

Stony Brook University



OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

32-bit Superconductor Integer and Floating-Point Multipliers

A Dissertation Presented

by

Artur Krzysztof Kasperek

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Engineering

Stony Brook University

May 2012

Copyright
by
Artur K Kasperek
2012

Stony Brook University

The Graduate School

Artur Krzysztof Kasperek

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Mikhail Dorojevets – Dissertation Advisor
Associate Professor, Department of Electrical and Computer Engineering

Alex Doboli – Chairperson of Defense
Associate Professor, Department of Electrical and Computer Engineering

Sangjin Hong
Associate Professor, Department of Electrical and Computer Engineering

Jennifer L. Wong
Assistant Professor, Department of Computer Science
Stony Brook University

This dissertation is accepted by the Graduate School.

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

32-bit Superconductor Integer and Floating-Point Multipliers

by

Artur Krzysztof Kasperek

Doctor of Philosophy

in

Computer Engineering

Stony Brook University

2012

The objective of this dissertation is to design and evaluate ultra-fast energy-efficient 32-bit integer and single-precision floating-point multipliers implemented with Rapid Single Flux Quantum (RSFQ) superconductor technology. Our goals in both multiplier designs were to design a wide datapath multipliers operating in 10 GHz+ frequencies with lowest possible latency and complexity below 100k Josephson junctions when implemented with Hypres 1.5 μm 4.5 kA/cm² fabrication process. To achieve this goal, various design techniques such as synchronous pipelining, asynchronous co-flow, and wave-pipelining are analyzed and applied throughout the design process. First, we will have a brief look at CMOS computing with its power and clock frequency challenges. Then, superconductor technology will be introduced, followed by a description of RSFQ logic. Next, traditional design and sequencing techniques for multiplier will be discussed. After a brief review of existing superconductor multipliers, the cell-level design of our 32-bit integer and floating-point multipliers will be presented. The microarchitec-

tures and implementations of the 32-bit multipliers are discussed in detail along with the choice of sequencing techniques used. Our multipliers were designed and evaluated using a SBU VHDL RSFQ cell-library tuned to the Hypres 1.5 μm 4.5 kA/cm² fabrication process. The simulation results for the 32-bit integer and floating-point multipliers will be presented along with statistical data about each design. Finally, we will present the design and experimental test results of an 8-bit integer RSFQ multiplier implemented with the Japanese CONNECT cell library and fabricated with ISTECH 1.0 μm 10 kA/cm² technology.

To my wife Joanna and our children

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Need for More Energy Efficient Technology	1
1.2 Superconductor Technology as an Alternative to CMOS	2
1.3 Superconductor Technology Overview	2
1.3.1 Josephson Junction	3
1.3.2 Latching Logic	4
1.3.3 RSFQ Logic	5
1.3.4 ERSFQ Logic	6
1.4 Overview of Previous Work in Superconductor Field	7
1.4.1 Analog and Digital RSFQ Circuits	7
1.4.2 Microprocessor Prototypes: FLUX-1 and CORE1	7
1.4.3 20 GHz 8-bit RSFQ Processor Datapath	10
2 General Approach to Multiplier Design	11
2.1 Objectives	11
2.2 Binary Multiplication	12
2.3 Shift-Add Multiplier	14
2.4 Classical Array Multiplier	15
2.5 Booth Algorithm	15
2.5.1 Partial Product Encoding with Booth-2	16
2.5.2 Advanced Multiplier Architectures	17
2.6 Multiplier Topology	18
2.6.1 Array Topologies	21
2.6.2 Irregular and Regular Tree Topologies	22
2.7 Summation with a Carry Propagate Adder	25
2.8 Related RSFQ Work	25

3	Techniques and Tools for Superconductor Circuit Design	29
3.1	Sequencing Techniques	29
3.1.1	Synchronous Clocking for Pipelined Designs	30
3.1.2	Asynchronous Co-flow Synchronization	31
3.1.3	Wave-pipelining	32
3.1.4	Hybrid Wave-pipelining	33
3.1.5	Other Sequencing Techniques	34
3.2	RSFQ Cell-level Library and Design Tools	34
3.2.1	Stony Brook Tunable RSFQ VHDL Cell Library	34
3.2.2	CONNECT Cell Library	39
4	11.4 GHz 32-bit RSFQ Integer Multiplier Design and Evaluation	41
4.1	Goals and Challenges	41
4.2	RSFQ Hybrid Wave-pipelined Asynchronous Multiplier Miroarchi- tecture and Cell-Level Implementation	42
4.2.1	Partial Product Generation	42
4.2.2	Partial Product Compression	50
4.2.3	Final Summation	56
4.3	Integer Multiplier Design Summary	58
5	11.1 GHz 32-bit Single-precision Floating-point Multiplier De- sign and Evaluation	63
5.1	Goals and Challenges	64
5.2	Floating-point Multiplication Basics	64
5.3	32-bit Floating-point Multiplier Structure	65
5.4	Sign Bit Calculation	65
5.5	Exponent Calculation Unit	67
5.5.1	Zero Value Detection	67
5.5.2	Initial Exponent Calculation Unit	67
5.5.3	Exponent Data Buffer	69
5.5.4	Exponent Adjustment Unit	69
5.6	Mantissa Calculation Unit	70
5.6.1	Partial Product Generation	70
5.6.2	Compression	76
5.6.3	Final Summation Unit	78

5.6.4	Sticky Bit Calculation	78
5.6.5	Normalization and Rounding Units	78
5.7	Floating-point Multiplier Design Summary	79
6	Physical Chip Design and Demonstration of 20 GHz 8-bit Integer Multiplier	85
6.1	Microarchitecture	86
6.2	Complexity and Power	86
6.3	Performance	88
6.4	Logical and Physical Layout Design	89
6.5	Experimental Test Results	92
7	Conclusions	95
	Bibliography	96

List of Figures

1.1	Josephson Junction	4
1.2	DC electrical characteristics of voltage level stage latching junctions.	5
1.3	DC electrical characteristics of shunted junctions.	5
1.4	D Flip-Flop JJ-level implementation.	6
1.5	FLUX-1R block diagram.	8
1.6	FLUX-1R chip micro-photograph.	9
2.1	16×16-bit Multiplication with 32-bit product.	13
2.2	Numerical example of 16×16-bit multiplication.	13
2.3	Simplified add-shift multiplier block diagram for 16×16-bit multiplication.	14
2.4	Carry-save array multiplier.	15
2.5	First example of Booth encoding.	17
2.6	Second example of Booth encoding.	17
2.7	16×16-bit Booth-2 multiply.	18
2.8	Multiplication steps.	19
2.9	Multiplier topologies.	19
2.10	Examples of counters.	20
2.11	(3,2) counter.	21
2.12	[4:2] compressor built with (4,3) and (3,2) counters.	21
2.13	Double array for reduction of a single column of partial products.	22
2.14	8-bit Wallace tree.	23
2.15	8-bit binary tree.	24
2.16	16-bit binary tree.	24
2.17	24 GHz 4-bit RSFQ array multiplier.	27
2.18	Bit-serial RSFQ floating-point multiplier.	28
3.1	Synchronously pipelined circuit.	30
3.2	Fan-out of 8 implementation with a binary tree.	31
3.3	Asynchronous co-flow sequencing.	32
3.4	Wave-pipelining.	33
3.5	Propagation delay model for D flip-flop.	35
3.6	T1 cell symbol.	36

3.7	T1 cell state diagram.	36
3.8	T1 cell simulation.	37
3.10	T1 cell schematic.	37
3.11	Example schematic and physical layout views of a circuit designed with the CONNECT cell library.	40
4.1	32-bit multiplier structure.	43
4.2	32-bit multiplication.	44
4.3	Clock distribution for top four partial products.	46
4.4	Clock distribution and timing for the partial product generator.	47
4.5	Step-by-step timing during partial product generation.	48
4.6	Routing four partial product bits to single PTL.	49
4.7	Time synchronization of partial product generator output.	49
4.8	Wave-pipelining of the partial product generator.	50
4.9	[4:2] compressor.	51
4.10	N-bit wide [4:2] compressor.	52
4.11	[4:2] compression example.	52
4.13	Wave-pipelining with co-flow sequencing of a [4:2] compressor.	55
4.14	Partial product grouping for compression tree.	56
4.15	Compressor tree structure for the 32-bit RSFQ integer multiplier.	57
4.16	Compression tree for the 32-bit wide column of partial products.	58
4.17	JJ distribution per component for the 32-bit RSFQ integer multiplier.	59
4.18	Latency breakdown for the 32-bit RSFQ integer multiplier.	60
4.19	Complexity and current distribution per cell for the 32-bit RSFQ integer multiplier.	62
4.20	Complexity breakdown by component for the 32-bit RSFQ integer multiplier.	62
5.1	IEEE-745 single-precision floating-point number representation.	65
5.2	Floating-point multiplier structure.	66
5.3	Exponent calculation with bias correction.	68
5.4	4-bit RCA used for exponent calculation.	68
5.5	Partial product generator for a 24-bit multiplier.	70
5.6	Rearranging partial product generator for symmetric data flow.	71
5.7	Dataflow inside the partial product generator.	72

5.8	Clock distribution and timing for the partial product generator for the floating-point multiplier.	73
5.9	Clock distribution for top four partial products.	74
5.10	Partial product synchronization.	75
5.11	Wave-pipelining of the partial product generator for the floating-point multiplier.	75
5.12	Partial products grouping for compression tree.	76
5.13	Compressor tree for the 32-bit RSFQ floating-point multiplier. . .	77
5.14	11.1 GHz RSFQ 32-bit single-precision floating-point multiplier. .	80
5.15	Complexity breakdown per stage for the RSFQ floating-point multiplier.	82
5.16	JJ and current breakdown per cell for the RSFQ floating-point multiplier.	83
5.17	Cell breakdown per stage for the RSFQ floating-point multiplier. .	84
6.2	Compressor tree structure for the 8-bit RSFQ integer multiplier. .	88
6.3	Operating margins of the 8-bit RSFQ integer multiplier.	89
6.4	8-bit multiplier layout.	90
6.5	Micro-photograph of the 8-bit RSFQ multiplier chip.	91
6.6	Oscilloscope waveforms.	93
6.7	8-bit RSFQ integer multiplier chip bonded to a chip holder. . . .	93
6.8	Pre-cooling of the 8-bit RSFQ multiplier chip in liquid nitrogen. .	94
6.9	Testing probe inside a cryostat during testing.	94

List of Tables

1.1	Representative RSFQ circuits demonstrated in 2-3 μm Nb.	8
2.1	Booth-2 partial product selection.	16
2.2	11 GHz RSFQ 4-bit serial multiplier characteristics.	26
3.1	Key characteristics of the Hypres 1.5 μm 4.5 kA/cm ² process. . .	36
3.2	RSFQ cells widely used in our multipliers.	38
3.3	Key characteristics of the ISTECH 1.0 μm 10 kA/cm ² process. . . .	39
4.1	T1 cell used as (4,3) and/or (3,2) counter.	51
4.2	Number of [4:2] compressors needed for each group of four partial products.	56
4.3	32-bit RSFQ integer multiplier characteristics at T = 4.2 K. . . .	59
4.4	JJ complexity breakdown and bias current distribution by component in the 32-bit RSFQ integer multiplier.	59
4.5	JJ and bias current breakdown per logic and interconnect for the 32-bit RSFQ integer multiplier.	60
4.6	Latency breakdown for the 32-bit RSFQ integer multiplier.	60
4.7	Total cell breakdown for the 32-bit RSFQ integer multiplier. . . .	61
4.8	Cell bias current distribution for the 32-bit RSFQ integer multiplier.	61
5.1	Sign bit calculation.	67
5.2	Rules for implementing the IEEE rounding modes.	79
5.3	RSFQ floating-point multiplier characteristics at T = 4.2 K. . . .	81
5.4	Latency breakdown for the RSFQ floating-point multiplier.	81
5.5	JJ and bias current breakdown per logic and interconnect for the RSFQ floating-point multiplier.	81
5.6	JJ distribution per stage for the RSFQ floating-point multiplier. . .	82
5.7	Cell use breakdown for the RSFQ floating-point multiplier.	83
5.8	Bias current distribution for the RSFQ floating-point multiplier. . .	84
6.1	Complexity, bias current, and area distribution for 8-bit multiplier.	88
6.2	8-bit RSFQ integer multiplier chip summary.	88
6.3	Testing equipment for low frequency testing.	92

Acknowledgements

First of all, I would like to thank my advisor and friend Professor Mikhail Dorojets for providing guidance and keeping me on track, but also allowing me the freedom of working at unusual hours and days as I had to coordinate between my full-time job, school, and family. Mikhail was always there when I needed someone to talk to, or needed support or asked for guidance.

Thanks to all of the colleagues from Ultra-High Speed Computing Lab at Stony Brook, especially to Christopher Ayala for helping me during my research on multiple subjects and for his great friendship. To Sheryeas Rajagopal for being a great friend during my graduate study.

I would like to thank the people from Yoshi Lab at Yokohama National University, especially to Professor Nobuyuki Yoshikawa for allowing me to visit his laboratory and for a very warm reception. I am also grateful to Taichi Kato for helping me with the testing during my visit. Thanks to all the people from Yoshi Lab and to Yuki Yamanashi for being great company during my stay in Japan.

I was fortunate to have two excellent managers at my full time job, Jay Greenrose and Robert Pang who provided their great support when I had to take time to cover up my coursework and thesis related work. It would not be possible for me to finish this study without their help. I would also like to thank my running crew at Motorola for keeping me engaged at running, proving the best stress relief. To all my coworkers, who were a great companion and had to deal with my unusual working habits.

I would like to extend thanks to the members of my committee, Professors Alex Doboli, Sangjin Hong, and Jennifer Wong for their time and patience, and for providing helpful suggestions.

Most importantly, I wish to thank my parents. They bore me, raised me, supported me, taught me, and loved me. Special thanks go to my sister and brothers for their great support and for believing in me.

Finally, I would like to thank my wife Joanna for being a great companion through all this years when my days were hard work and sometimes frustrating. Thanks to all my kids for their love. To all of my family I dedicate this thesis.

Introduction

Contents

1.1	Need for More Energy Efficient Technology	1
1.2	Superconductor Technology as an Alternative to CMOS	2
1.3	Superconductor Technology Overview	2
1.3.1	Josephson Junction	3
1.3.2	Latching Logic	4
1.3.3	RSFQ Logic	5
1.3.4	ERSFQ Logic	6
1.4	Overview of Previous Work in Superconductor Field . .	7
1.4.1	Analog and Digital RSFQ Circuits	7
1.4.2	Microprocessor Prototypes: FLUX-1 and CORE1	7
1.4.3	20 GHz 8-bit RSFQ Processor Datapath	10

1.1 Need for More Energy Efficient Technology

Engineering always involves trade-offs between performance, power, and price. However, even as transistors become smaller, they also become faster, dissipate less power, and become even cheaper to manufacture [1]. Meanwhile, everything has its limits and CMOS technology is reaching the point where technology cannot provide sufficient computational needs for critical governmental projects.

In 2005, the experts from the National Security Agency (NSA) have concluded in Superconducting Technology Assessment (STA) that semiconductor technology will not deliver the performance levels necessary for future government's

applications [2]. CMOS integrated circuits have become less of a technology performance horse with vendors such as Intel reluctant to seek 10 GHz clock speeds.

Although power dissipation of small CMOS semiconductor systems is relatively low, large data processing centers still require significant amounts of power to operate. For example, the Japanese K computer, the most advanced supercomputer requires a 4.3 Mega Watts of power on average [3], which corresponds to \$3.7 million per year at \$0.10/kW.

Furthermore, even if energy per logic function decreases as transistors get smaller, total power consumption does not decrease as rapidly as the total number of devices on the chip increases.

Recently, leakage related power consumption has become more significant as threshold voltage has scaled down. Furthermore, for sub-100nm technologies, temperature has been shown to have significant impact on leakage power consumption as the leakage power consumption increases exponentially with feature scaling [4].

1.2 Superconductor Technology as an Alternative to CMOS

Among all CMOS alternative candidate technologies available in research laboratories, the superconductor one is the most advanced. It offers the potential to achieve circuit speeds well above 100 GHz with much lower power requirements than semiconductor circuits. This makes superconductor technology not only very high speed, but also a more energy-efficient solution[2]. In fact, some experts believe that the circuits built with this technology consume less than 1/10-th of the power consumed by CMOS circuits [5].

1.3 Superconductor Technology Overview

Kamerlingh Onnes first discovered superconductivity in 1911 after he cooled solid mercury (Hg) wire with liquid Helium (He) to 4.2 K. He noticed that the wire resistance suddenly vanished. Immediately realizing the importance of his discovery, he published an article naming his discovery *superconductivity* [6].

Almost half a century later, in 1950, the first magnetic flux quantum circuit

was predicted by Fritz London. In 1961 Bacon Deaver and William Fairbank were able to experimentally prove Fritz's theory which was followed by David Brian Josephson's discovery of the Josephson Junction (JJ) in 1962 [7]. Josephson junction is the basic building block of today's superconductor circuits. Eventually, Brian Josephson received the Nobel Prize in Physics in 1973.

After Josephson's discovery, large scale superconductor research work involving 150 researchers was run by IBM from 1969-1983. At the time, IBM used Josephson *latching logic* which faced serious speed-limiting problems that became clear during the project [2].

A revolutionary step forward in superconductor computing was marked by the discovery of *Single-Flux-Quantum* (SFQ) logic in Moscow State University during the 1980's which was further described by Likharev and Semenov as *Rapid Single-Flux-Quantum* (RSFQ) in 1991 [8]. After this discovery, a wide variety of DSP blocks, logical functions, and microprocessors such as FLUX-1 [9] and CORE [10] were produced.

The key characteristics of RSFQ technology are [2]:

- extremely fast switching (few picoseconds switching time)
- very low dynamic power consumption
- availability of logic gates
- ultra-high-speed and loss-less superconducting wires
- low static power consumption
- cryogenic operating temperatures

RSFQ logic became a standard in superconductor computing and is used widely today.

1.3.1 Josephson Junction

Josephson Junction is built using two superconducting metals (usually Niobium) separated by a very thin insulating layer. This insulating layer must be very thin (30 angstroms to several microns) depending on the dielectric constant of the material from which it is made. The separating material can either be an insulator or another non-superconductor metal.

Once metal such as Niobium (Nb) is cooled down below its critical temperature T_c , the electrons in the metal become paired and the overall interaction between two electrons becomes slightly attractive.

This very slight attraction allows electrons to drop into lower energy states, opening up the energy gap in superconducting metal. Because of the larger energy gap and the lower energy state, electrons can move (generate current) without being scattered. Furthermore, there is no electrical resistance in superconductors and therefore no energy loss. There is however a maximum super-current that can flow called the critical current I_c .

Until a critical current is reached, electron pairs called Cooper pairs can tunnel across the barrier generating electric current without any resistance.

The AC or dynamic Josephson effect takes place when the current flowing through the junction is higher than the critical current ($I > I_c$). In this case voltage V exists across the junction giving continuous oscillations. This in turn will cause a lowering of the junction's critical current causing even more normal current to flow and a larger AC voltage. This AC voltage is nearly 500 GHz per mV (millivolt) across the junction [11].

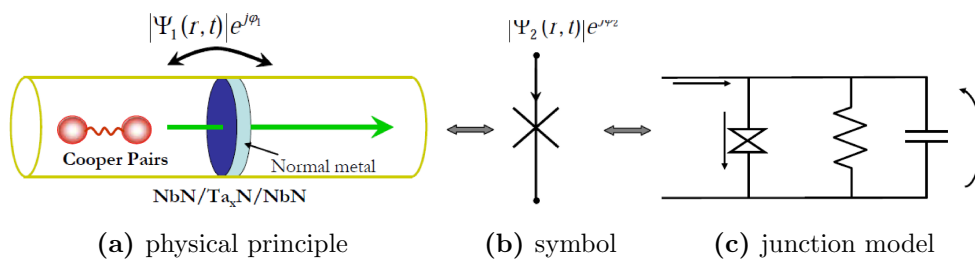


Figure 1.1: Josephson Junction [12].

1.3.2 Latching Logic

The first logical circuits used JJs in voltage-stage latching mode with I-V characteristics as shown in Figure 1.2. Here, the characteristics of the junction are multi-valued and hysteretic. The junction switches from $V = 0$ to V_g (equivalent to superconductor energy gap potential) at critical current I_c . Once the current is reduced to $I = 0$, the junction resets back to its original state providing two-state voltage levels similarly to CMOS logic.

The voltage-level logic was the basis of IBM and Japanese projects in the 1970's and 1980's which required an AC power system able to reset the junction in state $V = 0$. This technology was dropped as the operating speed was limited to 1 GHz [2].

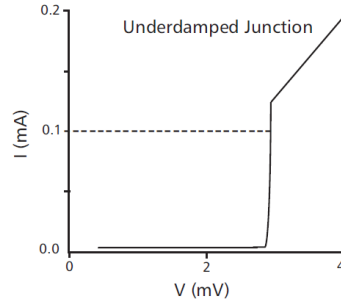


Figure 1.2: DC electrical characteristics of voltage level stage latching junctions [2].

1.3.3 RSFQ Logic

RSFQ circuits are not only faster than the circuits built with the latching logic but also dissipate less power. In the RSFQ logic, digital bits are coded in statics by the single quanta of magnetic flux, while in dynamics the data are transferred as picosecond SFQ pulses with quantized area [8]. Hence, the binary value of '1' is generally represented by a short pico-second wide RSFQ voltage pulse when the signal travels between RSFQ cells.

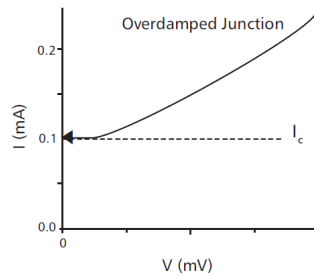


Figure 1.3: DC electrical characteristics of shunted junctions[2].

To implement this logic, a small shunt resistor is placed across the junction

and the I-V curves become non-hysteric, (see Figure 1.3). Since the damping is sufficiently high and the relaxation time is less than the period of plasma oscillations, the switching of the junction into the resistive state gives a periodic train of Single Flux Quantum voltage pulses with fixed area

$$\int V(t)dt \cong \Phi_0 = \frac{\pi\hbar}{e} = 2.07mV/ps \quad (1.1)$$

where Φ_0 is the *magnetic flux quantum*, \hbar is Planck constant and e is the charge of the electron [8].

