The BAW Instruction Set Manual

Bryant Herren, Austin Waddell, Wesley Ring

April 28, 2019

Contents

1 Introduction

BAW is the instruction set for a floating point co-processor designed in ECGR 3183 at the University of North Carolina at Charlotte. The co-processor was implemented with a single-cycle architecture, and a pipelined architecture. For the pipelined architecture, two branch prediction algorithms were implemented (1 static and 1 dynamic) as well as no branch prediction.

Provided in this document:

- The complete ISA
- Architecture and controller design (units, diagrams, etc)
- VHDL simulation
- Performance results and discussion for pipelined vs. unpipelined approaches

1.1 System Parameters

- 32 bits per instruction/register
- Data is stored using IEEE single-precision floating point numbers.
- Register file has 16 registers
- Timings:
  - Clock cycle (pipelined): 100ns
  - Register Read/Write: 100ns
  - Memory Read/Write: 300ns
  - Single ALU Op: 200ns

1.2 Memory

The system includes a data memory addressed 0-1023 and 16 Floating Point registers (Referenced as X0-X15). Each memory location and register uses a 32-bit value. The simulation can read an input file containing the operational parameters, code, and memory contents (there is an assembler).

1.3 Additional Features

- FP Multiply by -1, 1, or 0 takes 1 cycle
- FP Multiply by power of 2 takes 2 cycles
- Condition Codes:
  - Z - Zero
  - N - Negative
  - V - Overflow
  - C - Carry
  - E - Error (domain errors, etc.)
- All condition codes are set as needed on arithmetic operations
- The round, ceiling, and floor functions would round up to the nearest integer, expressing the result in floating point format.

2 ISA

2.1 Introduction

The ISA is based off of the ARMv8 / LEGv8 ISA. As a result, there will likely be similarities.

2.2 Instruction Format

There are four instruction formats: Register (R), Data (D), Immediate (I), and Branch (B). All processor instructions are 32 bits wide. Table 1 Contains information about the specific instruction formats.

Notes:

1. The Opcode is 8 bits long, allowing it to be easily read in hex format. This also allows additional instructions to be added in the future.
2.

Table 1: Instruction Format Descriptions

Format	Description	Example
R (Register)	An instruction whose inputs and outputs are both registers	Fadd X9, X21, X9
D (Data)	An instruction used when fetching or placing data in memory	Load X9, [X22, #64]
I (Immediate)	An instruction that carries specific additional data	Pow X9, #15
CB (Con. Branch)	An instruction that deals with changing the location of the PC directly, but conditionally	If (Ri == 0) PC ← LABEL (line)
UB (Unc. Branch)	An instruction that deals with changing the location of the PC directly and unconditionally	PC ← M[Ri]
S (Set)	An instruction that sets a register to a specific floating point value	Ri ← FPvalue

Please see Table 2 for specific information on the Instruction Formats.

Table 2: Instruction Formats

Instruction Format	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R (Register)	Opcode (8 bits)								Rm (4 bits)				Empty (12 bits)												Rn (4 bits)				Rd (4 bits)			
D (Data)	Opcode (8 bits)								Rm (4 bits)				Address (14 bits)														op2 (2 bits)		Rd (4 bits)			
I (Immediate)	Opcode (8 bits)								Rm (4 bits)				Immediate Data (16 bits)																Rd (4 bits)			
CB (Con. Branch)	Opcode (8 bits)								Rm (4 bits)				Address (20 bits)																			
UB (Unc. Branch)	Opcode (8 bits)								Empty (4 bits)				Address (20 bits)																			
S (Set)	Opcode (8 bits)								Empty (20 bits)																				Rd (4 bits)			
	Floating Point Value (32 bits)																															

2.3 Instructions

2.3.1 Set

Set

ASM	Opcode	Format	Description	Operation	ALU Cycles
Set	00000001	S	Sets Ri to given floating point value	Ri ← FPvalue	1

ASM Example: Set Ri, #FPvalue

Flags
- Zero
- Negitive

2.3.2 Load

Load

ASM	Opcode	Format	Description	Operation	ALU Cycles
Load	00000010	D	Copies Rj from memory and into Ri	Ri ← M[Rj]	1

ASM Example: Load Ri, Rj

Flags
- Zero
- Negitive

2.3.3 Store

Store

ASM	Opcode	Format	Description	Operation	ALU Cycles
Store	00000011	D	Copies data from register Rj into memory	M[Ri] ← Rj	1

