DESIGN AND IMPLEMENTATION OF THE SNAP FLOATING-POINT ADDER

Nhon Quach and Michael Flynn

Technical Report: CSL-TR-91-501

Dec. 1991

This work was supported in part by an IBM Graduate Fellowship and by NSF contract No. MIP88-22961.

DESIGN AND IMPLEMENTATION OF THE SNAP FLOATING-POINT ADDER

by

Nhon Quach and Michael Flynn

Technical Report: CSL-TR-91-501

Dec. 1991

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

This paper describes the design and implementation of the floating-point adder in the Stanford Nanosecond Arithmetic Processor (SNAP). The adder is capable of adding two double precision IEEE numbers in less than $20ns$ nominal with all IEEE rounding modes. Only round to nearest is described in this paper, however. The adder has been laid out in the HP CMOS26 1μm process with triple-layer metal, occupying a silicon area of roughly $14.5mm^2$.

The SNAP adder is smaller yet faster than existing floating-point adders in the literature because (a) it has a fast modified Ling integer adder, (b) it uses a fast leading-one prediction circuit, and (c) it uses a rounding method that allows a reduction in the number of operations required in the critical path. Logically, the adder still uses a two-path approach as suggested by Farmwald, but it merges the two paths to reduce the number of adders, thereby saving hardware.

The modified Ling integer adder has been shown to have a speed advantage over other adder organizations in the context of FP addition because (a) the carry out path in the adder is sped up and (b) IEEE FP numbers have a hidden one bit. This means that the bit propagate of the most significant bit in the Ling carry lookahead equation is always one for normalized numbers and therefore needs not be implemented.

Key Words and Phrases: High-speed floating-point adder, leading-one prediction, Modified Ling's adder, IEEE rounding

Copyright © 1996

by

Nhon Quach and Michael Flynn

Contents

1 Introduction **1**

2 Operation of the adder **1**

3 Design and Analysis of the Adder **3**

3.1 Notation 5
3.2 Significand swapper (*swapper*) 5
3.3 Absolute exponent adder (*ExpAdd*) 6
3.4 Absolute exponent muxes (*MuxAbs*) 6
3.5 Exponent adjust adder input selection muxes (*MuxEf*) 6
3.6 Exponent adjust adder input and result selection logic (*ELOGIC* and *CinEA*) 6
3.7 Exponent adjust adder (*ExpAdj*) 7
3.8 Sign logic (*SIGN*) 8
3.9 Path selection logic (*PSL*) 8
3.10 Path selection logic (*PSL*1) 8
3.11 Alignment Shifter and sticky bit logic (*Rs*55 and *SB*) 9
3.12 Significand adder input selection muxes (*Mux*53*tto*) 12
3.13 Significand adder (*ComAdd*) 12
3.13.1 Logic Block Diagram and Naming Convention 13
3.14 Normalize amount predictor or leading-one predictor (*LOP*) 18
3.15 Normalize amount encoder (*ENC*) 19
3.16 Round logic (*GRS*) 19
3.17 One's complementor (*OneCom*) 20
3.18 Normalization shift (*Ls*55) 21
3.19 Fine adjustment shift muxes (*MuxFA*) 21
3.20 Final result selection muxes (*Mux*53*fto*) 21

4 Simulation **21**

5 Implementation **22**

5.1 Area Estimate 22
5.2 Delay Estimate 23

6 Modification for Single-Precision Operation **24**

7 Pipelining the Adder **26**

8 Summary **26**

9 Acknowledgements **26**

List of Figures

1 The SNAP FP Adder. 4
2 Block diagram and naming convention for the significand adder in the SNAP adder 13
3 Circuit diagram of a 3-bit group 14
4 The process of carry generation in the SNAP FP adder. 15
5 Group generate circuits for groups other than Group 1 16
6 Group generate circuit for Group 1 with Cin=1 17
7 Block generate circuit 18
8 Circuit for GRS 20
9 Functional Simulation of the SNAP FP adder 22
10 A 2-stage pipelined implementation of the SNAP FP adder 27

List of Tables

1 Possible adjustments for the exponent 7
2 Control signals for computing the sticky bit 10
3 Partition of the Sticky Bit Logic Equations 11
4 Breakdown of area consumption 23
5 Simulated Delay 24

1 Introduction

The Stanford Nanosecond Arithmetic Processor (SNAP) project is an on-going research activity aiming at extending the current state-of-the-art in floating-point (FP) technology through efforts at all levels, including packaging, circuit, logic, architecture, algorithm, and CAD. The demonstration vehicle is an FP coprocessor of multiple chips, built with such diverse technologies as ECL, CMOS, BiCMOS, GaAs, etc. These chips are attached through surface tension of a fluid to an active substrate, which provides not only mechanical support and cooling, but also electrical connection. The chip set computes the primitive functions required by the IEEE 754 standard [2] — add, multiply, divide, remainder, and square root. High-order transcendental functions are computed using the CORDIC algorithm [1, 5]. The detail and the scope of the project have been described by Flynn *et al.* [6]; this paper describes the FP adder in the chip set.

The SNAP FP adder has the following features. First, it uses a fast rounding algorithm that reduces the number of operations in the critical path. The number of significand addition has been reduced to one from two in existing algorithms. It uses a modified Ling adder that speeds up the carry out path. Finally, it uses a slightly different leading one prediction (LOP) method from that used in the RS/6000. We believe that our LOP circuit consumes less hardware and is faster than the RS/6000 one. The net result is that the SNAP FP adder is smaller, yet faster than the existing ones reported in the literature. This paper focuses more on the implementation aspects of the adder. The reader is referred to [9] for a fuller discussion on the algorithm.

The remainder of this paper is organized as 7 sections. Section 2 describes the general operation of the adder. Section 3 presents the components in the adder in greater detail. Section 4 describes the functional simulator we have developed for the adder. In this simulator, all IEEE rounding modes were simulated. Section 5 addresses the implementation issues (e.g., layout, SPICE issues, etc.) of the adder. Most FP adders support both single and double precision operations and are usually pipelined to increase the throughput of the unit. In Section 6, we explain the modification needed for single-precision operations and show a way to pipeline the adder in Section 7. We then summarize in the final section. This paper was used as a design manual during the implementation stage of the adder and as such takes the form of a manual.

2 Operation of the adder

The SNAP adder uses the following fact to speed up its operation. When a massive right shift is needed during alignment, the result may need a left shift of at most 1 bit to normalize. On the other hand, when a right shift of 1-bit or less is needed for alignment, the result may require a massive left shift. Consequently, the major operations in the critical path of the adder consist of either a massive right shift followed by a significand add (the SA path) or a significand add followed by a left shift (the AS path).

This two-path operation has been known for quite some time [7] and has been used in most existing high-speed FP units in the industry [4, 3]. The SNAP adder extends the

algorithm in two ways. First, it doesn't have an explicit rounding step; the rounding step is performed with the significand addition step. Second, the SNAP adder has only one significand adder as opposed to two in the other FP units.[1]

Fig. 1 is a block diagram of the adder, which works as follows. In the exponent path, the exponent adder computes the absolute difference of the exponents, E_s, for the alignment shifter. The larger exponent is selected as the input into the exponent adjust adder. During normalization, the significand may need a right shift, no shift, or a variable number of left shift. The exponent adjust adder must be prepared to deal with all of these cases, the challenge being to manipulate the logic equations such that the exponent path does not become a critical one.

In the significand path, the operands are swapped as needed depending on the result of the exponent subtraction step. The *path_selection_unit* examines E_s to select the correct input. The inputs into the significand adder are: the significand of the larger operand and one of the three versions of the aligned significand of the smaller operand: unshifted, right shifted by 1 bit, and right shifted by many bits. The shifted by many bits version is provided by the right shifter, which also computes information needed for the sticky bit. We chose to swap the operands because it reduces the number of alignment shifters needed in the FP adder. An added benefit of this arrangement is that the larger operand is always normalized, allowing a simplification in the significand adder as we shall see.