In other words, if current through the junction is increased (presence of SFQ pulse), a short SFQ pulse is created on the output of Josephson junction.

The RSFQ gates are built by combining multiple Josephson junctions, inductors and bias resistors to form the circuit. A basic DFF SFQ circuit is shown in Figure 1.4. The DFF consists of: the input junction (J1), an inductor to store an SFQ pulse ('1'), and a comparator J2/J3 that determines whether or not an SFQ pulse will be transmitted to the output for each clock pulse. An SFQ pulse appearing at J1 will switch J1 and store one single flux quantum in the inductor between J1 and J2/J3. The stored flux quantum adds current Φ_0/L in J3. If there is a flux quantum in the latch when the clock pulse arrives, J3 switches, the SFQ pulse is transmitted, and the latch is reset to 0. If there is no flux quantum in the latch when the clock arrives, the current in J3 is insufficient to switch J3 and no SFQ pulse is transmitted [2].

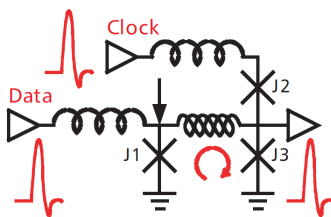


Figure 1.4: D Flip-Flop JJ-level implementation [2].

1.3.4 ERSFQ Logic

Energy-efficient RSFQ (ERSFQ) invented in 2010 by Hypres researchers [13, 5] is a novel energy-efficient zero-static-power SFQ technology which retains all

advantages of the current RSFQ logic family. Here, bias resistors are replaced with junction-based current distribution providing (excluding cryocooling factor) ERSFQ advantage of 10^4 in power consumption over standard CMOS circuits [13, 5] making the superconducting circuits suitable for IC manufacturing more than ever before.

1.4 Overview of Previous Work in Superconductor Field

Superconductor circuits have been studied in research laboratories over the past decade and a wide variety of RSFQ circuits were demonstrated as up to date. These include the fastest analog to digital converters (ADC), multipliers and microprocessors.

1.4.1 Analog and Digital RSFQ Circuits

Extremely fast RSFQ ADC converters were produced commercially by Hypres [14] and Northrop Grumman [15]. The RSFQ ADC converters are considered state of the art in ADC converter technology. Superconductor based 4-bit converters operate at 100 Giga-samples/second while the best operating speed achieved in semiconductors at this resolution is only 1 Giga-sample/second.

Other analog and arithmetic blocks including multiplexers, DAC, adders and multipliers were demonstrated with 2-3 μm line width niobium (Nb) and are listed in Table 1.1.

1.4.2 Microprocessor Prototypes: FLUX-1 and CORE1

Among all RSFQ circuits, microprocessors are the most complex requiring a great deal of knowledge as well as efficient CAD tools. Two RSFQ microprocessors have been produced to date. The FLUX-1 in the USA [9] and the CORE1 in Japan [10].

1.4.2.1 FLUX-1

FLUX-1 was the first RSFQ microprocessor to address the architectural and design challenges of 20 GHz processors. The design was a collaboration between

Table 1.1: Representative RSFQ circuits demonstrated in 2-3 μm Nb technologies [14].

Circuit Type	Circuit Metric(s)	Circuit Type	Circuit Metric(s)
Toggle flip-flop	144 GHz	2-bit counter	120 GHz
4-bit Shift register	66 GHz	1-kbit shift register	19 GHz
6-bit flash ADC	20 GHz	19-bit ADC	functional
14-bit 2MHz ADC	-100 dBc	1:2 demultiplexor	95 Gb/s
1:8 Demultiplexor	20 Gb/s	2-bit Full-adder	13 GHz
1-bit Half-adder	23 GHz	14-bit comb filter	20 GHz

Stony Brook University and Northrop Grumman Space Technology [9].

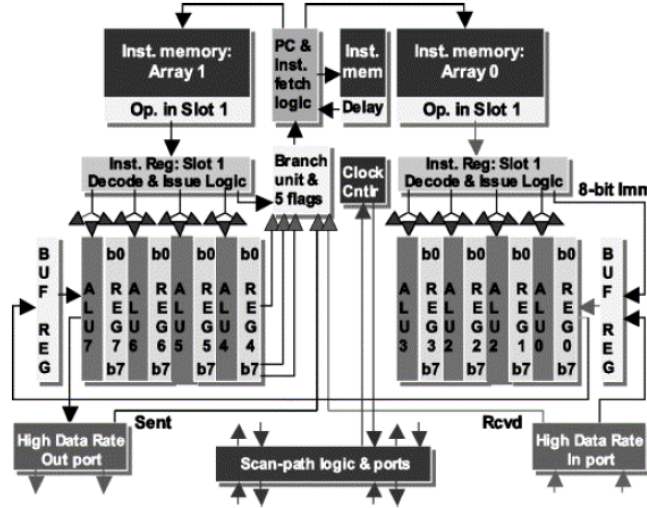


Figure 1.5: FLUX-1R block diagram [9].

New architecture was developed for FLUX-1 which included the following features [9]:

- ultra-pipelining to achieve 20 GHz clock rate
- two operations per cycle
- short-distance interaction and reduced connectivity between ALUs and registers
- bit-streaming (bit-chaining)

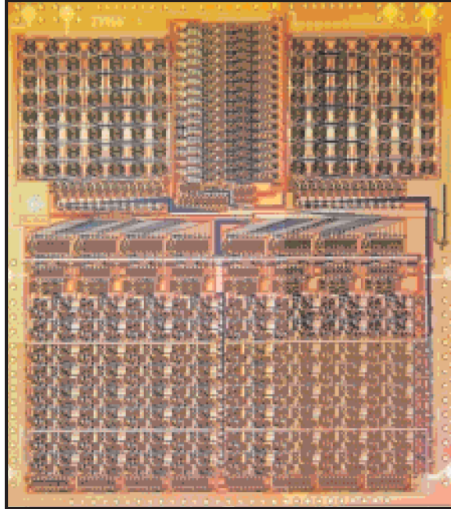


Figure 1.6: FLUX-1R chip micro-photograph. There are 63,107 Josephson junctions on $10.35 \times 10.65 \text{ mm}^2$ die [2]. Photo courtesy of Lynn Abelson and George Kerber, TRW, Inc.

- wave pipelining in the instruction memory
- modular design
- 25 control, logical and integer operations

The final FLUX-1 was called FLUX-1R and was fabricated in 2002. It consisted of 63,107 Josephson junctions on a $10.35 \times 10.65 \text{ mm}^2$ die (see Figure 1.6) with a total power consumption of 95 mW at 4.2 K [16].

At present, the largest circuit block successfully tested in the FLUX-1R chip was the ALU-register block (the most complex FLUX-1R component). No operational FLUX-1R chips were demonstrated by the end of the project in 2002 [17].

1.4.2.2 CORE1

The CORE1 α , a bit-serial microprocessor prototype (2002-2005) was designed, fabricated, and successfully tested at high speeds in the Japanese Superconductor Network Device project [10].

This microprocessor has relatively simple architecture consisting of two 8-bit wide registers and bit-serial ALU with a small shift register used as memory for 8-bit instructions.

The CORE1 processors used a relatively slow (1 GHz) system clock and fast (21 GHz) local clocks where bit-serial operations were carried out. The CORE1 α chip had 9,498 JJs with an area of 3.4 x 3.2 mm², and a total power consumption of 3.0 mW.

1.4.3 20 GHz 8-bit RSFQ Processor Datapath

The work on an 8-bit RSFQ datapath was carried out through a collaboration between Stony Brook University (SBU) and Hypres, Inc. in 2009-2012. This datapath implements an 8-bit version of the 32-bit Frontier data-flow architecture for RSFQ processors developed at Stony Brook [18].

The SBU team has done the complete cell-level design and verification of the 8-bit processor datapath using Hypres 1.5 μm 4.5 kA/cm² technology, while a team at Hypres did physical layout design, fabrication, and testing of several chips. Two chips with the 8-bit datapath components such as Arithmetic-Logic Unit [19] and Register file [20] were fabricated and demonstrated their operation at the target 20 GHz frequency.

General Approach to Multiplier Design

Contents

2.1	Objectives	11
2.2	Binary Multiplication	12
2.3	Shift-Add Multiplier	14
2.4	Classical Array Multiplier	15
2.5	Booth Algorithm	15
2.5.1	Partial Product Encoding with Booth-2	16
2.5.2	Advanced Multiplier Architectures	17
2.6	Multiplier Topology	18
2.6.1	Array Topologies	21
2.6.2	Irregular and Regular Tree Topologies	22
2.7	Summation with a Carry Propagate Adder	25
2.8	Related RSFQ Work	25

2.1 Objectives

Multiplication is a heavily used arithmetic operation that figures prominently in signal processing, imaging, science, and network security where the main criteria of interest are higher speed, low cost, and power consumption.

Since high efficiency is required, multiplier often represents one of the major bottlenecks as its hardware complexity grows with the input data size. It also

requires a vast amount of wires and logic gates, has a very long latency operation, and often very irregular layout structure.

Being one of the basic arithmetic operations, accounting for 8.72% of all instructions [21] in typical scientific program it is worth while to choose the best suited multiplier design for a particular application.

The major goal in multiplication is to speed up the multi-operand addition of multiple operands called *partial products* (shifted multiplicands or zeros), as shown in Figure 2.1.

Variety of multiplication algorithms and hardware designs starting with simple add-shift multiplier, Booth multiplier, array multipliers and tree-style compression schemes will be presented here.

2.2 Binary Multiplication

In binary multiplication, partial products are shifted versions of multiplicand or binary zeros depending on the value of the corresponding multiplier bit. If the multiplier bit is zero, then the partial product for this row has value zero, otherwise a shifted version of multiplicand is used which is often achieved using AND gates.

Once the partial products are generated, the main and most complex task in binary multiplication is to add these to make the final product. To achieve this goal, all bits in single column of multiple partial products have to be added together with any resulting carries propagating towards the most significant bit column.

The partial product generation for *simple multiplication* is illustrated in Figure 2.1 where each dot represents a single bit that can be either zero or one. The partial products are represented by horizontal dots, and the multiplier is represented by the column of vertical dots on the right. In Figure 2.1, the arrows to the left of multiplier are used for partial product value selection. The selection choice is shown in the selection table in the upper left corner, and the final product is represented by the double width row on the bottom.

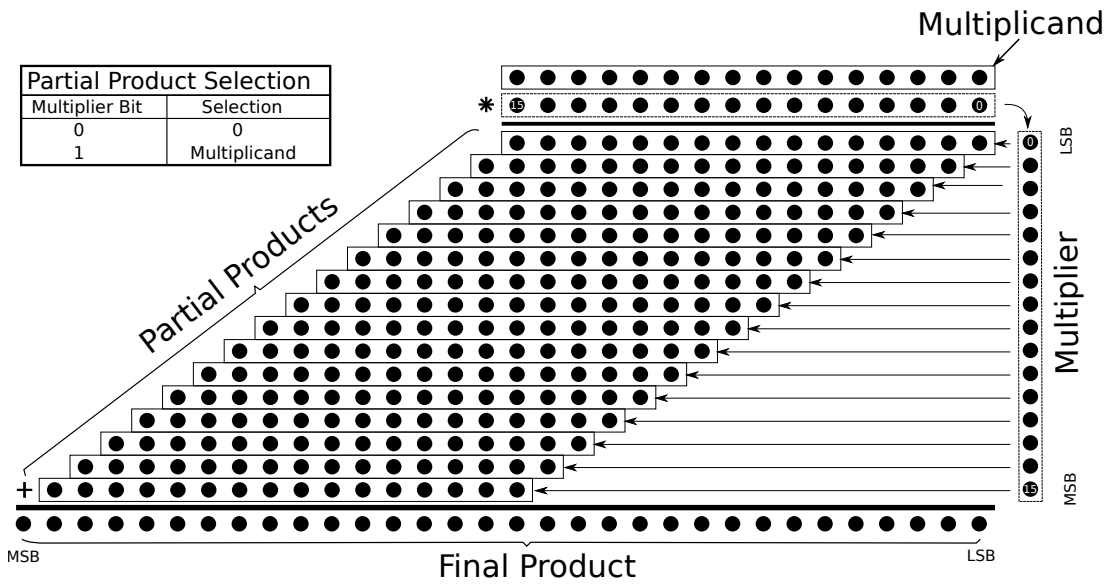


Figure 2.1: 16×16-bit Multiplication with 32-bit product [22].

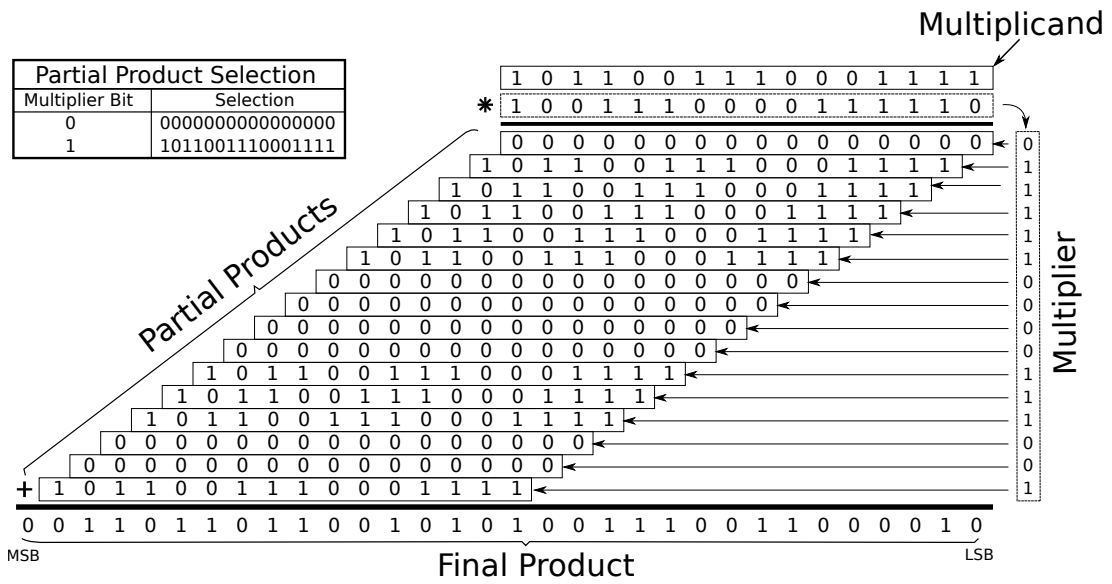


Figure 2.2: Numerical example of 16×16-bit multiplication.

2.3 Shift-Add Multiplier

The simplest multiplier design consists of two shift registers, an adder and a product register as shown in Figure 2.3. It is a sequential shift-add multiplier driven by control (write and shift) signals updated every step for a total of N stages, where N is the multiplier width in bits. Here, the partial products are generated using a shift register and are added together one-by-one, with intermediate results being saved in the product register.

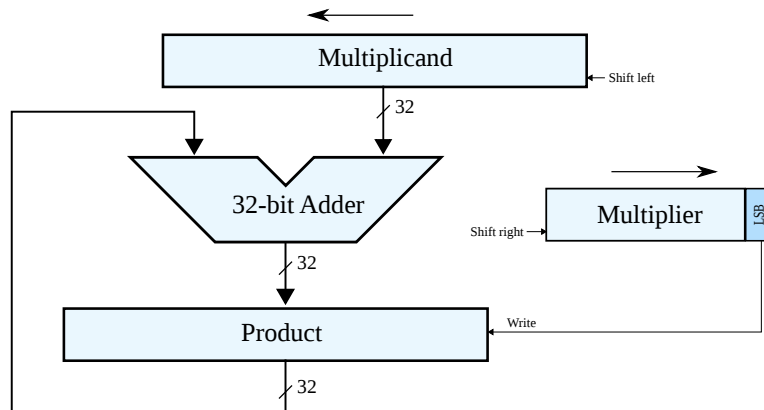


Figure 2.3: Simplified add-shift multiplier block diagram for 16×16 -bit multiplication.

Since the partial product value is just a shifted version of multiplicand or 0, the shifted multiplicand is added and written back to the product register only when the least significant bit (LSB) of currently shifted version of multiplier is 1.

Although the sequential add-shift multipliers can be built with very simple hardware taking a very small silicon area, if each step takes a clock cycle, then a 32-bit multiplier will require 32 clock cycles which is usually unacceptable for high-end multiplier designs.

The simplicity of this approach can provide a very low cost, and power efficient design, and was used in the past in first RISC type MIPS processors. The drawback of this design is that it can not be pipelined and is generally avoided in high speed processors.

2.4 Classical Array Multiplier

A classical N-bit array multiplier can be built with approximately N^2 adders connected using carry-save path between each row of adders as shown in Figure 2.4. The carry-save adder (CSA) is used very often in multiplier design. It is a fast adder that does not propagate carry from the least to most significant bit avoiding ripple-carry effect [23]. Carry-save adders will be discussed in greater detail in Section 2.6.

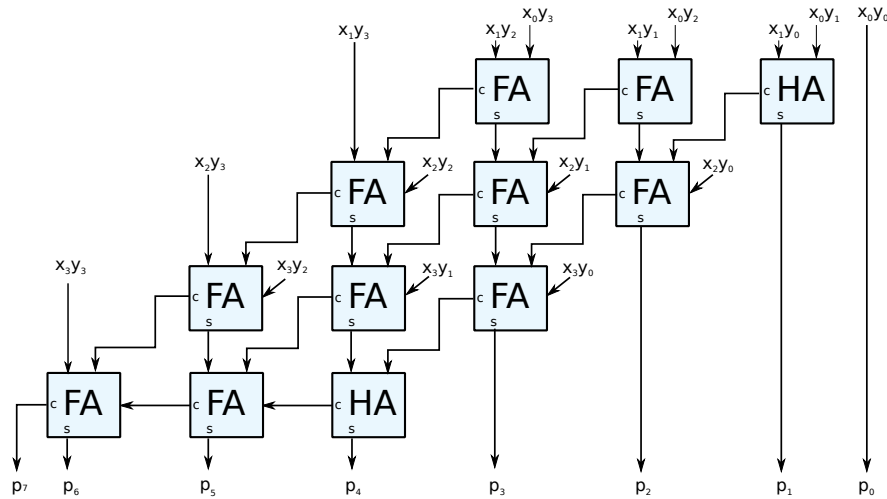


Figure 2.4: Carry-save array multiplier.

2.5 Booth Algorithm

An elegant approach to multiplying two numbers is called *Booth's algorithm*. Booth invented this approach in the quest for speed as the machines of his era were fast at shifting and slow on addition and subtraction, and for some patterns extra time was gained in computation when this algorithm was employed. The main idea comes from the fact that a sequence of ones in any binary number can be represented by a single subtraction of -1 in low bit position, and an addition of 1 in the next to highest bit position of the sequence. That is

$$2^j + 2^{j-1} + \dots + 2^i = 2^{j+1} - 2^i. \quad (2.1)$$

For example:

$$00111100_2 = 2^5 + 2^4 + 2^3 + 2^2 \equiv 2^6 - 2^2 = 64 - 4 = 60. \quad (2.2)$$

Now, more advanced versions of Booth's algorithm are used to reduce the number of partial product rows.

2.5.1 Partial Product Encoding with Booth-2

Modified Booth-2 algorithm which introduces parallel encoding of partial products was introduced in [24] and can be used to reduce the number of partial products by almost half by recoding the multiplicand. This reduction is done by making use of the run of 1s and 0s property which states that less partial products need to be produced for every run in multiplicand.

In Booth-2 encoding, the two consecutive rows of partial products are combined together into a single partial product row. where the multiplies of multiplicand are selected from ± 0 , $\pm \text{Multiplicand}$ and $\pm 2\text{Multiplicand}$ according to Table 2.1.

Table 2.1: Booth-2 partial product selection.

Multiplier bits	Partial product value	Sign value (S)
000	+0	0
001	+Multiplicand	
010		
011	+2Multiplicand	1
100	-2Multiplicand	
101	-Multiplicand	
110		
111	-0	

The partial products are shifted by two between each overlapping group and the different weights for partial products correctly produce the right result. Besides the two bits of multiplier being examined for each product, a third bit from the higher order bit is used to ensure that the end or middle of run of ones is detected correctly as shown in Figure 2.5.

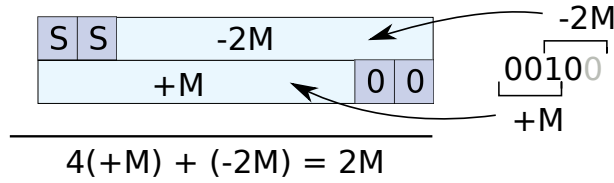


Figure 2.5: Booth encoding example when multiplier = $0010_2 = 2$.¹

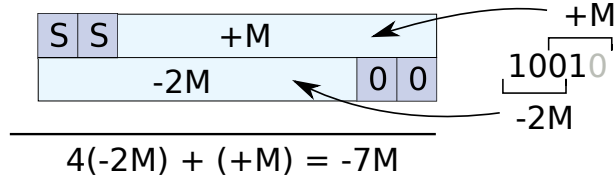


Figure 2.6: Booth encoding example when multiplier = $1001_2 = -7$.¹

The negative multiple of multiplicand given in the Table 2.1 are represented in 2's complement notation. The conversion of a number to 2's component can be performed by inverting the multiplicand and adding 1 to the least significant bit which can be inserted into the partial product below the current one without affecting the partial product generation speed. The sign extension and 2's complement can be replaced by what is shown in Figure 2.7 and a sufficient proof for the replacement is given in [25].