ASM Example: Store Ri, Rj

Flags
- None

2.3.4 Move

Move

ASM	Opcode	Format	Description	Operation	ALU Cycles
Move	00000100	R	Moves the value of Rj to Ri, deleting the original	Ri ← Rj	1

ASM Example: Move Ri, Rj

Flags
- Zero
- Negitive

2.3.5 Add

Fadd

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fadd	00000101	R	Adds Rj and Rk into Ri	Ri ← Rj + Rk	3

ASM Example: Fadd Ri, Rj, Rk

Flags
- Zero
- Negitive
- Overflow
- Carry

2.3.6 Subtract

Fsub

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fsub	00000110	R	Subtrcts Rk from Rj into Ri	Ri ← Rj – Rk	3

ASM Example: Fsub Ri, Rj, Rk

Flags
- Zero
- Negitive
- Overflow
- Carry

2.3.7 Negate

Fneg

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fneg	00000111	R	Sets Ri to the Opposite of Rj	Ri ← -Rj	1

ASM Example: Fneg Ri, Rj

Flags
- Zero
- Negitive

2.3.8 Multiply

Fmul

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fmul	00001000	R	Multiplies Rj and Rk into Ri	Ri ← Rj * Rk	5

ASM Example: Fmul Ri, Rj, Rk

Flags
- Zero
- Negitive
- Overflow
- Carry

2.3.9 Divide

Fdiv

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fdiv	00001001	R	Divides Rj by Rk into Ri	Ri ← Rj / Rk	8

ASM Example: Fdiv Ri, Rj, Rk

Flags
- Zero
- Negitive
- Overflow
- Carry
- Error - divide by zero

2.3.10 Floor

Floor

ASM	Opcode	Format	Description	Operation	ALU Cycles
Floor	00001010	R	Sets Ri to the floor of Rj	Ri ← ⌊ Rj⌋	1

ASM Example: Floor Ri, Rj

Flags

- Zero
- Negitive

2.3.11 Ceiling

Ceil

ASM	Opcode	Format	Description	Operation	ALU Cycles
Ceil	00001011	R	Sets Ri to the ceil of Rj	Ri ← ⌈ Rj⌉	1

ASM Example: Ceil Ri, Rj

Flags

- Zero
- Negitive

2.3.12 Round

Round

ASM	Opcode	Format	Description	Operation	ALU Cycles
Round	00001100	R	Sets Ri to Rj rounded to the nearest integer	Ri ← round(Rj)	1

ASM Example: Round Ri, Rj

Flags

- Zero
- Negitive

2.3.13 Absolute Value

Fabs

ASM	Opcode	Format	Description	Operation	ALU Cycles
Fabs	00001101	R	Sets Ri to the absolute value of Rj	Ri ← — Rj —	1

ASM Example: Fabs Ri, Rj

Flags

- Zero

2.3.14 Minimum

Min

ASM	Opcode	Format	Description	Operation	ALU Cycles
Min	00001110	R	Sets Ri to the minimum value between Rj and Rk	Ri ← min(Rj, Rk)	1

ASM Example: Min Ri, Rj, Rk

Flags

- Zero
- Negitive

2.3.15 Maximum

Max

ASM	Opcode	Format	Description	Operation	ALU Cycles
Max	00001111	R	Sets Ri to the maximum value between Rj and Rk	Ri ← max(Rj, Rk)	1

ASM Example: Max Ri, Rj, Rk

Flags

- Zero
- Negitive

2.3.16 Power

Pow

ASM	Opcode	Format	Description	Operation	ALU Cycles
Pow	00010000	I	Sets Ri to Rj raised to some given integer power	Ri ← Rjˆinteger_value	6

ASM Example: Pow Ri, Rj, #integer_value

Flags

- Zero
- Negitive
- Overflow
- Carry

2.3.17 Exponent

Exp

ASM	Opcode	Format	Description	Operation	ALU Cycles
Exp	00010001	R	Sets Ri to Rj exponentiated	Ri ← eˆRj	8

ASM Example: Exp Ri, Rj

Flags

- Overflow
- Carry

2.3.18 Square Root

Sqrt

ASM	Opcode	Format	Description	Operation	ALU Cycles
Sqrt	00010010	R	Sets Ri to the square root of Rj	Ri ← $\sqrt{}$ Rj	8

