UNIVERSITY OF WISCONSIN MADISON

MASTER'S PROJECT REPORT

Implementation of Complement Mode Execution

Author:

JATIN MITRA

Advisor:

MIKKO LIPASTI

AUGUST 23, 2011

Table of Contents

1. Introduction 6

2. Analytical Proof of Complement Mode Execution 8

2.1 Proof 1: Addition in complement mode 9

2.2 Proof 2: Multiplication in Complement Mode with operands sign extended to full width 12

2.3 Proof 3: Multiplication in Complement Mode with operands not sign extended 16

3. Case Study of Integer Multiplication in Complement Mode 21

3.1 Wallace Tree Multiplication 22

3.2 Booth Recoded Multiplication 24

3.2.1 Radix-4 Booth Recoding 24

4. OpenSPARC T1 Processor 27

4.1 Instruction Fetch Unit 28

4.2 Execution Unit 28

4.3 Load / Store Unit 28

4.4 Floating Point Unit 29

4.5 L2-Cache 29

5. Implementation of Complement Mode Execution in T1 30

5.1 Cache 31

5.1.1 Instruction Cache Data Unit 32

5.2 Instruction and Data Cache Tag Unit 35

5.3 Data Cache Data Unit 37

5.4 Decode 37

5.5 ALU 39

5.5.1 Logic Unit 41

5.5.2 Adder Unit 43

5.5.3 Shifter 44

5.5.4 Zero Comparator 46

5.5.5 Multiplier Unit 46

5.6 Register File 48

5.7 Floating Point Unit 49

5.7.1 Floating Point Multiplication 51

5.7.2 Floating Point Addition 52

6. Further Work 55

7. References 56

List of Figures

Figure 1 : Wallace Tree Multiplication [3] 23

Figure 2: SPARC T1 core pipeline [2] 27

Figure 3 : L2-cache simplified addressing and data overview[5] 31

Figure 4 : Dataflow from L2 cache to Data cache 32

Figure 5: Instruction Cache Dataflow 33

Figure 6: Instruction Data Cache 36

Figure 7 : Different Instruction formats in T1 Processor[5] 39

Figure 8: SPARC T1 ALU [5] 40

Figure 9: Integer condition codes (CCR_icc and CCR_xcc) 40

Figure 10: T1 ALU Unit 42

Figure 11: SPARC Arithmetic Unit[5] 43

Figure 12: SPARCT1 shifter unit [5] 45

Figure 13: MUL Block Diagram 46

Figure 14: Integer Register file read port along with the associated signals [5] 49

Figure 15 Top-Level FFU Block Diagram 50

Figure 16 Floating Point Unit (FPU) functional block diagram 51

Abstract

As technology goes deeper into the submicron range device aging effects become increasingly powerful. The Colt duty cycle equalizer gives a microarchitectural solution to this problem with execution of complemented data during alternate epochs. This results in almost balanced duty cycles in the internal nodes and aging is effectively slowed down.

This work adds to previous work done by Erika et al. [1] and presents an analytical proof of complement mode execution for addition and multiplication. We also implemented Colt on OpenSPARC T1 processor and identified the changes needed to be made in the datapath. Synthesis results from floating point addition indicate a delay penalty of 12% and an area penalty of 22.9%. In case of floating point multiplication the penalties were 10% and 11% for delay and area respectively. In addition, a gate level implementation of a 64-bit Wallace tree multiplier was also done to estimate the timing and the area penalty incurred in incorporating Colt. The results show that for a multiplier a delay penalty of 11% and an area penalty of 11.4% is incurred. The simulations were performed at 110 nm node.

1. Introduction

With technology scaling driving device dimensions deeper and deeper into the submicron range, reliability of the devices is becoming an imposing problem for architects. Effects like Bias Temperature Instability (BTI) and Hot Carrier Injection (HCI) make devices less reliable with time. In the work of Erika et al. [1] a microarchitectural technique was proposed to minimize the harmful impact of BTI effect. It was proposed that by operating the datapath and storage elements in true and complement mode, utilization of the devices can be equalized and the MTTF be increased by 40%. This was proposed under the name of Colt duty equalizer.

In this work we provide analytical proof of integer addition and multiplication in complement mode. While the aim of these proofs was primarily meant to generalize the result, this exercise gave valuable insight in the implementation too. The proofs lead to an optimization which resulted in major saving of hardware in case of multiplication.

To evaluate the design changes required to implement complementary mode operation we implement it in OpenSPARC T1 processor at the pipeline level for a single thread. T1 is a 4-way SMT 8-core multiprocessor with a floating point unit being shared by all the 8 cores. This case study helped identify various design challenges associated with the implementation as well as bring out new opportunities. In addition, impact on floating point addition and multiplication was also estimated by modifying the floating point unit in T1 to support complement mode. The implementation was not as tedious as expected as the code was divided into datapath and control units for almost every block. Since control unit was always maintained in true mode, the study of datapath elements could be done faster.

To work out a way around the integer multiplier in T1, a 64-bit Wallace tree multiplier was implemented in verilog to estimate the area and timing impact in complement mode execution. The results of this work show about 10% penalty in area and timing. The timing cost could be decreased with pipelining and using higher order carry save adders (CSA).

.

2. Analytical Proof of Complement Mode Execution

Assumption: Let us have a 'n' bit signed binary number X represented as $x_{n-1}\ x_{n-2}\ x_{n-3}\ldots.\ x_0$. The value of X in decimal system is

$$X = -2^{n-1}x^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Lemma 1: **Value of One's complement in decimal system**

One's complement of a binary number is obtained by reversing its digits. Let us consider a number X whose one's complement is represented as $\bar{X}$ where

$$\bar{X} = \overline{x_{n-1}}\ \overline{x_{n-2}}\ \overline{x_{n-3}} \ldots . \overline{x_0}$$

$$\bar{X} = -2^{n-1}\overline{x_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{x_i}$$

However every ($\overline{x_i}$) term can be written as (1 - x_i). On making this substitution in the above expression we get

$$\bar{X} = -2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i (1 - x_i)$$

$$\bar{X} = -2^{n-1} + 2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} 2^i x_i$$

$$\bar{X} = -2^{n-1} + \sum_{i=0}^{n-2} 2^i - \left(-2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i\, x_i\right)$$

We know that the sum $\sum_{i=0}^{n-2} 2^i$ is a geometric progression such that

$$\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$$

Substituting this value in the above expression yields

$$\bar{X} = -2^{n-1} + 2^{n-1} - 1 - \left(-2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \right)$$

$$\bar{X} = -1 - \left(-2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \right)$$

$$\bar{X} = -1 - X \quad or \quad \bar{X} + X + 1 = 0$$

The above is true for all binary numbers and hence the proof holds.

2.1 Proof 1: Addition in complement mode

Let us have two 'n' bit signed binary numbers X and Y represented as $x_{n-1}\ x_{n-2}\ x_{n-3}\ldots\ x_0$ and $y_{n-1}\ y_{n-2}\ y_{n-3}\ldots y_0$ respectively. Their values in decimal system are

$$X = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

$$Y = -2^{n-1} y_{n-1} + \sum_{i=0}^{n-2} 2^i y_i$$

The addition of these two number yields a new number Z, where Z is

$$Z = X + Y$$

$$Z = \left(-2^{n-1}\mathrm{x}_{n-1} + \sum_{i=0}^{n-2} 2^i \mathrm{x}_i\right) + \left(-2^{n-1}\mathrm{y}_{n-1} + \sum_{i=0}^{n-2} 2^i \mathrm{y}_i\right)$$

$$Z = \left(-2^{n-1}\mathrm{x}_{n-1} - 2^{n-1}\mathrm{y}_{n-1}\right) + \left(\sum_{i=0}^{n-2} 2^i \mathrm{x}_i + \sum_{i=0}^{n-2} 2^i \mathrm{y}_i\right)$$

$$Z = -2^{n-1}\left(\mathrm{x}_{n-1} + \mathrm{y}_{n-1}\right) + \sum_{i=0}^{n-2} 2^i \left(\mathrm{x}_i + \mathrm{y}_i\right)$$

In complement mode let the numbers be represented as $\bar{X}$ and $\bar{Y}$ where

$$\bar{X} = \overline{x_{n-1}}\,\overline{x_{n-2}}\,\overline{x_{n-3}} \ldots . \overline{x_0}$$

$$\bar{Y} = \overline{y_{n-1}}\,\overline{y_{n-2}}\,\overline{y_{n-3}} \ldots . \overline{y_0}$$

$$\bar{X} = -2^{n-1}\overline{x_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{\mathrm{x}_i}$$

The addition of these two number yields a new number $\bar{Z}$, where $\bar{Z}$ is

$$\bar{Z} = \bar{X} + \bar{Y}$$

$$\bar{Z} = \left(-2^{n-1}\overline{x_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{\mathrm{x}_i}\right) + \left(-2^{n-1}\overline{y_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{\mathrm{y}_i}\right)$$

Using the result derived above for the decimal representation of a number in one's complement, the above expression yields

$$\bar{Z} = \left(-1 \; + \; 2^{n-1} x_{n-1} \; \; - \sum_{i=0}^{n-2} 2^i \mathrm{x}_i \right) + \left(-1 \; + \; 2^{n-1} y_{n-1} - \sum_{i=0}^{n-2} 2^i \mathrm{y}_i \right)$$

$$\bar{Z} = -1 \; - \left(1 - \; 2^{n-1} (x_{n-1} + \; y_{n-1} \,) \; + \sum_{i=0}^{n-2} 2^i (\, \mathrm{x}_i + \; \mathrm{y}_i) \right)$$

$$\bar{Z} = -1 \; - \; (\, 1 \; + Z \,)$$

The quantity in the parentheses is the expected value. However the extra -1 suggests that a carry-in of +1 must be added to the input operands in complement form to get one's complement of the output in true mode.

Corollary: A corollary to above result is inclusion of (k-1) carry-in's in the addition of k numbers in complement mode.

$$\bar{Z} \; = \; \bar{A} + \bar{B} \; \ldots + \bar{Y} \qquad\qquad K \; such \; numbers$$

$$\bar{Z} = \left(-1 \; + \; 2^{n-1} a_{n-1} \; \; - \sum_{i=0}^{n-2} 2^i \mathrm{a}_i \right) + \left(-1 \; + \; 2^{n-1} b_{n-1} - \sum_{i=0}^{n-2} 2^i \mathrm{b}_i \right) \ldots .$$

$$+ \left(-1 \; + \; 2^{n-1} y_{n-1} - \sum_{i=0}^{n-2} 2^i \mathrm{y}_i \right)$$

$$\bar{Z} = (k-1) - \left(1 - \; 2^{n-1} (\, a_{n-1} + b_{n-1} \; \ldots + y_{n-1}) + \; \sum_{i=0}^{n-2} 2^i (\, \mathrm{a}_i \; + \; \mathrm{b}_i \ldots . + \mathrm{y}_i) \right)$$

$$\bar{Z} \; = (k-1) - (1 + Z \,)$$

2.2 Proof 2: Multiplication in Complement Mode with operands sign extended to full width

Let us have two n bit signed binary numbers X and Y represented as $x_{n-1}\ x_{n-2}\ x_{n-3}\ldots.\ x_0$ and $y_{n-1}\ y_{n-2}\ y_{n-3}\ldots y_0$ respectively. In multiplication the result is of 2n bits and hence the operands are also sign extended to 2n bits. Their values in decimal system are

$$X = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

$$Y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} 2^i y_i$$

The multiplication of these two number yields a new number Z, where Z is

$$Z = X * Y$$

$$Z = \left(-2^{2n-1}x_{2n-1} + \sum_{i=0}^{2n-2} 2^i x_i\right) * \left(-2^{2n-1}y_{2n-1} + \sum_{i=0}^{2n-2} 2^i y_i\right)$$

$$Z = \left(-2^{2n-1}x_{2n-1} + \sum_{i=0}^{2n-2} 2^i x_i\right) y_0 + \left(-2^{2n-1}x_{2n-1} + \sum_{i=0}^{2n-2} 2^i x_i\right) 2^1 y_1 \ldots.$$

$$- \left(-2^{2n-1}x_{2n-1} + \sum_{i=0}^{2n-2} 2^i x_i\right) 2^{n-1} y_{n-1}$$

$$Z = -\left[2^{2n-1}x_{2n-1}\left(\sum_{i=0}^{2n-2} 2^i y_i\right)\right] - \left[2^{2n-1}y^{2n-1}\left(\sum_{i=0}^{2n-2} 2^i x_i\right)\right] + \left(\sum_{i=0}^{2n-2} 2^i x_i\right)\left(\sum_{i=0}^{2n-2} 2^i y_i\right)$$

$$+ (2^{4n-2}x_{2n-1}y_{2n-1})$$

It is worth noting here that positive terms with exponent $\geq 2^{2n}$ will not affect the result for 2n bits. These terms come in existence as the n bit numbers were sign extended for completeness and correlation with the next result.

In complement mode let the numbers be represented as $\bar{X}$ and $\bar{Y}$ where

$$\bar{X} = \overline{x_{n-1}}\,\overline{x_{n-2}}\,\overline{x_{n-3}} \ldots . \overline{x_0}$$

$$\bar{Y} = \overline{y_{n-1}}\,\overline{y_{n-2}}\,\overline{y_{n-3}} \ldots . \overline{y_0}$$

In complementary mode multiplication the partial products are obtained by 'OR' operation rather the 'AND' operation (In normal multiplication the 'AND' operation occurs by default). However in decimal system there is no operator that acts as a Boolean analogue of 'OR' operator. The way the 'OR' operation is achieved is as follows.