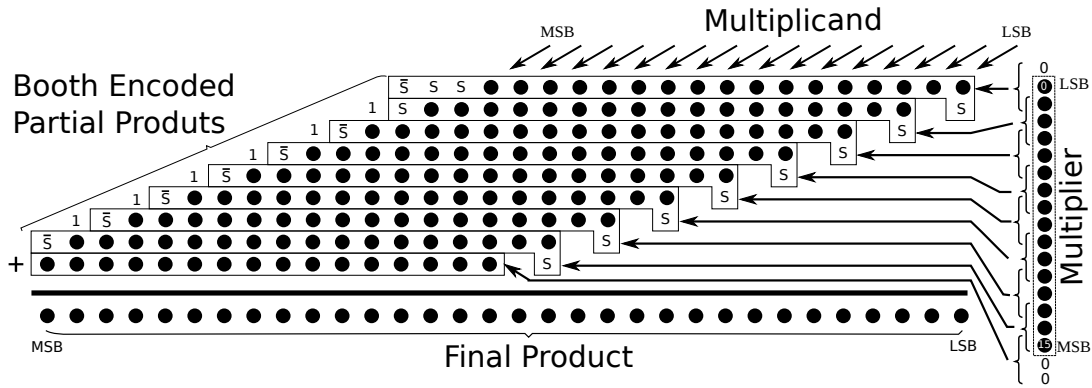
The amount of partial products generated using Booth-2 reduces the total number of N partial product rows into $\lceil \frac{N+1}{2} \rceil$. The extra 1 in the expression comes from the need to ensure that a last partial product is a positive multiple of the multiplicand [26].

Booth-2 and higher order booth algorithms like Booth-3 and Booth-4 are used very frequently in CMOS design, but are not used in this study due to the limitations described in Section 4.2.1.1.

2.5.2 Advanced Multiplier Architectures

Another approach to multiply two numbers is to partition multiplication into three operations which are:

¹Use Table 2.1 for partial product encoding selection.



Note:
Use Booth-2 Partial Product Selection Table to encode the partial products and to determine the value of S.

Figure 2.7: 16×16-bit Booth-2 multiply [22].

- 1) *Partial Product Generation* where partial products are generated in parallel using AND gates.
- 2) *Partial Product Compression* in which partial products are added together using carry-save adder to produce carry-sum operand pairs (one pair per column).
- 3) *Final Summation* is the last step, in which the carry and sum operands are added together using a standard arithmetic adder (usually CPA) to produce the final result.

2.6 Multiplier Topology

Most of the multiplier hardware is used for the summation of multiple rows of partial products called compression. It is a process in which the partial product bits with the same arithmetic weight are added together with intermediate carries propagating to higher order columns. This addition is carried out using either an array or compression tree which reduces each column to carry-sum operands. Tree topologies can be very fast and are usually preferred over array ones. Specific way in which these topologies process partial products are shown in Figure 2.9.

After compression is finished, a carry-propagate adder is used to add final carry-sum operands to produce a final product as shown in Figure 2.8.

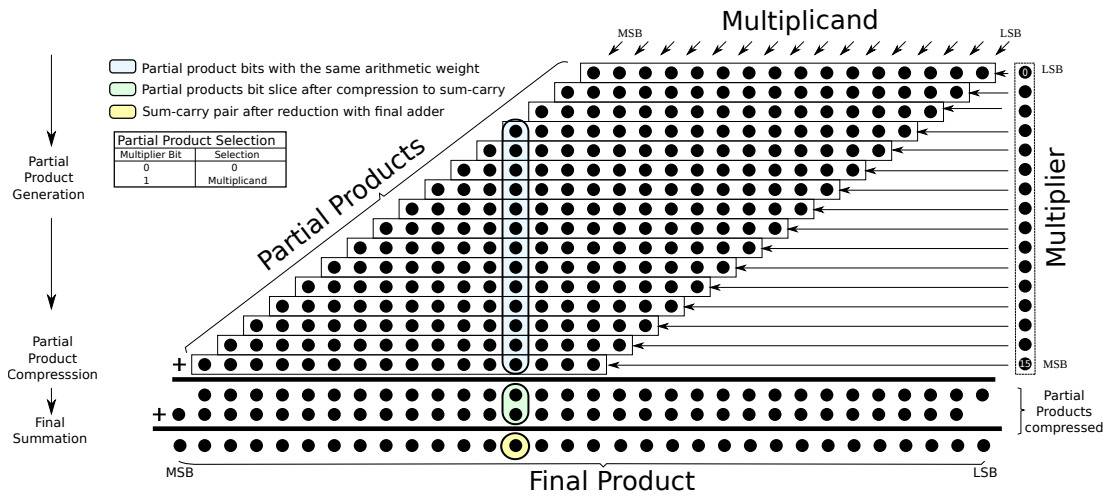


Figure 2.8: Main three steps of multiplication: partial product generation, compression and final summation.

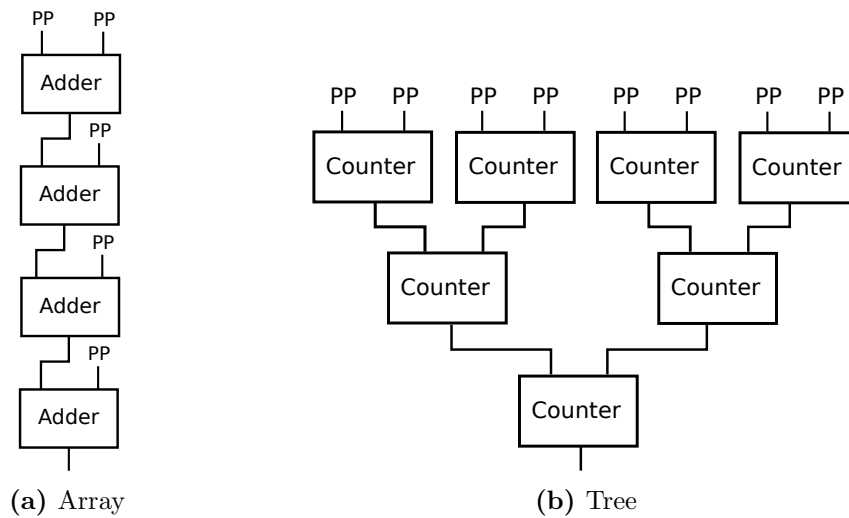


Figure 2.9: Multiplier topologies.

A basic building block used in reducing partial products is a carry-save adder (CSA) which is nothing more than an N-bit wide array of full adders connected in a way in which the carry does not propagate from the least to most significant bit avoiding ripple-carry effect. It also differs from other digital adders in that it has three or two input, and two output operands. One output operand is a sequence of partial sum bits and another a sequence of carry bits (i.e. carry-sum output operands) [23].

A full adder which is a building block of compressors discussed in this section is also referred to as a (3,2) counter (three-to-two counter) using Dadda's [27] and Stenzel's [28] terminology.

In general, any (M,N) counter is an adder which has M inputs and N outputs. An (M,N) counter counts M inputs from a single partial product column and produces an N-bit wide result where $\log_2(M) \leq N$. The output result is a binary representation of the number of inputs. For example, a (3,2) counter can reduce three inputs with the same arithmetic weight into a 2-bit wide result. Higher level counters such as (7,3) can also be designed. They are usually implemented using multiple (3,2) counters.

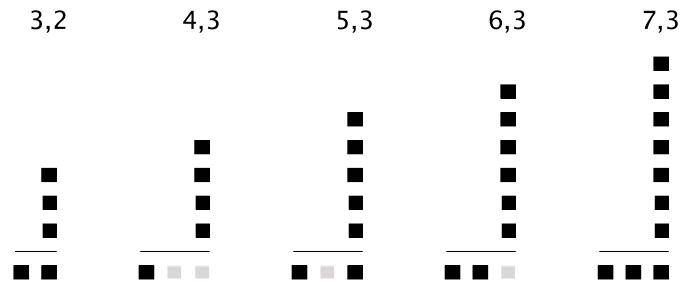


Figure 2.10: Examples of counters.

Compressors are a special form of counters which have multiple inputs, two outputs and any number of intermediate carries. The most common compressor is a [4:2] compressor shown in Figure 2.12. From now on, we will use [x:y] to indicate compressors and (x,y) to indicate counters.

The real distinction between counters and compressors is in the way these are used in compression trees and arrays. While counters always propagate the output down the tree, compressors have additional intermediate input/output carries which are propagated at the same level from the least significant bit col-

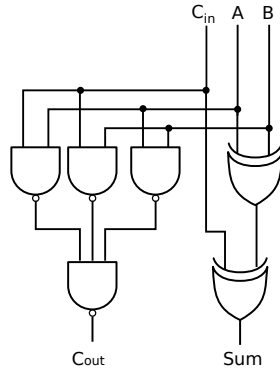


Figure 2.11: (3,2) counter, also known as full adder.

umn towards the most significant bit column.

The potential advantage of compressors is in their regularity and wirability which is described in Sections 2.6.1 and 2.6.2.

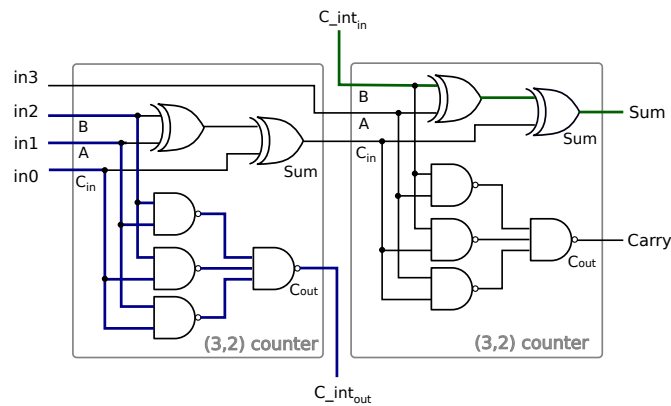


Figure 2.12: [4:2] compressor built with (4,3) and (3,2) counters connected using carry-save path.

2.6.1 Array Topologies

In a simple array, different partial products are processed at different stages (array rows) as was shown in Figure 2.9(a). Therefore, the total compression time delay is proportional to number of counters in each column, which is in order of $O(N)$, where N is the number of partial products. This delay can be halved using a double array shown in Figure 2.13, and higher order arrays are

also possible. The delay required to reduce partial products using higher order arrays is proportional to $2\sqrt{N}$, and is actually equal to $\lfloor \sqrt{N-9} \rfloor$, where N is the number of inputs [26].

Although, array multipliers are much faster than ripple-carry designs, the compression tree multipliers introduced in the next section are faster than array ones.

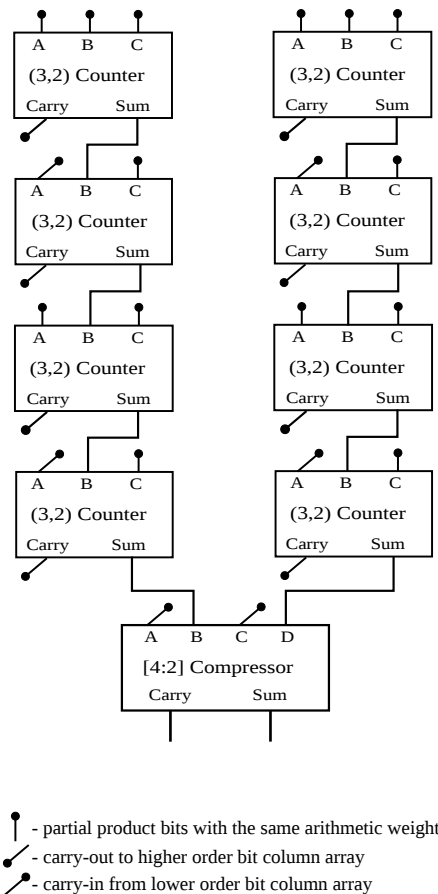


Figure 2.13: Double array for reduction of a single column of partial products.

2.6.2 Irregular and Regular Tree Topologies

Trees are extremely fast structures used for compression of partial products in which counters and compressors are used. Separate trees are built in parallel

for each column of partial products, and then are connected with intermediate carries propagating from one bit tree to the other with carry-save structure.

Irregular tree topologies are built with counters and compressors connected in a way to minimize the total delay without having any regular pattern.

The first trees were irregular Wallace trees [29] in which Wallace introduced the concept of adding partial products using (3,2) counters by connecting multiple (3,2) counters in parallel. An example of an 8-bit Wallace tree is shown in Figure 2.14.

The total number of counter stages in a Wallace tree is $\log_{3/2}(N)$, and the latency of the corresponding design is $O(\log_{3/2}(N))$, an impressive speed for multipliers. One of the drawbacks to using irregular trees is a very irregular layout in which the number of tracks per bit slice is greater than $\log(N)$ [26].

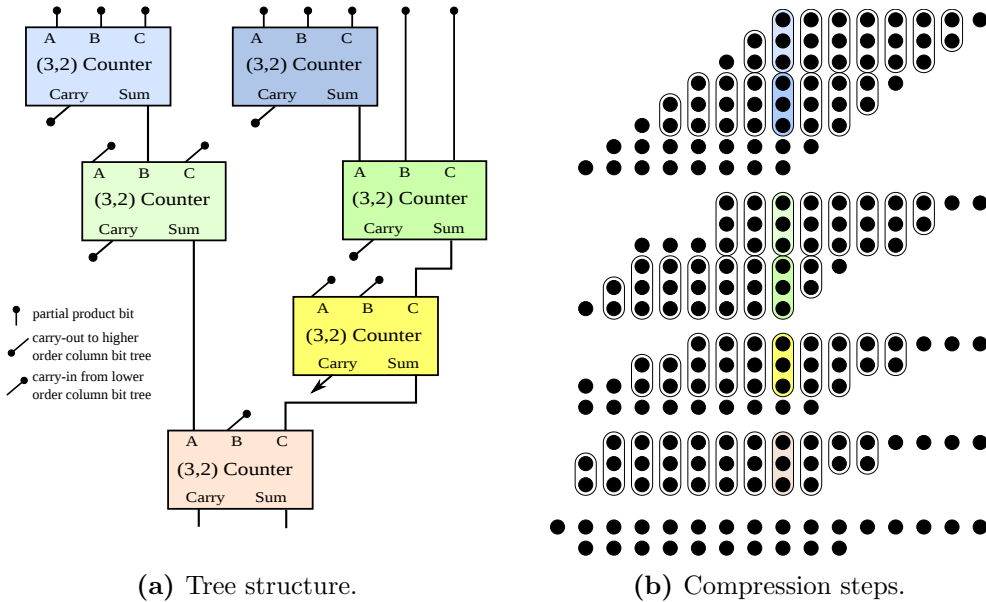


Figure 2.14: 8-bit Wallace tree which can be used to reduce a single column of partial products.

Regular tree structures have a 2:1 reduction ratio between each step providing for very regular and symmetric design structure. [4:2] compressors are very often the basic building blocks of regular trees. A binary tree reduces the partial products using $\lceil \log_2(\frac{N}{2}) \rceil$ [4:2] compression stages and the latency of the corresponding design is in order of $O(\log_2(N))$ with $2 \times (\lceil \log_2(N) \rceil - 2) + 4$ tracks per slice [26].

These type of trees suit RSFQ architecture much better than the irregular tree structures as all data are processed at the same time without any additional delay being added between each bit slice. Furthermore, regular trees are not only easy to be build, providing a very symmetric and regular structure, but also well suited for pipelining. Two examples of regular binary trees are shown in Figure 2.15 and Figure 2.16.

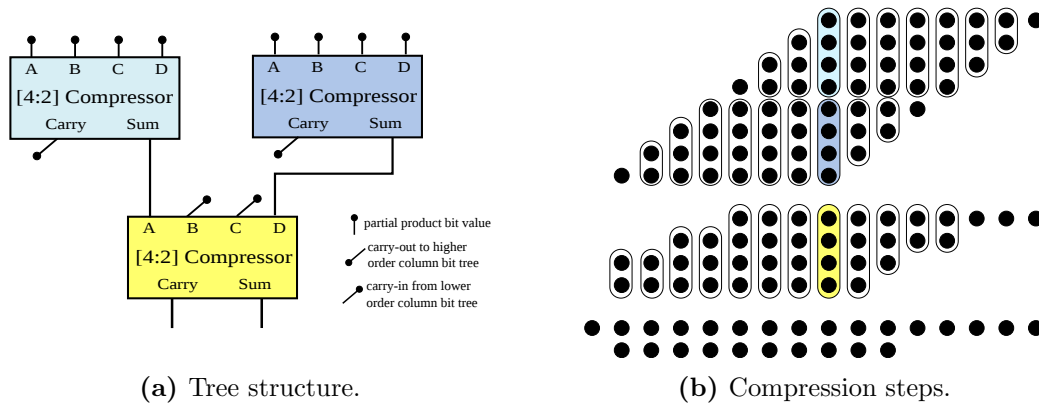


Figure 2.15: 8-bit binary tree which can be used to reduce a single column of partial products.

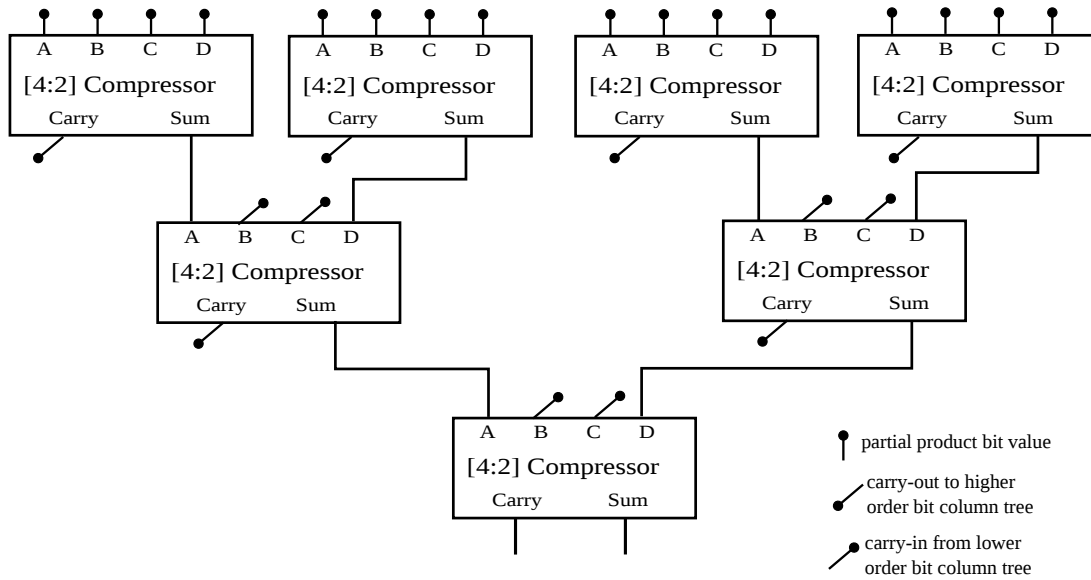


Figure 2.16: 16-bit binary tree.

2.7 Summation with a Carry Propagate Adder

Once all the partial products are reduced to two operands called sum and carry, these are then added together to produce a final product using a carry-propagate adder (CPA).

The CPA latency can be anywhere between $O(N)$ and $O(\log_2(N))$ depending on the adders selection with a ripple-carry adder being the slowest, and a parallel prefix tree adder the fastest.

Because of their lower latency, parallel-prefix adders are usually the best choice for the final adder in large multipliers. The CPA design is beyond the scope of this dissertation. A parallel-prefix tree adders were designed with RSFQ technology in SBU UHSC Lab [30]. These are the fastest types of parallel prefix tree adders and are used in both multipliers to reduce the final two rows into a single final product.

2.8 Related RSFQ Work

Since the introduction of superconductor RSFQ logic there have been several attempts to design and build different kinds of RSFQ multipliers. The immaturity of superconductor technology and design tools has limited the functional complexity, and width of the fabricated multiplier designs to 2-4 bits. Most of the proposed RSFQ multipliers were in fact design studies done without any actual fabrication [31, 32, 33].

The first conceptual RSFQ multiplier design proposed in 1989 was a bit-serial multiplier with a very basic carry-save path, and variable word length with possible expansion into parallel design [31]. An 11 GHz 4-bit RSFQ multiplier-accumulator was designed, fabricated and tested successfully in 1997 [34]. This design was built using a total of 1097 Josephson junctions (JJs). The multiplier characteristics are shown in Table 2.2.

A 10 GHz 32-bit parallel multiplier with Wallace-tree used for compression and carry look-ahead adder for final summation was proposed in 2001 [35]. Shortly after, an 8-bit enhanced version of this multiplier was proposed as well. The addition was that the Booth encoder was used to speed up partial product generation. Design simulations showed that the maximum rate of such a multiplier was limited to 10 GHz by its slowest final summation block, although

Table 2.2: 11 GHz RSFQ 4-bit serial multiplier characteristics.

Parameter	Value
Number of JJs	1,097
Power required	181 μ W
Area	2.6 x 0.8 mm ²
Maximum simulated clock frequency	11 GHz
Maximum simulated output rate	172 MHz
Measured global bias margins	\pm 5%

other components were designed to run at a faster rate. A 2-bit version of this multiplier was fabricated and tested successfully [36].

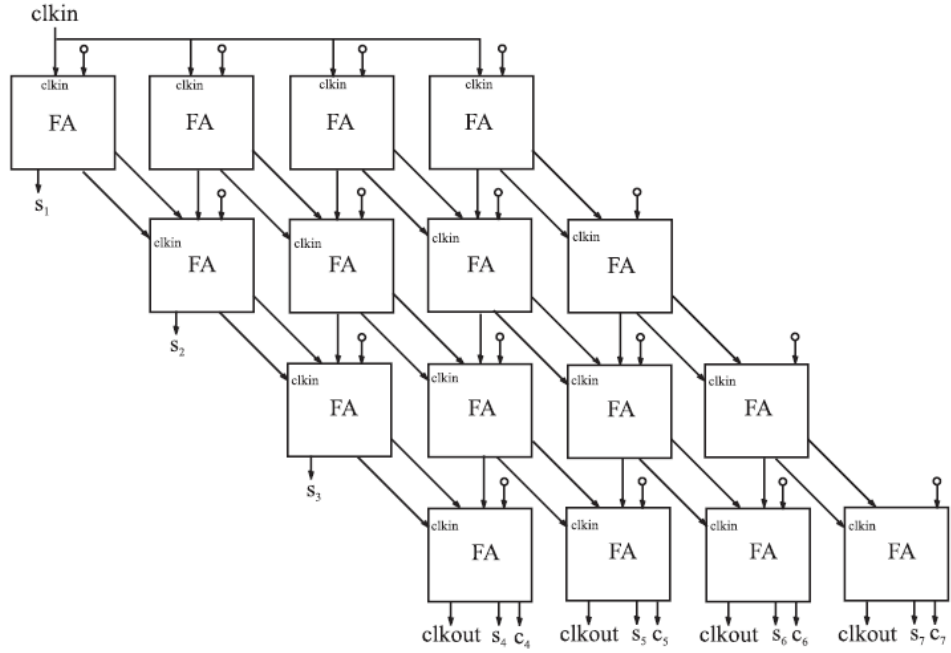
A 3-bit serial-parallel multiplier design with an AND-gate array used for product generation and sequential carry-save compression was designed (but not fabricated) by Akahori et al. in 2003 [32]. In the simulation, a 14 GHz processing rate was achieved and some suggestions were given on how this rate could be increased to 20 GHz.

