mins]

01/09/2019 - Met to finish up milestone 1, others finished relPrime and Euclids, I worked on converting our code snippets to machine code. [2 hours]

Milestone 2

1/15/2019 - Met with the lads to finish up the RTL. Helped develop new components for the data path. [1.5 hours]

1/16/2019 - Helped write up the components, specifically the comparator and the F cache. Designed a test to use in order to make sure the RTL is correct, then exectured said test to look for errors. [2 hours]

Milestone 3

1/20/2019 - Helped make the datapath [2.5 hours]

1/21/2019 - Started working on Verilog components, wrote test bench for comparator [2 hours]

1/22/2019 - Went over document with team, developed components in Verilog, helped teammates learn Verilog and develop other components + tests [4.5 hours]

Milestone 4

1/26/2019 - Debugged Lab 6 and fixed slt [1.5 hours]

1/27/2019 - Finished control state machine, wrote-up control unit test plan [3 hours]

1/28/2019 - Finished making a register file and test cases [4 hours]

1/30/2019 - Finished rewriting some tests for comparator, helped make the memory unit [2 hours]

Milestone 5

2/3/2019 - Discussed current issues ie $ra reg not getting backed up, had the regFile handle input/output [1 hour]

2/4/2019 - Worked on the control unit w/ Josh. [1 hour]

2/5/2019 - Realize that we monumentally fricked up the control unit, phat rip in the chat. Josh rebuilt it correctly, I helped write the test case. Navigated as Bailey drove implementing the PMS [4 hours]

2/6/2019 - Helped debug the ALU test case [1.5 hours]

2/9/2019 - Provided moral support and insulted Josh for a little bit [~30 mins-ish]

2/10/2019 - Helped debug recursive calls not return properly and helped write the recursive test case [2.5 hours]

Joshua Eckels

CSSE232 - Activity Journal

Monday, January 7, 2019 [50 min]: Decisions on architecture type. Load-store with accumulators considered. Listing necessary functions to implement.

Tuesday, January 8, 2019 [50 min (1), 90 min (2)]: Assigning registers (Function regs, args, etc.). 2 instruction types (G and I). Began work on implementing Euclid's algorithm in our language. Worked on desgin document.

Wednesday, January 9, 2019 [120 min]: Finishing translation of relPrime into BAEJ and machine language. Reformatting document.

Monday, January 14, 2019 [60 min]: Working on RTL for milestone 2. Dividing up instructions into multiple cycles. Excel document made for RTL.

Tuesday, January 15, 2019 [90 min]: Finished RTL table, discussed shopping list and control signals

Wednesday, January 16, 2019 [120 min]: Finished M2, discussion of each item in shopping list, method for verifying our RTL. Wrote up description for all of the registers we used in our RTL.

Sunday, January 20, 2019 [150 min]: Working on M3. Walked through RTL and drew datapath on whiteboard appropriately. Just a rough draft. Needs a lot of work tomorrow. Still need to document all of our changes and new updates.

Monday, January 21, 2019 [150 min]: Working on control signals, bit sizes, purposes, and functionality in our datapath. Writing this up in the Design document as a table with complete list and description of all control signals.

Tuesday, January 22, 2019 [250 min]: Fixing formatting of hardware components in design document. Learning verilog. Created verilog modules for 16-bit adder, 1-bit mux, and comparator (including test benches for each).

Monday, January 28, 2019 [330 min]: Finished implementations and extensive test benches on 1-bit mux, 2-bit mux, adder, and ALU. ALU still has some bugs in it. Need to do regs as well.

Wednesday, January 30, 2019 [60 min]: ALU bugs fixed. Decided we don't need actual reg hardware; they can just be instantiated whenever we need them.

Sunday, February 3, 2019 [60 min]: Discussed rest of project. Divided up work. Began on control unit. Will begin working on ALU sub-system for integration plan.

Monday, February 4, 2019 [90 min]: Worked with Alex. Finished control unit and test bench. Noticed a lot of inconsistencies in naming conventions.

Tuesday, February 5, 2019 [50 min]: Realized issues with control unit. Needs to be implemented as finite state machine.

Wednesday, February 6, 2019 [200 min]: Multicycle control unit finished. Test bench completed and extensive. Working on Arithmetic logic implementation system.

Wednesday, February 6, 2019 [120 min]: Meeting number 2 today. Finished ALS integration system with extensive testing. Unclocked ALU (causing a cycle delay).

Thursday, February 7, 2019 [180 min]: Massive bug fix in the register management system. Comparator was broken lol. Will finish tomorrow.

Friday, February 8, 2019 [120 min]: Register management system is 100% functional with extensive testing. Had to fix bugs in the comparator and the reg file.

Saturday, February 9, 2019 [60 min]: Testing relPrime on our processor. Lot of debugging. Realizing more issues with control unit.

Sunday, February 10, 2019 [120 min]: Scanning through and updating design document with work over the past couple weeks. Control unit was updated and is working. Also was having more issues with RA that needed to be fixed with restoring and backing up. Updated FSM and RTL.

Tuesday, February 12, 2019 [100 min]: Added code to have program pauses after finishing execution. Memory can now read programs from a .mem file. Synthesis works. Working on benchmarking.

Friday, February 15, 2019 [60 min]: Demo day. Working on a template for final report. Writing a few more quick test programs.

Bailey Morgan

Work log = Bailey Morgan

Milestone 1

Monday, December 7, 2019 [60 min] - Met with team, discussed initial design decisions, load store vs stack and instruction design.

Tuesday, December 8, 2019 [240 min] - Finalized instructions and register designs, translated programs to asm and to machine code.

Wednesday, December 9, 2019 [120 min] - Finished asm and machine code changes. Final touches on language specification.