Let us consider a term obtained from the multiplication of multiplicand ($\bar{X}$) and the first bit of the multiplier ($\overline{y_o}$). The term would look like

$$\left(-2^{n-1}\overline{x_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{x_i}\right)(1 - \overline{y_o}) + (2^{2n} - 1)\overline{y_o}$$

The above expression implies that if the value of the bit $\overline{y_o}$ is '0' then the multiplicand is to be considered. However if the value of $\overline{y_o}$ is '1' then the partial product should have all the bit values equal to '1'. This sequence implies adding $(2^{2n} - 1)$ in the decimal system. For subsequent terms, a factor of $\left(2^{2n} - 2^i\right)$ would occur in place of the second expression in parentheses above. In addition as discussed earlier in addition in complement mode a carry-in is added for each stage. However the carry-in's would be shifted as the successive partial products are shifted too.

$$\overline{Z} = \bar{X} * \bar{Y}$$

$$\overline{Z} = \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)\left(1 - (1 - \mathrm{y}_0)\right) + (2^{2n} - 2^0)(1 - \mathrm{y}_0)$$

$$+ \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)2^1\left(1 - (1 - \mathrm{y}_1)\right) + (2^{2n} - 2^1)(1 - \mathrm{y}_1) \ldots$$

$$- \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)2^{2n-1}\left(1 - (1 - \mathrm{y}_{2n-1})\right)$$

$$+ (2^{2n} - 2^{2n-1})(1 - \mathrm{y}_{2n-1}) + (\sum_{i=1}^{2n-1} 2^i)$$

$$\overline{Z} = \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)(\mathrm{y}_0) + (2^{2n} - 2^0)(1 - \mathrm{y}_0)$$

$$+ \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)2^1(\mathrm{y}_1) + (2^{2n} - 2^1)(1 - \mathrm{y}_1) \ldots$$

$$- \left(-2^{2n-1}(1 - x_{2n-1}) + \sum_{i=0}^{2n-2} 2^i(1 - \mathrm{x}_i)\right)2^{2n-1}(\mathrm{y}_{2n-1})$$

$$- (2^{2n} - 2^{2n-1})(1 - \mathrm{y}_{2n-1}) + (2^{2n} - 2)$$

$$\overline{Z} = \left(2^{2n-1}x_{2n-1}\sum_{i=0}^{2n-2}2^i y_i\right) - \left(2^{2n-1}\sum_{i=0}^{2n-2}2^i y_i\right) - \left(\sum_{i=0}^{2n-2}2^i y_i\right)\left(\sum_{i=0}^{2n-2}2^i x_i\right)$$

$$+ \left((2^{2n-1}-1)\sum_{i=0}^{2n-2}2^i y_i\right) + (2n-1)(2^{2n}) - \left(2^{2n}\sum_{i=0}^{2n-2}y_i\right) + \left(\sum_{i=0}^{2n-2}2^i y_i\right)$$

$$- \left(\sum_{i=0}^{2n-2}2^i\right) + (2^{4n-2}y_{2n-1}) - (2^{4n-2}x_{2n-1}y_{2n-1}) - \left(2^{2n-1}y_{2n-1}\sum_{i=0}^{2n-2}2^i\right)$$

$$+ \left(2^{2n-1}y_{2n-1}\sum_{i=0}^{2n-2}2^i x_i\right) - 2^{2n} + (2^{2n}y_{2n-1}) + 2^{2n-1} - (2^{2n-1}y_{2n-1}) + (2^{2n}$$

$$- 2)$$

$$\overline{Z} = \left(2^{2n-1}x_{2n-1}\sum_{i=0}^{2n-2}2^i y_i\right) - \left(2^{2n-1}\sum_{i=0}^{2n-2}2^i y_i\right) - \left(\sum_{i=0}^{2n-2}2^i y_i\right)\left(\sum_{i=0}^{2n-2}2^i x_i\right) + \left((2^{2n-1})\sum_{i=0}^{2n-2}2^i y_i\right)$$

$$- \left(2^{2n}\sum_{i=0}^{2n-2}y_i\right) + (2n-1)(2^{2n}) - \left((2^{2n-1}-1) - (2^{2n}-2)\right) + (2^{4n-2}y_{2n-1})$$

$$- (2^{4n-2}x_{2n-1}y_{2n-1}) - (2^{2n-1}y_{2n-1}(2^{2n-1}-1)) + \left(2^{2n-1}y_{2n-1}\sum_{i=0}^{2n-2}2^i x_i\right) - 2^{2n}$$

$$+ (2^{2n}y_{2n-1}) + 2^{2n-1} - (2^{2n-1}y_{2n-1})$$

$$\overline{Z} = \left(2^{2n-1}x_{2n-1}\sum_{i=0}^{2n-2}2^{i}\mathrm{y}_i\right) - \left(\sum_{i=0}^{2n-2}2^{i}\mathrm{y}_i\right)\left(\sum_{i=0}^{2n-2}2^{i}\mathrm{x}_i\right) - \left(2^{2n}\sum_{i=0}^{2n-2}\mathrm{y}_i\right) + (2n-1)(2^{2n})$$

$$+ \ (2^{2n-1}-1) + (2^{4n-2}\mathrm{y}_{2n-1}) - \ (2^{4n-2}\mathrm{x}_{2n-1}\mathrm{y}_{2n-1}) \ - (2^{2n-1}\mathrm{y}_{2n-1}(2^{2n-1}-1))$$

$$+\left(2^{2n-1}\mathrm{y}_{2n-1}\sum_{i=0}^{2n-2}2^{i}\mathrm{x}_i\right) - 2^{2n-1} + (2^{2n}\mathrm{y}_{2n-1}) - (2^{2n-1}\mathrm{y}_{2n-1})$$

$$\overline{Z} = \left(2^{2n-1}x_{2n-1}\sum_{i=0}^{2n-2}2^{i}\mathrm{y}_i\right) - \left(\sum_{i=0}^{2n-2}2^{i}\mathrm{y}_i\right)\left(\sum_{i=0}^{2n-2}2^{i}\mathrm{x}_i\right) - \left(2^{2n}\sum_{i=0}^{2n-2}\mathrm{y}_i\right) + \left(2^{2n-1}\mathrm{y}_{2n-1}\sum_{i=0}^{2n-2}2^{i}\mathrm{x}_i\right)$$

$$+ (2n-1)(2^{2n}) - 1 - \ (2^{4n-2}\mathrm{x}_{2n-1}\mathrm{y}_{2n-1}) + (2^{2n}\mathrm{y}_{2n-1})$$

$$\overline{Z} = -1 - Z + (2n-1)(2^{2n}) - \left(2^{2n}\sum_{i=0}^{2n-2}\mathrm{y}_i\right) + (2^{2n}\mathrm{y}_{2n-1})$$

$$\overline{Z} = -1 - Z + (2^{2n})\left(2n + \mathrm{y}_{2n-1} - 1 - \sum_{i=0}^{2n-2}\mathrm{y}_i\right)$$

The quantity in the parenthesis will always be positive and thus the product will be ≥ 2^{2n}. The third term would thus not affect the result of the previous 2n bits. Hence the proof holds for multiplication too.

2.3 Proof 3: Multiplication in Complement Mode with operands not sign extended

Let us have two n bit signed binary numbers X and Y represented as x_{n-1} x_{n-2} x_{n-3}…. x_0 and y_{n-1} y_{n-2} y_{n-3}….y_0 respectively. Their values in decimal system are

$$\mathrm{X} = -2^{n-1}\mathrm{x}_{n-1} + \sum_{i=0}^{n-2}2^{i}\mathrm{x}_i$$

$$\mathrm{Y} = -2^{n-1}\mathrm{y}_{n-1} + \sum_{i=0}^{n-2}2^{i}\mathrm{y}_i$$

The multiplication of these two number yields a new number Z, where Z is

$$Z = X * Y$$

$$Z = \left(-2^{n-1}\mathrm{x}_{n-1} + \sum_{i=0}^{n-2} 2^i \mathrm{x}_i\right) * \left(-2^{n-1}\mathrm{y}_{n-1} + \sum_{i=0}^{n-2} 2^i \mathrm{y}_i\right)$$

In normal mode the result of multiplication as shown above is

$$Z = -\left[2^{n-1}\mathrm{x}_{n-1}\left(\sum_{i=0}^{n-2} 2^i \mathrm{y}_i\right)\right] - \left[2^{n-1}\mathrm{y}_{n-1}\left(\sum_{i=0}^{n-2} 2^i \mathrm{x}_i\right)\right] + \left(\sum_{i=0}^{n-2} 2^i \mathrm{x}_i\right)\left(\sum_{i=0}^{n-2} 2^i \mathrm{y}_i\right) + (2^{2n-2}\mathrm{x}_{n-1}\mathrm{y}_{n-1})$$

In complement mode let the numbers be represented as $\bar{X}$ and $\bar{Y}$ where

$$\bar{X} = \overline{x_{n-1}}\,\overline{x_{n-2}}\,\overline{x_{n-3}} \ldots . \overline{x_0}$$

$$\bar{Y} = \overline{y_{n-1}}\,\overline{y_{n-2}}\,\overline{y_{n-3}} \ldots . \overline{y_0}$$

In complementary mode multiplication the partial products are obtained by 'OR' operation rather the 'AND' operation (in normal multiplication the 'AND' operation occurs by default). However in decimal system there is no operator that acts as a Boolean analogue of 'OR' operator. The way the 'OR' operation is achieved as follows.

Let us consider a term obtained from the multiplication of multiplicand ($\bar{X}$) and the first bit of the multiplier ($\overline{y_o}$). The term would look like

$$\left(-2^{n-1}\overline{x_{n-1}} + \sum_{i=0}^{n-2} 2^i \overline{\mathrm{x}_i}\right)(1 - \overline{y_o}) + (2^n - 1)\overline{y_o}$$

The above expression implies that if the value of the bit $\overline{y_o}$ is '0' then the multiplicand is to be considered. However if the value of $\overline{y_o}$ is '1' then the partial product should have all the bit values equal to '1'. This sequence implies adding $(2^n - 1)$ in the decimal system. For subsequent terms, a factor of $2^i(2^n - 1)$ would occur in place of the second expression in parentheses above. In addition as discussed earlier in complement mode a carry-in is added for each stage. However the carry-in's would be shifted as the successive partial products are shifted too.

$$\overline{Z} = \bar{X} * \bar{Y}$$

$$\begin{aligned}\overline{Z} = &\left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)\left(1 - (1 - \mathrm{y}_0)\right) + (2^n - 1)2^0(1 - \mathrm{y}_0) \\ &+ \left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)2^1\left(1 - (1 - \mathrm{y}_1)\right) + (2^n - 1)2^1(1 - \mathrm{y}_1) \ldots \\ &- \left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)2^{n-1}\left(1 - (1 - \mathrm{y}_{n-1})\right) \\ &+ (2^n - 1)2^{n-1}(1 - \mathrm{y}_{n-1}) + (\sum_{i=1}^{n-1} 2^i)\end{aligned}$$

$$\begin{aligned}\overline{Z} = &\left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)(\mathrm{y}_0) + (2^n - 2^0)(1 - \mathrm{y}_0) \\ &+ \left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)2^1(\mathrm{y}_1) + (2^n - 1)2^1(1 - \mathrm{y}_1) \ldots \\ &- \left(-2^{n-1}(1 - x_{n-1}) + \sum_{i=0}^{n-2} 2^i(1 - \mathrm{x}_i)\right)2^{n-1}(\mathrm{y}_{n-1}) - (2^n - 1)2^{n-1}(1 - \mathrm{y}_{n-1}) \\ &+ (2^n - 2)\end{aligned}$$

$$\overline{Z} = \left(2^{n-1}x_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(2^{n-1}\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right)\left(\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right) + \left((2^{n-1}-1)\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right)$$

$$+ (2^n)(2^{n-1}-1) - \left(2^n\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) + \left(\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(\sum_{i=0}^{n-2}2^i\right) + (2^{2n-2}\mathrm{y}_{n-1})$$

$$- (2^{2n-2}\mathrm{x}_{n-1}\mathrm{y}_{n-1}) - \left(2^{n-1}\mathrm{y}_{n-1}\sum_{i=0}^{n-2}2^i\right) + \left(2^{n-1}\mathrm{y}_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right) - 2^{2n-1}$$

$$+ (2^{2n-1}\mathrm{y}_{n-1}) + 2^{n-1} - (2^{n-1}\mathrm{y}_{n-1}) + (2^n - 2)$$

$$\overline{Z} = \left(2^{n-1}x_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(2^{n-1}\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right)\left(\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right) + \left((2^{n-1}-1)\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right)$$

$$- \left((2^n-1)\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) + (2^{2n-1} - 2^n - 2^{2n-1} + 2^{n-1} + 2^n - 2 - 2^{n-1} + 1)$$

$$+ (2^{2n-2}\mathrm{y}_{n-1}) - (2^{2n-2}\mathrm{x}_{n-1}\mathrm{y}_{n-1}) - (2^{n-1}\mathrm{y}_{n-1}(2^{n-1}-1)) + \left(2^{n-1}\mathrm{y}_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right)$$

$$+ (2^{2n-1}\mathrm{y}_{n-1}) - (2^{n-1}\mathrm{y}_{n-1})$$

$$\overline{Z} = \left(2^{n-1}x_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) - \left(\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right)\left(\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right) + \left(2^{n-1}\mathrm{y}_{n-1}\sum_{i=0}^{n-2}2^i\mathrm{x}_i\right) - (2^{2n-2}\mathrm{x}_{n-1}\mathrm{y}_{n-1})$$

$$- \left(2^n\sum_{i=0}^{n-2}2^i\mathrm{y}_i\right) + (-1) + (2^{2n-1}\mathrm{y}_{n-1})$$

$$Z = -\left[2^{n-1}\mathrm{x}_{n-1}\left(\sum_{i=0}^{n-2}2^{i}\mathrm{y}_{i}\right)\right] - \left[2^{n-1}\mathrm{y}_{n-1}\left(\sum_{i=0}^{n-2}2^{i}\mathrm{x}_{i}\right)\right] + \left(\sum_{i=0}^{n-2}2^{i}\mathrm{x}_{i}\right)\left(\sum_{i=0}^{n-2}2^{i}\mathrm{y}_{i}\right) + (2^{2n-2}\mathrm{x}_{n-1}\mathrm{y}_{n-1})$$

$$-\left(2^{n}\sum_{i=0}^{n-2}2^{i}\mathrm{y}_{i}\right) + (-1) \quad + (2^{2n-1}\mathrm{y}_{n-1})$$

$$\overline{Z} = -\mathrm{Z} - 1 - 2^{n}\left[-(2^{n-1}\mathrm{y}_{n-1}) + \left(\sum_{i=0}^{n-2}2^{i}\mathrm{y}_{i}\right)\ \right]$$

$$\overline{Z} = -\mathrm{Z} - 1 - 2^{n}\mathrm{Y}$$

3. Case Study of Integer Multiplication in Complement Mode

In Erika et al. [1] it was demonstrated that an adder suffers considerably from uneven duty cycles at the nodes. It was also discussed that multiplication would also suffer from the same effect of unbalanced duty cycles and that complement mode execution could be used in it too. The implementation aspect of multiplication in complement mode, however has certain caveats not explored in that paper.