The FP adder handles both subtraction and addition. Depending on the signs of the operands, an addition called for in an instruction may actually be performed as a subtraction. The actual operation performed by the adder is called the effective operation. During an effective subtraction step, the significand of the smaller operand is complemented.

The significand adder computes all possible outcomes needed to obtain the final result, normalized and rounded. In general, the result may require a rounding step or a complementation step (as explained below). Because of the possible right shift during normalization, the rounding step may require adding up to 2 *ulp*,[2] requiring 3 outcomes be computed in the significand adder for fast rounding. For the round to nearest mode, we were able to reduce the number of outcomes to two: $A + B$, and $A + B + 1(ulp)$.

The adder is a Ling type adder modified for CMOS technology [8]. Briefly, the idea of a Ling adder is to propagate a simplified carry to reduce the complexity of the group generate circuitry in the carry lookahead tree. Normally, the local sum logic has to recover the true carry from this simplified one at a cost of some hardware and delay. The modified Ling adder avoids these penalties through use of a clever organization. Further, because the larger operand is always normalized, the bit propagate at the most significant bit is one. This means that the factored p_i term in the Ling carry lookahead equation needs not be implemented.[3]

Though the significand adder computes both $A + B$ and $A + B + 1(ulp)$, only the carry chain needs to be duplicated as in a conditional sum adder. The outputs of the significand adder will be appropriately selected by the *GRS* unit to account for rounding

[1] Only in a 2-stage pipelined implementation does the unit require 2 adders. This is due to a resource conflict as pointed out in a later section. A 3-stage pipelined implementation requires only one adder.

[2] *ulp* stands for unit in the last place.

[3] Extra logic is needed to handle denormalized numbers.

or complementation. The GRS unit also computes the true guard, round, sticky bits, and the bit to be left shifted into the result during normalization. When the operands are close in magnitude, the result may need a massive left shift for normalization or a negation (for a negative result), or both. This gives rise to the complementation case mentioned above. GRS must also handle these cases. The development of the logic equation for GRS has been described in Quach and Flynn [9].

To remove an otherwise critical path during alignment, the amount of left shift is predicted by the LOP unit based on the inputs into the significand adder. The LOP predicts the shift amount to within one bit, which is later made up by an additional 1 bit fine adjustment shift. There are two ways to perform this fine adjustment shift. The first way is to predict such a case from the inputs into the LOP unit. This approach requires some hardware. The second approach is to simply wait for the result from LOP to arrive and, by examining the most significant bit of the result, the need for a 1-bit fine adjustment shift can be determined. This approach consumes no extra hardware, but pays a price in latency because operations are now serialized. SNAP uses the second approach and an LOP circuit that is similar in operation to the one used in the IBM RS/6000 processor.

Finally, the four possible results are selected to yield the final, rounded and normalized result. These possibilities arise because the result may need a 1-bit right shift, no shift, a 1-bit left shift, and a left shift of many bits. The control circuitry for selecting among these results depends on the following parameters: the difference in the exponent, the carry-out bit and the most significant bits of the results computed in the significand adder, the effective operation, and the round bits.

The overall control of the FP adder is simpler than that reported in [9]. In the following section, we go through each component in the adder in a greater detail.

3 Design and Analysis of the Adder

The FP adder consists of the following components:

- Significand swapper ($swapper$)
- Absolute exponent adder ($ExpAdd$)
- Absolute exponent muxes ($muxAbs$)
- Exponent adjust adder input selection muxes ($muxEf$)
- Exponent adjust adder input logic ($ELOGIC$)
- Exponent adjust adder results select logic ($CinEA$)
- Exponent adjust adder ($ExpAdj$)
- Sign logic ($SIGN$)
- Paths selection logic ($PSL1$ and PSL)

Figure 1: The SNAP FP Adder.

- Alignment shifter and sticky bit logic ($Rs55$ and SB)
- Significand adder input selection muxes ($mux53tto$)
- Significand adder ($ComAdd$)
- Normalize amount predictor or leading one predictor (LOP)
- Normalize amount encoder (ENC)
- Round logic (GRS)
- Normalization shifter ($Ls55$)
- One's complementor ($OneCom$)
- Fine adjustment muxes ($muxFA$)
- Final result selection muxes ($mux53fto$)

Referring again to Fig. 1, the significand result selection muxes ($MuxCA$) has been explicitly shown in the figure. In the current implementation, $muxCA$ is embedded in the adder. We describe each unit in more detail below. But first, an explanation on notation is in order.

3.1 Notation

$X\langle i\rangle$ denotes the ith bit and $\sim X$ the bit inversion of a bit vector X. $\overline{X}$ is the logical inversion of a logic signal X and X_c the result of an addition with a carry in c where $c \in [0,1]$. $X.Y$ is the Y bit of a quantity or a computation X. $OP(X,Y)$ is the logical operation OP of X and Y, where OP is one of the boolean operations. E_a, E_b, and E_f are the exponents; S_a, S_b, and S_f the signs; and F_a, F_b, and F_f the significands of the operands and the results, respectively. For equations, $\vee$ is used for logical OR, juxtaposition or $\wedge$ for AND, and $\oplus$ for XOR. For notational convenience, X_s and Xs are used interchangeably when there is no ambiguity. E_s and Es, for example, represents the same signal.

3.2 Significand swapper ($swapper$)

- *Inputs:* S_a, F_a, S_b, F_b, and C_{exp}
- *Outputs:* F_1 and F_2.
- *Function:* If C_{exp} then $F_1 = F_a$ and $F_2 = F_b$ else $F_1 = F_b$ and $F_2 = F_a$.
- *Explanation:* This unit accepts the 53-bit significands of the operands as inputs and depending on C_{exp}, which is the carry-out from the absolute exponent adder, performs a swap.

3.3 Absolute exponent adder ($ExpAdd$)

- *Inputs:* E_a and E_b.
- *Outputs:* E_s_0, E_s_1, and C_{exp}.
- *Function:*

$$E_s_0 = E_a + \sim E_b$$

$$E_s_1 = E_a + \sim E_b + 1$$

 and

$$C_{exp} = E_s_0.C_{out}$$

- *Explanation: ExpAdd* computes the two results needed for determining the absolute difference of the exponents so that the shift amount can be quickly determined. This adder computes both results (i.e., $A + B$ and $A + B + 1(ulp)$) because when the result is negative (i.e., when $C_{exp} = 0$), it needs to select the inverse of the $A+B(ulp)$ result.

3.4 Absolute exponent muxes ($MuxAbs$)

- *Inputs:* E_s_0, E_s_1, and C_{exp}.
- *Outputs:* Es.
- *Function:* If C_{exp}, $Es = E_s_1$ else $Es = \sim E_s_0$.
- *Explanation:* $MuxAbs$ selects the result of the exponent subtraction step. When $C_{exp} = 1$, the result is positive, hence Es_0 is chosen. Otherwise, the result is negative, and we need to select Es_1 and bit invert it to obtain a positive result.

3.5 Exponent adjust adder input selection muxes ($MuxEf$)

- *Inputs:* Ea, Eb, and C_{exp}.
- *Outputs:* E_{f1}.
- *Function:* If C_{exp}, $E_{f1} = E_b$ else $E_{f1} = E_a$.
- *Explanation:* $MuxEf$ selects the exponent of the larger operand.

