The BAW Instruction Set Manual

Bryant Herren, Austin Waddell, Wesley Ring

April 29, 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)
- Description of 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 round 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 six instruction formats: Register (R), Data (D), Immediate (I), Con. Branch (CB), Unc. Branch (UB), and Set (S). All processor instructions are 32 bits wide, with the exception of Set (see below). See Table 1 for descriptions of each 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. The Set instruction type is unique in that it accesses twice the amount of instruction memory as the other instructions (two 32 bit lines). Since the hardware described in this manual does not technically support this kind of memory access, the simulation considers it a one line instruction, accessing all necessary information at once.

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 $\leftarrow$ LABEL (line)
UB (Unc. Branch)	An instruction that deals with changing the location of the PC directly and unconditionally	PC $\leftarrow$ M[Ri]
S (Set)	An instruction that sets a register to a specific floating point value	Ri $\leftarrow$ FPvalue

Table 2 on the following page contains specific information on the bit mappings of each instruction format.

Table 2: Instruction Formats

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

In each of the six instruction formats, bits 31 down to 24 are reserved for the Opcode of the instruction. It is important that the Opcode appears consistently with each instruction because it will be sent to control before being decoded. The only exception is the 32 bit float appearing on the second line of the S type instruction.

After the Opcode, bits 23 down to 20 are reserved for Rm, the first register file input. Rm will always serve as input A in the ALU, which is the input that can be passed through the ALU. The Rm bits are reserved for R, D, I, CB, and S type instructions because they all utilize input A of the ALU. The Rm bits in the UB type instruction are left empty because it does not use input A of the ALU.

Beyond Rm, the bit assignments for each instruction start to vary more significantly. This also where more long form data (immediate, addresses) begin to appear. In R type instructions, since no long form data is used, bits 19 down to 8 are left empty. Bits 7 down to 4 are used for Rn, the second register file input, which is input B for the ALU in R type instructions. Lately, bits 3 down to 0 are used for Rd, the write back register, or result of the ALU in R type instructions. The bits for Rn and Rd are placed end the end of the instruction to maximize space for long form data when it is needed.

In D type instructions, bits 19 down to 4 are used for an address, with bits 5 down to 4 (op2) being hard set to 0 in order to make the address divisible by four. Most of the time, the entire address field will be set to zero since the current D type instructions don't support an address offset. Still, this field is included for the sake of future expandability. Like with R type instructions, bits 3 down to 0 are used for Rd, the write back register.

In I type instructions, bits 19 down to 4 are used for immediate data. This immediate data will be sent to input B of the ALU. Like with R type instructions, bits 3 down to 0 are used for Rd, the write back register.

In CB and UB type instructions, bits 19 down to 0 are complete reserved for branch addresses while in S type instructions, this same space in completely empty.

2.3 Instructions

2.3.1 Set

Set

ASM	Opcode	Format	Description	Operation	ALU Cycle
Set	00000001	S	Sets Ri to given floating point value	Ri $\leftarrow$ 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 $\leftarrow$ 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] $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ 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

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
- 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 $\leftarrow \lfloor$ Rj$\rfloor$	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 $\leftarrow \lceil$ Rj$\rceil$	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 $\leftarrow$ 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 $\leftarrow$ Rjˆinteger_value	6

ASM Example: Pow Ri, Rj, #integer_value

Flags

- Zero
- Negitive

2.3.17 Exponent

Exp

ASM	Opcode	Format	Description	Operation	ALU Cycles
Exp	00010001	R	Sets Ri to Rj exponentiated	Ri $\leftarrow$ 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 $\leftarrow \sqrt{}$ Rj	8

ASM Example: Sqrt Ri, Rj

Flags

- Zero
- 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.