In other design studies, a 24 GHz 4-bit RSFQ parallel multiplier-accumulator with a carry-save array type compressor [37] and a similar systolic array multiplier [33] were proposed. Only the first one was fabricated and demonstrated and is shown in Figure 2.17.

A 20 GHz 4-bit multiplier with the Booth-2 encoder was designed and fabricated using the ISTEK 10 kA/cm² process. The simulated frequency of this multiplier was 45 GHz [38].

Recently, using the same ISTEK process, a 25 GHz 16-bit bit-serial floating-point (FP) multiplier was designed and implemented using the CONNECT RSFQ cell library [39]. By re-using the same hardware during bit-serial computation, this FP multiplier needs 23 clock cycles (920 ps) to calculate a 16-bit FP result, thus processing 16-bit FP operands at 1 GHz rate [40]. The block diagram of this multiplier is shown in Figure 2.18.

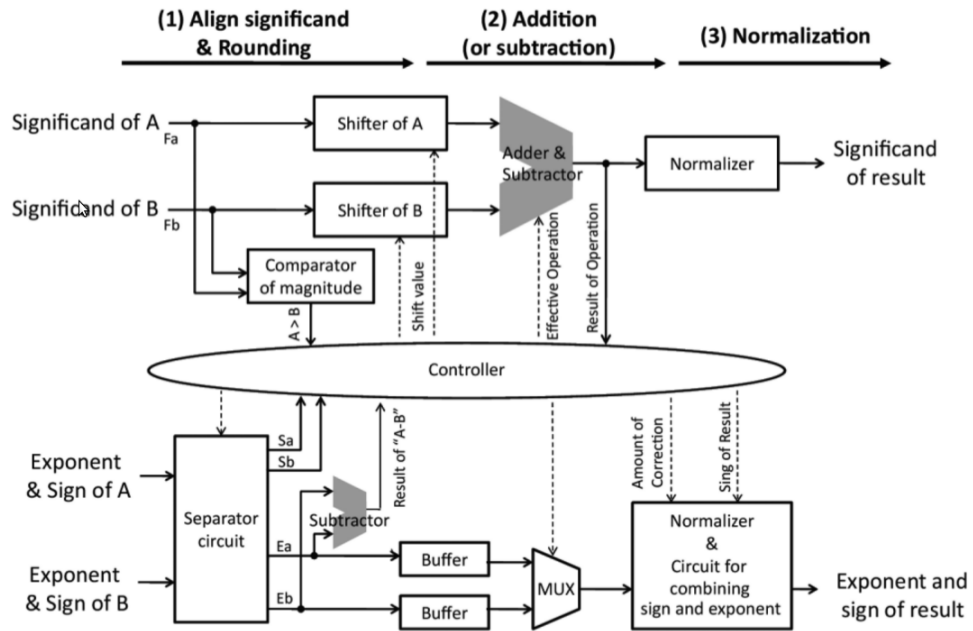
A traditional array-based compression which is used in most RSFQ multipliers proposed or demonstrated to date is suitable for multiplication of small numbers, but once the multiplier operand size is increased, the latency grows accordingly. Another drawback of the array multipliers is the relatively large size of their partial product compression logic where complexity grows quadratically



Parameter	Value
Number of JJs	704
Area	1.40 x 1.97 mm ²
Maximum simulated output rate	24 MHz
Measured global bias margins	±2%

Figure 2.17: 24 GHz 4-bit RSFQ array multiplier [37].

with respect to operand width. Finally, the delay fluctuations in the chains of adders used for compression could become problematic for large-size arrays built this way.



Parameter	Value
Number of JJs	11,044
Area	6.22 x 3.78 mm ²
Maximum simulated clock frequency	25 GHz
Measured global bias margins	± 3%

Figure 2.18: Bit-serial RSFQ floating-point multiplier [40].

Techniques and Tools for Superconductor Circuit Design

Contents

3.1 Sequencing Techniques	29
3.1.1 Synchronous Clocking for Pipelined Designs	30
3.1.2 Asynchronous Co-flow Synchronization	31
3.1.3 Wave-pipelining	32
3.1.4 Hybrid Wave-pipelining	33
3.1.5 Other Sequencing Techniques	34
3.2 RSFQ Cell-level Library and Design Tools	34
3.2.1 Stony Brook Tunable RSFQ VHDL Cell Library	34
3.2.2 CONNECT Cell Library	39

3.1 Sequencing Techniques

In RSFQ circuits, sequencing techniques play a very important role in overall system performance as they govern hardware usage and data flow affecting processing rate, latency as well as power consumption. The most popular design technique is *classical pipelining* which allows executions of multiple operations to be overlapped. Besides classical synchronous pipelining there are other sequencing methodologies such as wave-pipelining and co-flow. These two will be used in our work on multipliers to provide high-performance and energy-efficient designs.

3.1.1 Synchronous Clocking for Pipelined Designs

In conventional synchronous design multiple *pipeline registers* are inserted inside combinational circuits to synchronize the data between consecutive pipeline stages as shown in Figure 3.1.

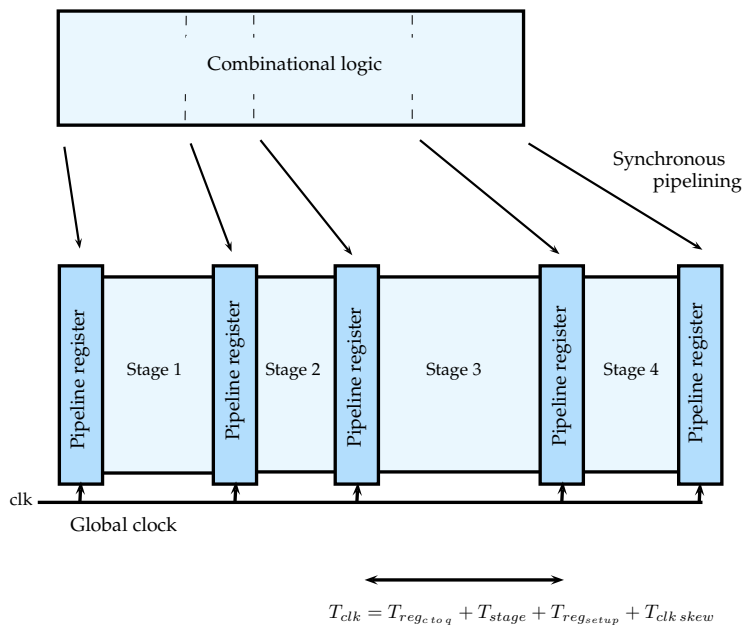


Figure 3.1: Synchronously pipelined circuit.

In order to achieve the best performance using synchronous pipelining, the combinational logic is divided into multiple stages separated by pipeline latches where the longest stage time delay must be minimized as much as possible without violating the setup/hold timing of the register buffer and clock skew.

Hence, the pipeline processing rate is limited by the stage with the longest critical path delay. For example, if the execution time of the stage 3 shown in Figure 3.1 takes much more time than execution of the stage 2, and the minimum clock frequency is limited by stage 4 in this example.

In conventional synchronous RSFQ pipelines, synchronization overhead can easily exceed logic latency due to the small amount of work per stage, and significant clock skew. Making the situation even more difficult is the unavoidable (temperature-induced) timing uncertainty in RSFQ cell delays [18].

Because of the peculiar way the RSFQ signal distribution works, we can not

connect a single signal wire to multiple gates as it is done in CMOS. On the contrary, clock distribution networks need to be build with active elements such as transmitters, receivers, splitters and JTLs as shown in Figure 3.2. As most of the RSFQ gates are synchronous, the global clock distribution network will have to be very big, and will consume vast amount of energy as each cell in clock distribution network will have to work every time clock switches. Additionally, such clock network would require enormous amount of wires, but the number of metal layers are very limited with current RSFQ fabrication technology. Furthermore, the clock skew resulting from such clock distribution would be unacceptable in our ultra-high speed RSFQ design. Therefore, we can not use a global synchronous pipelining for complex RSFQ designs, and only local clock distribution trees are possible.

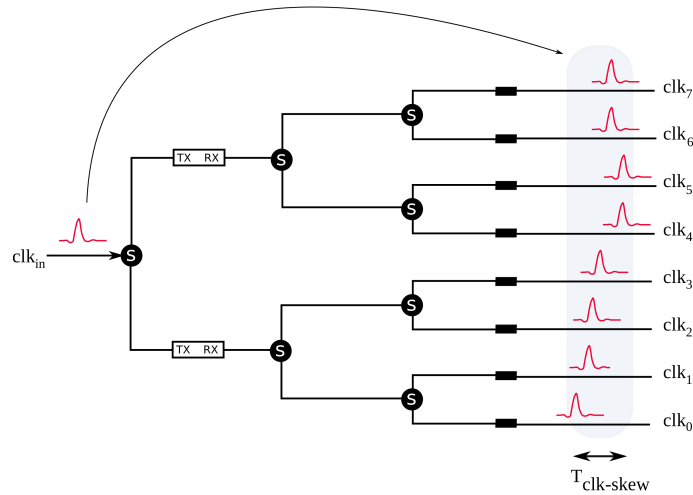


Figure 3.2: Fan-out of 8 implementation with a binary tree.

3.1.2 Asynchronous Co-flow Synchronization

In an *asynchronous co-flow* technique, there is no global clock generator and global clock distribution network. Instead, the clock follows the data inside each pipeline stage as shown in Figure 3.3.

The clock has to be slightly delayed with respect to the data in order not to violate the register setup time.

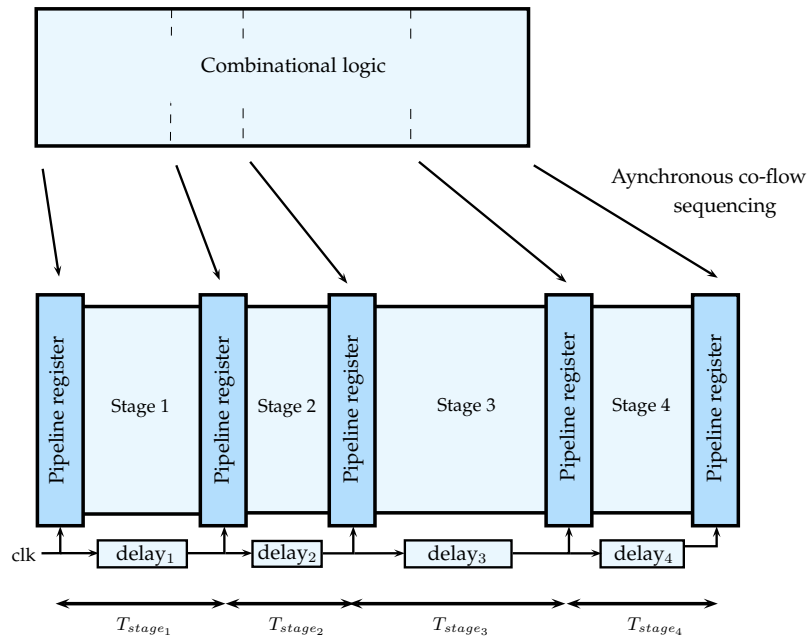


Figure 3.3: Asynchronous co-flow sequencing (clock follows data).

Here, clock fluctuations are smaller, because the data and clock are traveling through very similar paths and are subject to the same temperature and bias current distribution.

Another advantage of asynchronous co-flow sequencing is that the data buffers are clocked only when used (when data are processed), so that dynamic power consumption is cut to the minimum as the Josephson junctions are not switched continuously when unused in data buffers.

3.1.3 Wave-pipelining

Wave-pipelining is an approach aimed to achieve high performance in pipelined systems by removing intermediate latches [41]. This technique increases clock frequency by allowing multiple data waves to exist in any stage. By removing intermediate registers, the area and power associated with the clock are reduced. Wave-pipelining is used in modern CMOS design to improve processor cycle time by pipelining functional units and caches [42].

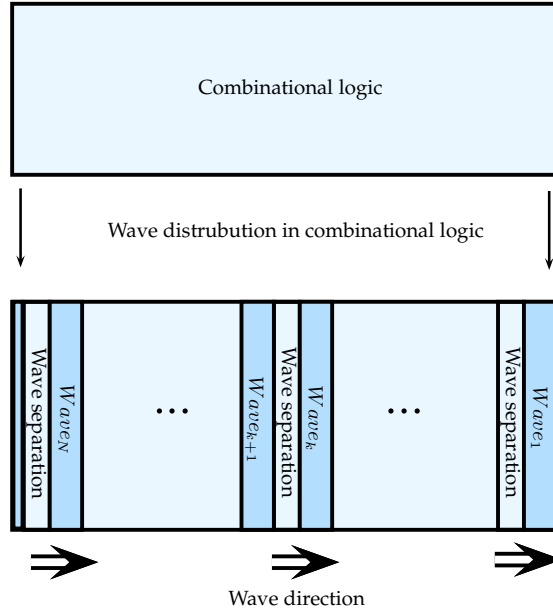


Figure 3.4: Wave-pipelining.

There are two major design challenges for wave-pipelined systems, namely

- 1) preventing collision of unrelated data waves, and
- 2) balancing (equalizing) delay paths in order to reduce differences between the longest and shortest delays through the combinational logic.

The differences can accumulate as the waves propagate through a pipeline, creating the potential for data overrun of unrelated data waves. Ultra-high-rate long RSFQ pipelines are especially prone to this problem.

These problems can be avoided by using the hybrid wave-pipelining approach, where signals are held so the next stage does not start operating until all the signals from the previous stage are available. Wave-pipelining is shown in Figure 3.4.

3.1.4 Hybrid Wave-pipelining

Many RSFQ designs used some form of hybrid wave-pipelining called co-flow synchronization [43, 30, 44, 18, 19, 45], with clock traveling with data across the pipeline, thus eliminating the need for complex central clock distribution.

The challenge for this co-flow synchronization is the necessity to insert carefully-calculated delays into the clock propagation path to honor set-up and hold time requirements for clocked RSFQ cells. The problem becomes more and more severe as the width, length, and complexity of the datapath grows, and timing uncertainty increases [43].

Our RSFQ multiplier designs use both wave-pipelining and co-flow synchronization.

3.1.5 Other Sequencing Techniques

Delay-insensitive dual-rail logic is yet another alternative sequencing technique introduced by Priyadarsan and Polonsky in 1997 [46]. In dual-rail delay insensitive design, the data are represented by two complementary values. Hence, CMOS-like combinational gates can be used in the design. The drawback of this technique is the vast amount of logic used. For example, a dual-rail XOR gate requires 30 junctions [47] while a regular RSFQ AND gate can be built with 8 gates [8]. This technique requires more space, more power, and time to switch each gate.

3.2 RSFQ Cell-level Library and Design Tools

3.2.1 Stony Brook Tunable RSFQ VHDL Cell Library

The purpose of the tunable VHDL RSFQ cell library developed in UHSCL (Ultra-High Speed Computing Lab) at Stony Brook University is to support the design, verification, and testing of wide datapath 10-50 GHz RSFQ processors. Each cell is described using a behavioral model based on finite state machines and/or logical truth tables when applicable. Parameters such as timing constraints, delay jitter, bias current, switching energy and complexity are provided by circuit-level designers. For the current study, the cell parameters have been tuned to the Hypres 1.5 μm 4.5 kA/cm² process.

All propagation delays for each cell are modeled using a normal distribution of the delays for a given value of delay variance. The resulting probability density function for D flip-flop cell is shown in Figure 3.5. We use Monte Carlo style simulation to calculate cell delays each time when the output pulse is generated

for any cell.

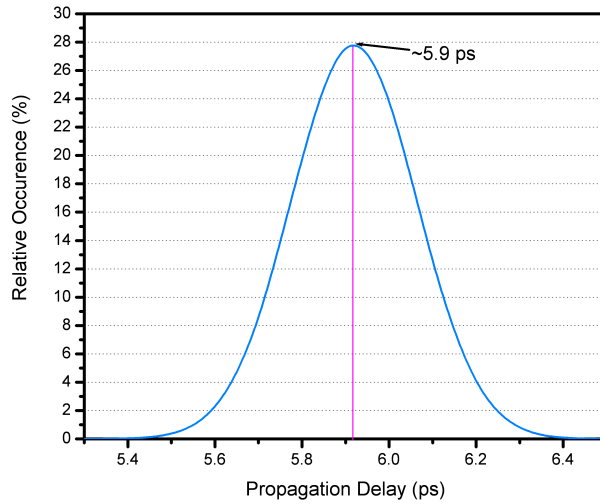


Figure 3.5: Propagation delay model for D flip-flop.

The cells are also checked for any timing violations during simulation and report these violations as a failure. A failure report consists of important information such as what kind of timing constraint has been violated (e.g. setup or hold time), where the failure occurred and which inputs caused it. Furthermore, all possible ways a cell can consume dynamic energy are stored in a switching table so that for a given switching event there is a corresponding amount of energy consumption which is locally accumulated within each instance of a cell. When the simulation is over, all cells in the design report their accumulated energy consumption and are summed up together to produce the final total amount of dynamic energy consumption for the entire design simulation. Finally, the cell library provides a set of procedures/functions for the logic designer to easily obtain complexities for a design and measure the different margins of clock rate that a design can support [18].

3.2.1.1 Key Cells Used in Multiplier Design

A T1 flip-flop (T1) serves as a basic building block in the compression tree of both multipliers. The state diagram of a T flip-flop is shown in Figure 3.8. It

Table 3.1: Key characteristics of the Hypres 1.5 μm 4.5 kA/cm² process.

Feature	Hypres [48]
Critical current (kA/cm ²)	4.5
Feature size (μm)	1.5
Number of Nb metal layers	4

works as a 2-to-1 frequency divider for input signal t , producing SFQ pulse at $q0$ output every other input pulse. Additionally, if clock c is applied after an odd number of input pulses was detected since the last clock, the SFQ pulse is produced at $q1$ output.

Since the T1 cell can detect an odd number of inputs, it can also be used as a *full adder* with an asynchronous *carry* and synchronous *sum* outputs as shown in Figure 3.9.

In 1997 a RSFQ T flip-flop fabricated at Stony Brook University was shown to operate at 770 GHz, and it is the fastest RSFQ gate demonstrated up to date [49].

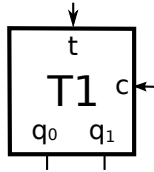


Figure 3.6: T1 cell symbol.

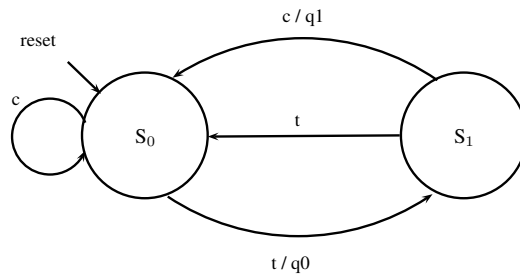


Figure 3.7: T1 cell state diagram.

Other cells used frequently in our multiplier designs are listed in the Table 3.2.

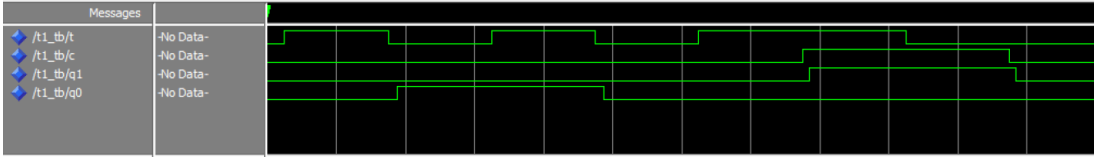


Figure 3.8: T1 cell simulation.

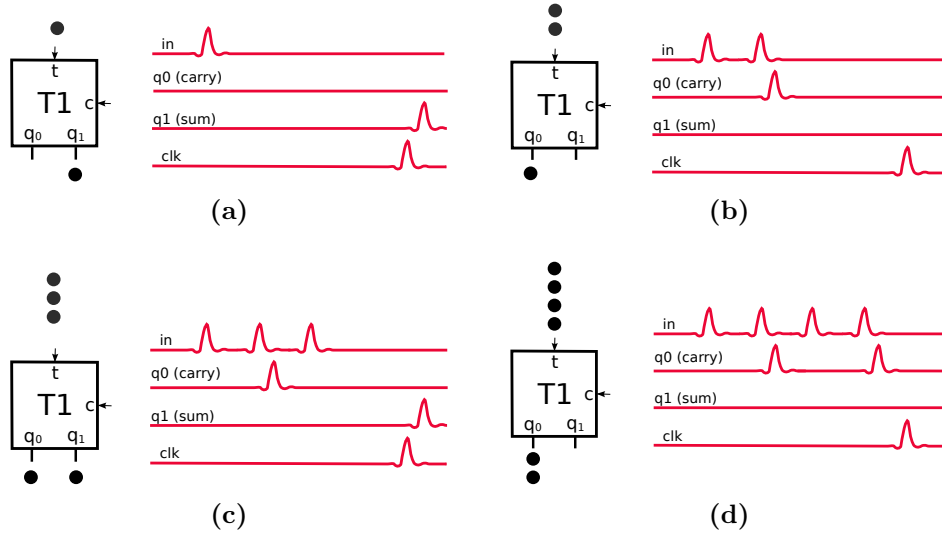


Figure 3.9: T1 cell wave diagram. Each dot represents a SFQ pulse.