3.6 Exponent adjust adder input and result selection logic ($ELOGIC$ and $CinEA$)

- *Inputs:* E_{LOP}, Eo, C_{fine}, NXS, MLS, and ORS.
- *Outputs:* E_{adj} and C_{in_EA}
- *Function: see explanation.*

Table 1: Possible adjustments for the exponent

$Case$	E_f	E_{adj}	Cin_EA
MLS	$E_{f1} - E_{LOP}$	$\sim E_{LOP}$	1
MLS	$E_{f1} - E_{LOP} - 1$	$\sim E_{LOP}$	0
OLS	$E_{f1} - 1$	All 1's	0
NXS	E_{f1}	All 1's	1
ORS	$E_{f1} + 1$	All 0's	1

- *Explanation:* $ELOGIC$ computes E_{adj}, the amount of adjustment to be added to E_{f1} to obtain the final exponent. There are five possibilities: (a) $E_{f1} - E_{LOP}$ for the case of a massive normalization shift, (b) $E_{f1} - E_{LOP} - 1$ for the same case as (a) with a fine adjustment shift, (c) $E_{f1} - 1$ for the case of a 1-bit left shift, (d) E_{f1} for the case of no shift, and (e) $E_{f1} + 1$ for the case of a 1-bit right shift. The unit takes care of these cases as shown in Table 1

  In the table, E_{adj} is the output of the $ELOGIC$ unit. The logic equations for E_{adj} and Cin_EA can be determined from the table as

$$E_{adj}\langle i\rangle = (Eo \wedge \overline{MLS}) \vee (MLS \wedge \overline{E_{LOP}\langle i\rangle})$$

  and

$$C_{in_EA} = ORS \vee (Eo \wedge NXS) \vee (MLS \wedge \overline{C}_{fine})$$

  where C_{fine} is the fine adjustment signal. Note that in the case of NXS, we first subtract and later add one to E_{f1} to avoid having NXS or OLS in the E_{adj} equation, because these signals are known late in the significand addition step and may cause additional delay in the SA path.

3.7 Exponent adjust adder ($ExpAdj$)

- *Inputs:* E_{f1}, E_{adj}, and Cin_EA.
- *Outputs:* E_f.
- *Function:*

$$Ef_0 = E_{f1} + E_{adj}$$

  and

$$Ef_1 = E_{f1} + E_{adj} + 1$$

  if Cin_EA then $E_f = Ef_1$ else $E_f = Ef_0$

- *Explanation:* $ExpAdj$ is the same as $ExpAdd$. In principle, the speed requirement on this adder is not as critical and a less hardware intensive, and slower adder, such as a carry-skip one, can be used to save hardware. This optimization is not performed in the current implementation due to time constraint.

3.8 Sign logic ($SIGN$)

- *Inputs:* Sa, Sb, C_{exp}, and C_{ca}.
- *Outputs:* S_f.
- *Function:*

$$S_f = (C_{exp} \vee \overline{C_{ca}})S_a \vee (\overline{C_{exp}}C_{ca})S_b;$$

- *Explanation:* The equation seems complicated because the sign may be affected by both C_{exp} and C_{ca}. We must consider the following cases: When $C_{exp} = 1$, we know that A is larger than B. But when $C_{exp} = 0$, all we can say is $A \leq B$ and the result from the significand adder may be negative. The equation is developed with this in mind.

3.9 Path selection logic (PSL)

- *Inputs:* E_s.
- *Outputs:* ns, os, and ms.
- *Function:*

$$E_{s09} = AND(\overline{E_s\langle 0:9\rangle})$$
$$E_{s10} = \overline{E_s\langle 10\rangle}$$
$$ns = E_{s09} \wedge E_{s10}$$
$$os = E_{s09} \wedge \overline{E_{s10}}$$
$$ms = \overline{E_{s09}}$$

- *Explanation:* PSL controls the inputs into the significand adder on the smaller operand side. It examines E_s to determine these conditions.

3.10 Path selection logic ($PSL1$)

- *Inputs:* E_{s09}, E_{s10}, E_o, $F2_ms\langle 53:55\rangle$, and C_{rs}.
- *Outputs:* MLS, ORS, NXS, OLS, b_n, b_{n+1}, and s.

- *Functions:*

$$b_n = \overline{E_{s09}} \wedge Fm\langle 53\rangle \vee E_{s09}\overline{E_{s10}}C_{rs}$$

$$b_{n+1} = \overline{E_{s09}} \wedge Fm\langle 54\rangle$$

$$s = \overline{E_{s09}} \wedge Fm\langle 55\rangle$$

$$ORS = \overline{E_o}C_{ca}$$

$$NXS = \overline{E}_o\overline{C}_{Ca} \vee E_o\overline{E_{s09}}\left[\overline{Fs_1\langle 0\rangle}(\overline{b_n \vee b_{n+1} \vee s}) \vee Fs_0\langle 0\rangle(b_n \vee b_{n+1} \vee s)\right]$$

$$MLS = E_oE_{s09}$$

and

$$OLS = E_o\overline{E_{s09}}\left[\overline{Fs_1\langle 0\rangle}\,(\overline{b_n \vee b_{n+1} \vee s}) \vee \overline{Fs_0\langle 0\rangle}(b_n \vee b_{n+1} \vee s)\right]$$

where $Fm\langle 55\rangle$ is the logical OR of all bits beyond the round bit.

- *Explanation:* $Fm\langle 55\rangle$ is the sticky bit computed by the shifter. Note that only the ORS, NXS, and OLS bits depend on the results from the significand adder. This means that MLS arrives early, allowing the $ELOGIC$ and $ExpAdj$ units to get ahead of the significand adder.

3.11 Alignment Shifter and sticky bit logic ($Rs55$ and SB)

- *Inputs:* $F2$ and E_s.
- *Outputs:* $F2_ms$
- *Functions:*

$$F2_ms = F2 \gg Es$$

- *Explanation:* $Rs55$ shifts $F2$ by E_s bits. SB determines the sticky bit. The guard and round bits are the least significant bits from $Rs55$. Together, these bits allow a proper determination of the b_n, b_{n1}, and the s bits, which are the final guard, round, and sticky bits.

  The shifter has 2 stages. The first stage performs shifts of a 8-bit multiple (0, 8, 16, 24, 32, 40, and 48) and the second stage a 1-bit multiple (0, 1, 2, 3, 4, 5, 6, and 7).

  An alternative design would be to use a 3-stage shifter. The first stage performs shifts of 0, 1, 2, or 3 bits; the second stage performs shifts of 0, 4, 8, or 12 bits; and the last stage performs shifts of 0, 16, 32, or 48 bits. But the two-stage shifter has the advantage that it has a pitch that matches more closely to that of the significand adder than a 3-stage one does.

  The 55th bit of $Rs55$ is the sticky bit, determined by ORing the control signals in the SB unit as shown in Table 2.

  In the table, $A = E_s\langle 5\rangle$, $B = E_s\langle 6\rangle$, $C = E_s\langle 7\rangle$, $D = E_s\langle 8\rangle$, $E = E_s\langle 9\rangle$, $F = E_s\langle 10\rangle$. Note that the logic equations in this table does not detect the case when $E_s > 53$. In