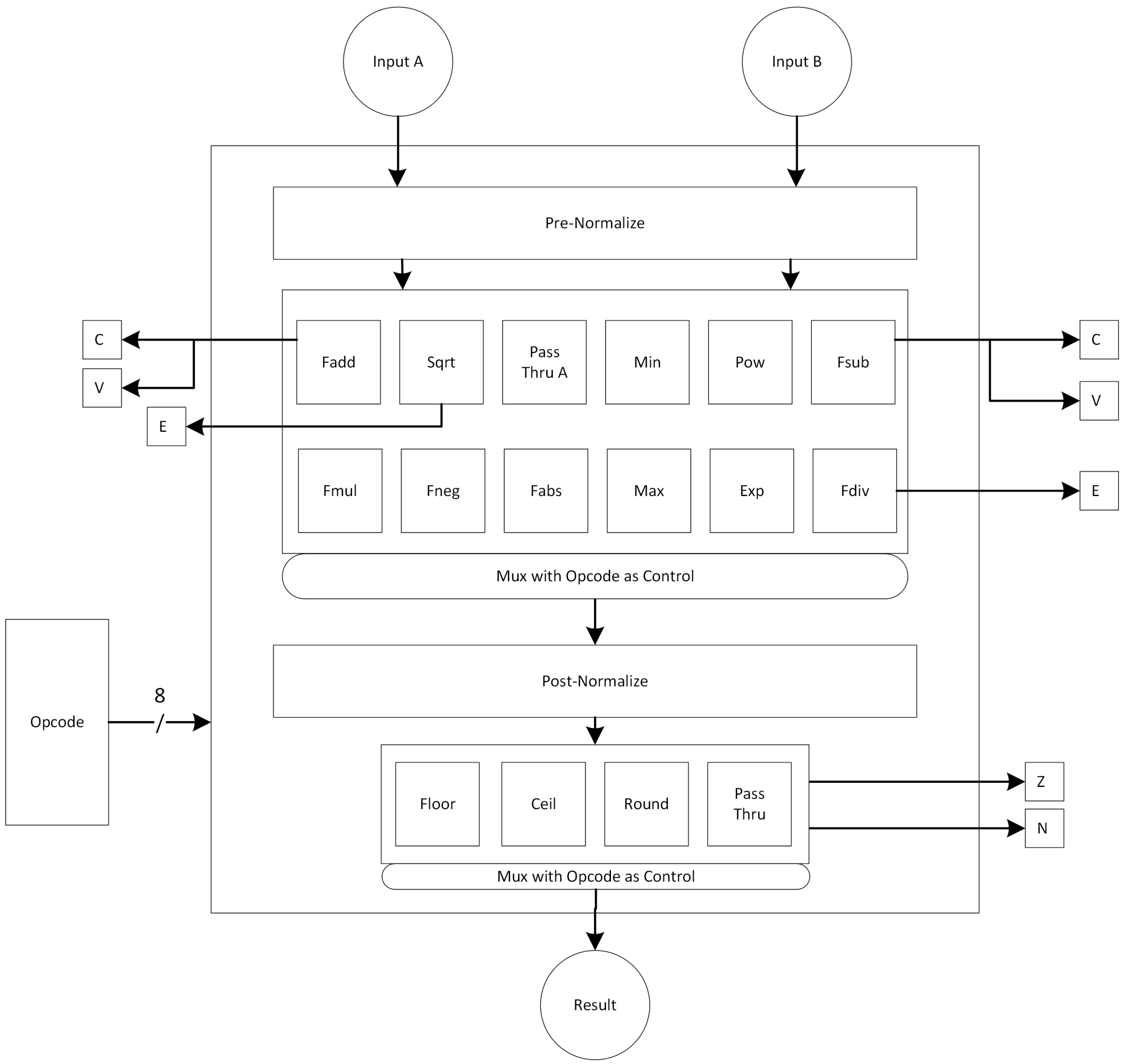


Figure 1: ALU Design

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). Another feature of this normalize process is setting either of the inputs to 0 before entering the first stage of arithmetic. This is useful S type instructions.

