Project Plan Report for EC772, Spring 2016

21 February 2016

Mahesh Bhutada

Darrin Ginoza

Omkar Nagendra

Kiran Vishal

By

Multicycle Cpu with approximate alu

**Abstract**

Our focus for this project was to construct a multicycle CPU with a 32-bit instruction set, while incorporating an approximate ALU. The CPU accepts both I-type and R-type instructions and can perform operations such as addition, subtraction, multiplication, and logical operations. Among the operations specified, multiplication consumes the most power for computation and time. For our project, we will focus on significantly improving the multiplication operation. During the multiplication computation, we plan to ignore a certain amount of bits of the multiplicands. We then concatenate zeros at the product to obtain our approximate result. We have estimated our worst case percentage error to be about 3% for a 16 bit multiplicands.

Contents

1. Introduction 1

1.1 Section head 1

2 Design 2

2.1 [Component or Block] 2

2.1.1 [Subcomponent or subblock] 2

3. Design Verification 3

3.1 [Component or Block] 3

3.1.1 [Subcomponent or subblock] 3

4.1 Labor 3

5. Conclusion 3

5.1 Accomplishments 3

5.2 Uncertainties 3

5.3 Ethical considerations 3

5.4 Future work 3

References 3

Appendix A Requirement and Verification Table 3

# 1. Introduction

The CPU is a standard construct with a 32-bit instruction set, accepting both I-type and R-type instructions. We plan to modify the ALU to be approximate, with the goal of saving power and computation time.

## 1.1 Standard CPU with approximate ALU

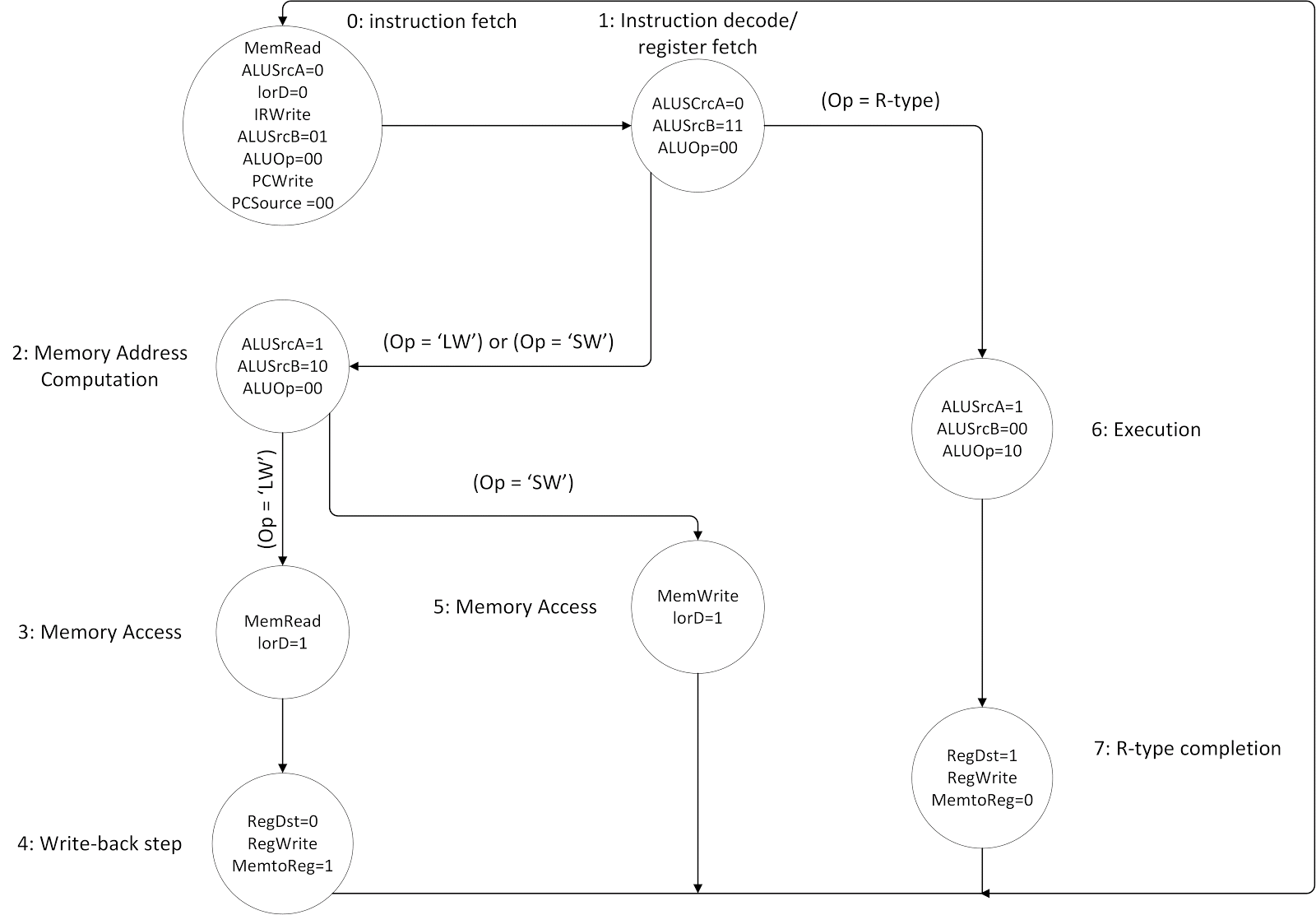
Our CPU will follow the flow in figure 1.