Table 2: Control signals for computing the sticky bit

Shift Amount	Equation	Bit No.	Shift Amount	Equation	Bit No.
3	$A+B+C+D+EF$	52	30	$A+BCDE$	25
4	$A+B+C+D$	51	31	$A+BCDEF$	24
5	$A+B+C+D(E+F)$	50	32	A	23
6	$A+B+C+DE$	49	33	$A(B+C+D+E+F)$	22
7	$A+B+C+DEF$	48	34	$A(B+C+D+E)$	21
8	$A+B+C$	47	35	$A(B+C+D+EF)$	20
9	$A+B+C(D+E+F)$	46	36	$A(B+C+D)$	19
10	$A+B+C(D+E)$	45	37	$A[B+C+D(E+F)]$	18
11	$A+B+C(D+EF)$	44	38	$A(B+C+DE)$	17
12	$A+B+CD$	43	39	$A(B+C+DEF)$	16
13	$A+B+CD(E+F)$	42	40	$A(B+C)$	15
14	$A+B+CDE$	41	41	$A[B+C(D+E+F)]$	14
15	$A+B+CDEF$	40	42	$A[B+C(D+E)]$	13
16	$A+B$	39	43	$A[B+C(D+EF)]$	12
17	$A+B(C+D+E+F)$	38	44	$A(B+CD)$	11
18	$A+B(C+D+E)$	37	45	$A[B+CD(E+F)]$	10
19	$A+B(C+D+EF)$	36	46	$A(B+CDE)$	9
20	$A+B(C+D)$	35	47	$A(B+CDEF)$	8
21	$A+B[C+D(E+F)]$	34	48	AB	7
22	$A+B(C+DE)$	33	49	$AB(C+D+E+F)$	6
23	$A+B(C+DEF)$	32	50	$AB(C+D+E)$	5
24	$A+BC$	31	51	$AB(C+D+EF)$	4
25	$A+BC(D+E+F)$	30	52	$AB(C+D)$	3
26	$A+BC(D+E)$	29	53	$AB[C+D(E+F)]$	2
27	$A+BC(D+EF)$	28			
28	$A+BCD$	27			
29	$A+BCD(E+F)$	26			

Table 3: Partition of the Sticky Bit Logic Equations

Shift Amount	Equation	Bit No.	Shift Amount	Equation	Bit No.
3	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{1})$	52	30	$\sim[\overline{A}(\overline{B}+\overline{Q}_{12})]$	25
4	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{2})$	51	31	$\sim[\overline{A}(\overline{B}+\overline{Q}_{13})]$	24
5	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{3})$	50	32	A	23
6	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{4})$	49	33	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{14})$	22
7	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{5})$	48	34	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{15})$	21
8	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{6})$	47	35	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{1})$	20
9	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{7})$	46	36	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{2})$	19
10	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{8})$	45	37	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{3})$	18
11	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{9})$	44	38	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{4})$	17
12	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{10})$	43	39	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{5})$	16
13	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{11})$	42	40	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{6})$	15
14	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{12})$	41	41	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{7})$	14
15	$\sim(\overline{A}\,\overline{B}\,\overline{Q}_{13})$	40	42	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{8})$	13
16	$\sim(\overline{A}\,\overline{B})$	39	43	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{9})$	12
17	$\sim[\overline{A}(\overline{B}+\overline{Q}_{14})]$	38	44	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{10})$	11
18	$\sim[\overline{A}(\overline{B}+\overline{Q}_{15})]$	37	45	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{11})$	10
19	$\sim[\overline{A}(\overline{B}+\overline{Q}_{1})]$	36	46	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{12})$	9
20	$\sim[\overline{A}(\overline{B}+\overline{Q}_{2})]$	35	47	$\sim(\overline{A}+\overline{B}\,\overline{Q}_{13})$	8
21	$\sim[\overline{A}(\overline{B}+\overline{Q}_{3})]$	34	48	$\sim(\overline{A}+\overline{B})$	7
22	$\sim[\overline{A}(\overline{B}+\overline{Q}_{4})]$	33	49	$\sim(\overline{A}+\overline{B}+\overline{Q_{14}})$	6
23	$\sim[\overline{A}(\overline{B}+\overline{Q}_{5})]$	32	50	$\sim(\overline{A}+\overline{B}+\overline{Q_{15}})$	5
24	$\sim[\overline{A}(\overline{B}+\overline{Q}_{6})]$	31	51	$\sim(\overline{A}+\overline{B}+\overline{Q_{1}})$	4
25	$\sim[\overline{A}(\overline{B}+\overline{Q}_{7})]$	30	52	$\sim(\overline{A}+\overline{B}+\overline{Q_{2}})$	3
26	$\sim[\overline{A}(\overline{B}+\overline{Q}_{8})]$	29	53	$\sim(\overline{A}+\overline{B}+\overline{Q_{3}})$	2
27	$\sim[\overline{A}(\overline{B}+\overline{Q}_{9})]$	28			
28	$\sim[\overline{A}(\overline{B}+\overline{Q}_{10})]$	27			
29	$\sim[\overline{A}(\overline{B}+\overline{Q}_{11})]$	26			

this case, $s = 1$, always. Because this sticky bit logic is to be implemented in random logic, we like to reduce the type of logic gates using the partition in Table 3.
Where

$$\begin{aligned}
Q_1 &= \overline{C + D + EF} \\
Q_2 &= \overline{C + D} \\
Q_3 &= \overline{C + D(E + F)} \\
Q_4 &= \overline{C + DE} \\
Q_5 &= \overline{C + DEF} \\
Q_6 &= \overline{C} \\
Q_7 &= \overline{C(D + E + F)} \\
Q_8 &= \overline{C(D + E)} \\
Q_9 &= \overline{C(D + EF)} \\
Q_{10} &= \overline{CD} \\
Q_{11} &= \overline{CD(E + F)} \\
Q_{12} &= \overline{CDE} \\
Q_{13} &= \overline{CDEF} \\
Q_{14} &= \overline{C + D + E + F} \\
Q_{15} &= \overline{C + D + E}
\end{aligned}$$

In this partition, we only need to implement 15 types of gates as opposed to 53 in the orginal equations. For shift amounts from 3 to 15, Q_{12} is used; for 16, Q_{10} is used; from 17 to 31, Q_8 is used; for 32, Q_6 (which is an inverter) is used; from 33 to 47, Q_4 is used; for 48, Q_2 is used; and finally for 49 to 53, Q_{15} is used. The output of each gate is to be AND'ed with the input signal (which is to be shifted in the shifter).

3.12 Significand adder input selection muxes ($Mux53tto$)

- *Inputs:* $F2$, $F2_os$, $F2_ms\langle 0:52\rangle$, ns, os, and ms.

- *Outputs:* Fin.

- *Function:*
  Case ns $Fin = F2$
  Case os, $Fin = F2_os$
  Case ms, $Fin = F2_ms\langle 0:52\rangle$

- *Explanation:* These muxes selects among the unshifted, shifted by 1 bit, and shifted by many bits version of $F2$ based on E_s. If the first 10 bits are not equal to 0, we select $F2_ms\langle 0:52\rangle$. If they are all equal to 0, then the 11th bit decides between the shifted and the unshifted versions.

3.13 Significand adder ($ComAdd$)

- *Inputs:* E_o, F_1, Fin, and GRS.

- *Outputs:* Fs and C_{ca}.
- *Function:* If E_o then $Fin = \sim Fin$

$$Fs_0 = F_1 + Fin$$

$$Fs_1 = F_1 + Fin + 1$$

$$C_{ca} = Fs_0.C_{out}$$

If C_{GRS} then $Fs = Fs_1$ else $Fs = Fs_0$.

- *Explanation:* It is basically a 53-bit adder with a duplicate carry-lookahead chain. We describe the adder in greater detail in the following subsections.