ASM Example: Sqrt Ri, Rj

Flags

- Zero
- Overflow
- Carry
- Error - complex domain

2.3.19 Branch (Uncond.)

B

ASM	Opcode	Format	Description	Operation	ALU Cycles
B	00010011	UB	Loads Ri from memory into PC	PC ← M[Ri]	1

ASM Example: B Ri

Flags

- None

2.3.20 Branch Zero

BZ

ASM	Opcode	Format	Description	Operation	ALU Cycles
BZ	00010100	CB	Sends the PC to a specific labeled line if Ri is zero	If (Ri == 0) PC ← LABEL (line)	3

ASM Example: BZ Ri, LABEL

Flags

- Zero

2.3.21 Branch Negative

BN

ASM	Opcode	Format	Description	Operation	ALU Cycles
BN	00010101	CB	Sends the PC to a specific labeled line if Ri is negative	If (Ri ¡ 0) PC ← LABEL (line)	3

ASM Example: BN Ri, LABEL

Flags

- Negitive

2.3.22 No-op

Nop

ASM	Opcode	Format	Description	Operation	ALU Cycles
Nop	00010110	-	No operation	No operation	1

ASM Example: Nop

Flags

- None

2.3.23 Halt

Halt

ASM	Opcode	Format	Description	Operation	ALU Cycles
Halt	00010111	-	Stop program	Stop Program	-

ASM Example: Halt

Flags

- None

3 Architecture

3.1 ALU

The ALU can perform fourteen unique operations, and also pass through input A. Eleven of these operations are done in the second phase of the ALU, directly after the pre-normalization process (Fadd, for example). The remaining three operations are done in the final stage of the ALU, directly after the post-normalization process (Floor, for example). The figure below shows each phase of the ALU in order from input to result.