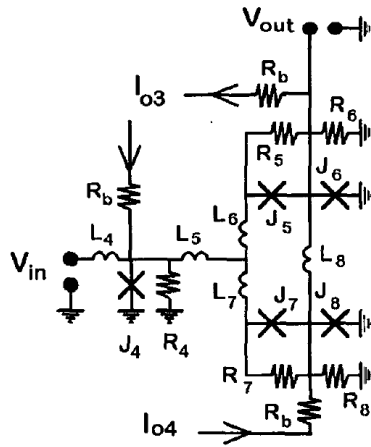

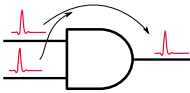
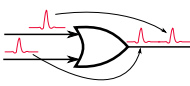
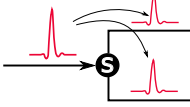
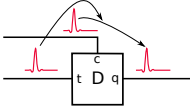
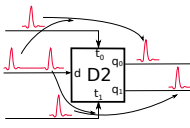
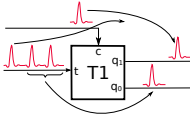


Figure 3.10: T1 cell schematic [49].

Table 3.2: RSFQ cells widely used in our multipliers.

Cell name	Symbol with SFQ diagram	Cell Description
JTL(x)		Josephson junction can be used to either introduce additional delay in the circuit or to amplify weak SFQ pulses. The x in JTL(x) signifies number of JTLs represented by the JTL symbol.
AAND		Asynchronous AND gate performs logical AND function. To get a SFQ pulse at the output, both inputs must arrive within $T_{\text{AAND_MINGAP}}$ time. Otherwise the output pulse is not generated.
MRG		Merger cell can be used to combine multiple SFQ signals onto single wire track. The input SFQ pulses must be separated by $T_{\text{MRG_MINGAP}}$ for correct operation.
SPL		Splitter cell is used to provide a fan-out of 2. Multiple SPL cells can be employed when larger fan-outs are needed.
DFF		D flip-flop is used to store a SFQ pulse. The input pulse is transferred to the output when clock signal is applied.
D2FF		D flip-flop with two clocks and two outputs. It is used to store the SFQ pulse, with an option to route the output to one of the two different outputs q_1 or q_0 depending whether t_1 or t_0 is asserted first.
T1		T1 flip-flop can be used as a half-adder, or as a frequency divider. The T1 cell generates SFQ output on q_0 for every two consecutive input SFQ pulses since last clock t was applied. The q_1 output pulse is generated only when the clock is applied and an odd number of SFQ pulses arrived on d since last clock was applied.

3.2.2 CONNECT Cell Library

Besides using our Stony Brook VHDL cell library we also use the CONNECT cell library developed in Nagoya University and Yokohama National University [39]. This library and associated CAD tools are used for logical and physical layout chip design of our 8-bit integer multiplier, and to generate a Verilog model used for circuit simulation. All cells in this library were fabricated using ISTECH process and characterized to get physical parameters (e.g. delay) necessary to model each cell. This library was demonstrated experimentally and was proven reliable for multiple designs [39].

Table 3.3: Key characteristics of the ISTECH 1.0 μm 10 kA/cm² process.

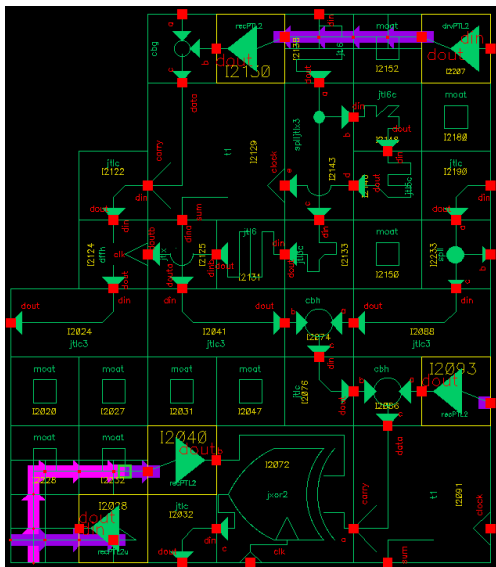
Feature	ISTECH [50]
Critical current density (kA/cm ²)	10
Minimum JJ feature size (μm)	1.0
Number of Nb metal layers	9

The CONNECT cell library has more than 300 cells at present, with multiple layout versions available for each cell. Every cell in the CONNECT library consists of a Verilog digital behavioral model, JJ circuit information, and a physical layout.

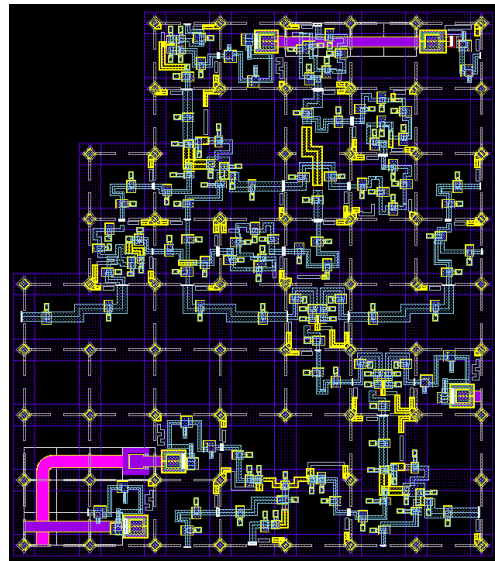
At the layout level, each cell is designed around a $30\mu\text{m}\times 30\mu\text{m}$ grid, and every cell size is an integer multiple of this size, so that cells can be placed next to each other. This way, it is simple to match the blocks together to provide more complex structures.

Using this semi-custom design flow, complex circuits can be designed in much faster way than when using full custom design approach with its non-custom shaped cells.

All cells have their schematic views that closely match their layout. The passive transmission lines and vias are also provided at this level and can be placed over JTL level blocks. A sample schematic built using this technique is shown in Figure 3.11(a). Once the schematic is built and verified, the layout and Verilog model are generated. The Verilog model can be used for circuit verification, as well as for bias margin and circuit rate calculations.



(a) schematic



(b) layout

Figure 3.11: Example schematic and physical layout views of a circuit designed with the CONNECT cell library.

11.4 GHz 32-bit RSFQ Integer Multiplier Design and Evaluation

Contents

4.1	Goals and Challenges	41
4.2	RSFQ Hybrid Wave-pipelined Asynchronous Multiplier Miroarchitecture and Cell-Level Implementation	42
4.2.1	Partial Product Generation	42
4.2.2	Partial Product Compression	50
4.2.3	Final Summation	56
4.3	Integer Multiplier Design Summary	58

4.1 Goals and Challenges

Our goal in a 32-bit integer multiplier design was to design a wide-datapath multiplier operating at 10 GHz+ frequencies with the lowest possible latency when implemented with the Hypres 1.5 μm 4.5 kA/cm² fabrication process. Due to existing fabrication technology limitations, the design had to have a reasonable complexity below 100,000 JJs.

To achieve this, an efficient 32-bit multiplier microarchitecture with appropriate clocking mechanism had to be developed.

Hence, our multiplier microarchitecture features wave-pipelining and co-flow sequencing without global synchronous clocking.

A novel T1 based design technique is used for building compressor trees. This allowed us to achieve 10 GHz+ processing rates with modest amount of hardware.

To achieve logarithmic execution time, we have connected individual compressors in a way that allows us to decompose partial product columns into multiple smaller groups which are processed in parallel.

4.2 RSFQ Hybrid Wave-pipelined Asynchronous Multiplier Miroarchitecture and Cell-Level Implementation

A 32-bit parallel integer multiplier presented in this dissertation utilizes a high performance parallel carry-save reduction tree with short logarithmic delay. To further achieve the best timing characteristics, an existing parallel prefix adder described in [30] was used for final summation. The nine least significant bits of the multiplication product are calculated in the compression tree and buffered using a partial sum buffer, while the other 23-bits are calculated by the parallel prefix tree adder. The block diagram of resulting 32-bit RSFQ integer multiplier is shown in Figure 4.1.

Hence, the multiplier is composed of four asynchronously clocked blocks:

- (1) A product generator with wave-pipelining,
- (2) Compression tree with wave-pipelining,
- (3) Data buffer,
- (4) A 23-bit asynchronous wave-pipelined sparse tree adder (STA).

4.2.1 Partial Product Generation

When two 32-bit integers are multiplied, a 64-bit result can be produced as shown in Figure 4.2. In such multiplier there are a total of $32^2 = 1024$ partial product bits which are used to produce a full 64-bit result.

The RSFQ multiplier designed here calculates low 32 bits of the product. In that approach only the right half of the product generator is used, thus reducing the number of partial products that need to be added from 1024 to $\sum_{k=1}^{32} k = 528$.

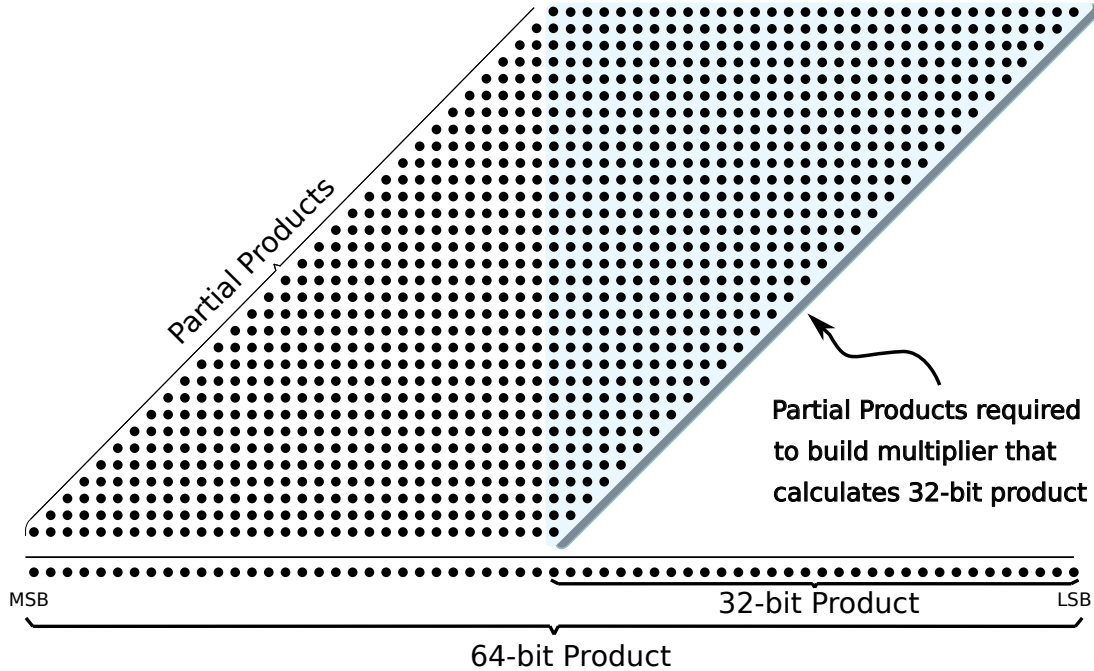


Figure 4.2: 32-bit multiplication. Here, each dot represents a single bit of either partial or final product.

4.2.1.2 Timing

The best way to achieve high performance would be to generate all bits in individual partial products simultaneously. However, this requires multiple parallel clock distribution trees capable of generating SFQ pulses for each bit, which would take most of the chip area. The large number of wire tracks would be needed to route clock signals from the main splitter tree to each partial product bit. This will be challenging due to a small number of metal layers available for the interconnect.

On the other hand, in series clocking of each partial product bit involves large delays between the most significant and the least significant bits. As mentioned in Section 3.2.1, whenever a fan-out of two or more is required, a splitter cell must be inserted. Inserting splitter cells in series, slows down the overall signal propagation as well. Also, due to the fact that the size of each partial product generator bit is relatively large, a passive transmission lines with their transmitters and receivers need to be placed between each partial product bit adding to

the delay which accumulates as the signal propagates. Therefore, a serial clock distribution scheme could not be used alone for the ultra-high speed processing, and a compromise which uses both schemes is employed. First, clock signals are distributed using a binary tree, and then a serial clock distribution inside each of the 4x4 groups of a partial product matrix is carried out as shown in Figure 4.3 and Figure 4.4(a).

As a result, the individual bits from the partial product matrix are generated over the relatively large period of time which spans over two clock cycles. The product generation time for each bit is shown in Figure 4.5.

A similar approach is also used for multiplicand and multiplier distribution where the operands are distributed by binary trees to each of the 4x4 groups, and then in series inside each group.

Another optimization which allows us to reduce the amount of wire tracks is to combine multiple partial product bits into a single wire track. Every column of the partial product generator is divided into groups of four partial products per group. Next, the four partial product bits from the same column are combined together using merger cells (MRG) and sent over a single wire track as shown in Figure 4.6.

As a result of uneven clock distribution, the bits in a single partial product are delayed with respect to each other. Therefore, they either need to be processed at different times or they must be synchronized before reaching the compression tree.

First, we have designed the compressor to be able to work with the resulting data skew. The related design worked at acceptable high clock rates, however required a large amount of delay elements inside each compressor and resulted in significant amount of glue logic which had to be inserted between every 4 and 16-bit groups.

To avoid this problem in the final design, the partial product generation timing was synchronized and all the bits in every single partial product are aligned before they reach the compression tree as shown in Figure 4.7. The resulting simplified compressor circuit is less complicated, uses less power, has better latency, and improved processing rate compared to the first design which had to deal with the skewed input data.

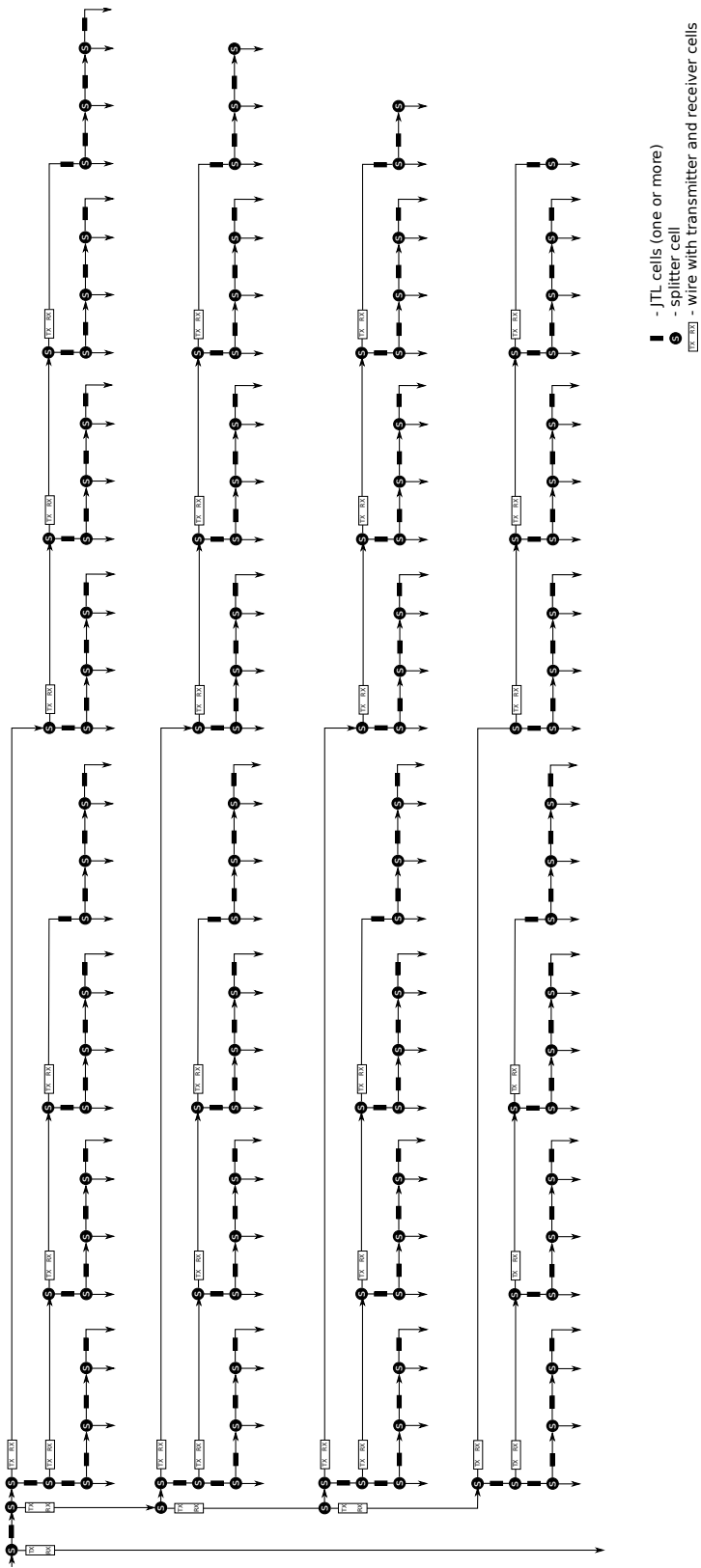
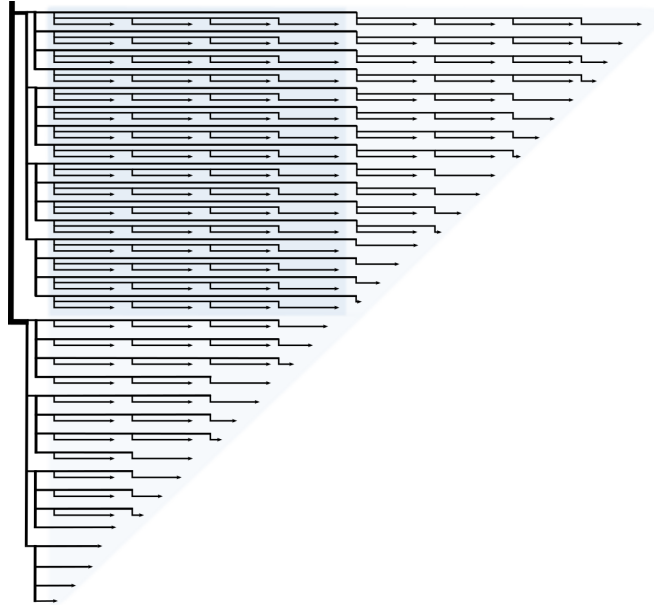
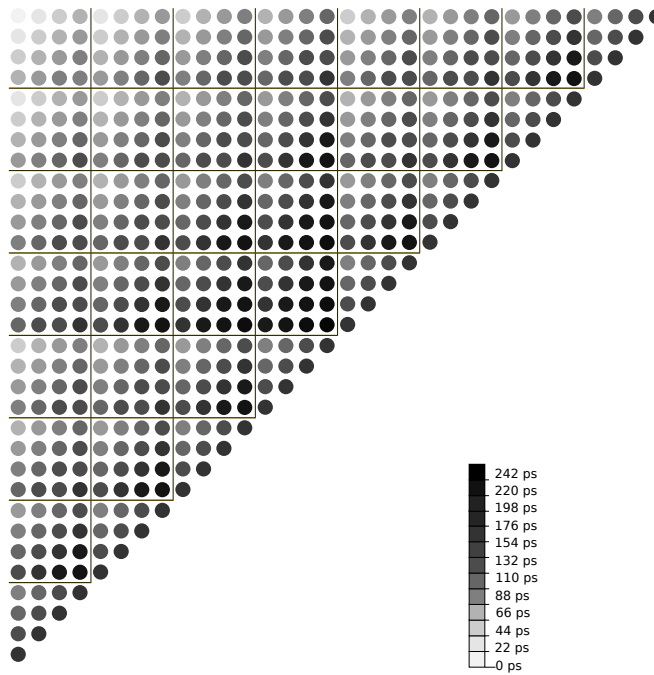


Figure 4.3: Clock distribution for top four partial products.



(a) Clock distribution for partial product generator.



(b) Partial product generation timing profile.

Figure 4.4: Clock distribution and timing for the partial product generator.

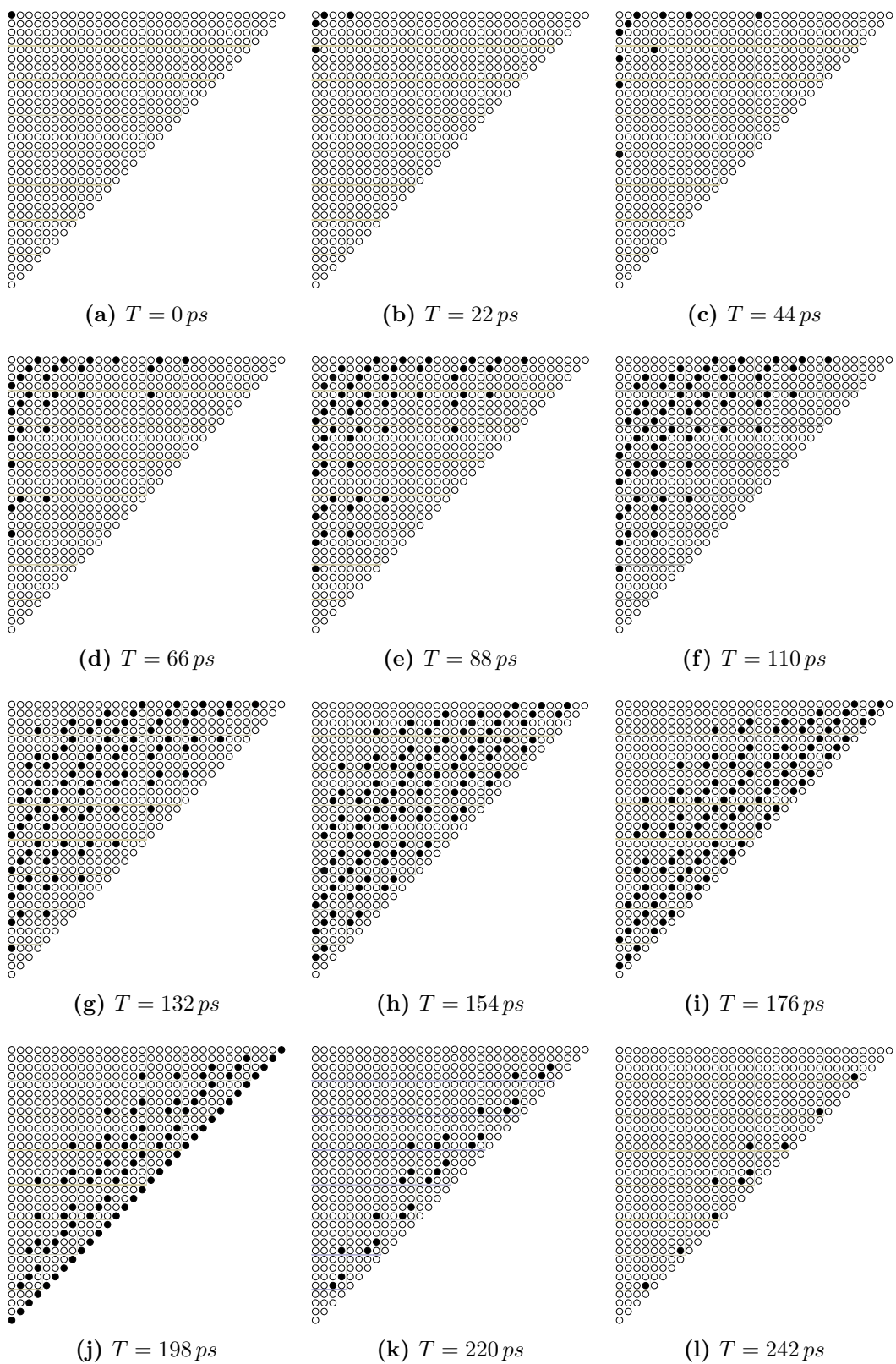


Figure 4.5: Step-by-step timing during partial product generation.