3.13.1 Logic Block Diagram and Naming Convention

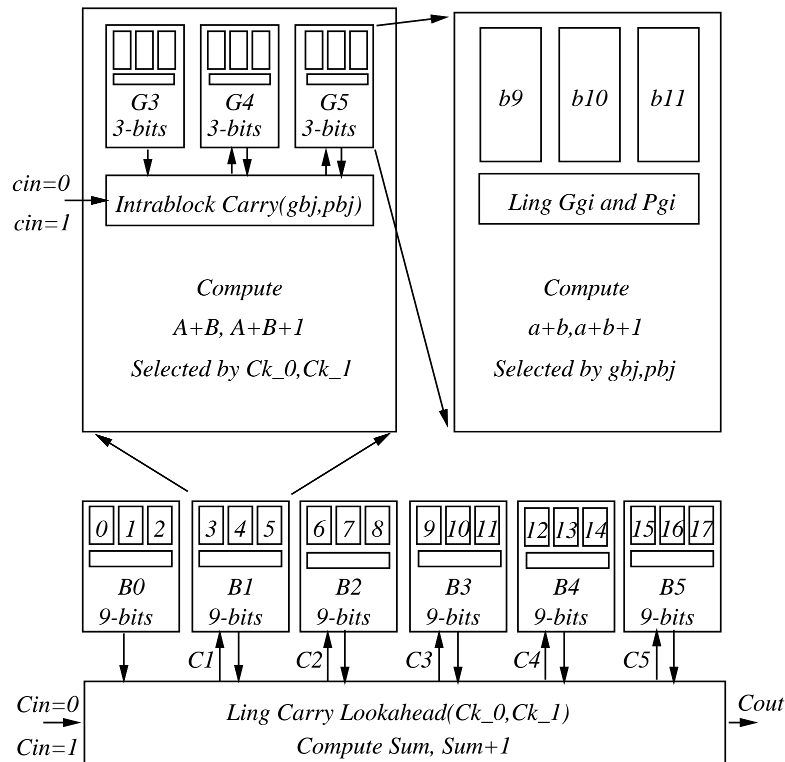

Figure 2: Block diagram and naming convention for the significand adder in the SNAP adder

Fig. 2 is the logic diagram of *ComAdd*. Note that the first group has only two bits. This is to reduce the number of serial transistors in the group generate gate when

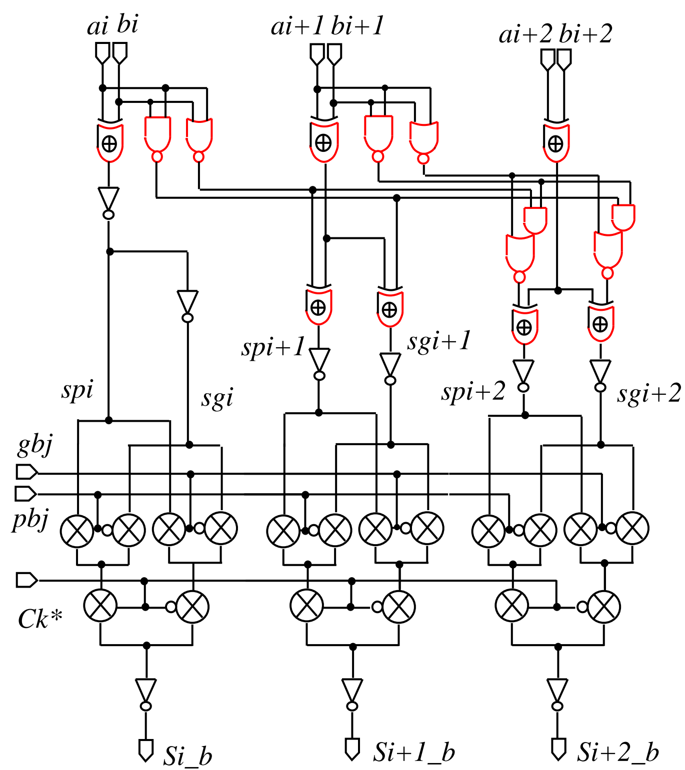

Figure 3: Circuit diagram of a 3-bit group

$C_{in} = 1$ as shown in a later figure. Fig. 3 is the circuit diagram of a 3-bit slice for Groups 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, and 17.

In principle, the circuits for Groups 3, 6, 9, 12, and 15 are simpler because the carries are known at those positions. However, to reduce layout time, we simply hardwire the circuits in Fig. 3. Groups 0 is special because it has only two bits and therefore must use a different circuit.

The process of carry generation is shown in Fig. 4. Fig. 6 and Fig. 5 show the group generate circuits for a two and three bit groups, respectively.

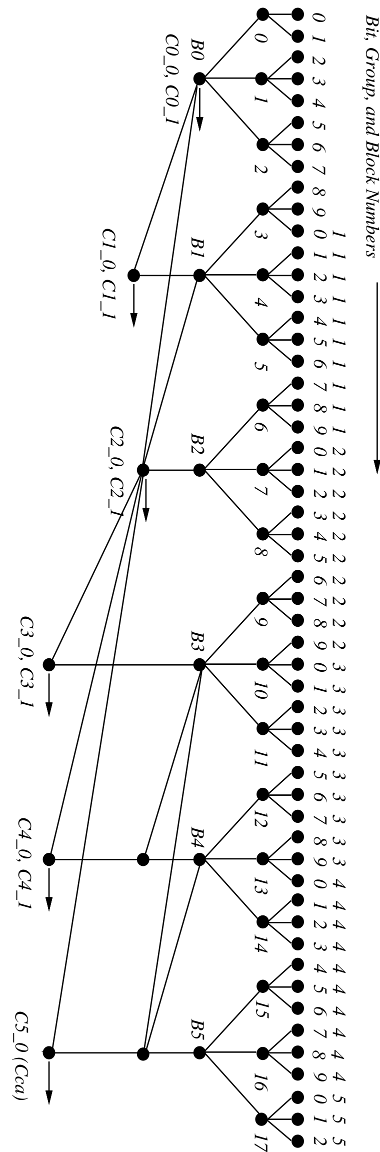

Figure 4: The process of carry generation in the SNAP FP adder.

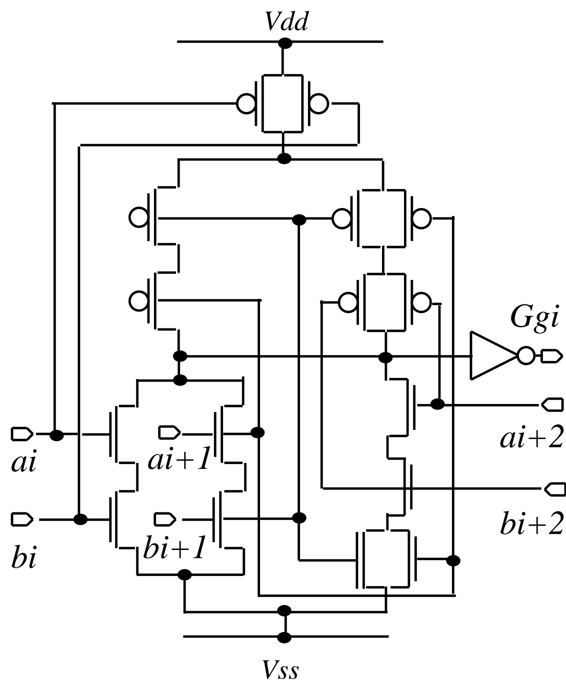

Figure 5: Group generate circuits for groups other than Group 1

Figure 6: Group generate circuit for Group 1 with Cin=1

Fig. 7 shows the block generate circuit. Roughly, the critical path of the adder consists of 4 such gates.