Next is the first stage of arithmetic where the inputs are added, multiplied, etc. The functional blocks of this section can output 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.

In the additional features section, it was noted that FP multiply by -1, 1, and 0 takes one cycle, and that FP multiply by a power of 2 takes two cycles. Normally, a multiplication operation would take five cycles in the ALU. However, because the above operations don't require explicit multiplication, just numerical manipulation, the cycle count can be dramatically reduced.

For FP multiply by -1, the ALU can just invert the sign bit. For FP multiply by 1, the ALU can just pass through the input. For FP multiply by 0, the ALU can just pass through a 0. The multiplication by power of 2 is a bit more complicated though. For this operation, the ALU will take the power of 2 (n), and calculate $\Delta_2 = \log_2(n)$. The Δ_2 value will then be added to the exponent of the input, and the mantissa will be shifted to the left by the same value Δ_2.

Another additional feature was that the Round, Floor, and Ceil functions would each round to the nearest integer. This is done by determining the location and value of the "half bit", the "half bit" being the first digit to the right of the binary point in the original input. The "half bit" exists somewhere in the mantissa. To determine its location, the ALU subtracts 126 from the biased exponent. The resulting value Δ_h denotes the location of the "half bit" in the mantissa (going from left to right). For example, if $\Delta_h = 1$, then the most significant mantissa bit is the "half bit". If $\Delta_h \leq 0$, then the "half bit" location is irrelevant, and its value is 0.

For the Floor function, the "half bit" is set to 0, and all mantissa bits that are less significant are also set to 0.

For the Ceil function, the "half bit" is set to 0, and all mantissa bits that are less significant are also set to 0. A binary 1.0 is then added to the resulting value.

For the Round function, one of two things will happen depending on the value of the "half bit". If the "half bit" is a 1, the Round function will perform the Ceil operation. If the "half bit" is a 0, the Round function will perform the Floor operation.

3.2 Datapath

3.2.1 Single Cycle

The figure below shows the single cycle implementation of the BAW architecture. This section will describe the functional blocks of the architecture, but only briefly mention their control signals. See Section 3.3 for further elaboration on the control signals.