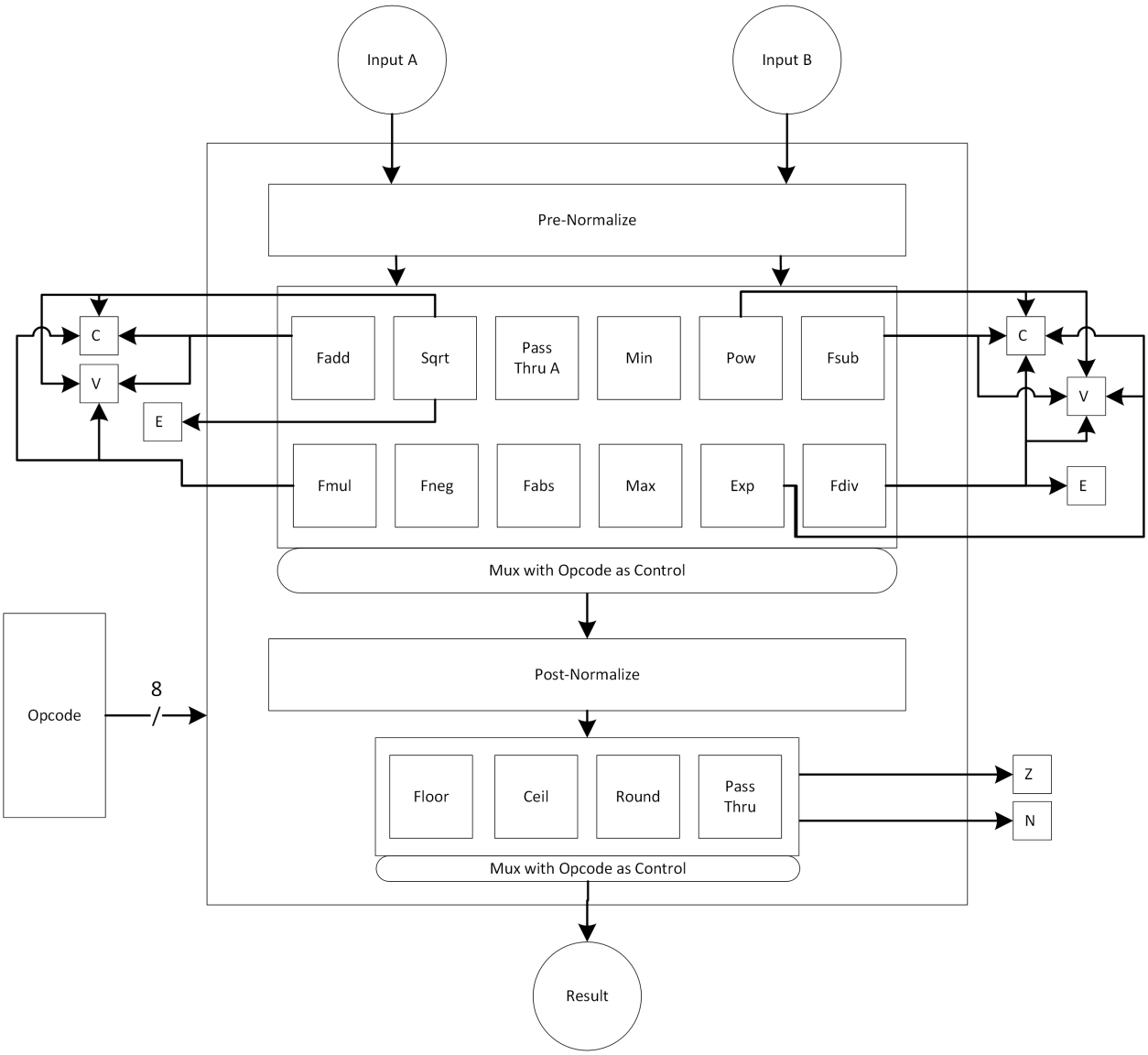


Figure 1: ALU Design

When the two inputs enter the ALU, their exponents are normalized to whichever input is larger. The larger input is chosen because it's less likely to result in a loss of precision. Depending on the instruction, the two inputs may bypass the normalization process in order to save on ALU cycles (Fabs, for example).

Next is the first stage of arithmetic where the inputs are added, multiplied, etc. The functional blocks of this section can out any combination of the following three flags: carry, overflow, and error. The figure

above depicts specifically what blocks can trigger what flags. The error flag is exclusively used for domain errors, either when dividing by zero or entering the complex domain.

After the first stage of arithmetic, the result enters the post-normalize stage (which can be bypassed like before) and then is passed through to the result output of the ALU. This is also where the zero and negative flags are set if necessary. Alternatively, if the first stage of arithmetic is skipped (pass-through), then the second stage may be used for the Round, Floor, and Ceil functions. It's worth noting that having two stages of arithmetic allows for combination instructions to be implemented with little to no additional hardware (Fadd + Round, for example) in order to make the common case fast. This is why more Opcode bits than necessary exist.

The entire ALU is controlled by the Opcode control signal, which comes directly out of the instruction. See Table 3 to determine which Opcodes correspond to which ALU functions.