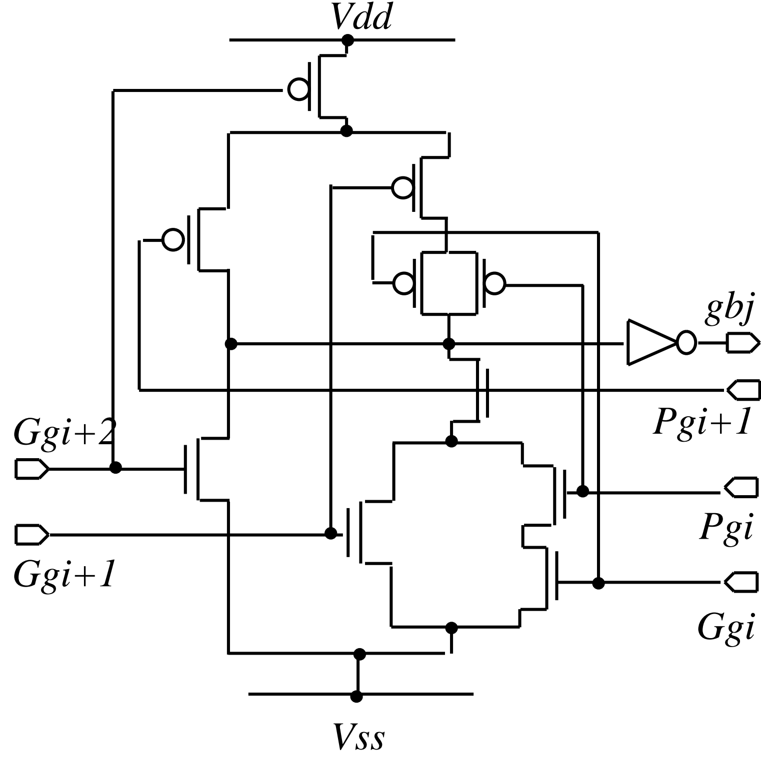

Figure 7: Block generate circuit

3.14 Normalize amount predictor or leading-one predictor (LOP)

- *Inputs:* T_i, G_i, P_i, and Z_i.
- *Outputs:* F_{LOP}, and C_{fine}.
- *Function:*
- *Explanation:* This unit performs leading one prediction.

  The equation for U_i is

$$\overline{U_i} = (\overline{T_{i-1} \oplus G_i})Z_{i+1} \vee (T_{i-1} \oplus G_i)G_{i+1}$$

An additional parallel means must be provided to locate the first U_i that is 0. This can be done as follows:

$$F_{LOPi} = AND(U_{\langle N:i+1\rangle}, \overline{U_{i-1}})$$

The logic equations for the U_i have been explained in [10].[4]

After the input is shifted by an amount indicated by F_{LOP}, the output is at most off by one bit; hence, we can simply observe the MSB of the result and determine if an additional 1b left shift is needed

3.15 Normalize amount encoder (ENC)

- *Inputs:* F_{LOP}
- *Outputs:* E_{LOP}
- *Functions:*

$$E_{LOP}\langle 0\rangle = OR(F_{LOP}\langle 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51\rangle)$$

$$E_{LOP}\langle 1\rangle = OR(F_{LOP}\langle 2,3,6,7,10,11,14,15,18,19,22,23,26,27,30,31,34,35,38,39,42,43,46,47,50,51\rangle)$$

$$E_{LOP}\langle 2\rangle = OR(F_{LOP}\langle 4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31,36,37,38,39,44,45,46,47,52\rangle)$$

$$E_{LOP}\langle 3\rangle = OR(F_{LOP}\langle 8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31,40,41,42,43,44,45,46,47\rangle)$$

$$E_{LOP}\langle 4\rangle = OR(F_{LOP}\langle 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,48,49,50,51,52\rangle)$$

$$E_{LOP}\langle 5\rangle = OR(F_{LOP}\langle 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52\rangle)$$

- *Explanation:* F_{LOP} is stored as a 1-of-N code. ENC encodes it into a 2's complement number, E_{LOP}. Again, our numbering convention starts with zero at the MSB.

3.16 Round logic (GRS)

- *Inputs:* b_n, b_{n+1}, s, E_{s09}, E_{s10}, S_{52}, S_{51}, C_{ca}, E_o, and $Fs_0\langle 0\rangle$.
- *Outputs:* C_{GRS} and q.
- *Function:* This unit determines the C_{GRS} and the q bits. C_{GRS} selects between Fs_0 and Fs_1. q is the bit to be left shifted into the result during normalization and its logic equation is:

$$q = \overline{b}_n\, b_{n+1}\, s \vee b_n \overline{b}_{n+1}$$

The derivation of C_{GRS}'s logic equation is based on a case by case analysis as described in [9]:

$$\begin{aligned} C_{GRS} \quad = \quad & \overline{E}_o \left[C_{ca} S_{52} (b_n \vee b_{n+1} \vee s \vee S_{51}) \vee \overline{C}_{ca} b_n (b_{n+1} \vee s \vee S_{52}) \right] \vee \\ & E_o \overline{E_{s09}} \left\{ \overline{b}_n\, \overline{b}_{n+1}\, \overline{s} \vee Fs_0\langle 0\rangle \left[\overline{b}_n (b_{n+1} \vee s) \vee b_n \overline{b}_{n+1} \overline{s} S_{52} \right] \vee \overline{Fs_0\langle 0\rangle}\, \overline{b}_n (b_{n+1} \oplus s) \right\} \vee \\ & E_o E_{s09} \left[\overline{E_{s10}} (Fs_0\langle 0\rangle b_n S_{52} \vee \overline{b}_n) \vee E_{s10} C_{ca} \right] \end{aligned}$$

Where $E_o = S_a \oplus S_b \oplus Op$ is the effective operation performed by the significand adder. $S_{52} = a_{52} \oplus b_{52}$ and $S_{51} = a_{51} \oplus b_{51}$ are the local sums of the two LSBs of the adder.[5] Because C_{ca} and $Fs_0\langle 0\rangle$ arrive late, we rearrange the above equation as:

[4] In [10], there is a redundancy in the logic equation. But this would not significantly change the picture.

[5] In [9], the MSB is numbered as bit 52. Here, it is numbered as bit 0.

$$
\begin{aligned}
C_{GRS} \quad = \quad & E_o E_{s09} \overline{E_{s10}}\, \overline{b}_n \vee \\
& C_{ca} \left[\overline{E}_o S_{52} (b_n \vee b_{n+1} \vee s \vee S_{51}) \vee E_o E_{s09} E_{s10} \right] \vee \overline{C_{ca}} \left[\overline{E}_o b_n (b_{n+1} \vee s \vee S_{52}) \right] \vee \\
& Fs_0\langle 0 \rangle \left[E_o \overline{E_{s09}} (\overline{b}_n \vee b_n \overline{b}_{n+1} \overline{s} S_{52}) \vee E_o E_{s09} \overline{E_{s10}} b_n S_{52} \right] \vee \overline{Fs_0\langle 0 \rangle} \left[E_o \overline{E_{s09}} b_n (\overline{b}_{n+1} \vee \overline{s}) \right]
\end{aligned}
$$

The above equation is of the form

$$C_{GRS} = (A) \vee C_{ca}(B) \vee \overline{C_{ca}}(C) \vee Fs_0\langle 0 \rangle (D) \vee \overline{Fs_0\langle 0 \rangle}(E)$$

where

$$A = E_o E_{s09} \overline{E_{s10} b_n}$$

$$B = \overline{E}_o S_{52} (b_n \vee b_{n+1} \vee s \vee S_{51}) \vee E_o E_{s09} E_{s10}$$

$$C = \overline{E}_o b_n (b_{n+1} \vee s \vee S_{52})$$

$$D = E_o \overline{E_{s09}} (\overline{b}_n \vee b_n \overline{b}_{n+1} \overline{s} S_{52}) \vee E_o E_{s09} \overline{E_{s10}} b_n S_{52}$$

and

$$E = E_o \overline{E_{s09} b_n} (\overline{b}_{n+1} \vee \overline{s})$$

which can be implemented as in Fig. 8.

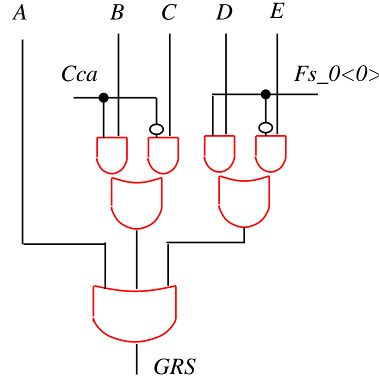

Figure 8: Circuit for GRS

- *Explanation:* The C_{GRS} bit selects between the results computed by the significand adder.