Milestone 2

Monday, January 14, 2019 [60 min] - Developed starting RTL for I types. Broke up instructions into cycles. Started RTL summary charts.

Tuesday, January 15, 2019 [90 min] - Finished RTL summary charts. Listed the needed hardware and logic units.

Wednesday, January 16, 2019 [120 min] - Finished hardware descriptions and methods for testing RTL description. Completed M2.

Milestone 3

Sunday, January 20, 2019 [150 min] - Worked on M3, developed datapath using rtl. Traced many instructions to verify they worked.

Monday, January 21, 2019 [180 min] - Created datapath svg and Makefile to convert both datapath diagram and design documents to pdfs. Touched up things in the data path.

Tuesday, January 21, 2019 [340 min] - Finished up datapath design and pdf conversions. Updated components list and created unit test plans and Integration test plans. Worked on subcomponents with Eric.

Milestone 4

Sunday, January 27, 2019 [180 min] - Worked on finite state machine design. Updated rtl and component descriptions based off of feedback

Monday, January 28, 2019 [300 min] - Finished Fcache and helped Alex with reg file, and Josh with alu. Made updates to the design doc.

Wednesday, January 30, 2019 [330 min] - Finished all parts and went over tests with everyone to make sure they were extensive enough. Made sure they also matched what was described in the design doc. Created PCS integration and wrote testbench with Eric

Milestone 5

Sunday, Febuary 3, 2019 [90 min] - Met to discuss what we need to do for the rest of the week. Discovered a few issues with our design and discussed fixes. Helped divide and assign integration work

Tuesday, Febuary 5, 2019 [150 min] - Updated PCS, integrated MMS and PMS

Wednesday, Febuary 6, 2019 [60 min] - Wrote test bench and tested FBS.

Thursday, Febuary 7, 2019 [180 min] - Helped with RMS integration. Helped debug control unit.

Friday, Febuary 8, 2019 [180 min] - With josh integrated IES and wrote test benches. After, integrated IES, PMS, and control unit to form the data path

Sunday, Febuary 10, 2019 [240 min] - Wrote some programs to test and debug datapath with including relprime. More control issues found and fixed. Processor seems to work fine. Added a reset input to the datapath to reset to processor to pc = 0.

Friday, February 15, 2019 [60 min] – Demo given to Micah

Saturday, February 16, 2019 [180 min] – optimizations and worked on presentation

Sunday, February 17, 2019 [20 min] – Recorded video demo

Eric Tu

Eric Tu

CSSE232 - Activity Journal

Monday, January 7, 2019 [20 min]:

Discussed implementation strategy for the BAEJ Assembly Language.

Tuesday, January 8, 2019 [50 min (1), 200 min (2)]:

Implementing modulus using BAEJ and helped with the summation function.

When challenges arose, I discussed certain implementations that were needed to satisfy

the requirements for the functions.

Began work on implementing Euclid's algorithm in our language.

Wednesday, January 9, 2019 [120 min]:

Finishing translation of relPrime into BAEJ and machine language. Reformatting document.

01/14/2019 - 5:10 pm [60 min]

Helped write the skeleton of the RTL table.

01/15/2019 - 5:30 pm [90 min]

Helped finish the RTL write-up for all instructions. Reviewed the instructions.

01/16/2019 - 1:30pm [120 min]

Wrote out the hardware component spec for the ALU. Reformated and reviewed the document.

01/20/2019 - 2:30pm [150 min]

Helped draw up a rough draft of datapath via RTL.

Discussed implementation method of backing of registers.

01/21/2019 - 5:10pm [120 min]:

Wrote up a rough draft of component unit testing methods.

01/22/2019 - 7:00 pm [270 min]:

Drew up and named subsystems for integration testing. Peer reviewed.

Rewrote methods of testing the RTL.

Milestone 4

01/27/2019 - 6:00 pm [180 min]:

Drafted the finite state machine for the datapath control signals. Described different

testing methods for the finite state machine with Alex.

01/28/2019 - 6:00 pm [240 min]:

Implemented the memory unit. Memory unit test still has bugs.

01/30/2019 - 3:00 pm [300 min]:

Finished implementing memory unit, along with its unit tests.

Milestone 5

02/03/2019 - 1:00 pm [90 min]:

Discussed the plans for the rest of the week. Communicated an issue about ra being lost upon nested function calls.

02/04/2019 - 8:00 pm [120 min]:

Worked on integrating and testing the F Register Backup System. Discovered that there were errors with the testbench and not the module itself.

02/05/2019 - 9:00 pm [120 min]:

Worked on integrating and testing the F Register Backup System. Bailey helped debug the test cases and found out that it was a timing issue. The FBS has been finalized.

02/06/2019 - 11:00 pm [150 min]:

Helped Josh debug the ALUS. Updated documentation and started the presentation powerpoint.

02/07/2019 - 8:00 pm [60 min]:

Helped Josh and Bailey debug the Register Management System.

02/08/2019 - 7:00 pm [60 min]:

Helped debug the datapath and discussed methods of I/O.

02/10/2019 - 4:00 pm [240 min]:

Updated the team journal and updated design document according to the current milestone with Josh.

Appendix C: Benchmarking

The following data was collected from the simulator when the relative prime algorithm was run with an input of 5040:

```
============================================================
                        METRICS
============================================================

Latency:
    Transfered from memory (bytes):  203944
    Transfered to memory   (bytes):  0
    Program size in memory (bytes):  70

Performance:
    Total instructions executed   :  50986
    Total cycles executed         :  163114
    Average cycles per instruction:  3.20
    Simulation clock rate    (MHz):  87.60
    Execution time            (ms):  1.86

Output:    11
```

Figure C.1: Benchmarking data on relPrime (5040)