3.2 Datapath

3.2.1 Single Cycle

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

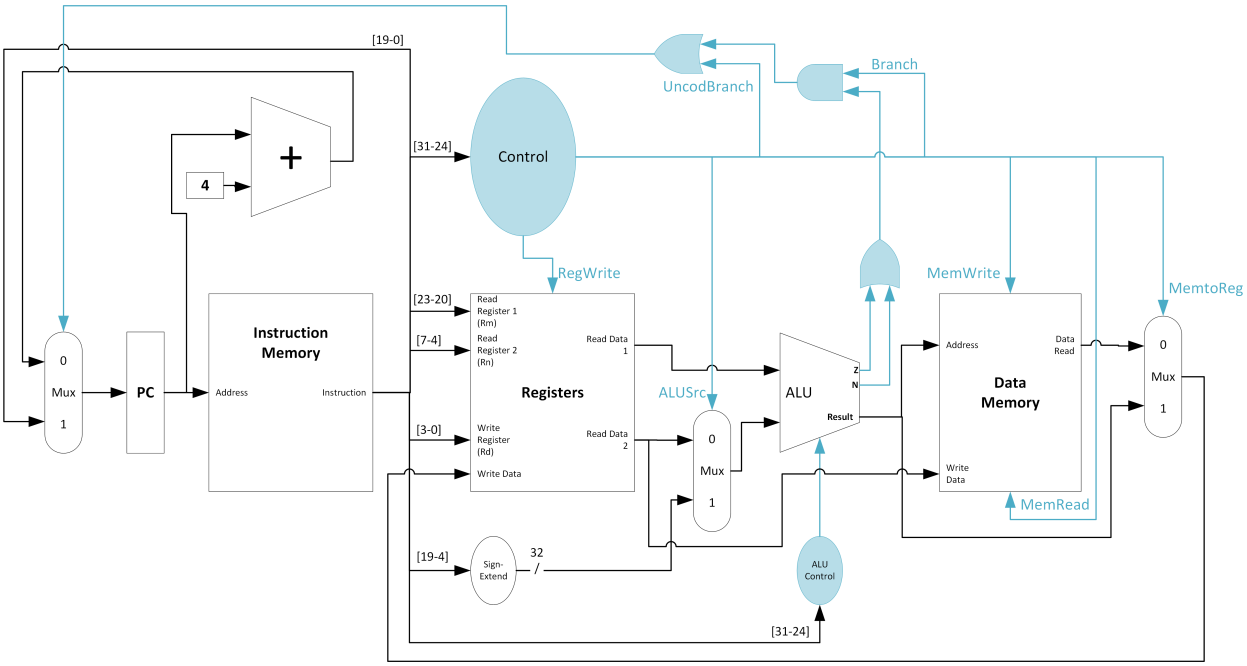


Figure 2: Single Cycle Architecture

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

3.2.2 Pipelined

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

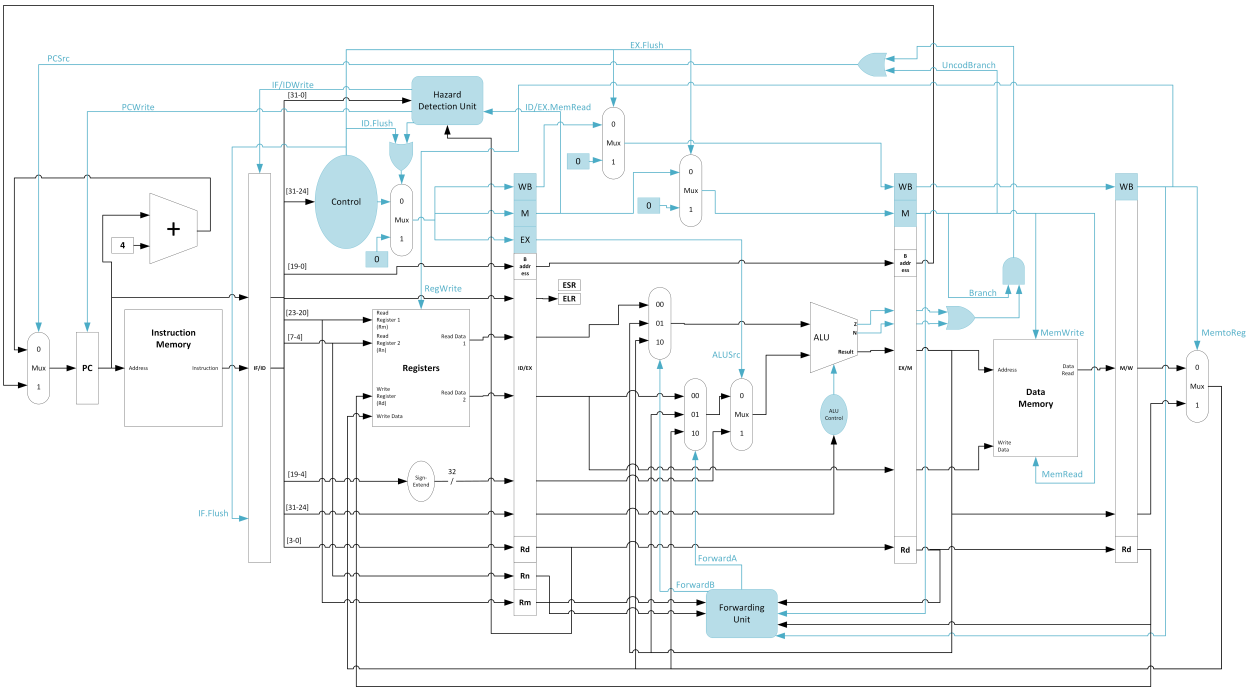


Figure 3: Pipelined Architecture

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

3.3 Controller

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

Table 3: Datapath Control

ASM	Opcode	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	UncodBranch	ALU Control Description
Set	00000001	1	0	1	0	0	0	0	Use Adder
Load	00000010	1	1	1	1	0	0	0	Use Adder
Store	00000011	1	X	0	0	1	0	0	Use Adder
Move	00000100	0	0	1	0	0	0	0	Pass Through
Fadd	00000101	0	0	1	0	0	0	0	Use Adder
Fsub	00000110	0	0	1	0	0	0	0	Use Subtracter
Fneg	00000111	0	0	1	0	0	0	0	Negate
Fmul	00001000	0	0	1	0	0	0	0	Use Multiplier
Fdiv	00001001	0	0	1	0	0	0	0	Use Divider
Floor	00001010	0	0	1	0	0	0	0	Floor Result
Ceil	00001011	0	0	1	0	0	0	0	Ceil Result
Round	00001100	0	0	1	0	0	0	0	Round Result
Fabs	00001101	0	0	1	0	0	0	0	Take Absolute Value
Min	00001110	0	0	1	0	0	0	0	Take Minimum Input
Max	00001111	0	0	1	0	0	0	0	Take Maximum Output
Pow	00010000	0	0	1	0	0	0	0	Take Power
Exp	00010001	0	0	1	0	0	0	0	Exponentiate
Sqrt	00010010	0	0	1	0	0	0	0	Take Square Root
B	00010011	1	X	X	X	X	X	1	Pass Through
BZ	00010100	1	X	0	0	0	1	0	Pass Through
BN	00010101	1	X	0	0	0	1	0	Pass Through
Nop	00010110	0	0	0	0	0	0	0	Pass Through
Halt	00010111	0	0	0	0	0	0	0	Pass Through