As shown above in the analytical discussion of multiplication, the implantation changes can be summarized as follows –

1. A carry-in needs to be added for each pair of partial product addition. This arises from the fact that addition in complement mode needs a carry-in to get the correct result.
2. At the cost of a large overhead in hardware, the operands should be extended to full width. For example in a 32x32 bit multiplication giving a 64-bit result in normal mode, in complement mode the operands need to be sign extended to 64 bits (before multiplication) and then the first 64-bits of the results would be complement of the actual result.
3. Another implementation approach that does not require sign extension and which is implemented in this work aims at reducing the hardware cost of multiplying operands in full width. In this case, the operands are multiplied as such (32-bits) and a carry-in is needed in the 33^{rd} bit (n+1) of the result. As is obvious the first 32 bits are always complement of the true result.

The example below shows multiplication of two 4-bits operands giving an 8-bit result.

Table 1: Multiplication in Normal and Complement Mode without sign extension

	Normal Mode								Complement Mode							
Mutiplier					1	1	0	1					0	0	1	0
Multiplicand					0	1	0	1					1	0	1	0
Carry-in's													1	1	1	
Partial Products					1	1	0	1					0	0	1	0
				0	0	0	0	0				1	1	1	1	0
			1	1	0	1	0	0			0	0	1	0	0	0
	0	0	0	0	0	0	0	0		1	1	1	1	0	0	0
	0	1	0	0	0	0	0	1	1	0	1	0	0	1	1	0
Add '1' to 5th bit in complement mode												1				
Corrected Result	0	1	0	0	0	0	0	1	1	0	1	1	0	1	1	0

As can be seen from the above example, multiplication in complement mode involves generation of the partial products using 'OR' operation instead of 'AND' operation. The partial products are then added together with the carry-in shown in light grey. After the addition, the first 4 bits are complement of the true result, while the latter 4 are not. However with the addition of '1' at the 5^{th} bit position, the result has all the bits complement of the true result.

3.1 Wallace Tree Multiplication

Wallace tree multiplication is one of the most commonly used techniques for adding the partial products in multiplication [2]. The example above demonstrated multiplication in complement mode and it can be seen that it is suitable to be used for Wallace tree addition of the partial products. The Wallace tree implemented (for 32-bit multiplication) has a partial product generation stage, 5 reduction stages and a final carry propagate stage to get the result. The implementation can be understood from the figure below. The figure shows partial product generation for '4' bit operands with addition using the CSA tree. The final stage consists of carry look-ahead adder.

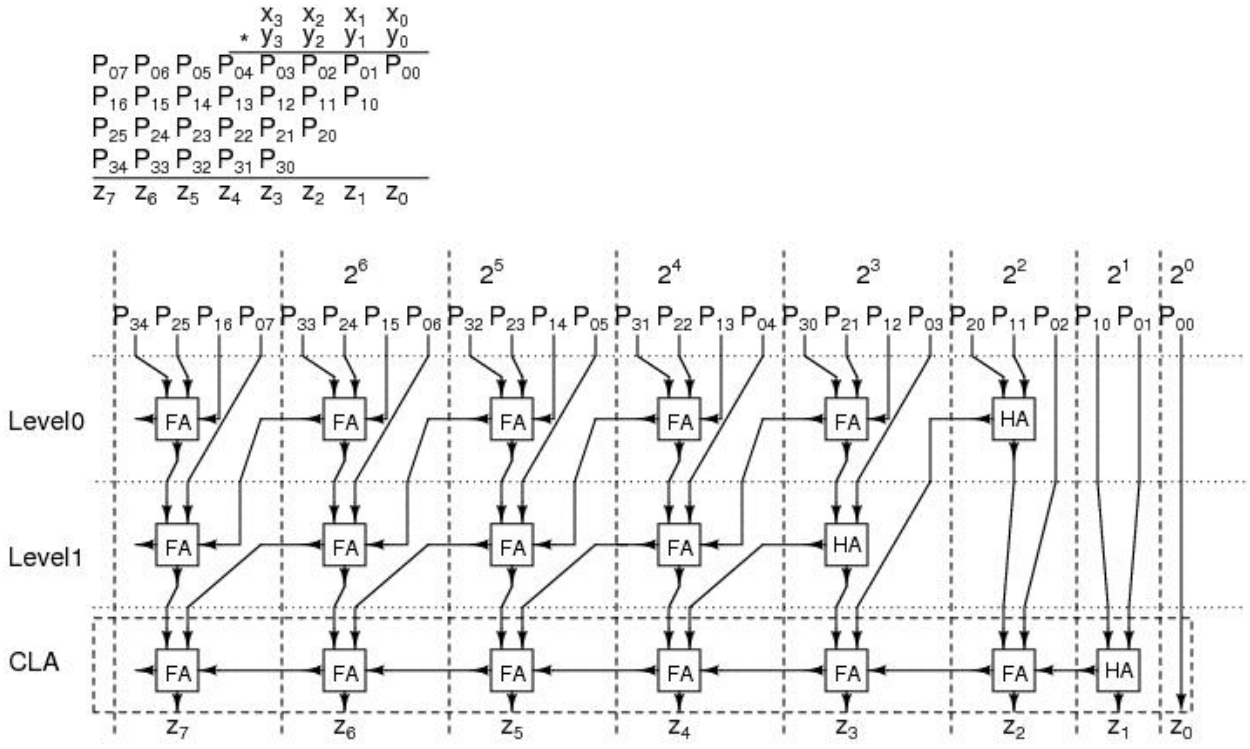


Figure 1 : Wallace Tree Multiplication [3]

To implement multiplication in complement mode with support for true mode multiplication, a complement bit input acts a select for partial product generation from 'AND'/'OR' operations, depending upon the mode. The required carry-in of '1' for the partial products is used from the complement bit. In different stages of the Wallace tree the complement bit is added to half adders, thus converting those to full adders. In this way, with minimal impact on timing and area, the carry-in's are included in the design. It must be noted that the complement bit is added only once for a bit weight. There are cases where a half adder is not used in any stage for a bit weight and the complement bit is added in the final ripple carry stage. (In the documentation under integer multiplication an excel sheet has been added which demonstrates the complete break-down of each level of the 32x32 bit multiplier implemented in this design).

The designs were synthesized at 110nm technology. As expected complement mode multiplier incurs higher area and delay due to extra circuitry in the critical path. The complement mode multiplier incurs a delay penalty of 11% and an area penalty of 11.4%. The modules were synthesized with max_area parameter set to '0' and area_effor low.

3.2 Booth Recoded Multiplication

While Wallace tree reduction of partial products is an effective technique to speedup multiplication, there exists a method which can reduce the number of partial products itself. This method involves re-coding the multiplier and using that encoding to generate the partial products and is known as booth recoding [4].

3.2.1 Radix-4 Booth Recoding

To recode the multiplier term in Booth radix-4 encoding, bits in blocks of three are considered, such that each block overlaps the previous block by one bit. Grouping starts from the LSB, and the first block only uses two bits of the multiplier (since there is no previous block to overlap) and a '0' is appended at the LSB end. The overlap is necessary so that information from what happened in the last block is known, as the MSB of the block acts like a sign bit. The recoding is shown in the following table

Table 2 : Booth Recoding

Block	Partial Product
000,111	0
001	1 * Multiplicand
010	1 * Multiplicand
011	2 * Multiplicand
100	-2 * Multiplicand
101	-1 * Multiplicand
110	-1 * Multiplicand

For ‘000’ and ‘111’ the partial product ‘0’ implies no change in the ongoing product. For 100, 101, and 110 encodings the negative sign implies 2’s complement of the multiplicand followed by a left shift in case of ‘100’.

Implementation of complement mode multiplication using booth encoding requires careful attention to ensure complementation of the partial products. In complemented mode the multiplier comes complemented and hence the recoding would also be opposite to the recoding in normal mode. For example recoding for a bit pattern ‘110’ booth recoding would be add 2’s compliment of the multiplicand while in complement mode this bit pattern would be ‘011’ and the encoding would be to add the multiplicand as it is. However the problem here is that since the multiplicand is itself complemented its gets normalized and no compensation occurs in the duty cycle during complemented mode as the partial products are again in true form. To prevent this, the multiplier is always kept in true form. Another caveat includes ‘XORing’ carry-in bits with the complemented bit for partial product generation. This is expected as addition in complement mode requires a carry-in if it is ‘0’ in normal mode and vice versa. This is illustrated in following example.

Table 3: Example demonstrating Booth 4-bit radix-4 multiplication in normal and complement mode

	Normal Mode								Complement Mode							
Multiplicand	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1
Multiplier					0	1	0	1					0	1	0	1
Booth Recoding					1		1						1		1	
Partial Products														1		
	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1
	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
Result	1	1	1	0	1	1	0	0	0	0	0	1	0	0	1	1

The multiplicand needs to be sign extended to get the correct result. The optimization found in case of normal multiplication has not been explored in this case and could be pursued in future for booth multiplication.

The booth encoded multiplier was implemented as a multi-cycled unit. The implementation involves adding a partial product on each cycle to the running product. This takes a total of $\lceil n/2 \rceil$ cycles for n bit numbers. Synthesis at 110 nm node shows an area penalty of 22%. The booth radix-4 multiplier was studied as it is implemented in OpenSPARC T1. The study could not be completed in time and the section on OpenSPARC T1 integer multiplication only includes a discussion of the multiplier used there.

4. OpenSPARC T1 Processor

A SPARC T1 core is composed of four threads and each thread consists of an eight-windowed register file. The threads share the instruction cache, the data caches, and the TLB. Each instruction cache is 16Kbytes with 32-bytes line size. The data caches are write-through, 8 Kbytes, and have 16-byte line size. Figure 2 shows the in-order pipeline implemented in a SPARC T1 core. A T1 core is a six 6 stage pipeline with one extra thread select stage as compared to a simple MIPS pipeline.

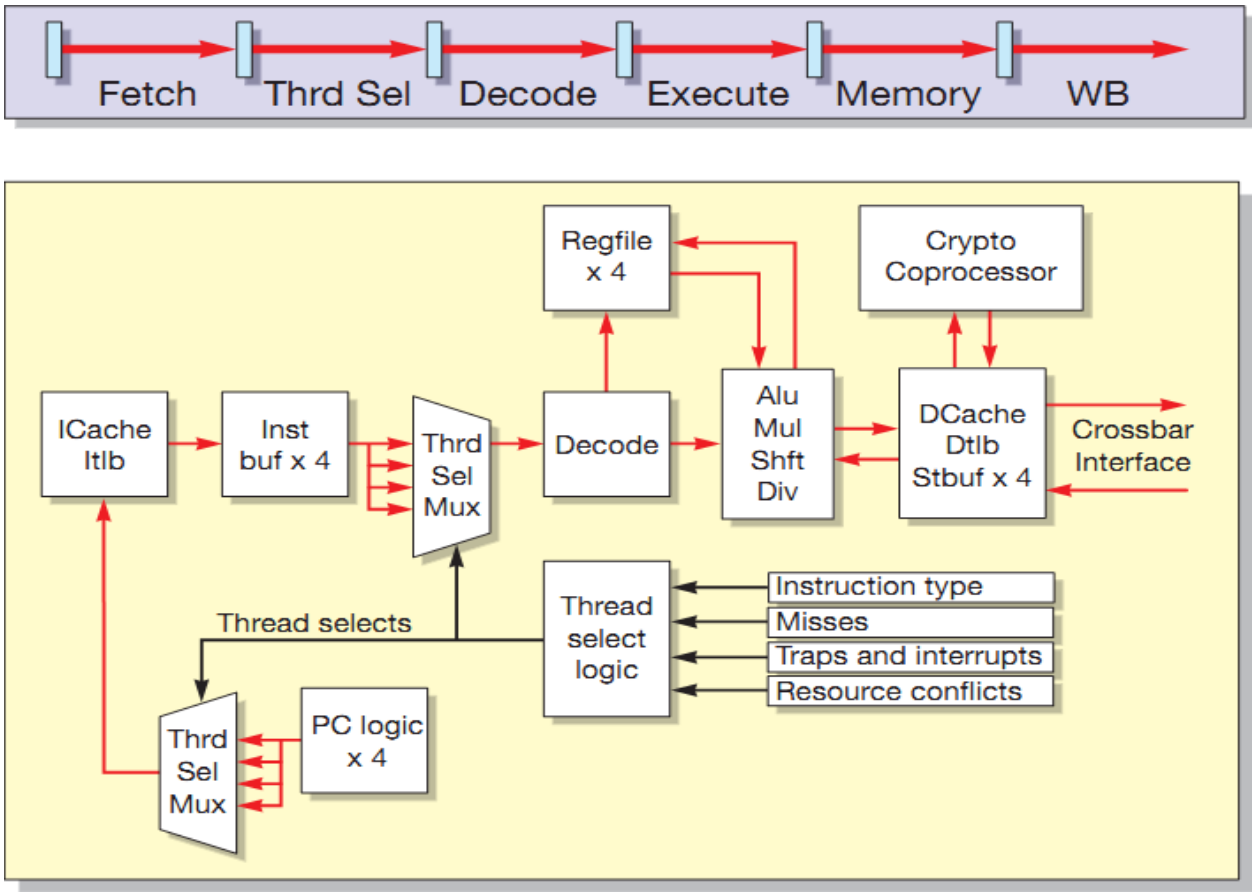


Figure 2: SPARC T1 core pipeline [2]

4.1 Instruction Fetch Unit

The instruction unit encompasses the following stages of the pipeline – fetch, thread select, and decode. This stage also includes the instruction cache and the TLB. Instruction cache is divided into Tag, Valid and Data units. The Tag unit has a single port while the Valid unit has a read and a write port. In case of an invalidation the Valid unit is accessed and not the Tag unit.