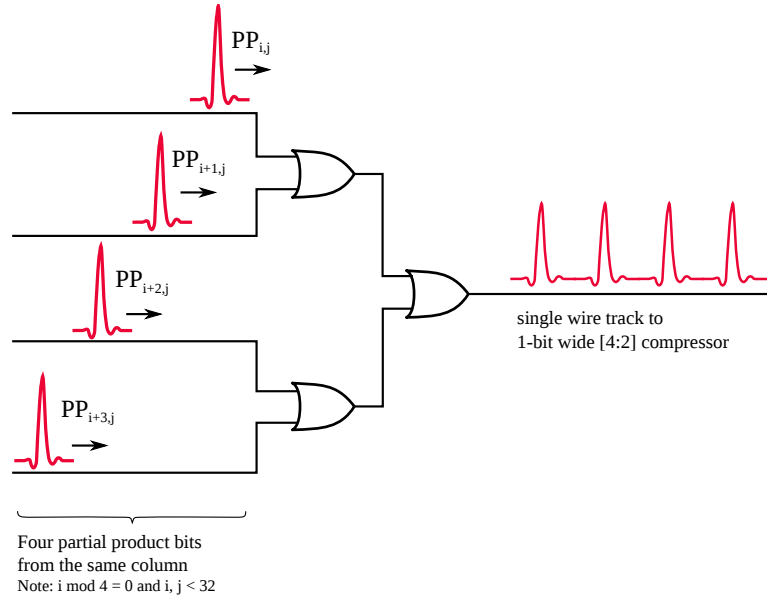


Figure 4.6: Routing four partial product bits with the same arithmetic weight onto a single transmission line.

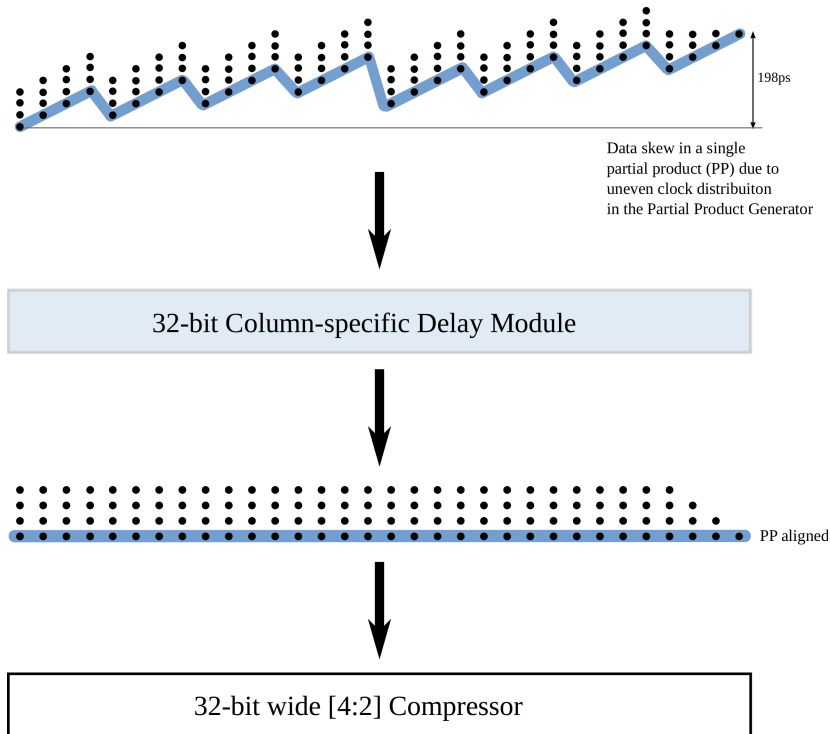


Figure 4.7: Time synchronization of partial product generator output.

4.2.1.3 Pipelining

Due to the relatively large latency of this unit, the partial product generator is pipelined with two pipeline stages to provide the fastest possible processing rate.

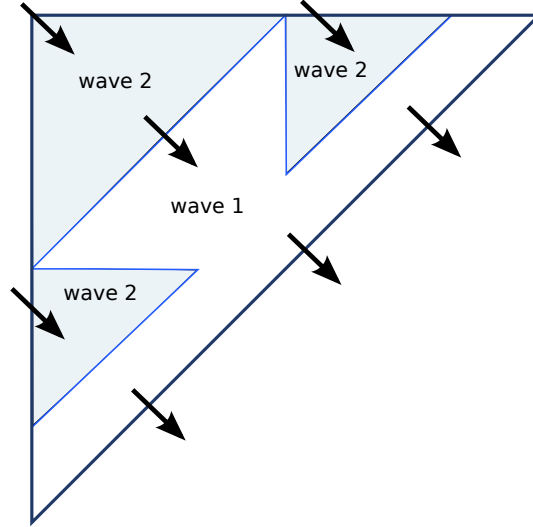


Figure 4.8: Wave-pipelining of the partial product generator.

Pipelining is done with wave-pipelining. Two pipeline stages are shown in Figure 4.8, where two consecutive operand waves can be processed at the same time. All partial products are generated after both stages are completed.

4.2.2 Partial Product Compression

4.2.2.1 [4:2] Compressor

Once partial products are generated, these need to be added together (compressed). The compression is done by adding the partial products column by column, with carries from each column propagating toward the most significant column. The addition is done with a use of [4:2] compressors. In RSFQ, we build the [4:2] compressor by connecting (4,3) and (3,2) counters together with carry-save path as shown in Figure 4.9.

We use the T1 cell to build both counters. This cell was described in Section 3.2.1.1, but the functionality is also shown in Table 4.1. Whether the T1 cell serves as (4,3) or (3,2) counter depends on its position in a [4:2] compressor

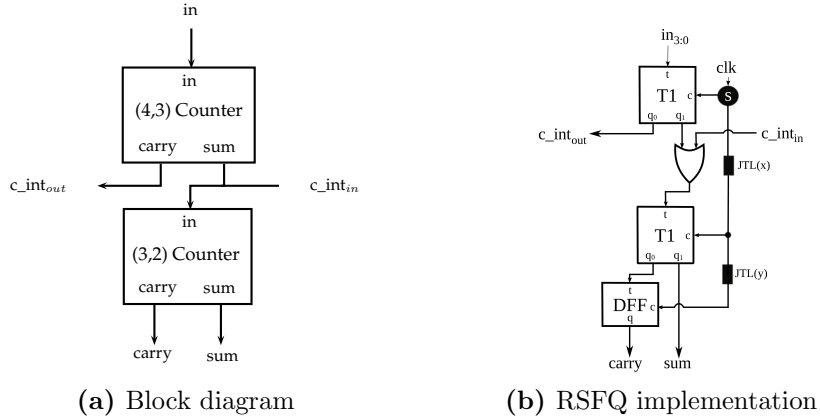


Figure 4.9: [4:2] compressor.

as shown in Figure 4.9. When placed on the top, it counts up to 4 inputs serving as a (4,3) counter. On the other hand, the bottom T1 cell handles up to 3 input SFQ pulses implementing a (3,2) counter.

Like any clocked RSFQ cell, the T1 cell has a built in buffer for the sum output. This buffer is used for co-flow synchronization.

Table 4.1: T1 cell used as (4,3) and/or (3,2) counter.

Input Sequence	Output	
	Carry(q0)	Sum(q1)
0	0	0
1	0	1
1 1	1	0
1 1 1	1	1
1 1 1 1	1 1	1

It is also possible to build larger compressors like a [7:3] compressor this way, but the processing rates of T1 cell will drop significantly if each T1 cell will have to process more than four partial product bits per cycle. This comes from the fact that each input signal must be separated by the time required for a correct T1 cell operation. In fact, the operating frequencies of 10 GHz or above are attainable only if four or less data input bits are processed in one period.

To build an N-bit wide [4:2] compressor, multiple [4:2] compressors are placed

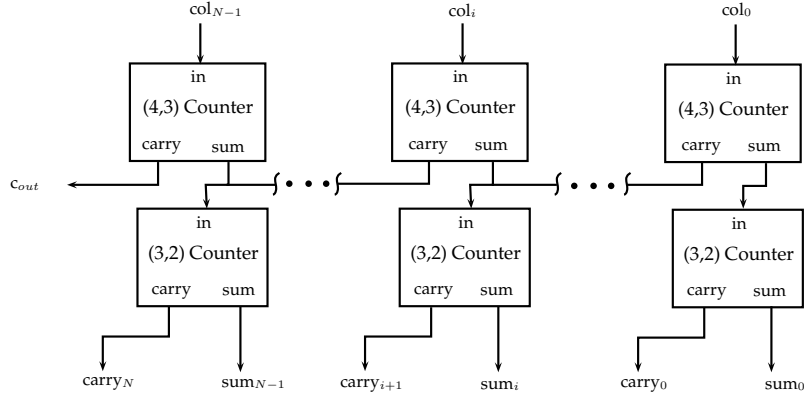


Figure 4.10: N-bit wide [4:2] compressor, $i < N : i, N \in \mathbb{N}$.

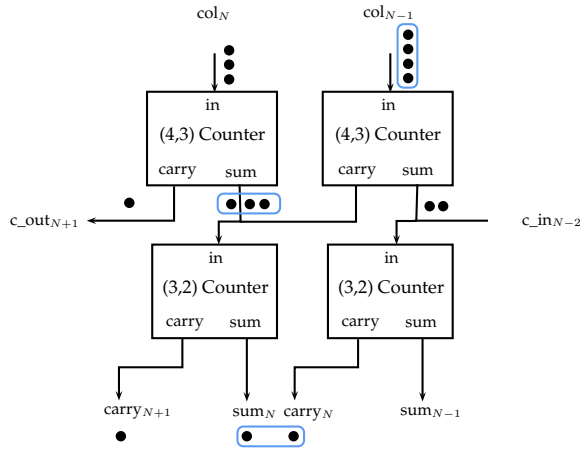


Figure 4.11: [4:2] compression example.

next to each other and connected with intermediate carries as shown in Figure 4.10. The N-bit wide [4:2] compressor can compress up to four operands (i.e. partial products) into two operands.

The 4-to-3 and 3-to-2 compression steps are illustrated in an example shown in Figure 4.11.

Our [4:2] compression is a six step process in which up to four partial products are processed asynchronously followed by two clock signals which are used to complete and synchronize the compression process as shown in Figure 4.12.

Here, after four inputs arrive, these are reduced from 4 to 3 using the top (4,3) counter. Next, the asynchronous co-flow clock is sent for an additional (3,2)

reduction in the second stage. Finally, the resulting carry-sum pair is generated after the second co-flow clock is applied to bottom (3,2) counter and the D flip-flop buffer.

It is also worth to mention that the carry does not have to be buffered in first stage but only in the second stage. This comes from the fact that only one asynchronous carry-out is possible after the compression.

4.2.2.2 Compressor Pipelining

The multiplication operations can be sent to our compressor at 11.4 GHz rate, and each operation takes 1.5 cycles to execute. The execution is ultra-pipelined with four 45.6 GHz micro-stages per 11.4 GHz stage as shown in Figure 4.13.

The first four micro-stages are ultra-pipelined using wave-pipelining without any clocking. The last two stages are pipelined using internal sum buffers which are available as a part of T1 cell. An additional D flip-flop cell is used to synchronize the carry output as mentioned. These last two micro-stages are synchronized with the co-flow clocking.

Another feature of our pipelining is that the execution of two multiplication instructions is partially overlapped within each compressor.

4.2.2.3 Tree Structure

Due to the fact that each [4:2] compressor (bit slice) in an N-bit wide [4:2] compressor can compress only up to four bits, the partial product generator was divided into eight groups of four partial products per group. Next, each group of four partial products is handled by a separate N-bit wide [4:2] compressor as shown in Figure 4.14.

Since each N-bit wide [4:2] compressor produces two bits per column, the outputs from the top-level N-bit wide [4:2] compressors are combined together in the second stage and processed by another compressor. The compression process continues until the final 32-bit carry-sum pairs are generated as shown in Figure 4.15.

A single bit-slice cross-section view of the compression tree for the most significant bit column is shown in Figure 4.16. Here, all the partial products in a single column of 32 products are reduced into 16, then into 8 at the second level, 4 at the third level, and finally into a single carry sum pair. The carry-sum pair

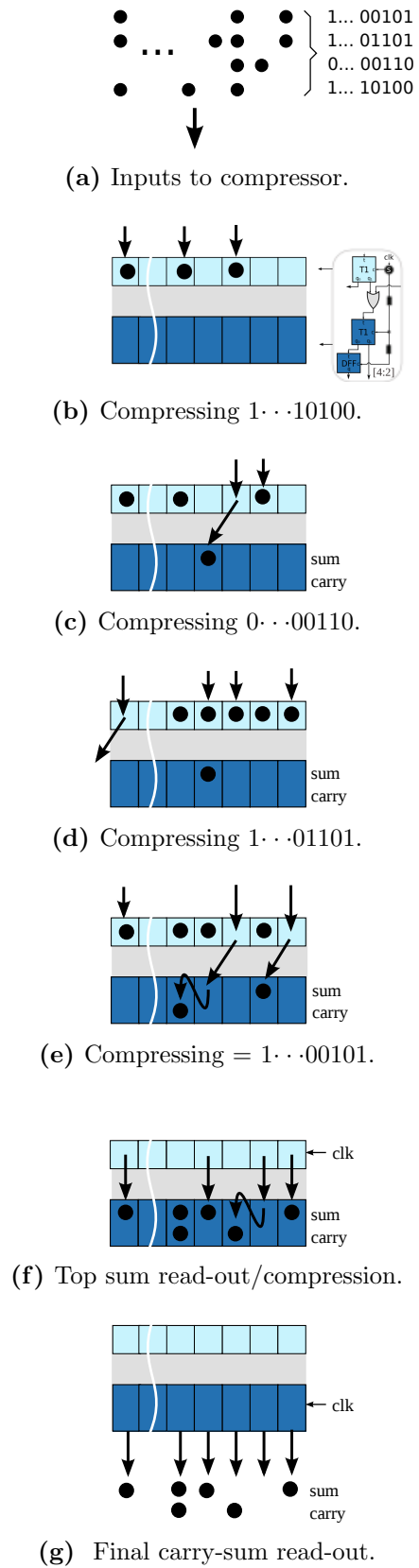


Figure 4.12: Step-by-step compression process inside an N-bit wide [4:2] compressor.

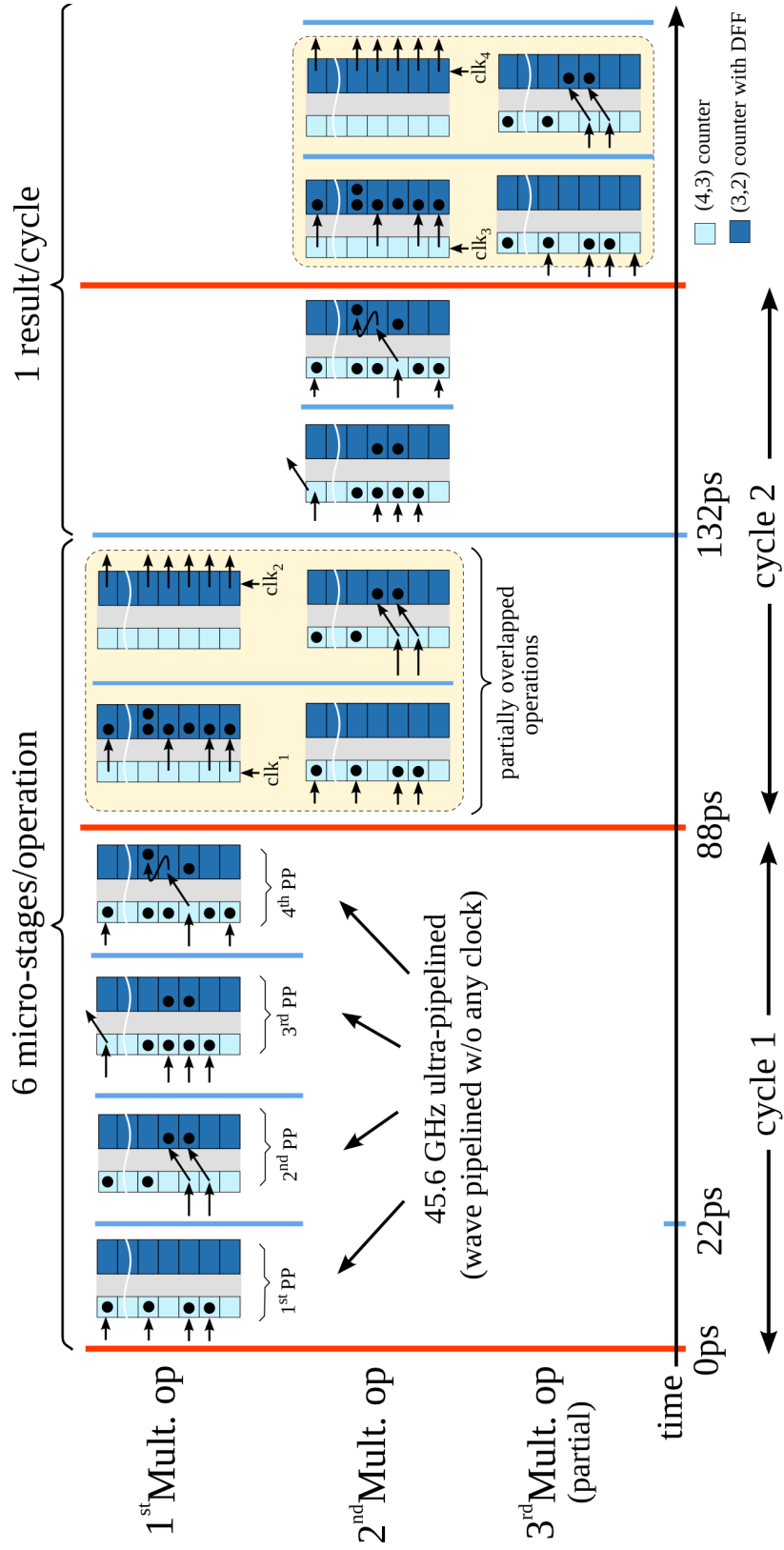


Figure 4.13: Hybrid wave-pipelined and asynchronous co-flow sequencing of an N-bit wide [4:2] compressor.

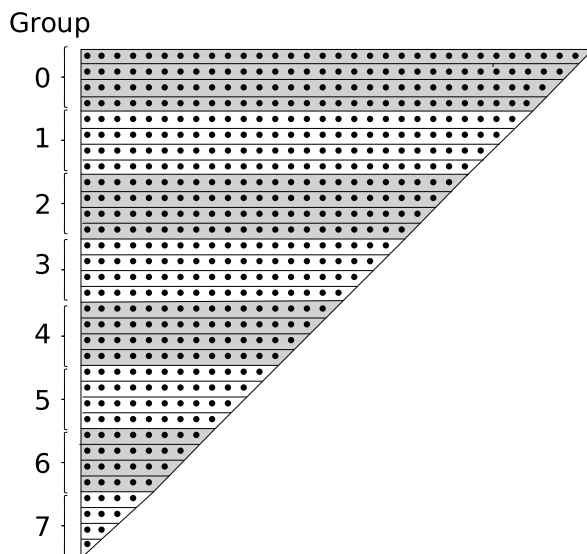


Figure 4.14: Partial product grouping for compression tree.

Table 4.2: Number of [4:2] compressors needed to compress each group of four partial products.

Group from Figure 4.14	Number of [4:2] compressors needed
0	32
1	28
2	24
3	20
4	16
5	12
6	8
7	4

is completely reduced into a single bit output with the final adder.