Figure 1 CPU state diagram without approximation

1. The instruction is pre-fetched.
2. Instruction is decoded to determine R-type(01) or I-type(10) or I-type arithmetic/logical (11). These instructions are from the first two bits of op-code.
3. Determines address of register, and determines either load word (LW) or store word(SW).
4. Memory access for load word
5. Actually loading the word
6. Memory access for store word
7. Execution of logic computation
   1. Op = 010000 : R1 = R2 -- MOVE
   2. Op = 010010 : R1 = R2 + R3 – ADD
   3. Op = 010011 : R1 = R2 - R3 -- SUBTRACT
   4. Op = 010100 : R1 = R2 | R3 -- OR
   5. Op = 010101 : R1 = R2 & R3 -- AND
   6. Op = 010110 : R1 = R2 \* R3 -- MULTIPLY

Same instructions for I-type (except first two bits = 11)

1. Write back answer to memory

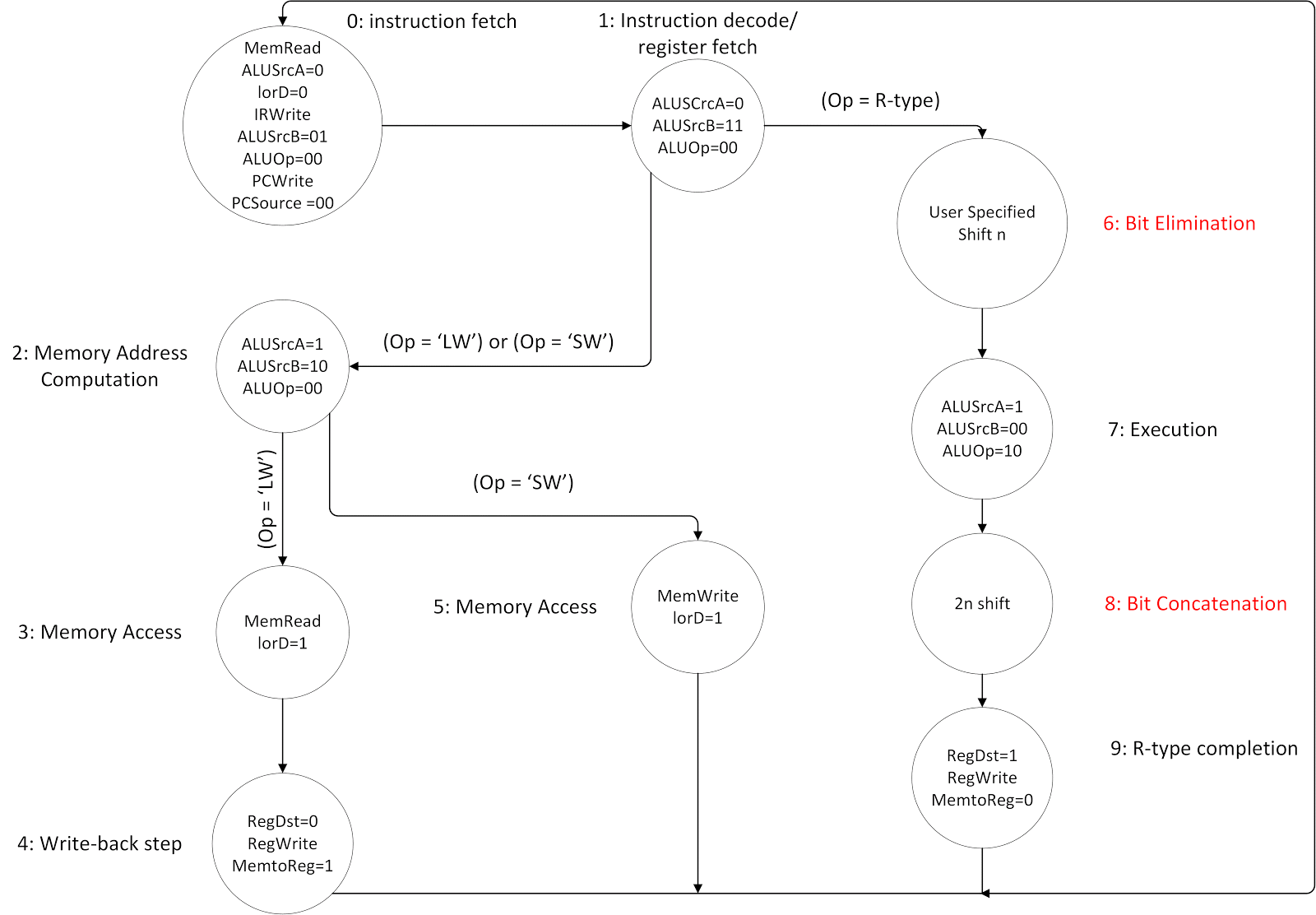


Figure 2 Modified CPU state diagram with addition of approximation