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

4 VHDL Description

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

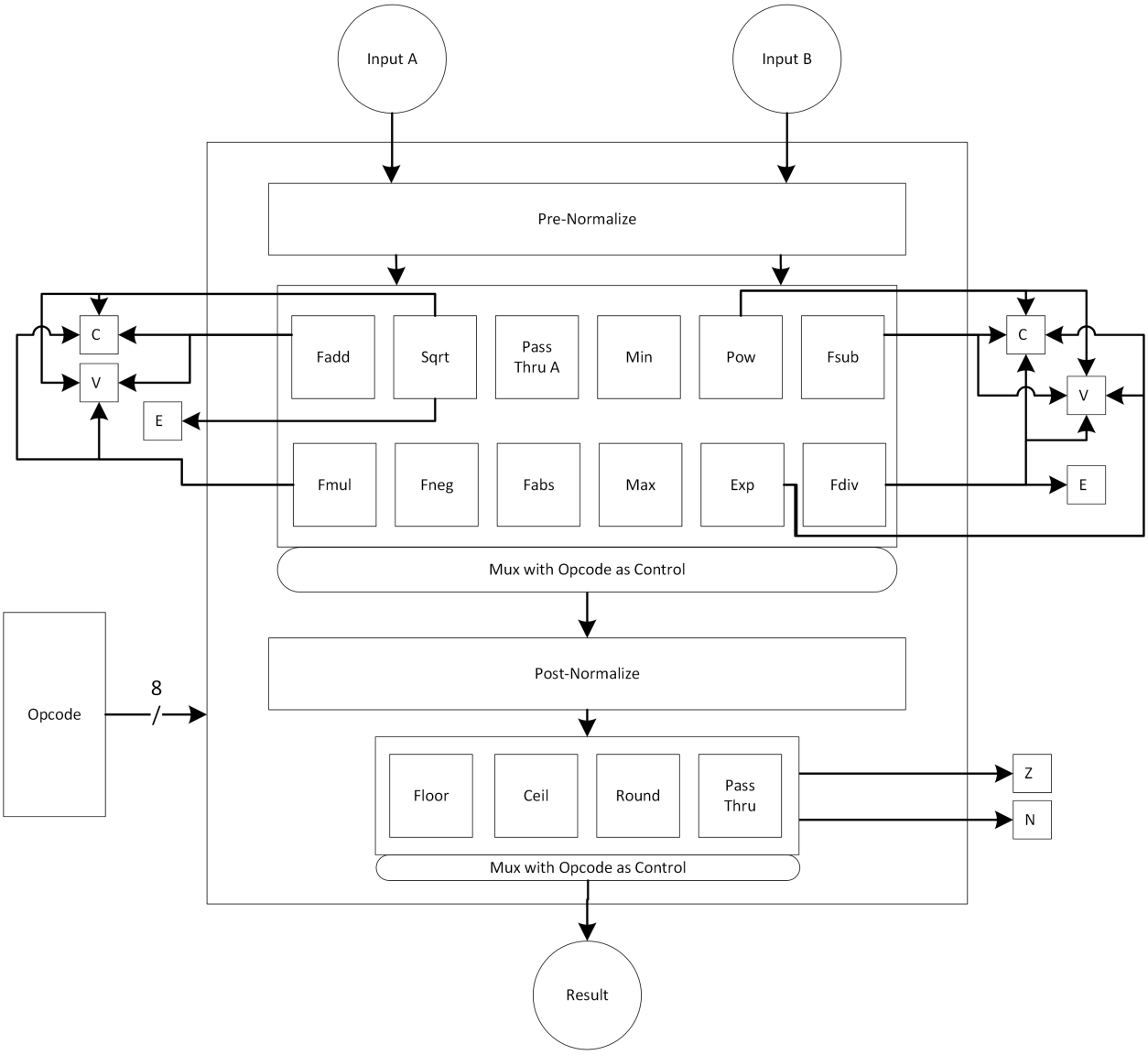