The fetch unit also consists of a 64 entry associative TLB with autodemap to prevent multiple hits when a new entry is installed with the same virtual number (VPN). The integer register file consists of 3 read / 2 write / 1 transport port and 5Kbytes. In total there are 640 64-bit registers with ECC. At a time a thread sees its 32 registers from the current window. A window change occurs with a thread switch.

4.2 Execution Unit

The execution unit has a single arithmetic unit (ALU) and a shifter unit. The ALU is reused with the branch prediction and virtual address translation. Multiplier is a separate unit and is pipelined to increase throughput. The multiplier gives a throughput of half-per-cycle and has a 5 cycle latency. There is also a divide units which allows one divide per outstanding core. A thread issuing a multiply / divide operation will be switched out if the units are busy.

4.3 Load / Store Unit

The data cache is 8Kbytes, 4-way and 16-byte line size. It also has a single port Tag unit. Similar to the instruction cache the data cache also has a 1 read / 1 write Valid unit to hold cache state for invalid or valid blocks. The cache has a pseudo- random replacement. The cache follows write-no-allocate policy while loads are allocating. The data TLB operates similar the instruction TLB.

The load store unit has 8 store buffer units for each thread making a total of 32 per core. The LSU has the interface logic to talk to the CPU-cache crossbar (CCX). The unit performs several functions which include prioritizing requests to the cross, forming packets for the processor-cache crossbar (pcx) and request priority. The LSU handles the ordering for cache invalidations and updates.

4.4 Floating Point Unit

The single floating unit is shared between all the eight SPARC cores in T1. The floating point unit has a front end unit that queues request from different cores. The unit is fully pipelined with a throughput of one instruction per cycle.

4.5 L2-Cache

The L2 cache is a four-way banked cache with a total size of 3-Mbytes and a line size of 64-bytes. The cache is 12-way set-associative with pseudo LRU replacement policy. The 64 bytes are interleaved between different banks. Pipelined latency in the L2-cache is 8 clocks for a load, 9 clocks for an I-miss, with the critical word first. The L2-cache directory shadows the L1 tags. A data cache miss in L1 not present in L2 cache would take 23 cycles access time.

5. Implementation of Complement Mode Execution in T1

To implement the Colt duty cycle equalizer several changes need to be implemented at the microarchitectural level. Erika et al. outline the salient features that would need to be incorporated in an existing architecture [1]. Prominent amongst these changes are:

1. Complementary mode operation of the data path, control path and storage elements. True and complementary mode operation ensures an almost completely balanced duty cycle and utilization of PMOS and NMOS devices. Thus negative impact of BTI effect is minimized. Some changes in the microarchitecture include complimenting register file data in complemented mode, normalizing opcode in the complement mode to avoid complexity in decode, issuing proper control signals for the logic elements in the ALU, proper sign extension in the complemented mode etc.
2. Rotation of cache sets using a Linear Feedback Shift Register (LFSR) to randomize the index between different epochs. LFSR generates pseudo random patterns such that the patterns can be made to cover all the sets of the cache.
3. It was observed that the part of the data path serving the immediate operands or unary operations was much more active that the right part, resulting in an imbalance in the duty cycle. Swapping of the left and right operands in the data path can be done to equalize the duty cycle for the immediate operands.

In the following subsections the implementation details of some stages of the T1 pipeline is presented.

5.1 Cache

The implementation of complement mode operation in the caches would involve complementing the data coming from L2 cache and flushing the cache between complement and true mode epochs. Figure 2 below shows the microarchitecture of L2 cache implemented in T1 processor.

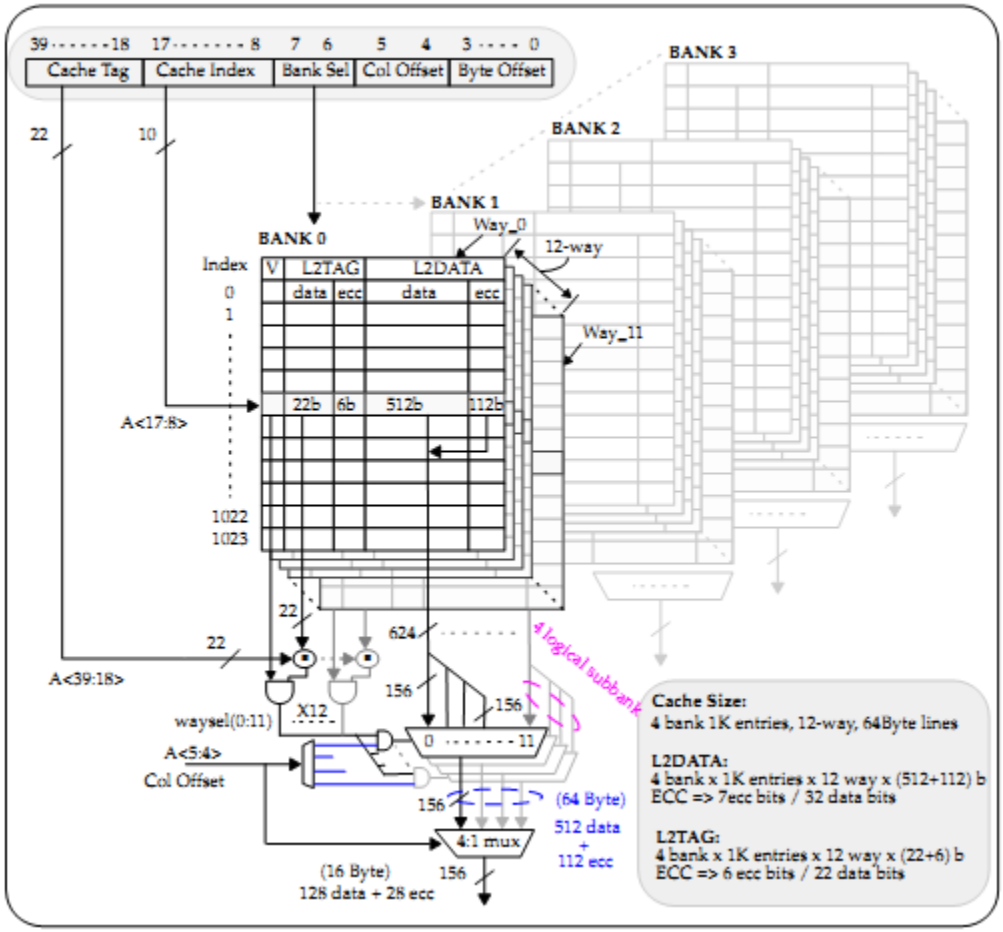


Figure 3 : L2-cache simplified addressing and data overview [5]

The implementation would involve normalizing the column offset to always keep it in the true form. Data coming from the L2 cache (*dcache_wdata_e[143:0]*) would be complemented in the complement mode operation. Further the way select input in L1 cache (*dcache_wr_way_e[3:0]*)

would be normalized to keep in the true form. This is essential as the correct way should be read from the cache. On a cache read in complemented mode, complemented data is read out.

The flow of data is as follows: - The fetch queue gets data from 4 sources – forwarded data packet which is previous entry in the queue, in queue packet from load-store unit (flopped *lsu_ifu_cpxpkt_i1*), extended store instruction packet, or load-stores unit packet via crossbar. The bits 95:0 in the cpx packet represent data bits and the remaining are control/status bits. Inside the *lsu_qdp2* module the output *dcache_iob_addr_e* is used to enable the bytes to be written to the data cache which should be in true form.

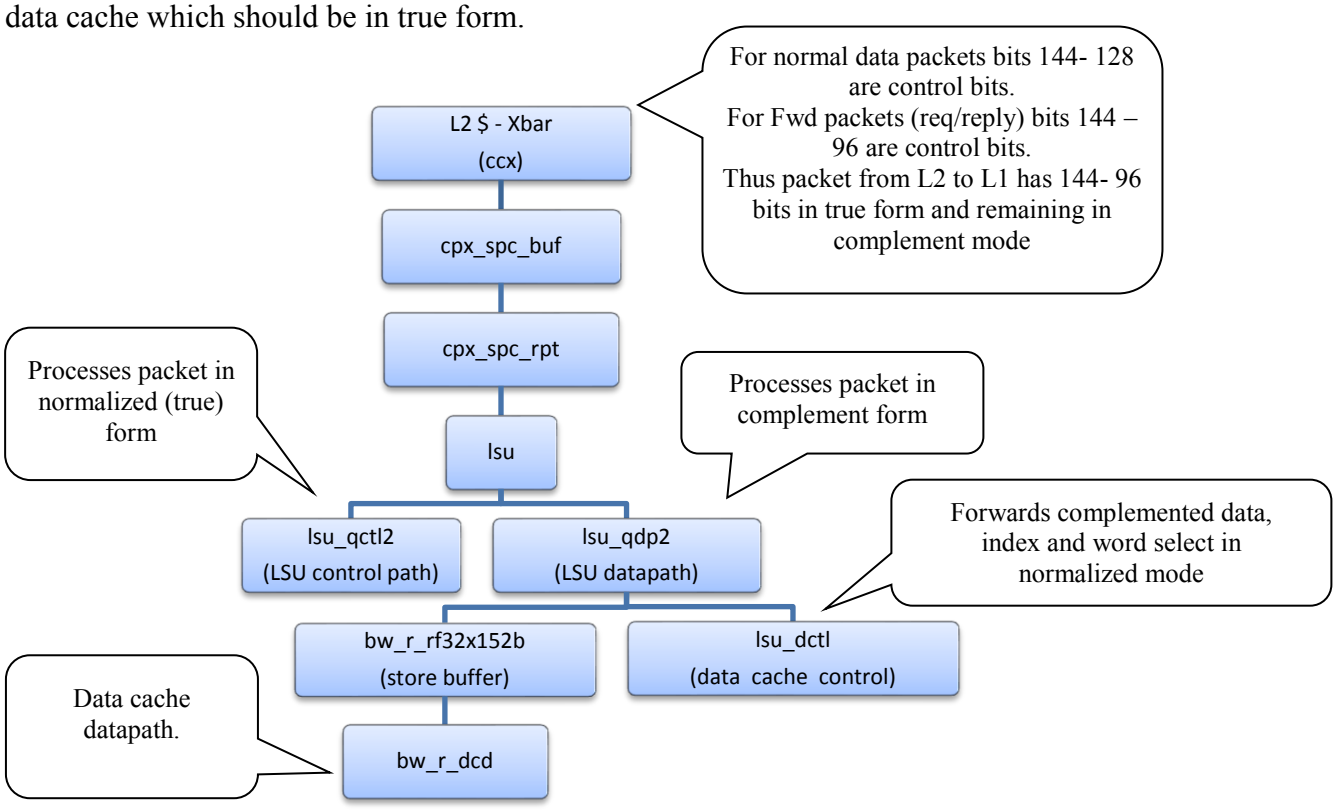


Figure 4 : Dataflow from L2 cache to Data cache

5.1.1 Instruction Cache Data Unit

The instruction cache is physically divided into data and tag parts. The instruction cache is a single ported SRAM with 4 ways x 128 entries x 272 bits (16 parity bits). The data written is 34b

(32b inst + parity + predec bit). In total it's a 17KB cache with 16KB data and 1KB parity and predecode bits. Each cache line is contained in one bank and each bank stores eight instructions (with 2 parity bits per instruction).

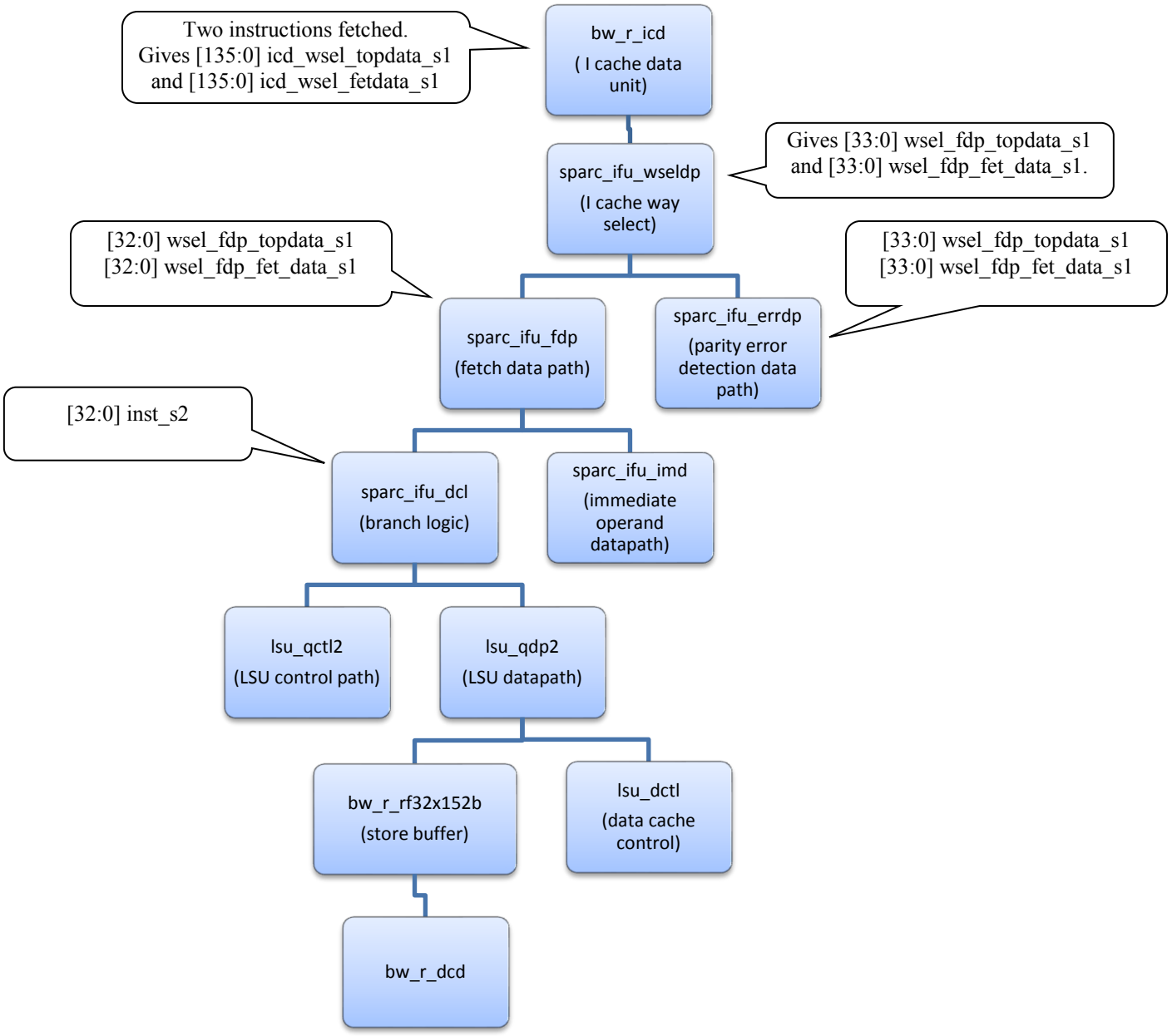


Figure 5: Instruction Cache Dataflow

On a cache read the upper seven bits of the index are used for selecting one of the 128 entries and last 3 bits are used for selecting one word out of the 8 words read from a bank. Each bank finally gives two instructions. *Wrway* signal is then used to select the appropriate way.