Figure 2 outlines the flow of the CPU with approximate ALU.

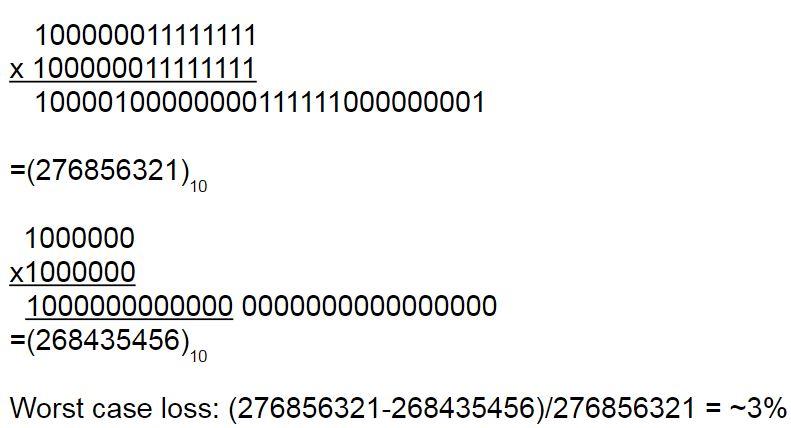
1. The instruction is pre-fetched.
2. Instruction is decoded to determine R-type(01) or I-type(10) or I-type arithmetic/logical (11). These instructions are from the first two bits of op-code.
3. Determines address of register, and determines either load word (LW) or store word(SW).
4. Memory access for load word
5. Actually loading the word
6. Memory access for store word
7. Eliminate user specified number of bits for approximation
8. Execution of logic computation
   1. Op = 010000 : R1 = R2 -- MOVE
   2. Op = 010010 : R1 = R2 + R3 – ADD
   3. Op = 010011 : R1 = R2 - R3 -- SUBTRACT
   4. Op = 010100 : R1 = R2 | R3 -- OR
   5. Op = 010101 : R1 = R2 & R3 -- AND
   6. Op = 010110 : R1 = R2 \* R3 – MULTIPLY
   7. Op = 010111 to 011111 : SHIFT THEN MULTIPLY

Same instructions for I-type (except first two bits = 11)