5 Testing

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

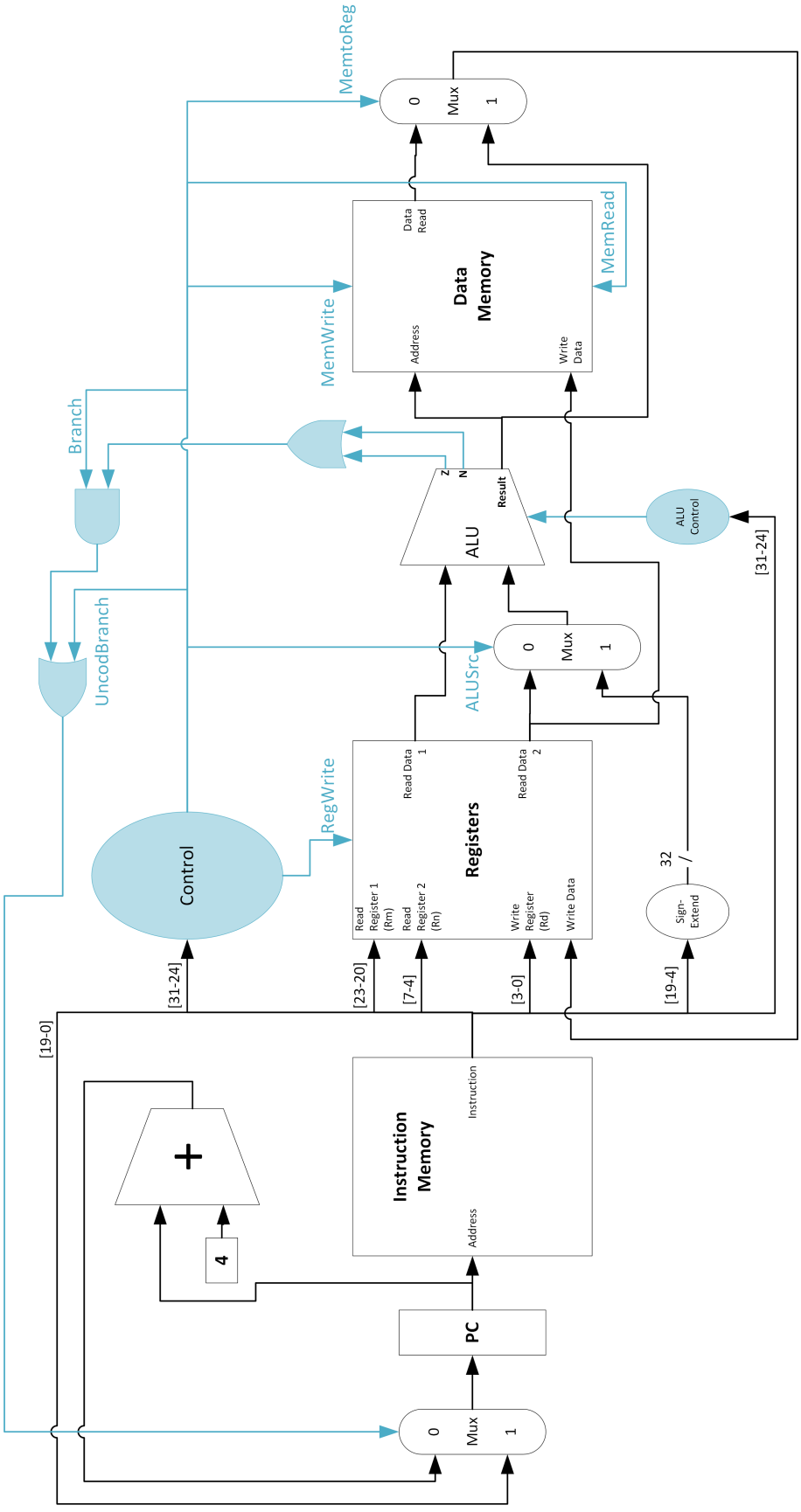
6 Conclusion

Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah Blah

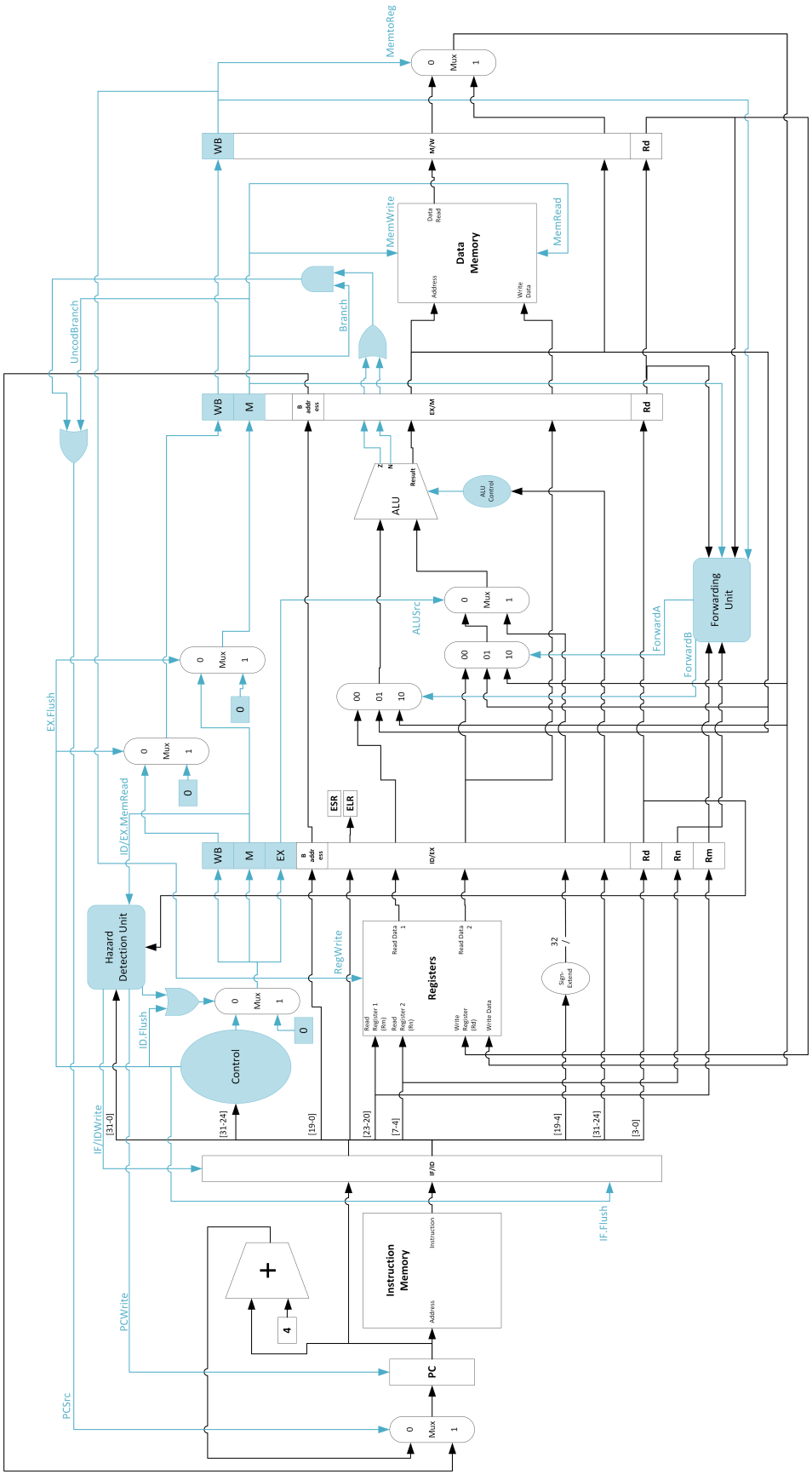
Appendices



ALU Design



Single Cycle Architecture



Pipelined Architecture