On a write, 136 bits are selected to be written. The signal *ifq_icd_wrway_bf* decides in which way would the word be written while *ifq_icd_worden* decides which 34 bits of the incoming bits would be written.

In the complement mode the data being written into the cache from the L2 cache needs to be complemented. The instruction to be fetched is obtained from the *LSU* unit. The instruction could either be in the data fill queue or could be coming from the L2 cache across the crossbar.

The data to be written is selected from previous data (wrdata_f), L2 cache fill (ifq_icd_wrdata_i2), or bist data (bist_data_expand). The L2 cache fill data needs to be inverted in complement mode operation. However it is not that straightforward as the data should not have been used by any other unit before its complemented. The data comes from cpx_spc_buf as cpx_spc_data_cx2 into the SPARC module. Thus this is the most appropriate place to complement the data. Inside the SPARC module the data is sent to the *LSU* which handles load/store misses. The unit has separate datapath and control path modules. While data path module gets the complemented data the control unit gets the normalized data.

The data coming in the data fill queue (*dfq_rdata [151:0]*) is filled either from data coming from the processor pipeline (store operation) or from the L2 cache.

However the implementation does not completely decouple control logic from data and some code in the data path unit does contain control information. In the datapath module (*lsu_qdp2*) the bits of the packet from 135- 96 contain control information and hence need to be normalized. The output of the LSU unit finally goes to the fetch queue datapath (*SPARC_ifu_ifqdp*) in the

fetch unit (*SPARC_ifu*). The instruction field is partially decoded for switch condition before it is sent to the L1 cache. This decode should be done on the true form and hence the instruction is normalized (*SPARC_ifu_ifqdp*). However parity can be calculated on the complemented instruction.

The signal for the selection of the word to be written in the instruction cache must be in true form. The signal *ifq_icd_worden_bf[3:0]* is used to select the word to be written. The bits are obtained from the virtual address bits [4:3] in the *SPARC_ifq_dp* module which is in turn obtained from the *EXU* unit from the *ALU* output or bypass output. This virtual address could be from the register file, from *ALU* output or from the load store unit. These bits are sent in true form but stored in complemented form. In *lsu_excpctl* unit the last bit of the address is used for half-word alignment, bits [1:0] for word, bits [2:0] for double word and [3:0] for quadword alignment. In addition bits [5:0] are used for a cache block aligned request and hence all these bits must be in true form.

Since write can be done to a byte granularity the offset must be in true form. The offset bit is derived from the packet coming from L2 (bit 68) and the signal *wr_size* in *lsu_qctl*.

For the address part of the instruction cache the L2 cache gives four words out of which the appropriate word needs to be selected. The word selection bits (*worden[3:0]*) need to be in the true form to be consistent with L2 which is always in true form. The signal is obtained from the instruction fetch unit control logic (*SPARC_ifu_ifqctl*) and is normalized along with the rest of the field for address space identifier.

5.2 Instruction and Data Cache Tag Unit

For the tag unit all the four words are processed on every cache access. Thus there is no need for a word select in these units and hence no change is warranted here.

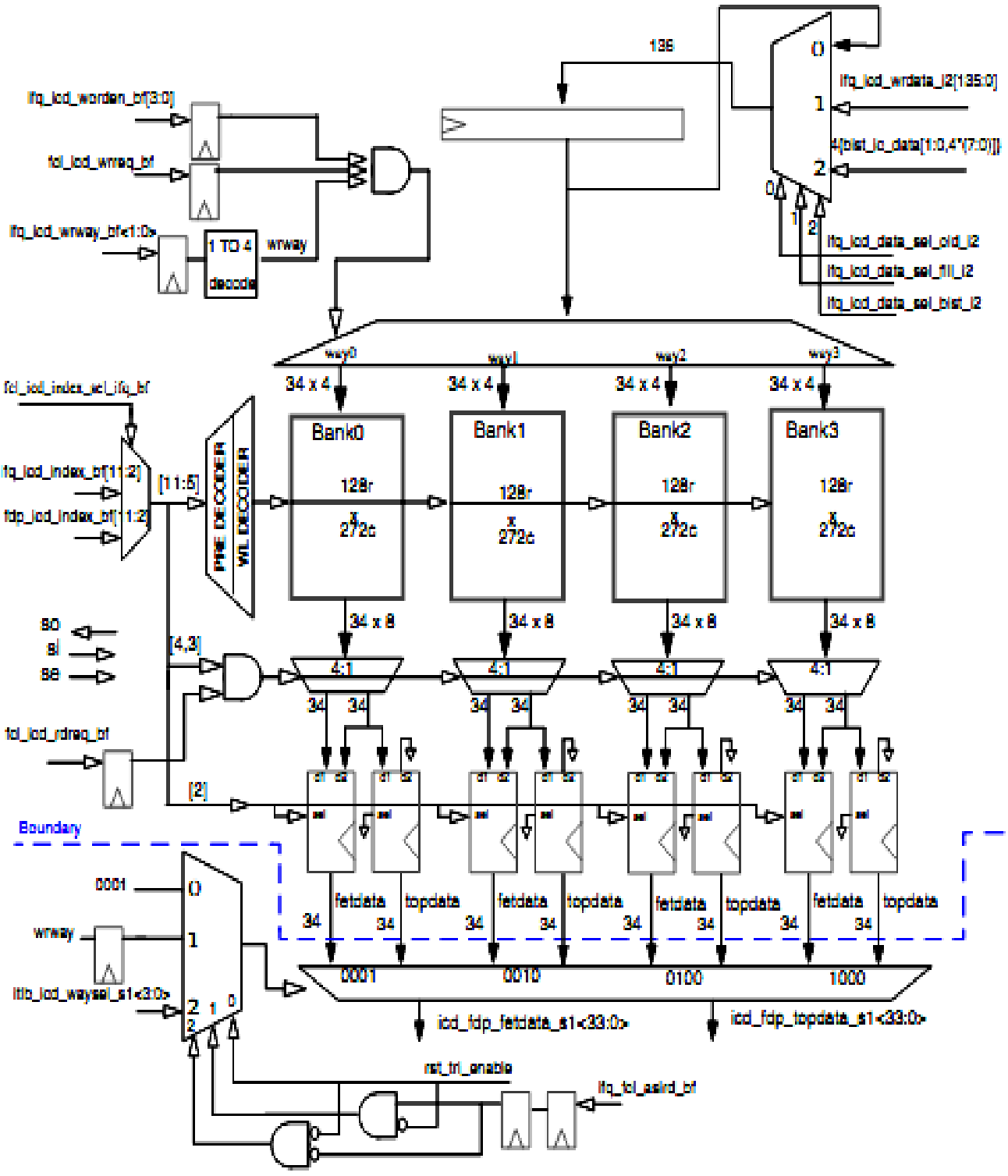


Figure 6: Instruction Data Cache

5.3 Data Cache Data Unit

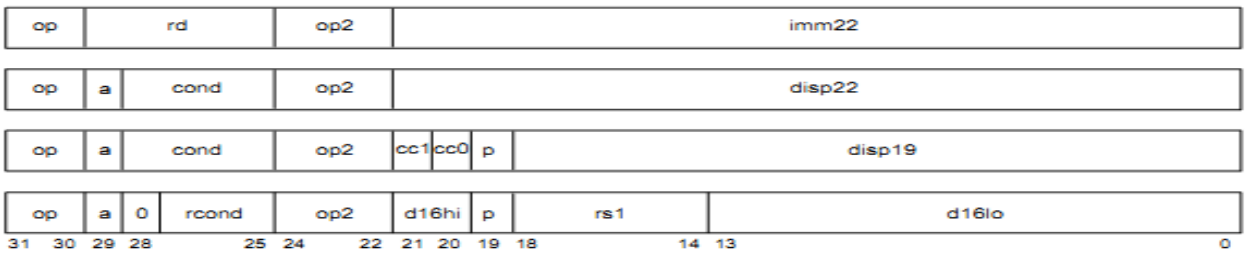
The data cache in T1 is arranged as two banks of 128x288. The signal *dcache_wr_rway_e* is used to select the way to be written and the signal *dcache_byte_wr_en_e* is used as a select/enable for each of the 16 words in one way. This signal should be in true form as the byte inside a word is not being shuffle as it would essentially amount to make a little endian machine a big endian machine. The signal *dcache_byte_wr_en_e* is an output from the module *lsu_dctl.* The Load/Store unit manages data coming from the crossbar (CCX) and forwards it to the appropriate unit (Figure 4). The threaded architecture of the LSU can process four loads, four stores, one fetch, one FP operation, one stream operation, one interrupt, and one forward packet. Therefore, thirteen sources supply data to the LSU.

5.4 Decode

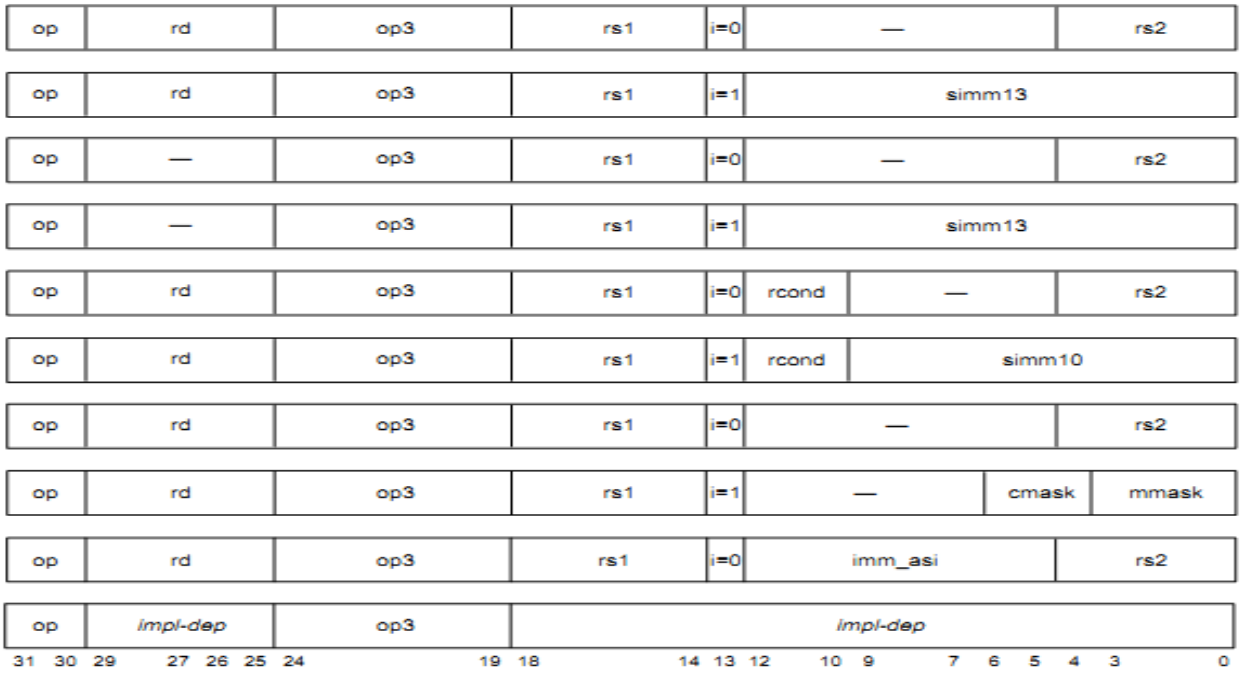
Several changes were needed for the decode unit as the control signals were to be sent out in true form. Figure 7 shows various instruction formats in T1 processor.