4.2.3 Final Summation

As mentioned in Section 2.7 the adder design is beyond the scope of this dissertation. Therefore, in the final summation to calculate the product from carry-sum

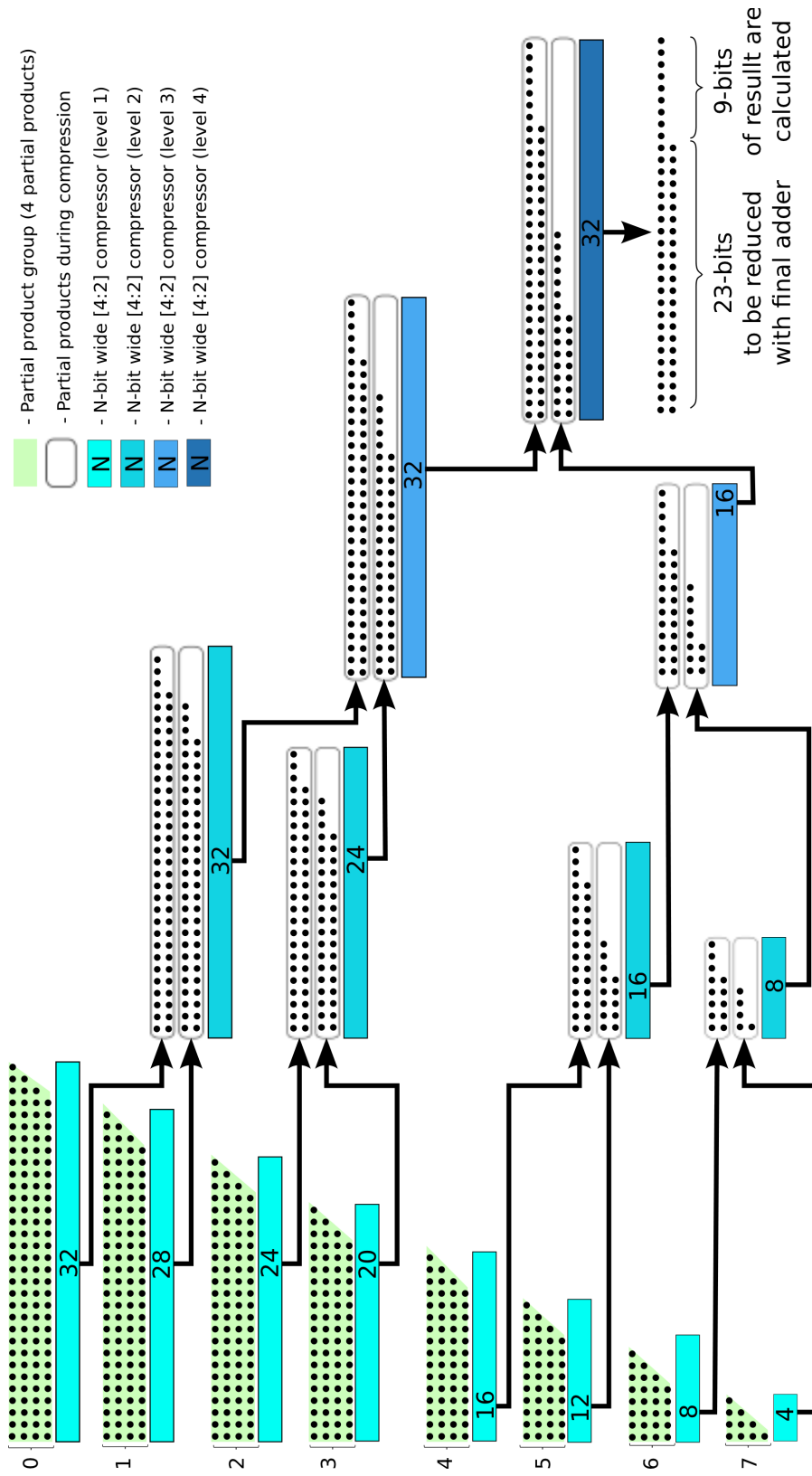


Figure 4.15: Compressor tree structure for the 32-bit RSFQ integer multiplier.

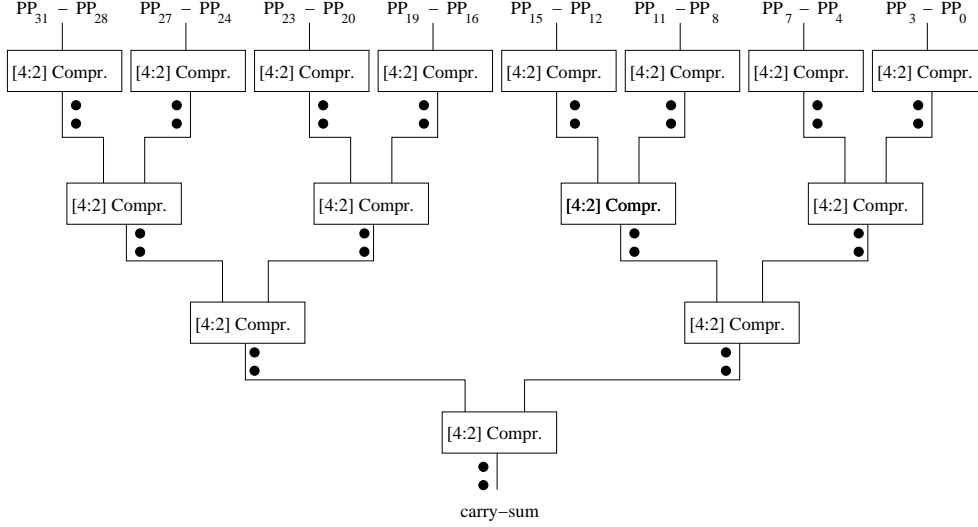


Figure 4.16: Compression tree for the leftmost vertical bit column of the partial product generator. The intermediate carries which are sent to higher-order bits (the next vertical column to the left of the current column) are not shown. Single carry-sum pairs are represented by two dots. Here, PP_i is the multiplicand bit $(31-i)$ ANDed with i^{th} bit of a multiplier (i.e. $PP_{31}-PP_0$ is a single column of partial product generator) where $i < 32$ and $i \in \mathbb{N}$.

operands a high performance RSFQ 23-bit sparse tree adder developed by M. Dorojevets and C. Ayala [30] is used in this design.

4.3 Integer Multiplier Design Summary

The cell-level design of a RSFQ 32-bit integer multiplier was verified using VHDL simulation with over 100,000 random operands at the processing rate of 11.4 GHz with the total latency of 1.4 ns.

The current distribution for each part of the multiplier is given in Table 4.4, and the complexity of the 32-bit integer multiplier can be found in Table 4.5.

Taking into account that the final design has around 76K JJs, we consider our design goals achieved for this 32-bit RSFQ integer multiplier.

Table 4.3: 32-bit RSFQ integer multiplier characteristics at $T = 4.2$ K.

Maximum frequency (GHz)	11.4
Total latency (ps)	1,409
Total complexity (JJs)	75,811
Total bias current (A)	11.065

Table 4.4: JJ complexity breakdown and bias current distribution by component in the 32-bit RSFQ integer multiplier.

Stage	Complexity (JJs)	I_{bias} (A)	% Total bias
Partial Product Gen.	23,265	3.60	32.49
Compressor	42,841	6.16	55.67
Final Adder	9,220	1.24	11.24
Other ¹	275	0.06	0.60
Total	75,811	11.07	100.00

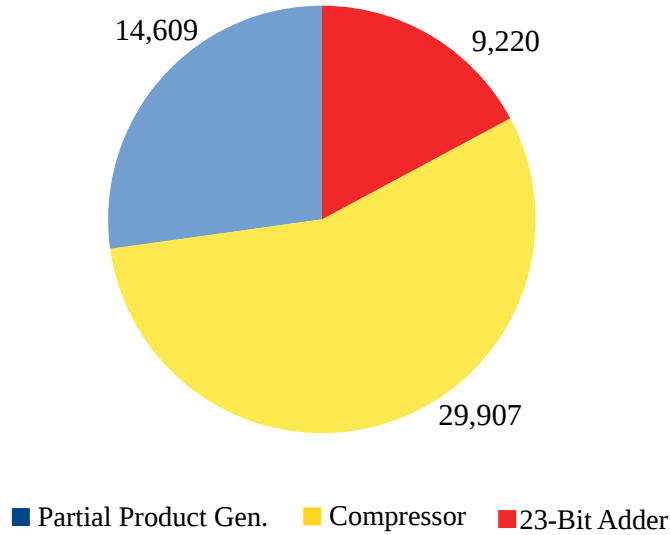


Figure 4.17: JJ distribution per component for the 32-bit RSFQ integer multiplier.

¹Includes JJs used for cell separation, delay lines and clocking.

Table 4.5: JJ and bias current breakdown per logic and interconnect for the 32-bit RSFQ integer multiplier.

Category	JJ Count	% JJs	Total I_{bias}	% I_{bias}
Logic	25,767	33.99	2,221.63	20.10
Splitters	4,674	6.17	1,238.61	11.20
TX/RX	8,653	11.41	1,169.12	10.58
Other ²	36,717	48.43	6,425.48	58.12
Total	75,811	100.00	11,054.83	100.00

Table 4.6: Latency breakdown for the 32-bit RSFQ integer multiplier.

Stage	% Latency	Latency (ps)
Partial Product Generator	22.00	310
Compressor	62.10	875
Final Adder (STA)	27.89	393
Overlapped ³	11.99	-169
Total critical path latency:		1,409

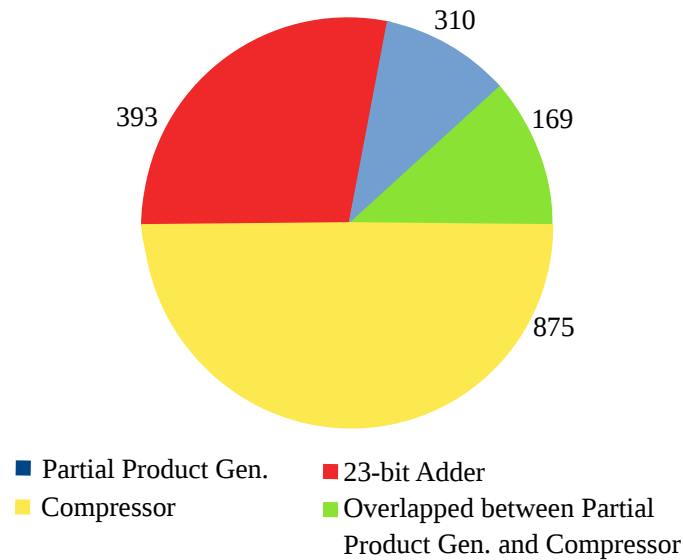


Figure 4.18: Latency breakdown for the 32-bit RSFQ integer multiplier.

²Includes JJs used for cell separation, delay lines and clocking.

³Partial product generation and compression are partially overlapped.

Table 4.7: Total cell breakdown for each cell in the 32-bit RSFQ integer multiplier.

Cell	Cell Count	JJs/Cell	Total JJs/Cell	% Cell	% JJs
AAND	528	6	3,168	1.05	4.18
CFF	119	16	1,904	0.24	2.51
T1	771	9	6,939	1.53	9.15
DFF	1,607	4	6,428	3.20	8.48
JL	36,717	1	36,717	73.09	48.43
MRG	1,354	5	6,770	2.70	8.93
RX	2,124	3	6,372	4.23	8.41
CXOR	62	9	558	0.12	0.74
SPL	4,674	1	4,674	9.30	6.17
TX	2,281	1	2,281	4.54	3.01
Total	50,237	N/A	75,811	100.00	100.00

Table 4.8: Cell bias current distribution for the 32-bit RSFQ integer multiplier.

Cell	Current/Cell (mA)	Total Current/Cell (mA)	Total Current (%)
AAND	0.85	448.80	4.06
CFF	1.14	136.10	1.23
T1	0.50	385.50	3.48
DFF	0.35	562.45	5.08
JL	0.18	6,425.48	58.07
MRG	0.51	688.78	6.22
RX	0.31	655.89	5.93
CXOR	0.18	10.85	0.10
SPL	0.27	1,238.61	11.19
TX	0.23	513.23	4.64
Total	N/A	11,065.68	100.00

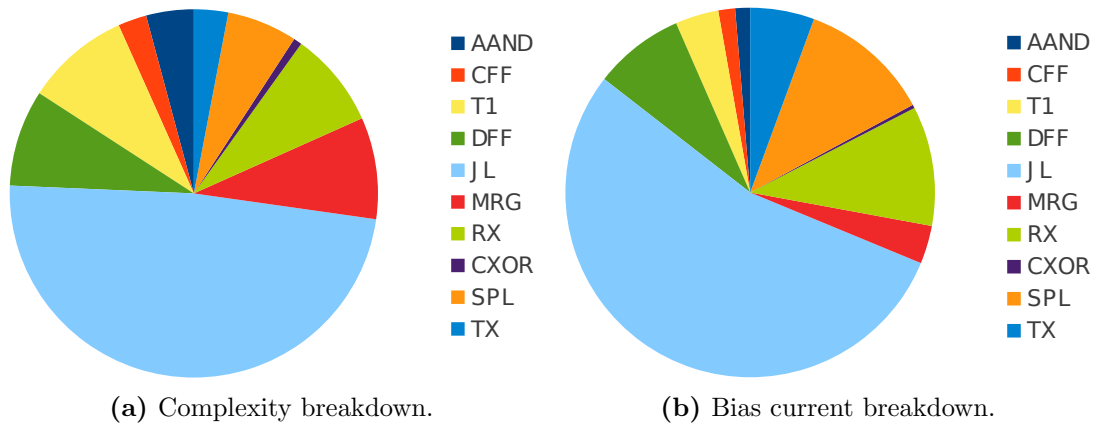


Figure 4.19: Complexity and current distribution per cell for the 32-bit RSFQ integer multiplier.

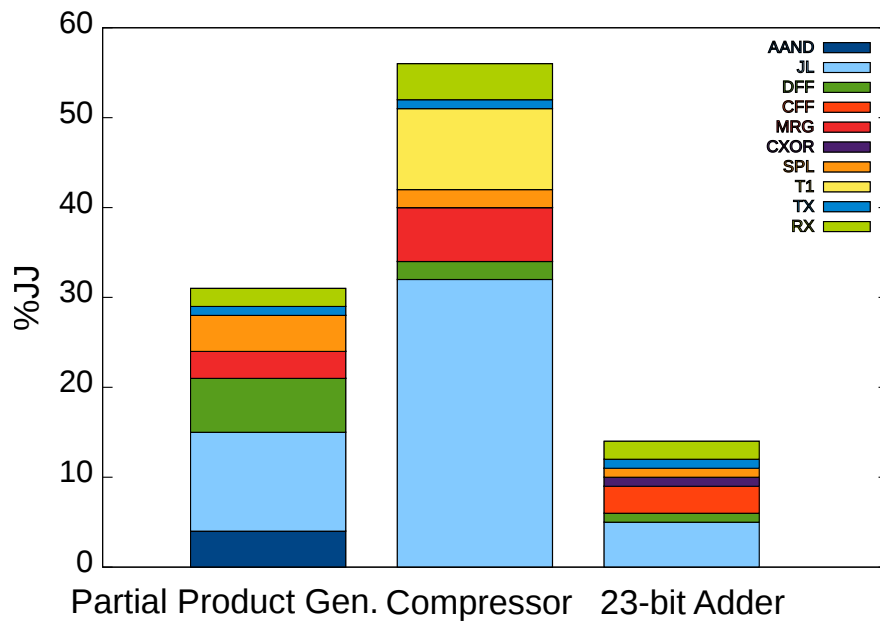


Figure 4.20: Complexity breakdown by component for the 32-bit RSFQ integer multiplier.

11.1 GHz 32-bit Single-precision Floating-point Multiplier Design and Evaluation

Contents

5.1	Goals and Challenges	64
5.2	Floating-point Multiplication Basics	64
5.3	32-bit Floating-point Multiplier Structure	65
5.4	Sign Bit Calculation	65
5.5	Exponent Calculation Unit	67
5.5.1	Zero Value Detection	67
5.5.2	Initial Exponent Calculation Unit	67
5.5.3	Exponent Data Buffer	69
5.5.4	Exponent Adjustment Unit	69
5.6	Mantissa Calculation Unit	70
5.6.1	Partial Product Generation	70
5.6.2	Compression	76
5.6.3	Final Summation Unit	78
5.6.4	Sticky Bit Calculation	78
5.6.5	Normalization and Rounding Units	78
5.7	Floating-point Multiplier Design Summary	79

5.1 Goals and Challenges

Our goals in 32-bit floating-point multiplier design were similar to those of 32-bit integer multiplier. We aimed at a wide-datapath multiplier operating at the 10 GHz+ frequencies with the lowest possible latency when implemented with Hypres 1.5 μm 4.5 kA/cm² fabrication process. The design required a complexity below 100,000 JJs.

Our multiplier microarchitecture features wave-pipelining and co-flow sequencing without global synchronous clocking, parallel execution, and synchronization between exponent and mantissa calculation blocks.

To achieve logarithmic execution time, we used binary compression tree and carry-propagate adder. Furthermore, we connected all individual blocks in a way that enabled us to deliver calculated results with the best timing possible.

5.2 Floating-point Multiplication Basics

The 32-bit floating point (FP) multiplier designed in this chapter implements the single-precision IEEE-754 floating-point arithmetic standard supporting the most complex rounding mode (rounding to the nearest) and excluding denormalized numbers.

In IEEE-754 notation, floating-point numbers are represented using a sign bit, an 8-bit wide exponent, and a 23-bit wide fraction as shown in Figure 5.1.

The sign bit determines the sign of the number, which is the sign of the significand. The exponent is an 8 bit signed integer from -128 to 127 (2's complement). In this case an exponent value of 127 represents the actual zero.

The true significand also referred to as *mantissa* includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros in which case the denormalized number is represented (denormalized numbers are unsupported in this design). Thus only 23 fraction bits of the significand appear in memory format but the total precision of floating-point number is 24 bits.

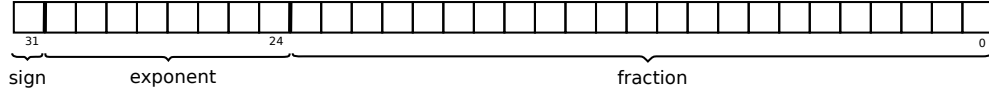


Figure 5.1: IEEE-754 single-precision floating-point number representation.

5.3 32-bit Floating-point Multiplier Structure

The floating-point multiplier is composed of the sign bit selection hardware, exponent calculation, adjustment, and the integer multiplier for the mantissa as shown in Figure 5.2.

If a binary floating-point number x is represented as a significand s , and exponent e as in $x = s \times 2^e$ the formula

$$(s_1 \times 2^{e_1}) \cdot (s_2 \times 2^{e_2}) = (s_1 \cdot s_2) \times 2^{e_1+e_2} \quad (5.1)$$

shows that the floating point multiplication has several parts.

The first part multiplies significands using an ordinary integer multiplier. Because the floating-point numbers are stored in sign magnitude form, the multiplier has to deal only with unsigned numbers.

The second part rounds the result. If the significands are unsigned 24-bit numbers (for 32-bit floating-point multiplier), then the product can have as many as 48 bits and must be rounded to a 24-bit number.

The third part computes the new exponent. Because exponents are stored with bias, this involves subtracting the bias from the sum of the biased exponents [51].

The top level 32-bit RSFQ floating-point multiplier structure with all three parts is shown in Figure 5.2.

5.4 Sign Bit Calculation

The sign bit is either positive when the sign value is '0' or negative otherwise, and therefore a XOR gate can be used to provide the correct sign bit for the result (see Table 5.1).

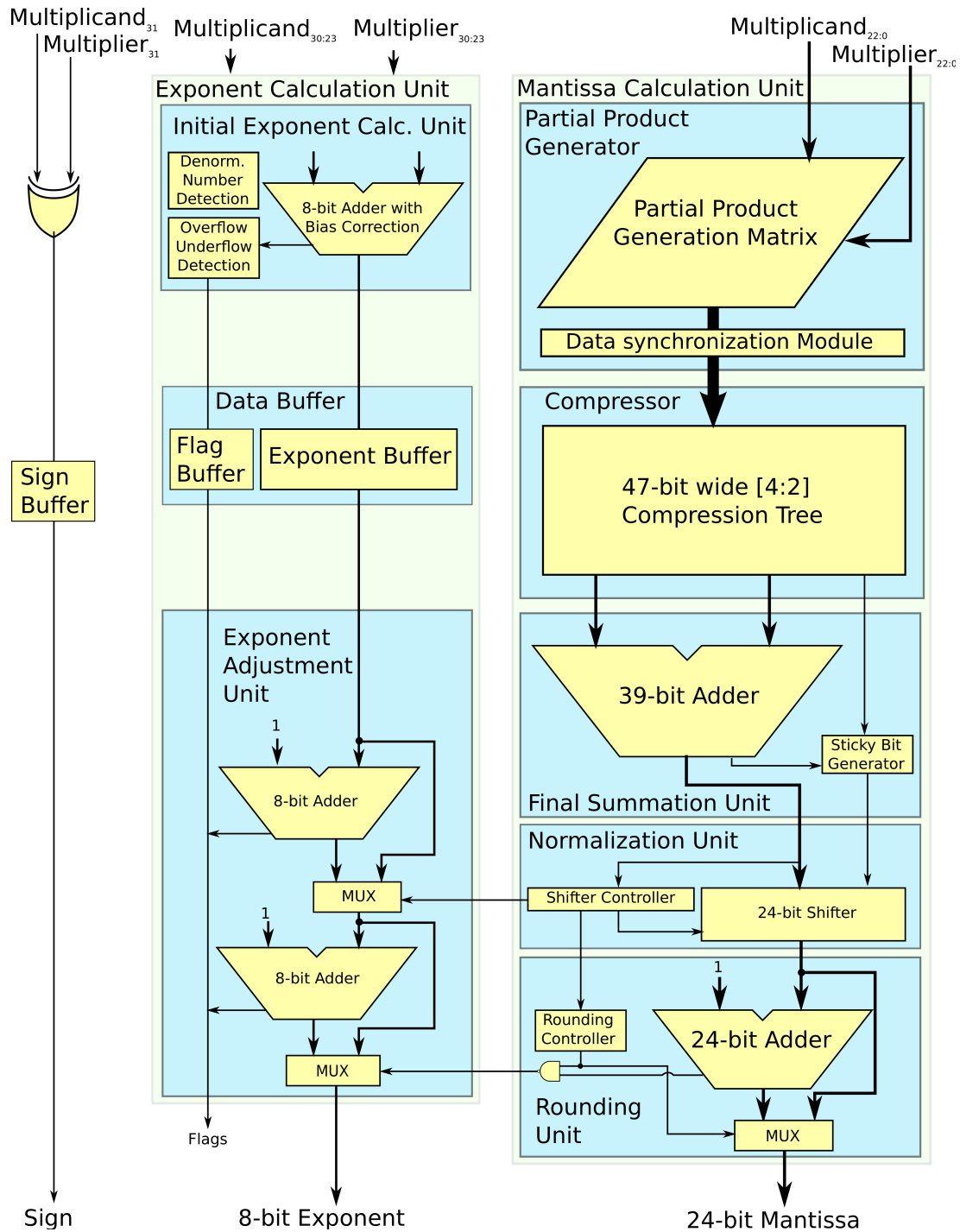


Figure 5.2: Floating-point multiplier structure. The 39-bit adder, blocks inside Normalization, Rounding, and Exponent Adjustment units are part of 32-bit floating-point adder described in [44] and are not a subject of this dissertation. These blocks were adopted and reused in either exact or modified form.