3.17 One's complementor ($OneCom$)

- *Inputs:* Fs, C_{ca}, $Es09$ and $Es10$.
- *Outputs:* $Fscom$.
- *Function:* If $(Es09 \wedge Es10 \wedge \overline{Cca})$, then $Fscom = \sim Fs$
- *Explanation:* $OneCom$ conditionally performs a bit inversion on the result from the significand adder when it is negative.

3.18 Normalization shift ($Ls55$)

- *Inputs:* $Fscom$, E_{LOP}, C_{fine}, and q.
- *Outputs:* $Fsfine$
- *Functions:*
$$Fsfine = Concatenate(Fscom, q) \ll E_{LOP}$$
- *Explanation:* $Ls55$ left shifts the result by E_{LOP} bits as needed. E_{LOP} may be off be 1 bit, which is later made up by C_{fine}.

3.19 Fine adjustment shift muxes ($MuxFA$)

- *Inputs:* $Fsfine$, $Fsfine \gg 1$, and C_{fine}.
- *Outputs:* Fs_mls.
- *Function:* If C_{fine}, $Fs_mls = Fsfine \gg 1$ else $Fs_mls = Fsfine$.
- *Explanation:* $MuxFA$ selects between the unshifted and shifted version of $Fsfine$, depending on whether the most significant bit of $Fsfine$ is zero.

3.20 Final result selection muxes ($Mux53fto$)

- *Inputs:* Fs_ors, Fs, Fs_ols, Fs_mls, ORS, NXS, OLS, and MLS.
- *Outputs:* F_f.
- *Function:*
Case ORS, $F_f = Fs_ors$
Case NXS, $F_f = Fs$
Case OLS, $F_f = Fs_ols$
Case MLS, $F_f = Fs_mls$
- *Explanation:* These muxes selects among the right shifted, no shifted, left shifted by 1 bit, and left shifted by many bits version of Fs.

4 Simulation

We have developed a functional simulator to verify the addition and rounding algorithms. This functional simulator works at both algorithmic and gate levels. It has proven to be useful in developing the algorithm and in studying the design options.

Fig. 4 shows how simulation is performed. For functional correctness, a random number generator is used to create two number streams, which are coverted into bit vectors and then fed into the simulator. Independently, the numbers are also passed to the floating-point hardware on the host machine to perform the same operation. The simulated and the computed results are then compared. In the simulator, each component is the adder is

described by its logic equation. This allows the simulator to verify not only the algorithm but also the hardware implementation. To simulate other rounding modes, the control register for the rounding mode is set accordingly.

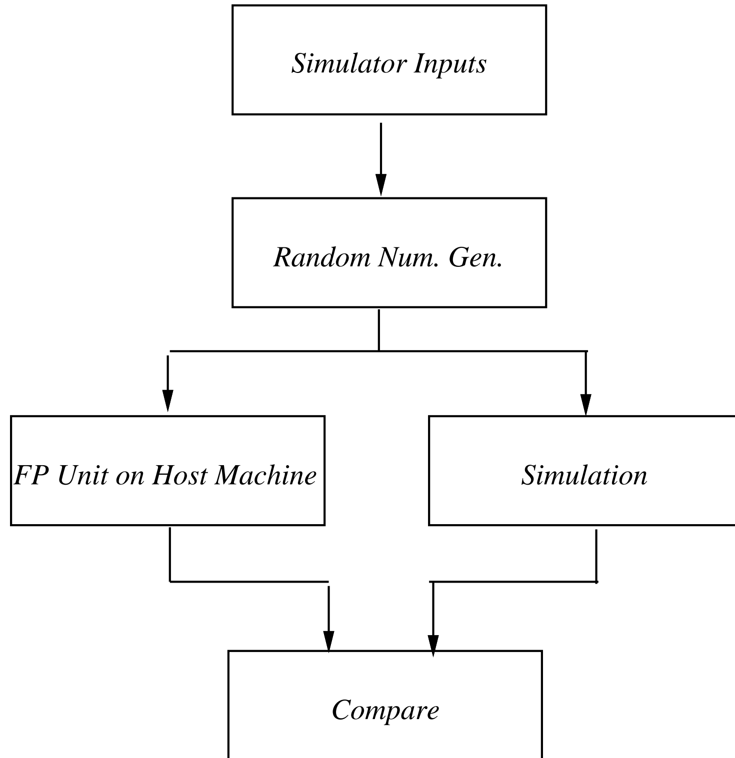

Figure 9: Functional Simulation of the SNAP FP adder

5 Implementation

The adder is implemented using the HP CMOS26 $1\mu m$ process with 3 layer metals. The third layer metal is only used for power and ground in the current implementation. The adder is fully static and laid out in a standard cell fashion without compaction.

5.1 Area Estimate

The area of the whole FP adder occupies $14.2mm^2$. Table 4 is the breakdown of the area consumed by each component. In the table, the muxes and the buffers items are the total area consumed by the muxes and buffers in the adder. Buffers within the components are not included. These external muxes and buffers can be considered as the cost of decision making and communications. In relation to the table, several observations can be made. First, wires connecting the components, power and ground, and wasted area occupy as much as 45% of the adder area. Second, LOP consumes roughly the same area as the significand

Table 4: Breakdown of area consumption

Component	$X \times Y \mu m^2$	$K \mu m^2$	Percent
Exponent adder	592x459	272	2
Swapper	2150x142	921	6.6
Alignment shifter	2952x282	833	6
Normalization shifter	2952x282	833	6
Shift decoders	423x170	72	0.5
Sticky bit logic	2172x424	921	6.6
Significand adder	3089x450	1390	10
LOP	3000x430	1290	10
One's Complementor	3000x71	213	1.5
Round Logic	212x208	44	0.3
Sign logic	92x120	11	0.01
Exponent adjust logic	592x58	34	0.2
Exponent adjust adder	592x459	272	2
Muxes		225	1.6
Buffers		327.6	2.3
Total		7700	55

adder. Third, the area consumed by the rounding logic is negligible. Fourth, sticky bit computation required to round for the IEEE standard consumed as much area as a shifter. Fifth, the significand adder consumes the most area in the whole FP adder.

5.2 Delay Estimate

Table 5 lists the delay in the critical paths. In the SA path, *ExpAdd* computes the absolute magnitude of the exponent difference, which is then decoded (*Dec*) to drive the alignment shifter. The *ExpAdd* delay includes selecting the proper result to obtain absolute value. The output then goes through a 3-1 mux, which also performs inversion if needed. The delay for computing the multiple results in the significand adder is 3.7ns. The round logic takes 3.6ns, which includes delay for signals travelling from the MSB to the round logic and delay for driving the final selection muxes. Finally, The result is buffered to drive the final selection mux. One's complementation is not needed in this path. In the current implementation, the round logic is placed on the LSB side of the significand adder. By placing the round logic on the MSB side, the signal travelling delay, about 1ns, can be saved. The final result selection muxes should not have been counted as part of the round delay because this delay is needed anyway to normalize the result before rounding had a more conventional rounding algorithm been used.

In the AS path, we need to compute the sign of the result of the exponent subtraction

Table 5: Simulated Delay

SA Path		AS Path			
Path	ns	Path 1	ns	Path 2	ns
ExpAdd	3.6	ExpAdd.Cexp	2.6	Same	2.6
Dec	1.2	Swapper + buffer	2.0	Same	2.0
Alignment	1.6	3-1 mux	0.6	Same	0.6
3-1 mux + buffer	0.8	LOP	6.5	ComAdd + Round	7.3
Significand add	3.7	Dec	1.2	One's Complement	0.8
Round logic	3.6	Normalize (Control)	1.6	Normalize (Data)	1.5
One's Complement / buffer	0.8	Fine Adjust	1.4	Same	1.4
4-1 mux + buffer	1.0	4-1 mux + buffer	1.0	Same	1.0
Total	16.3		16.9		17.2

and swap the significands if needed. The result may be negative when the exponents are equal. The output of the swapper then goes to the 3-1 mux. At this point, we need to examine two paths to determine the critical path. The first path is LOP plus normalization shift and the second is significand add, round, and one's complement. As shown in the table, the latter turns out to be worse, becoming the critical path of the whole adder. After one's complement, the result result is selected.

