**Amar Kumar-CED17I029**

# Pipelining of Half Precision Floating Point adder

## Introduction

Floating-point adders are critically important components in modern microprocessors and digital signal processors. The architectures developed so far for floating point adders are based on sequence of significant operations: Swap, shift, add, and normalize. Due to these operations, the overall process of addition slows down. Floating-point adders must be fast to match with the increasing clock rates demanded by deep sub-micron technologies, also they must be small for being used in parallel processing systems. Since, in the traditional adders, all the stages were performed with single clock cycle, but the frequency of this clock was restricted due to the circuit constraints. Hence, whenever, the addition of a large number of values was performed, the traditional floating-point adders proved to be inefficient. This latency could be overcome if the concept of pipelining in the simple adder is introduced.

## Half Precision Floating Point Representation

The IEEE half precision floating point standard representation requires a 16 bit word, which may be represented from 0 to 15, left to right. The first bit is the sign bit, S, the next five bits are the exponent bits, 'E', and the final 10 bits are the fraction 'F' i.e. also known as mantissa:

```
S  EEEEE  FFFFFFFFFF
0  1      5  6          15
```

## Floating Point Algorithm

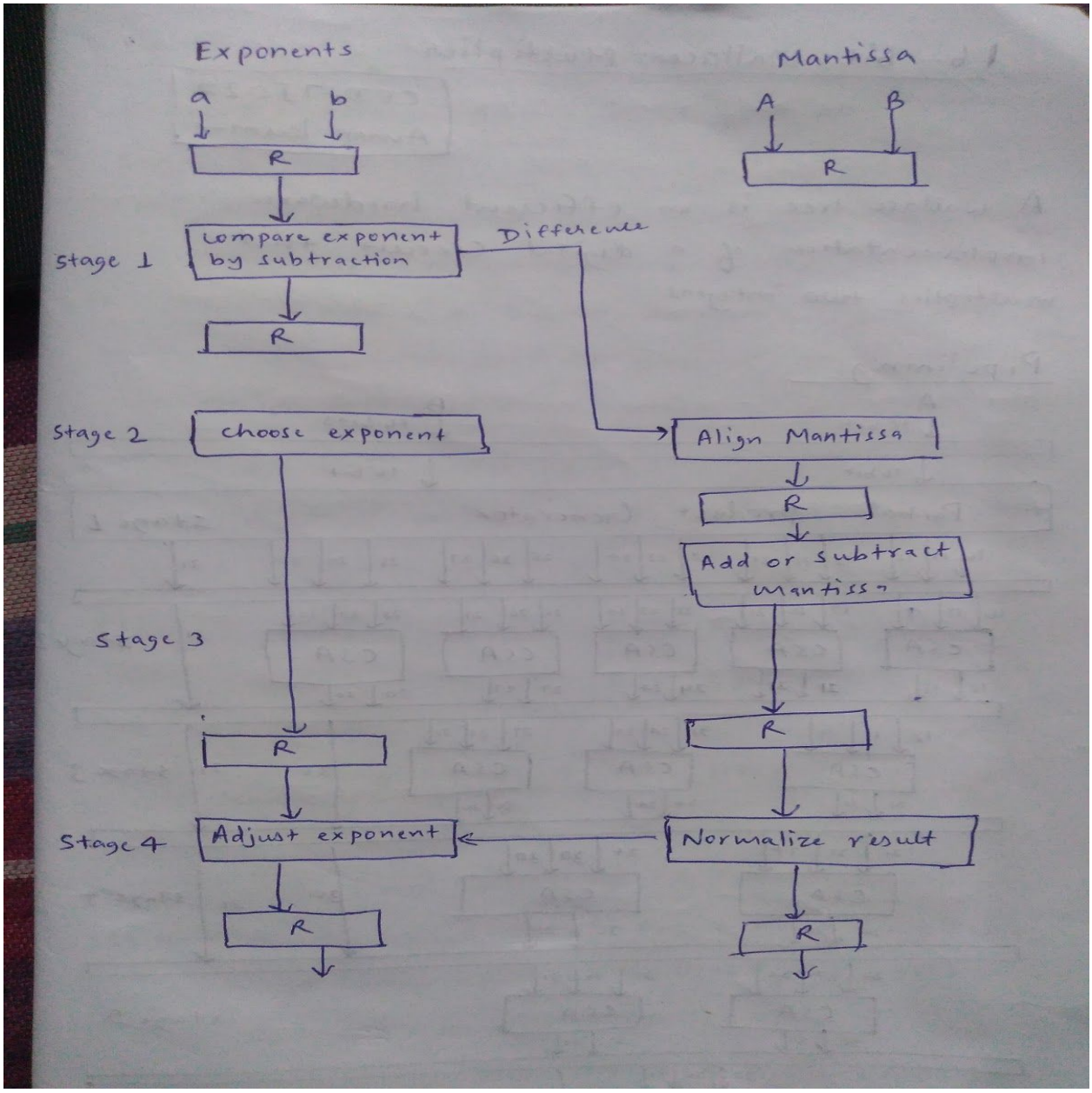
The floating-point representation of a number includes three fields: Sign bit (Si,), exponent (Ei,)and the mantissa (Fi). The algorithm for floating point addition/subtraction consists of the following steps:

(i) Load the inputs and compares the exponents and generates the exponent difference.
(ii) Align the mantissas (right shift the significant of the smaller operand)
(iii) Add or subtract the mantissas
(iv) Normalize the results

## Pipelining

I have four stages of pipelining in half precision floating point adder.

**First stage of the pipeline**. It performs the function of loading the operands, compares the exponents and generates the exponent difference.
**Second stage in the pipeline**. Its function is to shift the mantissa according to the exponent difference value generated in the previous stage.
**Third stage in the pipeline**. It performs the basic addition or subtraction as required.
**Fourth stage in the pipeline**. Its function is to convert the result into normalized form.



## Logic components

(assume delay of the gates as following:- NOT-1D, AND-1D, OR-1D, XOR-2D)

In the first stage, we are performing exponent difference.
Let us suppose we are performing A-B. For this we need 5 NOT gate to complement B, then add 1 to B to make B's complement and then add A and B to find the result of A-B.
For addition i am using 16bit recursive doubling, i require a maximum of 64 AND gate which will run parallely, 32 OR gate which will run parallely and 32 XOR gate which will also run parallely according to my code for recursive doubling in VLSI lab.
Since all 64 NAND gates, 32 OR gates and 32 XOR gates are running parallely so we are taking delay of only one gate.
For N bit number we have log2(N) stages in recursive doubling algorithm. So for 16 bit number we will have 4 stages hence we are taking 4D for AND gate, 4D for OR gate and 4*2D=8D for XOR gate. So a total of 16D for addition of two 16bit number using recursive doubling.
In first stage, we are using recursive doubling two times(firstly for addition of 1 and secondly for final addition to find A-B)
So total delay in first stage is 1D for NOT gate since all five NOT gate are running parallely, 16*2 = 32D for two times recursive doubling addition. Hence a total of 33D.

In second stage, we are shifting mantissa according to the difference of exponent. Maximum difference can be 31. So i need a shifting logic.
For shifting i am using 2x1 multiplexer which has a delay of 1D for NOT gate, 1D for two AND gate(since both running parallely) and 1D for OR gate. So total of 3D for a 2x1 multiplexer. Maximum delay can be 5*3 = 15D(we have 5 stages in 32bit shifter using mux; for shifting 1bit,2bit,4bit,8bit,16bit each using 32 2x1 mux in each stages and all 32 mux will run in parallel). Hence Total delay is 15D.

In third stage, we are performing addition of mantissa using recursive doubling.
So total delay for this stage will be the delay for recursive doubling which is 16D.

In fourth stage, we are normalizing the result.
For this we may need to shift mantissa by 1 bit and add 1 to the exponent(if addition of mantissa produces a carry otherwise no need to normalize).
For shifting by one bit we are using sixteen 2x1 multiplexer parallely so it will have a delay of 3D.For addition of 1 to exponent we are using 16 bit recursive doubling so 16D for addition. Hence a total of 19D may require in fourth stage

## Comparison of pipelined and non-pipelined 16 bit half precision floating point adder

Let us assume that we have 1000 instructions to add.
In pipelined version, first instruction will complete after 83D and after that every instruction will take only a delay of 33D to perform half precision fp addition. So overall, it has a delay of 33D*1000 on an average which is 33000D(accurate delay is 33D*999 + 83D*1 = 33050D).
In non-pipelined version, each instruction will take a delay of 83D. Hence for 1000 instructions, a total delay will be 83000D

## % of Improvement can be achieved with respect to non-pipelined architecture.

For 1000 instructions, pipelined architecture was taking 33000D delay on an average and non-pipelined architecture was taking 83000D delay.
Hence improvement using pipelined architecture is 83000D - 33000D = 50000D
So %improvement using pipelined architecture is $\frac{83000}{33000} x\ 100 \approx 251\%$