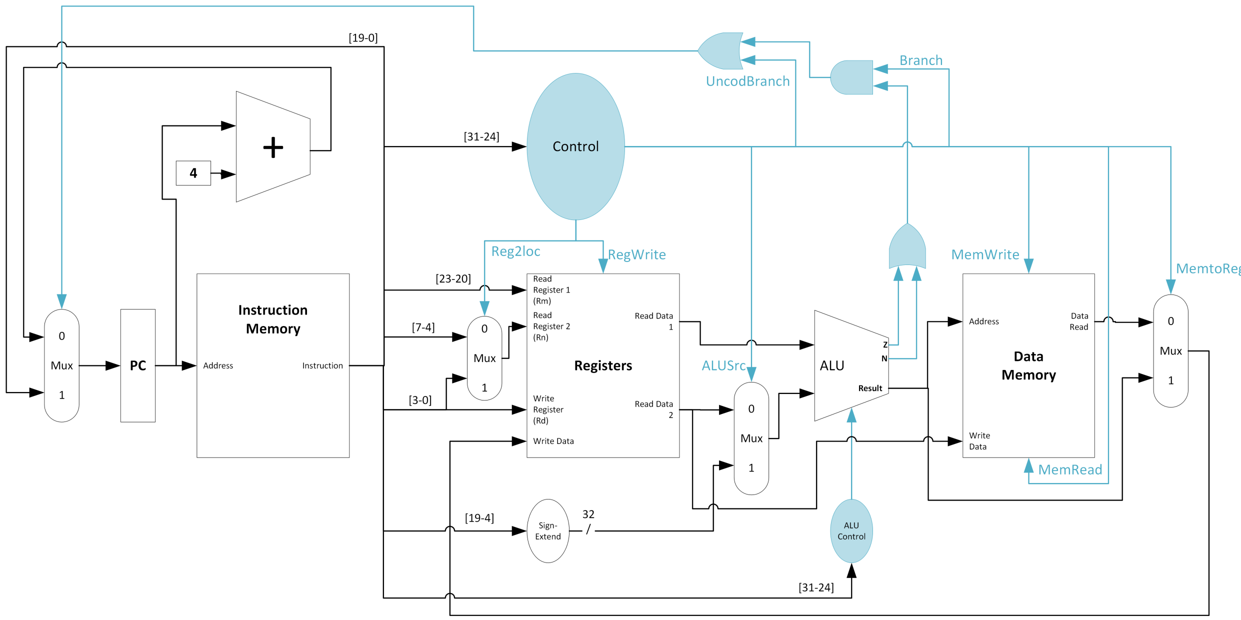


Figure 2: Single Cycle Architecture

The Instruction Fetch phase has four functional blocks: the Instruction Memory, the PC, a multiplexer, and an adder. The instruction memory is byte addressable, and holds 32 bit instructions formatted as seen in Table 2. The PC takes an address as an input, and outputs that same value to the instruction memory input. The output of the PC is also sent to an adder, where it is incremented by 4 bytes, and sent back to the PC input, bringing it to the beginning of the next instruction. Alternatively, if a branch instruction is being used, the resulting address from the branch will be sent to the PC input. In order to handle the multiple PC inputs, a multiplexer is used with branch related signals as control.

The Instruction Decode phase also has four functional blocks: the Register File, the sign extension unit, the control unit, and a multiplexer. The control unit takes the Opcode of the instruction as input, and distributes control signals as needed. The sign extension unit takes a 16 bit input from the instruction (an address or immediate), and outputs the same number sign extended to 32 bits. This is important because the ALU can only take in 32 bit inputs. The Register File takes in specific bits from the instruction that denote which register to use and where. The Read Register 2 (Rn) input is unique in that its value can come from two different locations in the instruction. In order to handle this, a multiplexer controlled by Reg2loc is introduced. Having this functionality is necessary for the Store instruction to work properly.

The outputs of the Instruction Decode phase are Read Data 1, Read Data 2, and the sign extended value. Read Data 1 always goes to input A of the ALU while input B of the ALU can either be Read Data 2 or the sign extended value. A multiplexer controlled by ALUsrc determines input B of the ALU. Read Data 2 is also routed to the Write Data input of Data Memory. This is necessary for the Store instruction.

The Execute phase has one input excluding Read Data 2 as mentioned above: the ALU Result. The ALU Result is sent to the Address input of Data Memory, and the Write Back multiplexer. Depending on the instruction, a value can be loaded from or stored in the calculated address from ALU Result, or the ALU Result can be directly sent to the Write Register input on the register file from the Write Back multiplexer.