Without LOP, the AS Path 2 is worse because two more operations are needed for normalization. The first operation is leading one detection (LOD)[6] on the result from the one's complementor. The second is encoding, whose result is then used to drive the normalization shifter.[7] On the positive side, however, we no longer need a fine adjustment step for a 1.4ns saving. The critical path would then have been 22.3ns[8]; hence, LOP is important for high-speed FP adders.

The worst case delay shown in Table 5 does not include other rounding modes. Round to infinity requires computing up to 3 outcomes in parallel in the significand adder, requiring a row of 3-2 counters as shown in [9]. Assuming a 1.2ns delay for a 3-2 counter, the worst case delay for an adder that performs all rounding modes is roughly 18.5ns.

6 Modification for Single-Precision Operation

For machines like the IBM RS/6000, single-precision is not supported; all numbers are treated as double precision during computation. For others, however, direct support for

[6] LOD requires a priority encoder.

[7] A possible optimization is to merge the encoder logic and the decoder in the shifter.

[8] $Delay = T_{Path\ 2} + T_{LOD} + T_{encoding} + T_{normalize(control)} - T_{normalize(data)} - T_{Fine_adjust} = 17.2 + 2.8 + 2.4 + 2.8 - 1.5 - 1.4 = 22.3ns$

single-precision operations is required. In this section, we describe the changes for single precision operations.

The changes needed are:

- The b_n, b_{n+1}, and s bits have to be computed separately. This is relatively easy because we already have a 53-bit shifters in place. We have

$$b_n = Fin\langle 24\rangle$$

$$b_{n+1} = Fin\langle 25\rangle$$

  and

$$s = OR(Fin\langle 26 : 52\rangle)$$

  We have chosen to observe Fin, rather than Fm, to compute these bits. This is for conversion from double precision to single. To perform such an operation, we have to subtract the operand from 1, set the exponents equal, force the MSB to be 1, and select the result to be single. It will be rounded properly because the lower order bits of Fin are observed by the adder. This is also the reason why we want to observe up to the 52th bit to determine the sticky bit.

  Extra logic must be provided to select between the b_n, b_{n+1}, and s bits for double precision and for single operations.

- The LSBs of $F1$ into the significand adder must be forced to 1 so that when computing $A + B + 1$, the 1 will ripple into the correct place. That is,

$$F1\langle 24 : 52\rangle = 1$$

- The LSBs of Fin into the significand adder must be forced to zero to avoid generating a false carry into the MSBs.

$$Fin\langle 24 : 52\rangle = 0$$

- The LSBs of Fs, the result from the significand adder must be forced to zero to avoid shifting invalid bits into the top 23 bits during normalization.

$$Fs\langle 24 : 52\rangle = 0$$

All of these changes are relatively minor and have a negligible speed impact on the adder.

In principle, it is possible to modify the adder to perform operation on two double precision operands and returns a single-precision result. This, however, will come at the expense of delay. Most of the architectures, therefore, do not support such an instruction. Instead, it is performed as a double precision operation and the result is then converted into single precision. Fortunately, such a need for conversion occurs relatively infrequently.

7 Pipelining the Adder

To increase throughput, the current adder can be pipelined at 3 stages. For the SA path, we do exponent subtract and shift in the first cycle, add in the second cycle, and select the final result in the last cycle. For the AS path, we do exponent subtract in the first cycle, add in the second, then normalize and select the final result in the last cycle. Only one adder is needed in this design. The cycle time in this pipelining scheme is likely determined by the adder stage.

To reduce latency, the adder can be pipelined at 2-stage. We have to use another significand adder in the AS path because the two paths use the adder in different stages of the pipeline, creating a resource conflict. Fig. 7 shows a possible 2-stage implementation. In this 2-stage implementation, the AS path can be slightly sped up because the operands can be swapped simply by examining the least 2 significant bits of the exponents. These bits may be different for single and double-precision operations, however.

8 Summary

In this paper, we have presented the design and implementation of a high-speed floating-point adder used in the Stanford Nanosecond Arithmetic Processor. The SNAP FP adder uses a state of the art rounding method to reduce the number of operations in the shift-add critical path and uses leading one prediction and a fast complementation technique to speed up the add-shift path. The key in the rounding method is to compute multiple results and select the correct one based on the rounding bits. These multiple results have also been used to perform fast complementation when the result is negative in the SNAP adder. Leading one prediction – LOP –is a mothed of predicting the amount of normalization shift needed to align a result before its arrival. This allows LOP and significand addition to be performed in parallel, rather than in serial. SNAP uses a LOP circuit that is similar in operation to that of the RS/6000. But we believe that our LOP circuit consumes less hardware.

The SNAP FP adder has been implemented in the HP CMOS26 1μm process with triple layer metals. The third metal layer is used only for ground and power. The adder is fully static and consumes less than 1 watt at a nominal delay of less than 20ns. The adder is laid out in standard cells using Mantor the GDT tool from Mantor Graphics without compaction; the size of the adder is 14.5 mm^2.

9 Acknowledgements

The authors wish to thank Yoshiu Nishi, John Kelly, Dennis Brzezinski, and Ruby Lee of HP for their help in the fabrication of the adder. Mark Horowitz and Dennis Brzezinski provided advice during the implementation of the adder.

Figure 10: A 2-stage pipelined implementation of the SNAP FP adder

References

[1] A. Naseem and P. D. Fisher. The modified cordic algorithm. In *Proc. of the 7th Symposium on Computer Arithmetic*, pages 144–152, 1985.

[2] ANSI/IEEE Standard No. 754, American National Standrads Institute, Washington, DC. *An American National Standard: IEEE Standard for Binary Floating-Point Arithmetic*, 1988.

[3] B. J. Benschneider, W. J. Bowhill, E. M. Cooper, M. N. Gavrielov, P. E. Gronowski, V. K. Maheshwari, V. Peng, J. D. Pickholtz, and S. Samudrala. A pipelined 50-mhz cmos 64-bit floating-point arithmetic processor. *IEEE Transactions on Computers*, 24(5):1317–1323, Oct. 1989.

[4] H. P. Sit, M. R. Nofai, and S. Kim. An 80mflops floating-point engine in the i860 processor. In *Proc. of International Conference on Computer Design*, pages 374–379, 1989.

[5] J. E. Volder. The cordic trigonometric computing technique. *IEEE Transactions on Computers*, EC-8:330–334, Sep. 1959.

[6] M. Flynn, G. DeMicheli, R. Dutton, B. Wooley, and F. Pease. Sub-nanosecond arithmetic. Technical Report CSL-TR-90-428, Stanford University, May 1990.

[7] M. P. Farmwald. *On the Design of High Perfromance Digital Arithmetic Units.* PhD thesis, Stanford University, Aug. 1981.

[8] N. T. Quach and M. J. Flynn. High-speed addition in cmos. Technical Report CSL-TR-90-415, Stanford University, Fab. 1990.

[9] N. T. Quach and M. J. Flynn. An improved algorithm for high-speed floating-point addition. Technical Report CSL-TR-90-442, Stanford University, Aug. 1990.

[10] N. T. Quach and M. J. Flynn. Leading one prediction — implementation, generalization, and application. Technical Report CSL-TR-91-463, Stanford University, March 1991.