Format 1 (*op* = 1): CALL

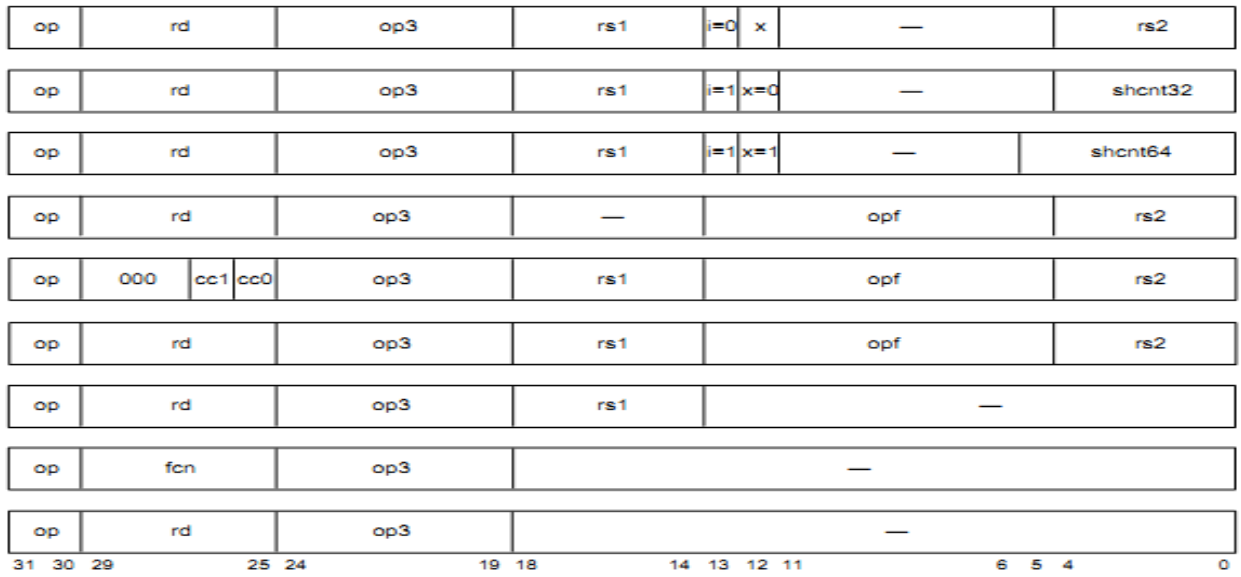
Format 2 (*op* = 0): SETHI & Branches (Bicc, BPcc, BPr, FBfcc, FBPfcc)



Format 3 (*op* = 2 or 3): Arithmetic, Logical, MOVr, MEMBAR, Load, and Store



Format 3 (op = 2 or 3): *Continued*



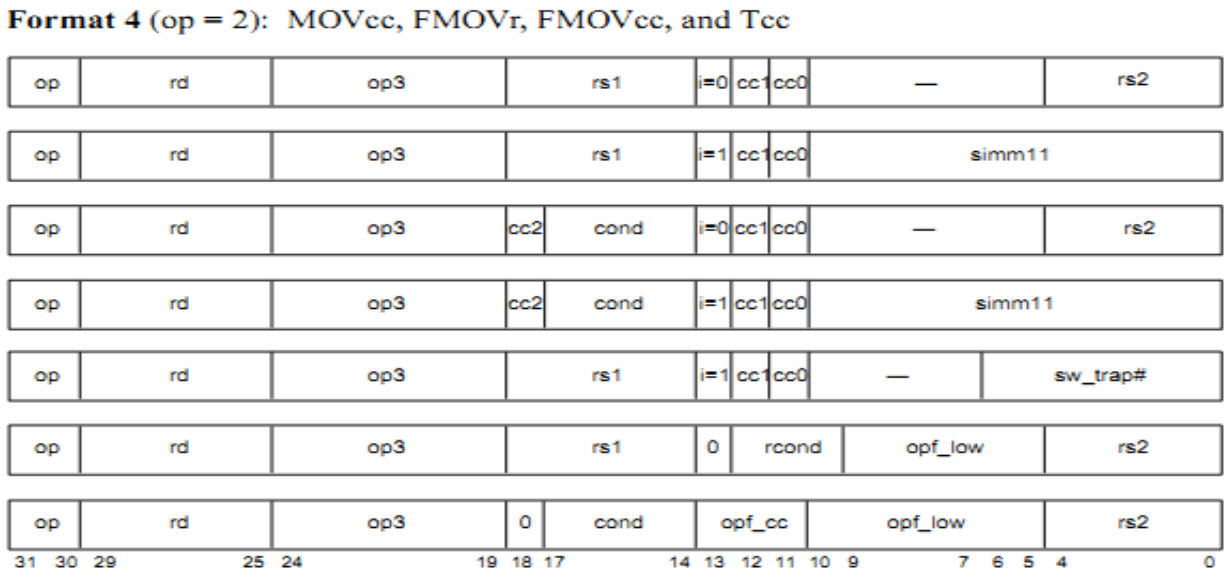


Figure 7 : Different Instruction formats in T1 Processor[5]

The decision to have control signals in true form was based on ease of implementation and verification. Further data cycle for these signals is expected to be balanced so they can be in true form. The changes made are listed for every module in the documentation provided on the disk. Some of the important features are changing the opcode to the normalized form, complementing instruction bits (other than opcodes) used for condition codes (cc), normalizing the register file identifiers. Effectively bits 31-30 (Op), 24-22 (Op2), 24-19 (Op3) and 13-5 (Opf) were normalized. Since the instruction format is fixed length the changes to the decode stage were quite straightforward.

5.5 ALU

The T1 ALU is controlled by the execute control logic (ECL). Figure 8 shows the high level design of the ALU with a few control signals. Figure 5 shows the logic unit in the ALU. The inputs bits to the MUX would need to be normalized and so is the inv_logic. The signal

ifu_exu_sethi_inst_e is used in case of a NOP as a SETHI instruction with rd = 0 and imm22 = 0 is defined to be a NOP instruction.

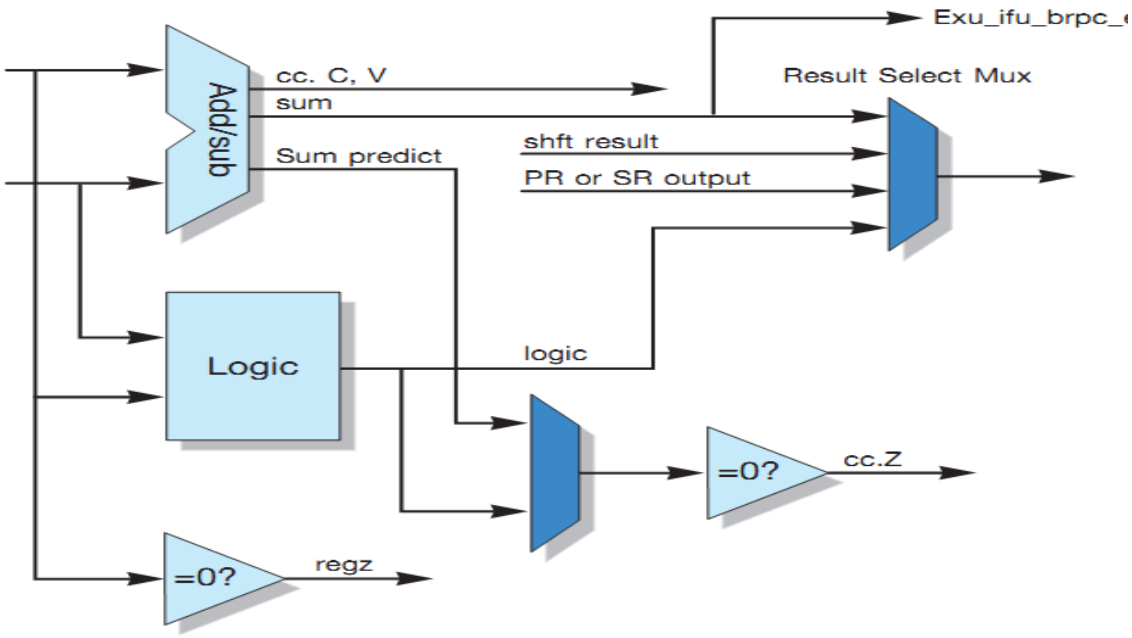


Figure 8: SPARC T1 ALU [5]

Most of the control signals originate from *ECL* unit. Numerous changes have been identified to be made in the logic like zero comparison, shift amount, changing control signals for ADD to OR etc and so on. Many instructions in SPARC V9 ISA modify the condition codes register (CCR) which holds integer condition codes. The condition codes register is as shown in Figure 9.

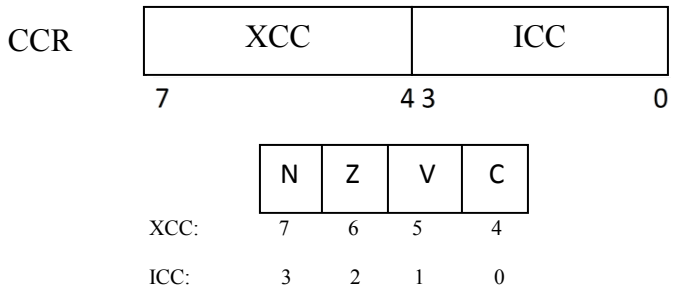


Figure 9: Integer condition codes (CCR_icc and CCR_xcc)

The xcc (icc) condition codes indicate the result of an operation when viewed as a 64-bit (32-bit) operation. The n bits indicate whether the 2's-complement ALU result was negative for the last instruction that modified the integer condition codes, 1 = negative, 0 = not negative. The z bits indicate whether the ALU result was zero for the last instruction that modified the integer condition codes, 1 = zero, 0 = nonzero. The v bits indicate whether the ALU result was within the range of (was representable in) 64-bit(xcc) or 32-bit (icc) 2's complement notation for the last instruction that modified the integer condition codes, 1 = overflow, 0 = no overflow. The c bits indicate whether a 2's complement carry (or borrow) occurred during the last instruction that modified the integer condition codes. Carry is set on addition if there is a carry out of bit 63 (xcc) or bit 31 (icc). Carry is set on subtraction if there is a borrow into bit 63 (xcc) or bit 31 (icc), 1 = carry, 0 = no carry.

5.5.1 Logic Unit

The logic unit implemented in SPARCT1 is illustrated in Figure10. The changes to be incorporated are in the control signals for the mux that selects the output for the logic block. In SPARC V9 ISA there are few instructions that involve logical negation of one of the operands before the logic operation is performed (andn, andcc, orn, and ornc). In a logical operation in complement mode normally instead of an AND an OR operation would be performed as the inputs are complemented. For instructions like 'andn' this operation would involve normalizing the second operand before the OR operation is performed in complement in place of an AND operation. There is however no change in the implementation of the complement mode and only the logical operation needs to be changed. This can be seen as follows for 'andn' instruction:

$$a \ \& \ \bar{b} = \overline{(\bar{a} \ + \ \bar{\bar{b}})}$$

In the implementation the logic for complementing the second operand (subtraction in 2's complement or addn etc) is in the decode stage. Since this decoding is done on normalized instruction there is no change here. The bit for sign extension (*ecl_alu_cin_e*) originates from decode stage and is processed in the execute control logic (ECL) unit of the *ALU.* The signal is complemented in the decode stage to take care complemented mode operation. Logic for two of the selection bits 'isand' and 'isor' for the mux shown in Figure 10, is modified in the *SPARC_exu_ecl* unit. This unit has the decoded alu_op signal from decode stage to decode further the control signals. The changes involve addition of a 2-1 MUX with 'complement' bit as the selector to select the appropriate logic type. 'XOR' operation does not involve any change neither does data forwarding with the *sethi* instruction. However the *sethi* instruction should now set the specified bits 'low' in complemented mode. Thus the bit *ifu_exu_sethi_inst_e* is 'xored' with complement bit to achieve this.

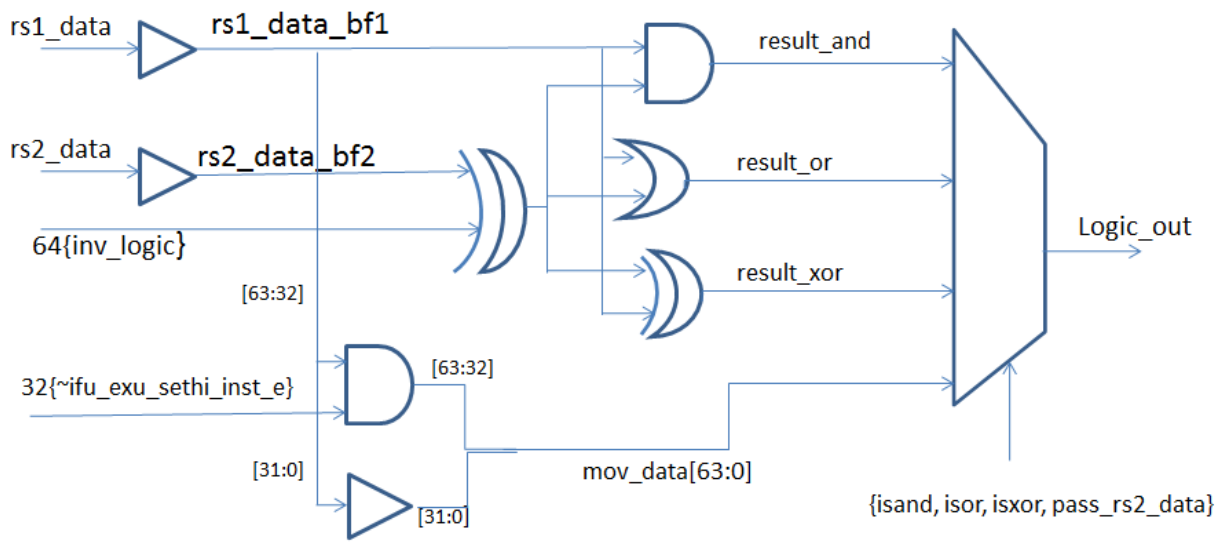


Figure 10: T1 ALU Unit

5.5.2 Adder Unit

Microarchitectural details of the arithmetic unit are depicted in Figure 11. The inputs from the register file are already complemented. The adder out would be obtained in complemented or true form based on the mode of operation. The adder unit (SPARC_exu_aluadder64) in T1 is not modeled at the gate level but at the rtl level (with a + sign). Hence additional details about the duty cycle of the nodes cannot be obtained. In 32 bit addition mode the upper 32 bits are made to zero in true form and one in complement mode.