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

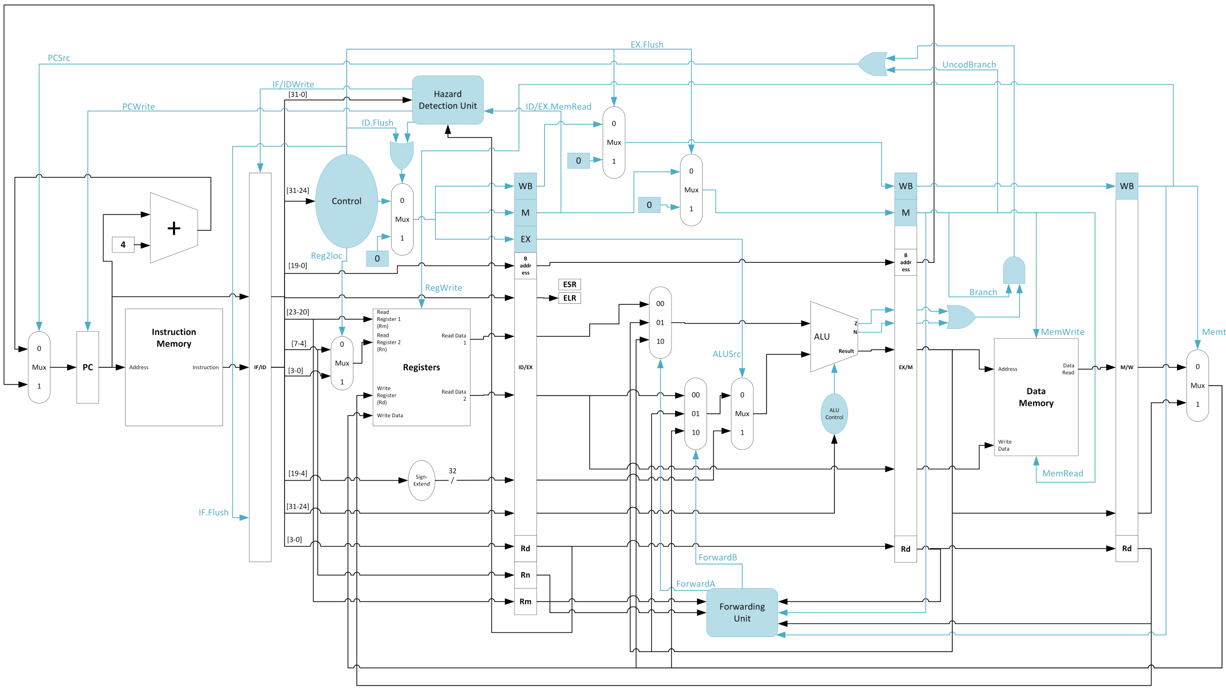


Figure 3: Pipelined Architecture

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

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

Table 3: Datapath Control

ASM	Opcode	Reg2loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	UncodBranch	ALU Function
Set	00000001	0	1	0	1	0	0	0	0	Use Adder
Load	00000010	0	1	1	1	1	0	0	0	Use Adder
Store	00000011	1	1	X	0	0	1	0	0	Use Adder
Move	00000100	0	0	0	1	0	0	0	0	Pass Through
Fadd	00000101	0	0	0	1	0	0	0	0	Use Adder
Fsub	00000110	0	0	0	1	0	0	0	0	Use Subtracter
Fneg	00000111	0	0	0	1	0	0	0	0	Negate
Fmul	00001000	0	0	0	1	0	0	0	0	Use Multiplier
Fdiv	00001001	0	0	0	1	0	0	0	0	Use Divider
Floor	00001010	0	0	0	1	0	0	0	0	Floor Result
Ceil	00001011	0	0	0	1	0	0	0	0	Ceil Result
Round	00001100	0	0	0	1	0	0	0	0	Round Result
Fabs	00001101	0	0	0	1	0	0	0	0	Take Absolute Value
Min	00001110	0	0	0	1	0	0	0	0	Take Minimum Input
Max	00001111	0	0	0	1	0	0	0	0	Take Maximum Output
Pow	00010000	0	0	0	1	0	0	0	0	Take Power
Exp	00010001	0	0	0	1	0	0	0	0	Exponentiate
Sqrt	00010010	0	0	0	1	0	0	0	0	Take Square Root
B	00010011	0	1	X	X	X	X	X	1	Pass Through
BZ	00010100	0	1	X	0	0	0	1	0	Pass Through
BN	00010101	0	1	X	0	0	0	1	0	Pass Through
Nop	00010110	0	0	0	0	0	0	0	0	Pass Through
Halt	00010111	0	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