1. Concatenates zeros to get 32 bit approximate product
2. Write back answer to memory

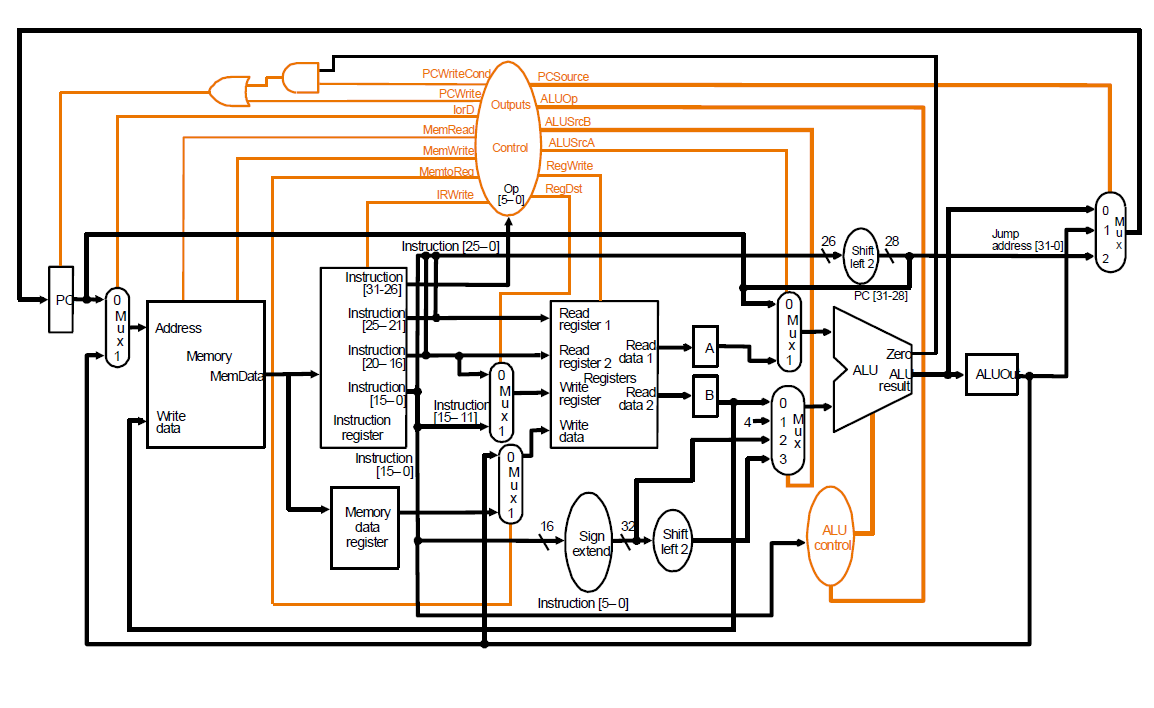
## 1.2 Approximating Multiplication

Our approach to approximating computation was to focus on the multiplication operation. Multiplication not only takes up the most computation time, but also consumes a lot of power. Our proposed method will reduce the power used as well as cut down on computation time, while sacrificing accuracy. The user will be able to choose the resolution of approximation, which is simply how many bits to ignore in the multiplication computation. For example, if we have 16-bit multiplicands, the user can specify that 8-bits be ignored, as in figure 3. After the computation, zeros are concatenated at the end of the product to form the approximate 32-bit product. For the 16-bit example, the worst case error was observed to be about 3%.

Figure 3 Example of approximate multiplication

# 2 Design

The block diagram for our multicycle CPU can be observed in figure 4. Refer back to figure 1 for explanation of the block diagram. Our modification to this design to incorporate approximation will be centered around the ALU.

 Figure 4 Block diagram for multicycle CPU[1]

## 2.1 [Component or Block]

Figure 5 describes our proposed design implementation of approximate multiplication. There will be two n-bit shift registers for the multiplicands, where the user specifies how many bits will be ignored. The modified multiplicands are sent to the ALU for computation, and then the product is sent to a 2n-bit shifter to concatenate 2n zeros to form the approximate product.