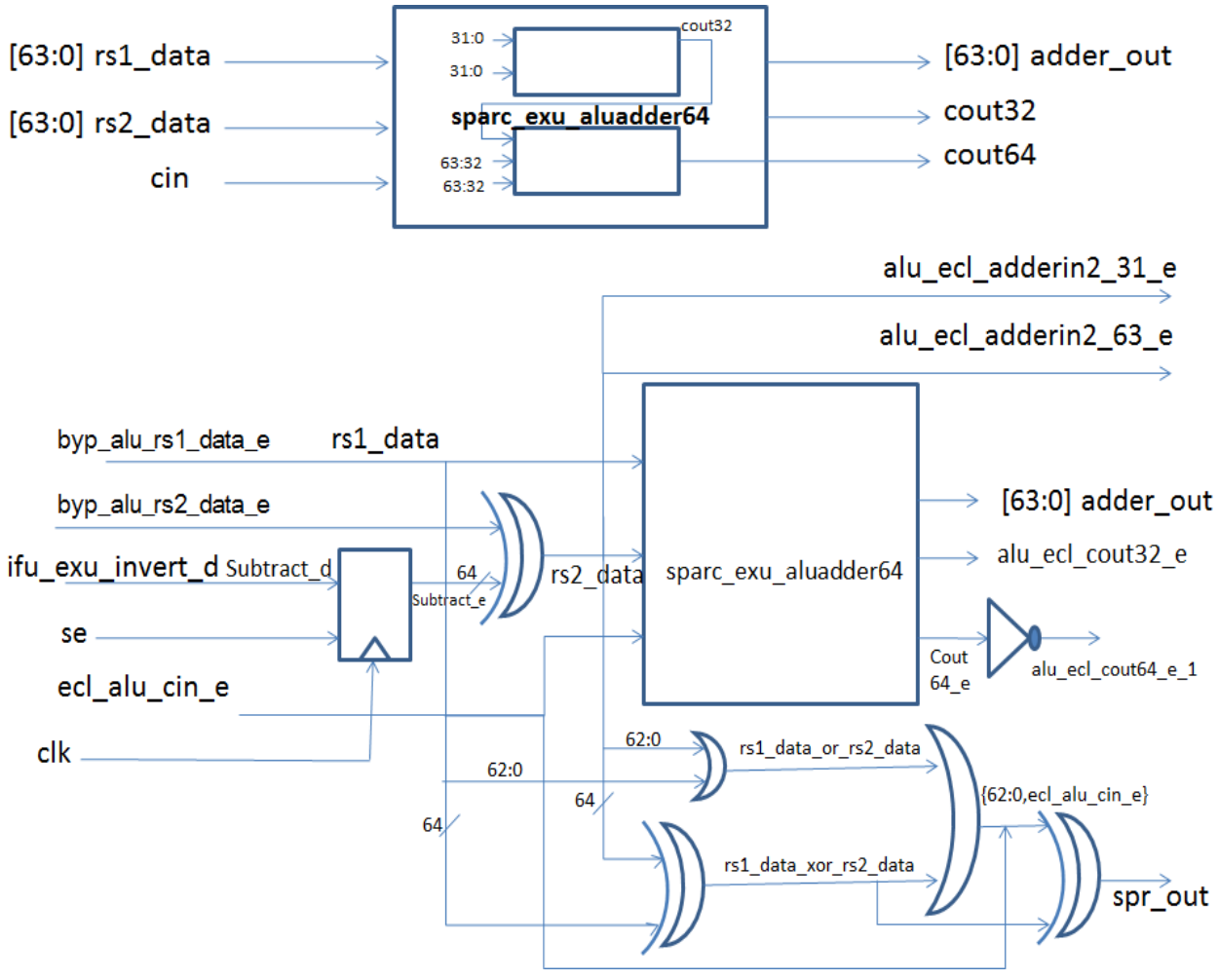


Figure 11: SPARC Arithmetic Unit[5]

5.5.3 Shifter

The shift unit is a 3 stage barrel shifter with a shift amount broken down into multiples of 16, 4 and eventually by mod 4. The left shift and the right shift is broken split in to two identical units with a MUX selecting the appropriate output (obtained from decode in true form). Figure 12 shows the implementation of the shifter with the internal signals. As such no change is needed in this unit and all the changes are implemented in ECL part of the ALU where it is essential to keep the shift amount in true form.

In SPARC V9 ISA the instruction format provides for shift from *rs2[5:0]* or immediate shift amounts ([6:0]). While the first four bits are decoded in *ecl* unit, the last 2 bits are decoded in the shift module (*SPARC_exu_shft*) itself. The shifter supports 32 and 64 bit operations and the selector bit *shift16_e* is derived from *ecl_shft_op32_e,* decoded in true (normalized) form.

During the shift operation, the instruction is decoded to find out whether it's an arithmetic or logical operation. The bit is logically 'anded' with the msb (separately for 32 and 64 bit operation) to get the sign extend bit. In complement mode this would remain in the complement state. Thus the sign extension bit for 32 bit operands *byp_ecl_rs1_31_e* and 64 bit operands *byp_ecl_rs1_63_e* does not need to be normalized. However during left shift operation the bits being shifted are by default '0' as shown in Figure 12 below. In complemented mode these bits should be '1'. This is implemented by replacing the '0' bit vectors with the complement bit vectors.

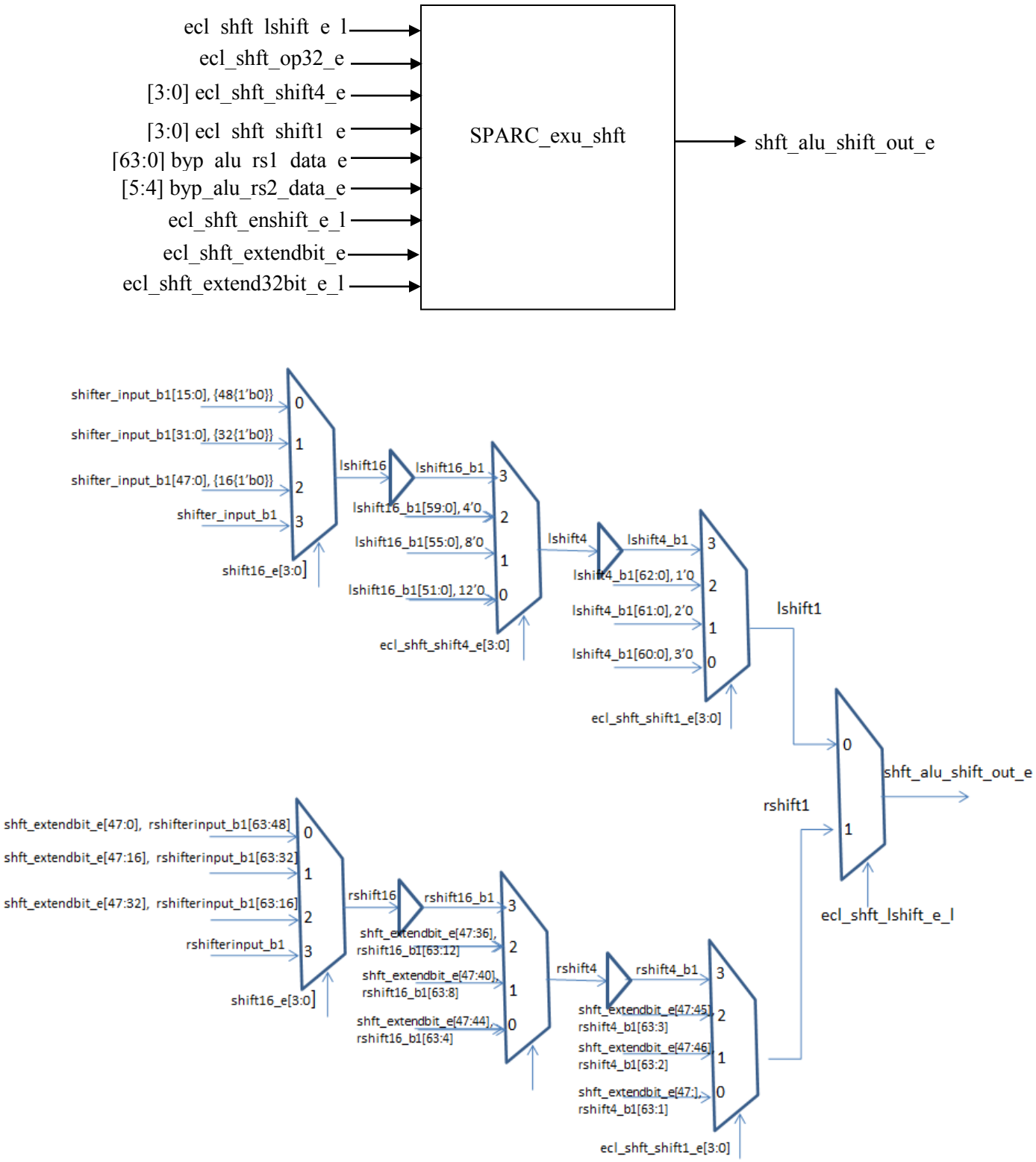


Figure 12: SPARCT1 shifter unit [5]

5.5.4 Zero Comparator

The zero comparator is used to check set the Z flag in the condition code register (refer Figure 9). The logic is based on a 2-input NOR of the input data. This checks if there is atleast one '1' in the input. This is followed by a 2-input 'NAND' followed by inversion. The results bits are finally 'NANDed' to get active low output.

Implementing this logic in complement mode can be done in two ways. One could be to normalize the input data there by adding another XOR in the datapath. The other could be implementing a parallel logic (using DeMorgan's Theorem) and finally using a MUX with select as the complement bit. This would depend upon the primitives in synthesizer and would affect timing and area. However for the current implementation the first choice was selected. A designer looking for possibly better cycle time could go for the second approach. The change was implemented in *SPARC_exu_alu* module where input to module *regzcmp* was normalized by XORing with complement bit.

5.5.5 Multiplier Unit

The multiplier unit (MUL) is shared between the stream processing unit (SPU) and the execute unit (SPU). MUL performs multiplication of two 64 bit operands. Figure 13 shows the interaction of the multiplier unit with EXU and SPU.

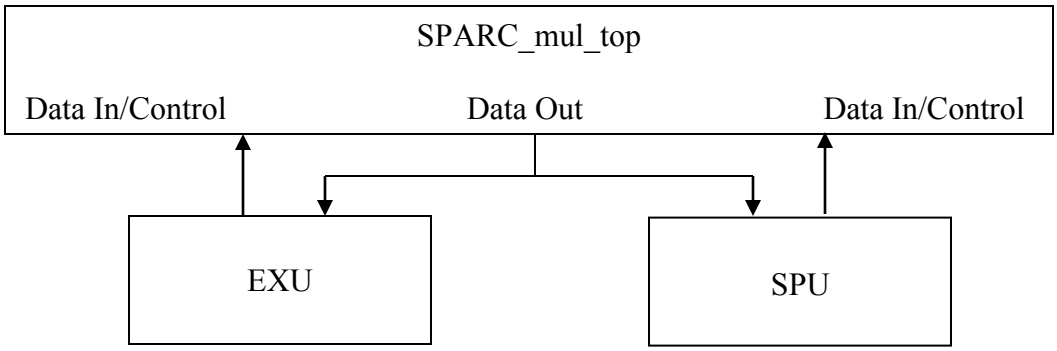


Figure 13: MUL Block Diagram

The multiplier for the floating point unit is the same module (*mul64*) as the integer multiplier and the difference is in terms of the control signals. SPARC_*mul_top* unit instantiates the control (*SPARC_mul_cntl*) and datapath (*SPARC_mul_dp*) modules for the multiplier. The multiplier is a six stage multiplier booth multiplier. The booth multiplier is a radix-4 multiplier [7-8].

The steps involved in integer multiplication process are outlined in the table below. In the first cycle the multiplier is enabled which indicates beginning of operation. In this step the booth radix-4 recoding is done. The formatting of operands is done in floating point multiplication. The documentation does not specify the implementation details of the booth recoding implementation and partial product generation. However it looks similar to the multiplier proposed by Ki-seon et al. [8] as it does recoding as 'direction', 'shift' and 'add'.

Table 4: Stages in integer multiplication

Stage	Action
M1	• Format input operands, booth recoder
M2 – M4	• Generate partial products using a radix-4 booth algorithm • Accumulate partial products using a Wallace tree configuration • Add the two Wallace tree outputs using a carry-propagate adder

The changes needed to be made in multiplier are surprisingly not much. To combat BTI the duty cycle of the internal nodes needs to be balanced (around 50%). In booth recoding, based on the encoding bits, the multiplicand may be complimented. In complemented mode the multiplier comes as complemented and hence the recoding would also be opposite to the recoding in normal mode. For example recoding for a bit pattern '110' booth recoding would be add 2's compliment of the multiplicand while in complement mode this bit pattern would be '011' and the encoding would be to add the multiplicand as it is. However the problem here is that since the multiplicand is itself complemented its gets normalized and no compensation occurs in the

duty cycle during complemented mode. To prevent this, the multiplier is always kept in true form. Other caveat include 'XORing' carry-in bit with the complement bit..

5.6 Register File

The register file in T1 is a windowed implementation with dedicated instructions like Save, Restore, Saved and Restored to manage it [6]. Some features of Save and Restore instructions are

- If Save does not trap, it increments CWP (mod NWINDOWS) by one , decrements CANSAVE and increments CANRESTORE.
- If new window is occupied a spill trap is generated. The trap is vectored based on value of OTHERWIN and WSTATE. CWP points to the window to be spilled (old CWP + 2).
- If CANSAVE !=0 … the SAVE checks if the new window needs to be cleaned. If yes then it invokes a clean_window trap with CWP set to the window to be cleaned which is old CWP + 1.
- %o6 (%r14) is used as %sp. Here the system can store %r16 … %r31 when a spill trap occurs in memory pointed by %sp.
- If Restore does not trap, it decrements CWP (mod NWINDOWS) by one , increments CANSAVE and decrements CANRESTORE. This restores the window that was in use prior to the last SAVE instruction executed by the current process.
- If new window is occupied a fill trap is generated (CANRESTORE == 0). The trap is vectored based on value of OTHERWIN and WSTATE. CWP points to the window to be filled (old CWP - 1).

In the complemented mode the output of the register file is complemented and complemented data is written into it. The zero register will now have all 1's stored in it. This can be done at ease as the implementation of the register file is quite clear in the code. Figure 14 shows the microarchitectural details of the read port and the signals associated with it. The output *irf_byp_rs1_data_d* would be complemented in complement mode operation.