3.4 Branch Prediction

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

3.4.1 Static

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

3.4.2 Dynamic

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

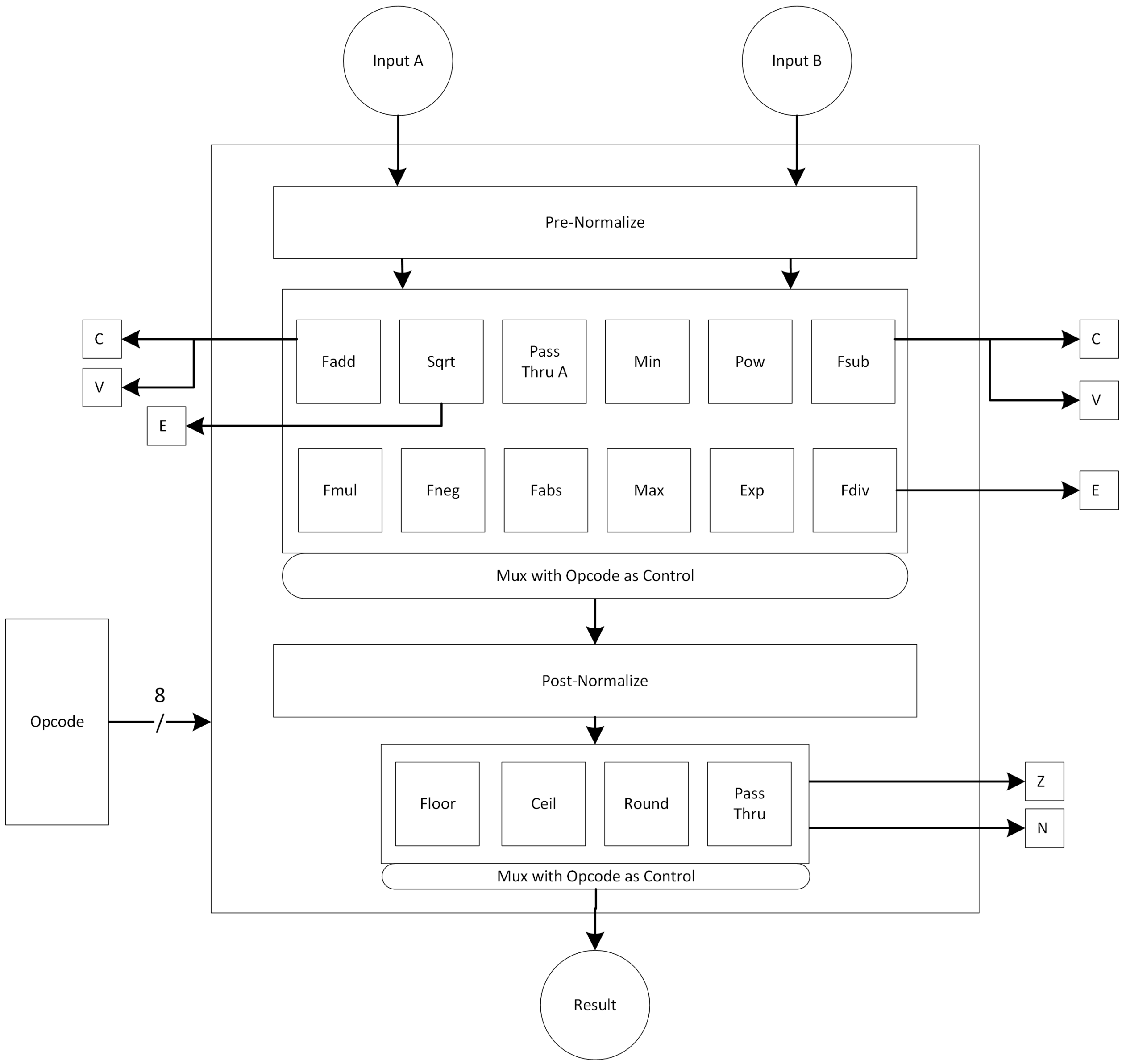
5 Testing

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

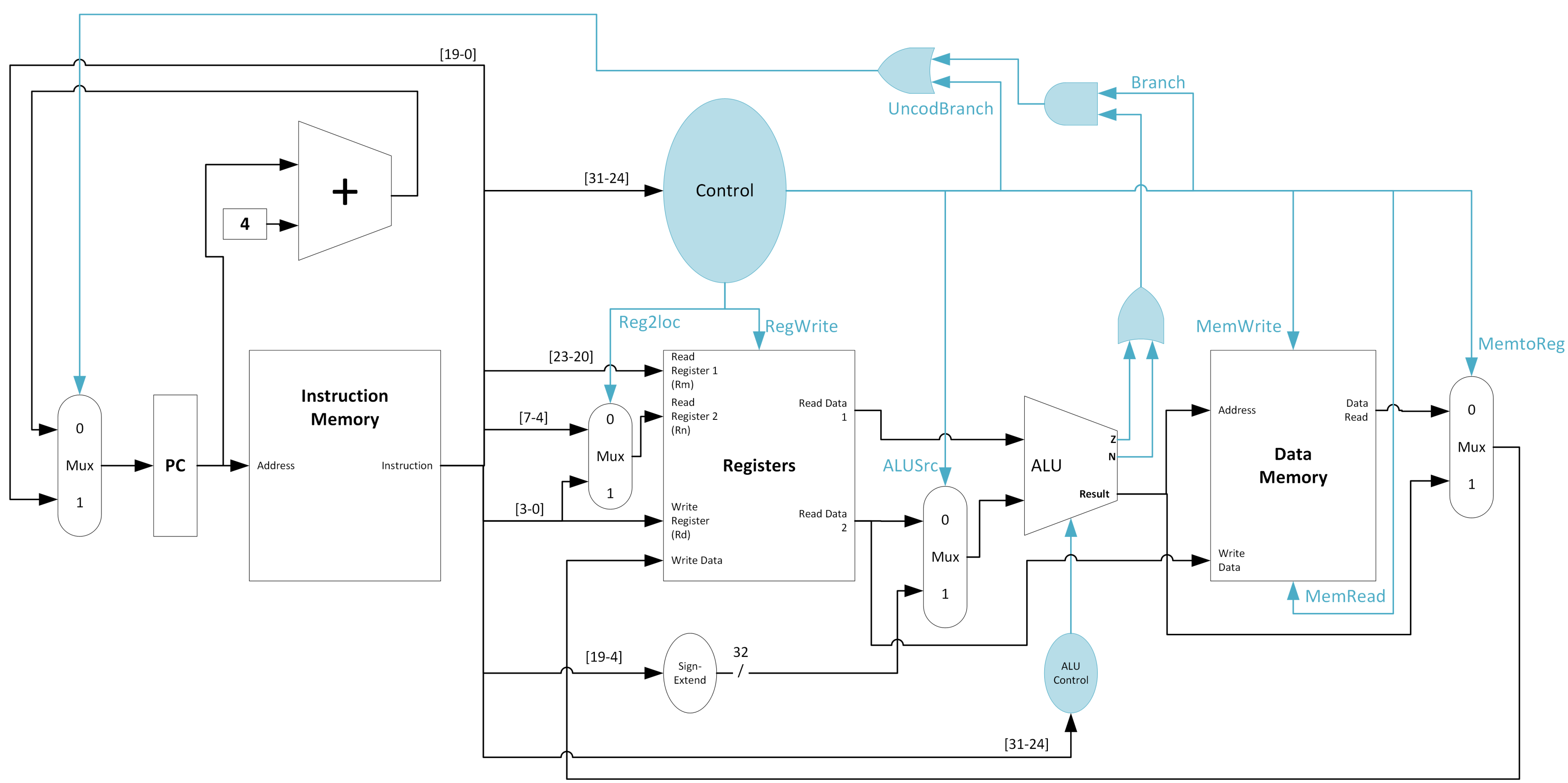