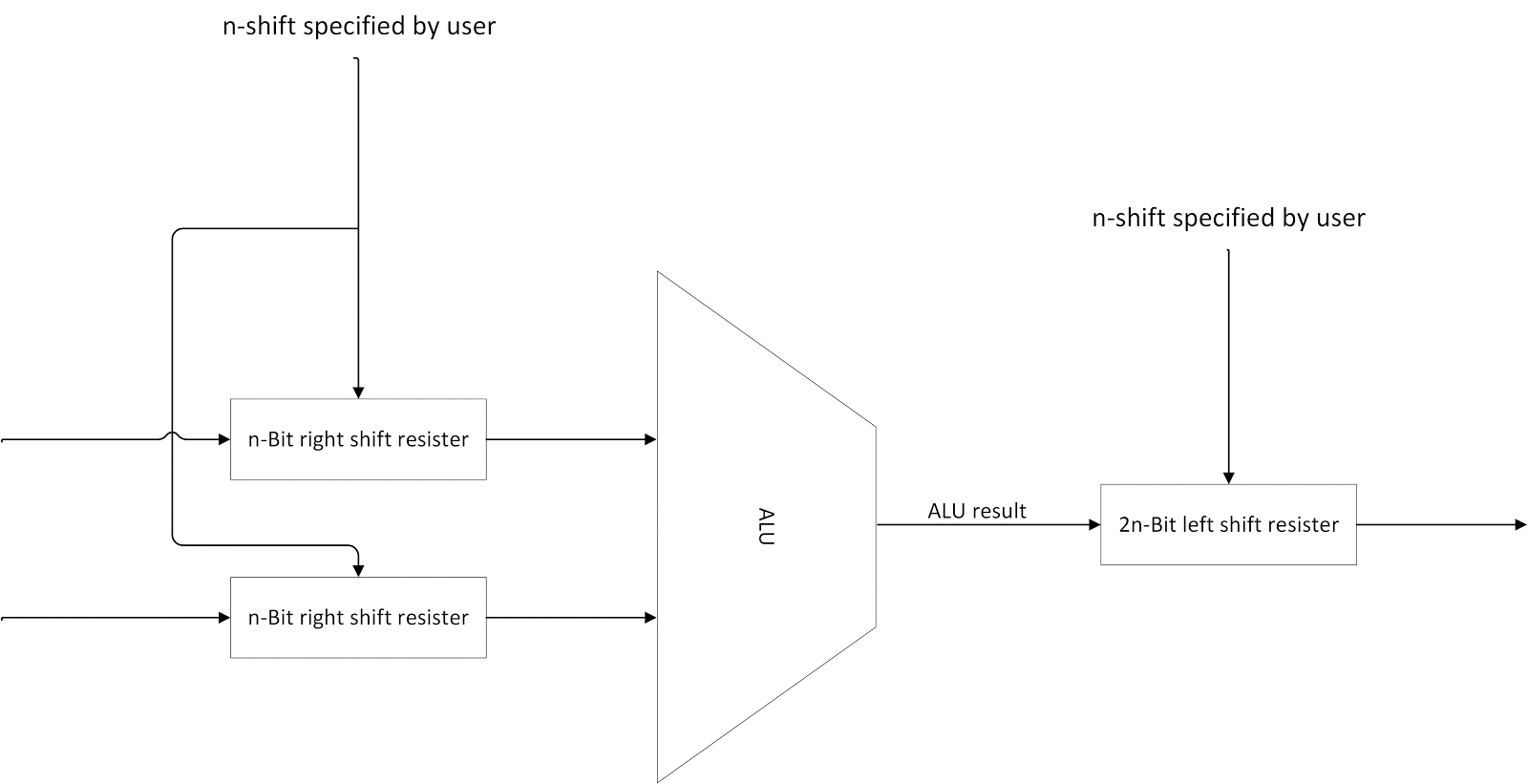


Figure 5 Block diagram of proposed approximate ALU

# 3. Design Verification

TBD

## 3.1 [Component or Block]

TBD

### 3.1.1 [Subcomponent or subblock]

TBD

## 4.1 Labor(tentative)

Regular CPU:

VERILOG: Mahesh, Darrin, Kiran, Omkar

CADENCE TOOLFLOW: Mahesh, Darrin, Kiran, Omkar

Approximate CPU

VERILOG: Darrin, Omkar

CADENCE TOOLFLOW: Mahesh, Kiran

# 5. Conclusion

TBD

## 5.1 Accomplishments

TBD

## 5.2 Uncertainties

TBD

## 5.3 Ethical considerations

TBD

## 5.4 Future work

TBD

# References

[1] EC551, Prof. Martin C Herbordt

# Appendix A Requirement and Verification Table

TBD