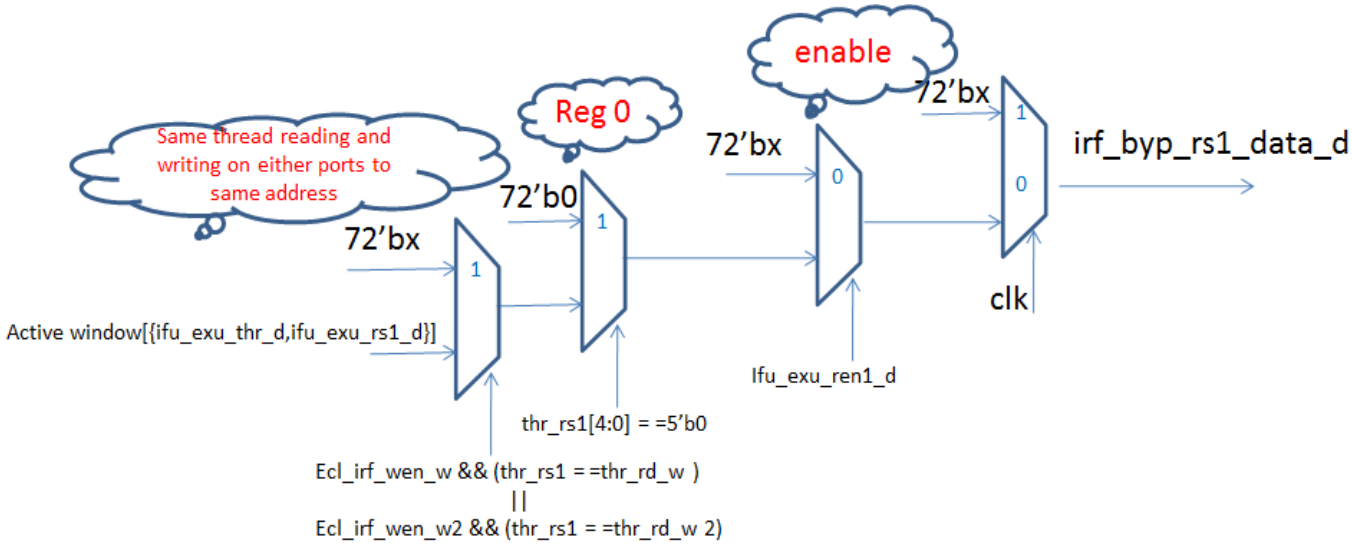


Figure 14: Integer Register file read port along with the associated signals [5]

5.7 Floating Point Unit

The floating point unit is shared by all the 8 cores and can support at the most one floating point operation. Floating point Front end Unit (FFU) manages simple instructions like mov, abs, negative etc while add, division and multiplication are managed by the Floating Point Unit (FPU). Figure 15 shows the FFU and its interaction with different logic blocks. The floating point register file (FRF) is managed by the FFU and is private to every core. The operands are sent over the interconnect to the FPU for the related instructions. In case the operand is 32-bit, it is encoded in the higher order bits of the 64-bit word. FRF has one R/W port and is 78 bits (64

bit data, 14 bit ECC). Bits [38:32] are the ECC bits for the lower word (data [31:0]) and bits [77:71] are the ECC bits for the upper word (data [70:39]). FFU_CTRL is the control unit of the FFU and generates the select signal for the multiplexors. It also decodes the opcode and contains the FSM for the FFU pipeline. The FFU_DP implements instructions like FMOV, FABS and FNEG. It also checks and generates ECC and directs data to be written to and read from FRF. The FFU_VIS is used to implement limited visual instruction set (VIS) instructions including partitioned add/subtract, logical operations, and falign data. All inputs and outputs are controlled by FFU_DP as shown.

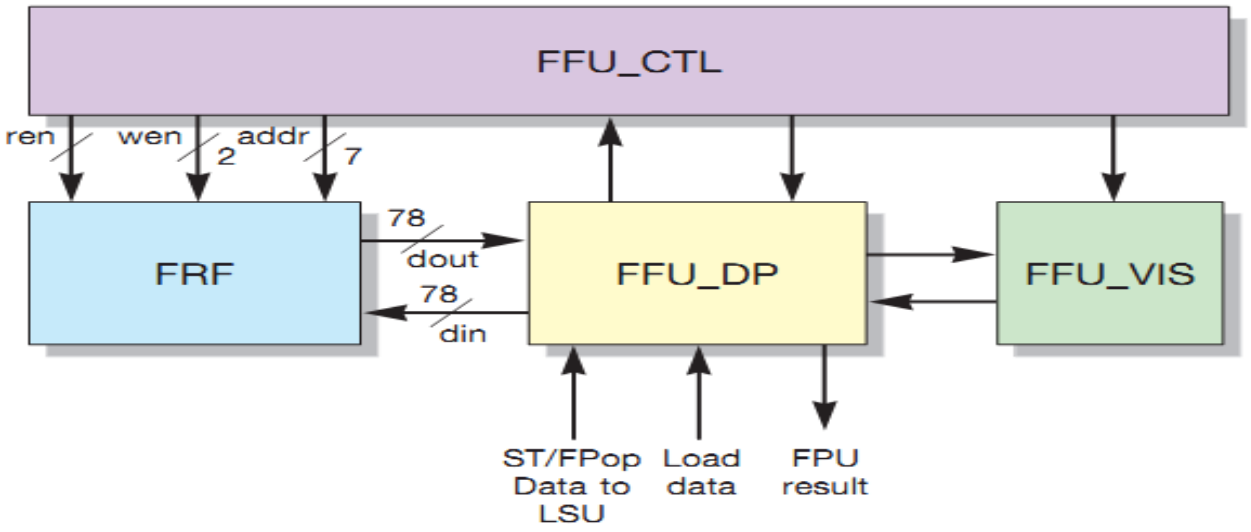


Figure 15 Top-Level FFU Block Diagram

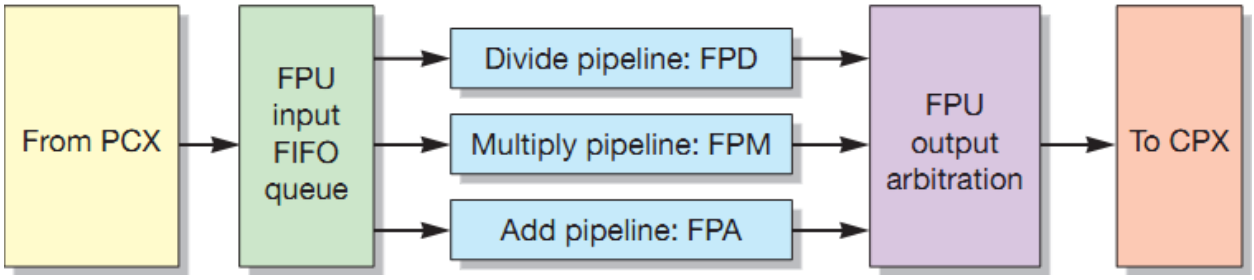


Figure 16 Floating Point Unit (FPU) functional block diagram

5.7.1 Floating Point Multiplication

Floating point multiplication is a 6 stage pipelined unit. The multiplication unit (*fpu_mul*) is split into three units viz. control (*fpu_mul_ctl*), exponent data path (*fpu_mul_exp_dp*) and mantissa data path (*fpu_mul_frac_dp*). The control unit has the FSM for the multiplication stages and also controls inputs to selection muxes for single precision or double precision multiplication. Numerous special cases (not covered in FFU) are also covered in the control block.

Exponent datapath - The exponent multiplication datapath unit is basically an adder unit with rounding logic included in it. The changes to this unit assume true signals coming from the control unit irrespective of complement or true mode operation. This implies that the control unit always operates on true data. The exponent data path first involves adding 2 extra bits in double precision and 5 extra bits in single precision to the MSB side for resolution (internally) to detect overflows. The addition of these bits in complemented mode is done as 2'b11 instead of 2'b00 etc. In addition 1'b1 each is added to the two inputs to treat them as 2's complement of the original exponents. The two exponents are added and 2's complement of the bias, 13'h1c00 for double and 13'h1f80 for single precision in true mode, is added from the result to keep it correctly biased. In complement mode this constant is the true value of the bias ie. 13'h03fe and 13'h007e for single and double precision respectively.

Fractional Datapath – The fractional datapath involves instantiation of the integer multiplier in the third stage. The integer multiplier is a 3 stage multiplier followed by a normalization stage in the end. The integer multiplier was replaced by a '*' sign as the T1 integer multiplier could not be changed to incorporate complement mode execution.

5.7.2 Floating Point Addition

The floating-point adder (FPA) performs addition and subtraction on single and double precision floating-point numbers, conversions between floating point and integer formats, and floating-point compares. The unit is implemented as separate exponent, fractional and control units. The implementation strategy is based on normalizing the data in the control unit in complement mode. Although this is a pessimistic approaching it was adopted for two reasons. Firstly working on complemented data would have involved duplicating logic at almost each and every stage in the control unit. Secondly debugging the operation of the control unit in complement mode would have involved a lot of time which in the limited time span could not have been possible. In the fractional and exponent units it was assumed that the data is obtained in complement mode. FPA characteristics include:

- The FPA execution datapath is implemented in four pipeline stages (A1, A2, A3, and A4).
- Certain integer conversions to floating-point instructions require a second pass through the final stage.
- All FPA instructions are fixed latency, and independent of operand values.
- A post-normalization incrementer is used for rounding (*late round* organization).

- NaN source propagation is supported by steering the appropriate NaN source through the datapath to the result.
- The FPA unit also includes conversion from integer to single or double precision float, and single or double precision float to integer. The control signals (a1stg_f(dors)toi) are generated in the control unit.

Exponent Datapath – In T1 a single or a double precision floating point word is encoded in double precision format with unused bits left for double precision. The encoding has bit 63 as sign bit, bits 62-52 as biased exponent and remaining 51-0 bits as fractional part of the number. The two input exponents ([62:52] *inq_in1, inq_in2*) are flopped in the first stage and different parts of it are masked as per the control bits. Control bits identify double precision operation, single precision operation, signaling operation, special cases like all infinity or NAN etc. The mask for extracting the required bits remains unchanged. To the second exponent input (*a1stg_expadd2*) a bias of '1023' is subtracted so that the addition of the two exponents remains correctly biased. The exponent stage then has multiple stages of exponent alignment, increment or decrement adjustment, rounding and normalization followed by rounding.

Fractional Datapath – The fractional parts accepts the inputs ([51:0] *in1, in2*) and checks which of the two fractions is greater. Based on this comparison the fractional part is shifted to the left or right in stage 2. For conversion from or to integer and floating point numbers, control signal with suffix '_nx' are maintained. These indicate inexactness of the conversion. In stage 4 the addition of the two aligned fractional parts in performed. After this stage, separate modules identify whether the result is denormalized or not. The number of leading zeros is then found out and the count is used to shift the fractional part. Once the shift

is done the result is rounded. The area of the add unit with support for complement mode execution is 22.9% and worst case timing is 12% more than normal mode floating point addition.

6. Further Work

- Implement complement mode execution in a fully pipelined booth multiplier. The implementation in T1 is different from the algorithmic specifications. This it would take more probing to understand the encoding and confirm whether it is straight forward to implement Colt on it or not.
- Assess the changes to be performed in the non restoring DIV (divide) unit.
- The interaction of the special purpose unit (SPU) in T1 with the processor pipeline is not straightforward. The SPU supporting the asymmetric cryptography operations (public-key RSA) for up to a 2048-bit key size. The SPU initiates streaming load or streaming store operations to the level 2 cache (L2) and compute operations to the integer multiplier. The details of this operation were too complicated to be addressed in this work.
- The functionality to complement data in register files has not been implemented.
- It is not essential (and often not required) to implement the complementary Boolean logic. For example in most cases a Boolean expression involves masking a few bits of the data which can be done by keeping the mask in true form, data is in complement form and doing the same AND operation.
- It was found that some bits are very critical and hence may not sustain complement mode and incur timing delays. Thus a trade-off needs to be established such that only a part of the higher order bits are implemented in complement mode while the lower order bits are kept as such. This would involve a very critical analysis of timing and experiments and was beyond the scope of this work.

7. References

[1]. Erika Gunadi, Abhishek A. Sinkar, Nam Sung Kim, Mikko H. Lipasti. Combating Aging with the Colt Duty Cycle Equalizer. In Proceedings of MICRO'2010. pp.103~114.

[2]. Wallace, C. S.,"A Suggestion for a Fast Multiplier," *Electronic Computers, IEEE Transactions on* , vol.EC-13, no.1, pp.14-17, Feb. 1964.

[3]. http://www.eecs.tufts.edu/~ryun01/vlsi/verilog_simulation.htm

[4]. J. Fadavi-Ardekani, "M ´ N Booth Encoded MultiplierGenerator Using Optimized Wallace Trees," IEEE Trans.VLSI Systems, June 1993, pp. 120-125.

[5]. OpenSPARC T1 documentation suite. http://www.OpenSPARC.net/OpenSPARC-t1/download.html

[6]. P. *Kongetira,* K. Aingaran, K. Olukotun,. *Niagara: A 32-Way* Multithreaded SPARC Processor, IEEE. Micro, March-April 2005, pp. 21-29.

[7]. Weste, Neil H.E. and Eshraghian, Kamran, *Principles of CMOS VLSI Design : A systems perspective,* Addison-Wesley Publishing Company, 2nd ed., 1993, pp547-555.

[8]. Ki-seon Cho, Jong-on Park, Jin-seok Hong, Goang-seog Choi, "54x54-bit Radix-4 Multiplier based on Modified Booth Algorithm," ACM, Proceedings of the 13th ACM Great Lakes symposium on VLSI, pp. 233-236, April 2003.

[9]. Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems, Jean-Pierre Deschamps, Gery J.A. Bioul, Gustavo D. Sutter.

[10]. "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture". April 2008. pp. 118–125, 266–267, 334–335.