6 Conclusion

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

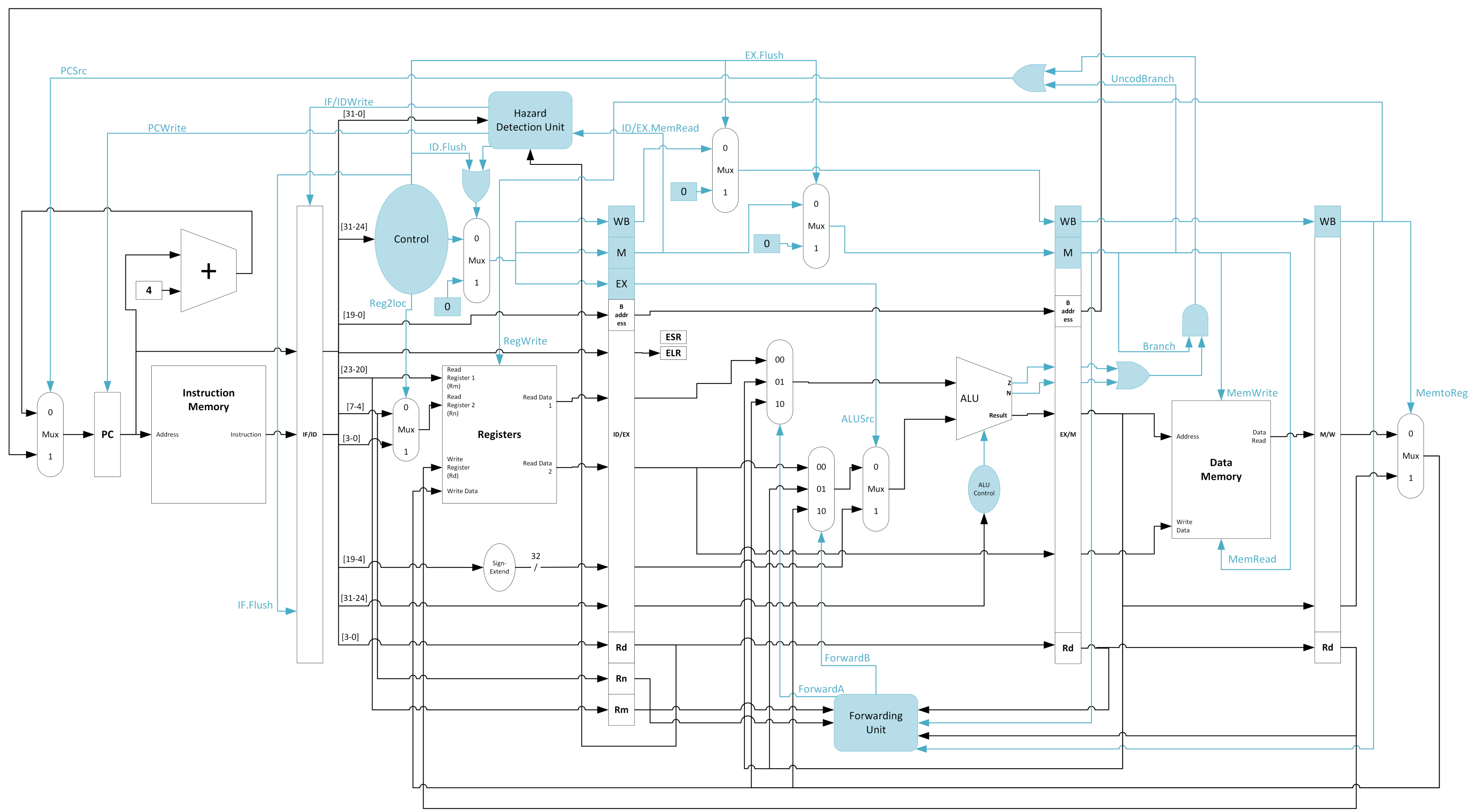
Appendices



ALU Design



Single Cycle Architecture



Pipelined Architecture