Table 5.1: Sign bit calculation.

Sign ₁	Sign ₂	New sign
+	+	+
+	-	-
+	+	+
+	-	-

5.5 Exponent Calculation Unit

The Exponent Calculation Unit as its name suggests deals with the calculation of exponents and also detects whether any input exponent value is zero. It is composed of multiple smaller blocks which have to be synchronized with the Mantissa Calculation Unit to provide the best processing rate and latency.

5.5.1 Zero Value Detection

When either of the exponents has a value of zero, this is detected and the appropriate signal is propagated through the Exponent Calculation Unit. In this case, the floating-point output is flushed to 0 and the zero flag is asserted.

5.5.2 Initial Exponent Calculation Unit

Because exponents are biased, the bias has to be subtracted from the sum of two biased exponents to get the new biased value. That is, for two numbers A and B the new exponent e can be calculated using following formula

$$e = e_A + e_B - bias. \quad (5.2)$$

This equation can be modified to use two adders as in

$$e = e_A + e_B + (-bias) \quad (5.3)$$

where $bias = 127 = 0b10000001_{2s}$.

The first adder is used to add biased exponents. The second one adds negative bias to the result from the first addition to form a new biased exponent.

Since the exponent calculation is not on the critical path, a low complexity ripple-carry adder is used for this purpose. As ripple-carry addition takes more

than one cycle, the addition had to be pipelined with the four least significant bits calculated in the first stage, and the other four bits in the second stage. Furthermore, the second part of the first addition is overlapped with the first part of the second addition as shown in Figure 5.3.

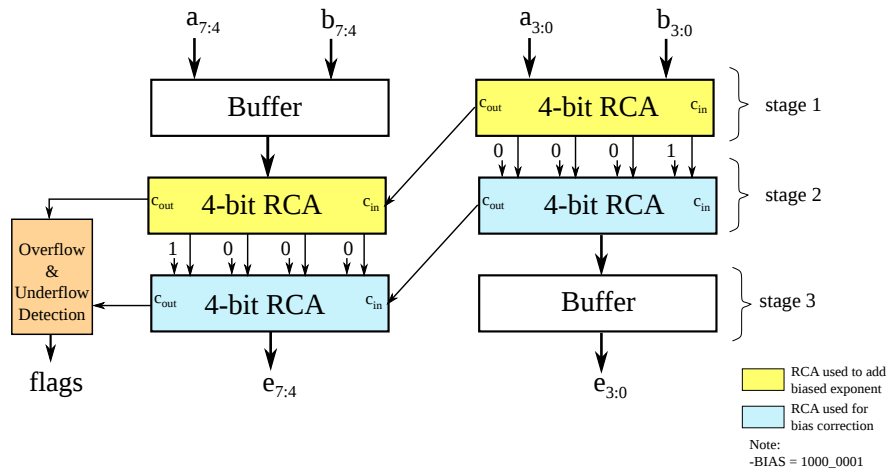


Figure 5.3: Exponent calculation with bias correction. Exponents $a_{7:0}$, $b_{7:0}$ and final exponent $e_{7:0}$ are in biased form.

A 4-bit ripple-carry adder (RCA) used for initial exponent calculation is shown in Figure 5.4.

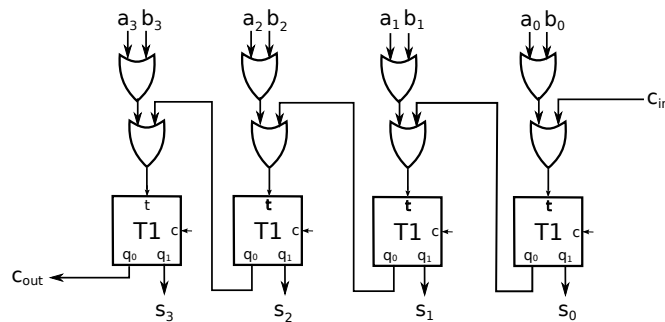


Figure 5.4: 4-bit RCA used for exponent calculation.

Overflow occurs when the resulting exponent is too large to be represented, that is when the new unbiased exponent value is 128 or larger. On the other hand, the underflow is detected when the result becomes negative. The case where the result is 0 (internal denormalized number) can be corrected if the exponent is

incremented after the normalization and rounding stage and is handled correctly in our implementation.

5.5.3 Exponent Data Buffer

Once calculated, the exponent is not used for the next 600 ps until the normalization and rounding stage. Sending the exponent to the rounding block without any buffering would corrupt the exponent value for previous multiplication. Hence, the exponent is buffered using a ten-stage data buffer.

5.5.4 Exponent Adjustment Unit

This block consist of two 8-bit adders with multiplexers. The adders are used to increment the exponent.

The first increment operation is done in parallel with normalization inside the Mantissa Calculation Unit which provides a control signal for the multiplexer. That is, it selects between the original and incremented value of the exponent based on normalization outcome.

Similarly, the second increment operation is done in parallel with the rounding step inside the Mantissa Calculation Unit to provide the lowest possible latency of this stage. The overflow in mantissa calculation causes the incremented value to be propagated to the output buffer, otherwise the non-incremented value is used.

A carry-out from each adder is used for overflow detection and is propagated to the flag register.

If the initial exponent value of 0 is incremented due to either normalization or rounding, the zero flag from initial exponent calculation block is cleared and a correct normalized/rounded result is given.

The adders and multiplexers used in this block are not a subject of this dissertation. These were designed as a part of the 32-bit floating-point adder described in [44] and were only reused here.

5.6 Mantissa Calculation Unit

The most complex floating-point multiplier blocks such as the partial product generator (33.2% JJs) and compressor (32.8% JJs) are the main focus in this section. On the other hand, the summation, normalization, and rounding stages were adopted from floating-point multiplier designed at Stony Brook at USCL [44] with slight modifications required due to rounding and normalization specific to the multiplier.

5.6.1 Partial Product Generation

Multiplying two 24-bit integers results in a 48-bit product as shown in Figure 5.5 with a total of $24^2 = 576$ partial product bits.

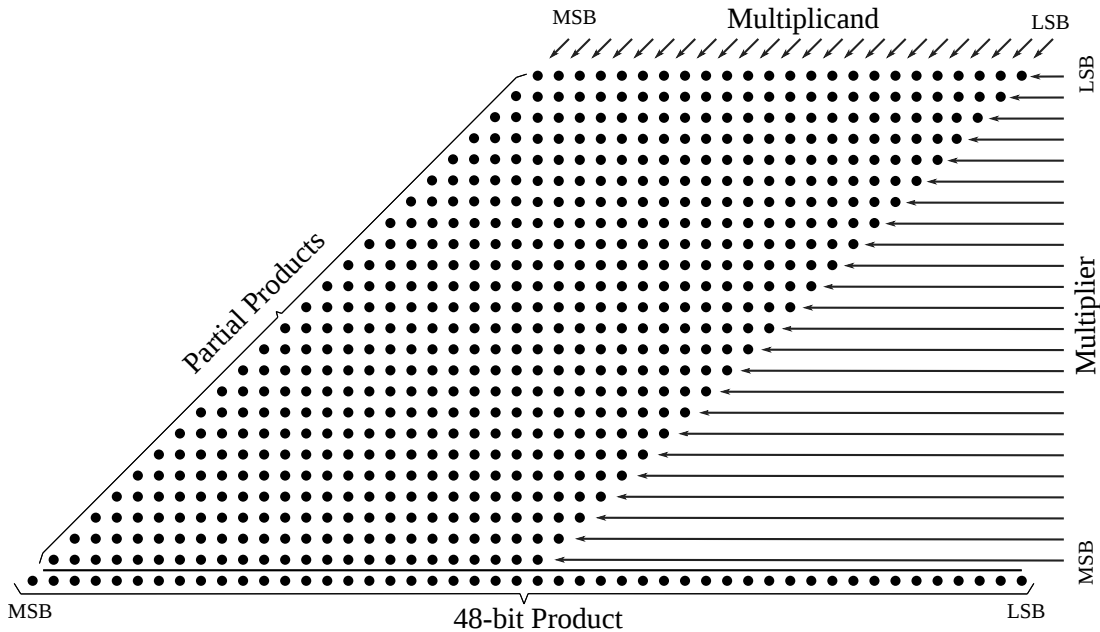


Figure 5.5: Partial product generator for a 24-bit multiplier used for mantissa calculation.

The straightforward implementation of the resulting partial product parallelogram leads to the asymmetric design for which it is difficult to provide adequate timing. Therefore, to provide a more uniform symmetric clock and data distribution we have rearranged the partial product parallelogram by moving up the bit columns on the left side toward the input as shown in Figure 5.6.

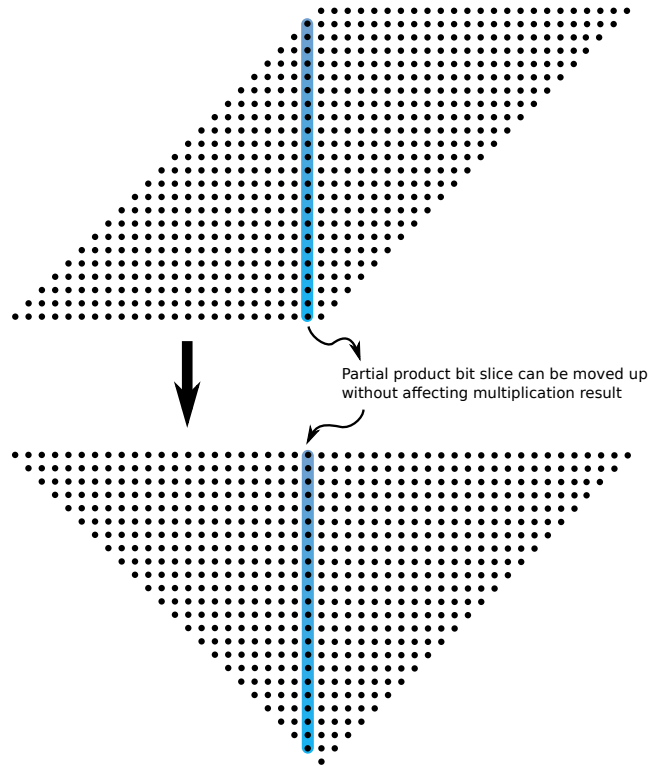
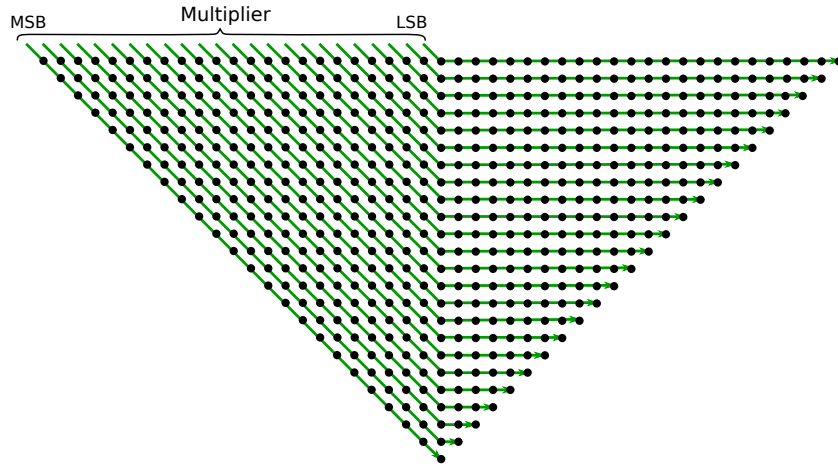


Figure 5.6: Rearranging partial product generator for symmetric data flow.

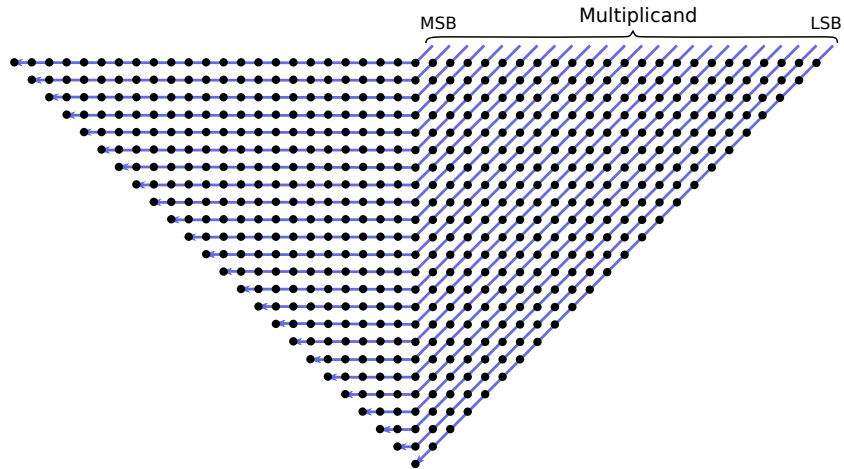
Since the arithmetic weight of each bit in every single column is preserved in this rearrangement, such modification does not impact the final product calculation result. Rearranging partial products this way provides for symmetric clock distribution. As a consequence, the physical data does not travel in straight lines, but rather makes a turn in the middle as shown in Figure 5.7. This way, the timing for multiplicand and multiplier is balanced as well.

Another advantage of this arrangement is that the number of bits per row decreases as data move down. With the number of bits in each row decreasing, the time difference between most and least significant bit in each row decreases as well. This is important because the signal fluctuations are growing as the data travel inside the product generator. These fluctuations can be better accommodated when the worst case delay between the least and most significant bit is shorter.

Once the data reaches the middle column of the partial product generator, they are distributed the same way as the clock is distributed, providing an addi-



(a) Multiplier dataflow.



(b) Multiplicand dataflow.

Figure 5.7: Dataflow inside the partial product generator.

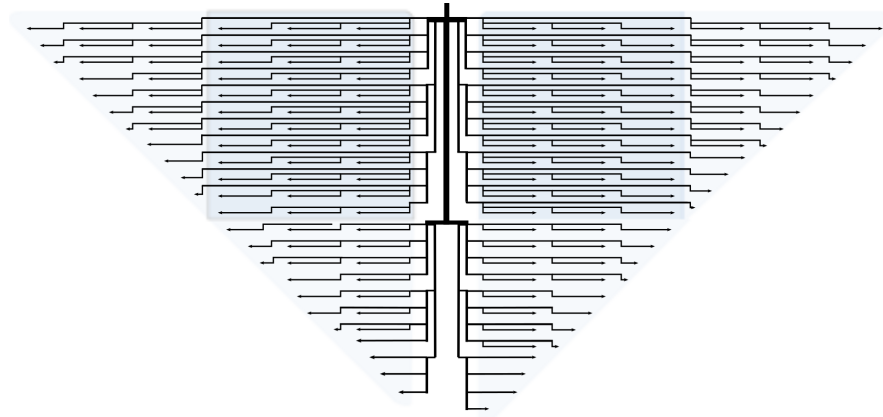
tional clock and data synchronization. Furthermore, once the layout for the right side of the circuit is completed, it can be reused for the left side.

5.6.1.1 Timing

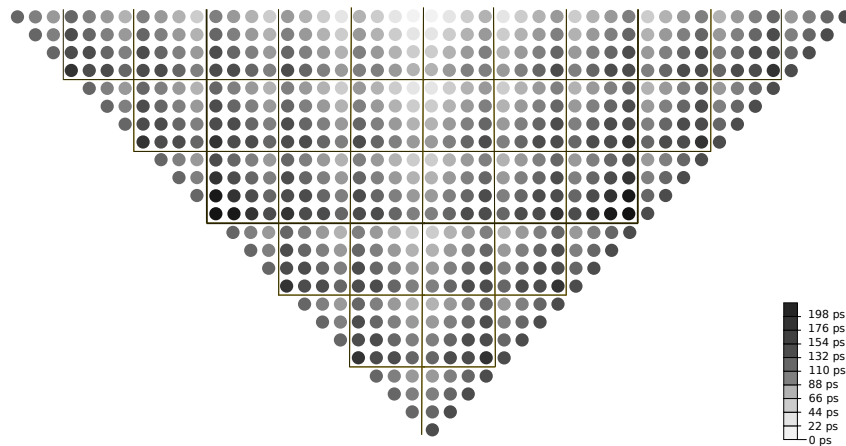
The large number of wires needed and area limitations similar to those that were described in Section 4.2.1.2 for the integer multiplier apply here as well. Therefore, the operand and clock distribution is similar to that used for the the 32-bit integer multiplier.

As a result, the operands are distributed using binary trees to each of the 4x4

group of the partial product and then in series inside each group. The partial product clock distribution and timing profile are shown in Figure 5.8.



(a) Clock distribution for partial product generator.



(b) Readout timing for each bit.

Figure 5.8: Clock distribution and timing for the partial product generator for the floating-point multiplier.

The partial product synchronization mechanism for the partial product generator is also similar to the one used in 32-bit integer multiplier. The partial product generation and synchronization for the floating-point multiplier is shown in Figure 5.10.

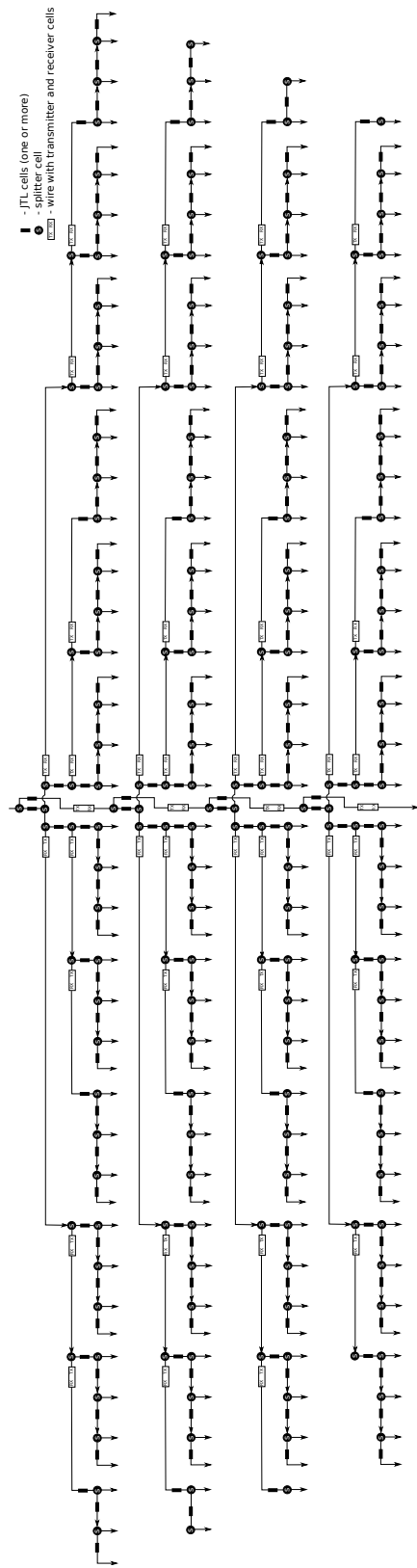


Figure 5.9: Clock distribution for top four partial products.

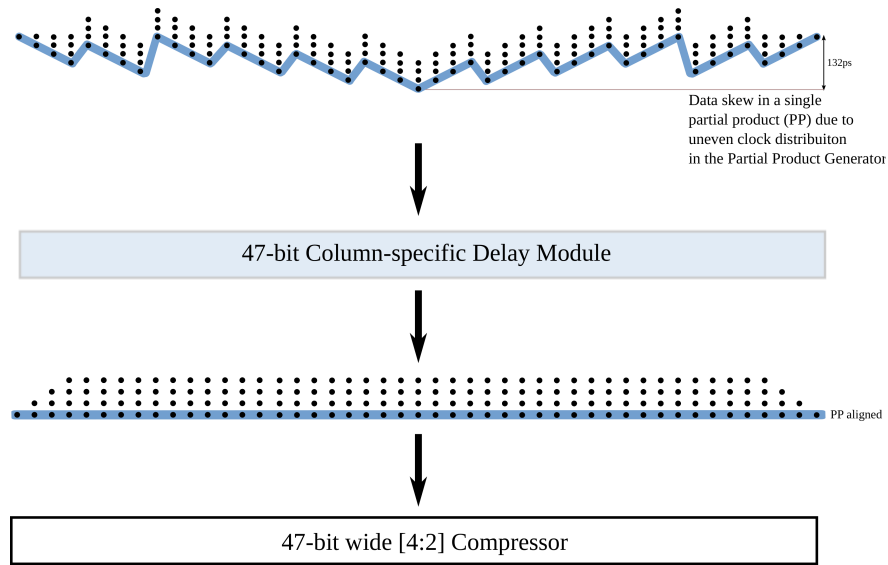


Figure 5.10: Partial product synchronization.

5.6.1.2 Pipelining

In order to provide the best possible processing rate, two pipeline stages were utilized. The pipeline stages are shown in Figure 5.11. It is similar to that of the 32-bit integer multiplier described in Chapter 4.

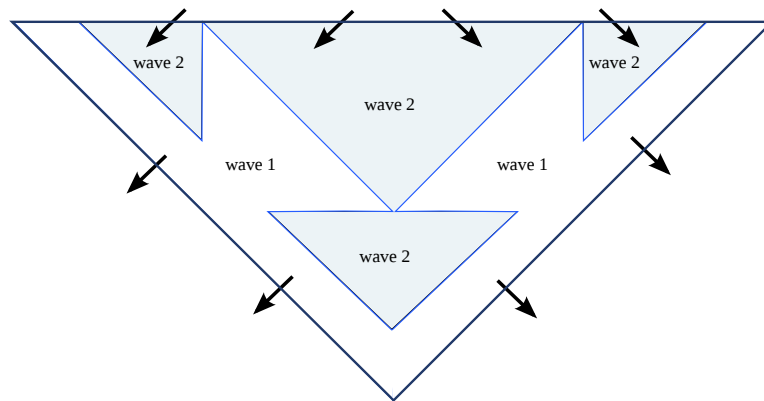


Figure 5.11: Wave-pipelining of the partial product generator for the floating-point multiplier.

5.6.2 Compression

5.6.2.1 Compressor Pipelining

Operation of each compressor is synchronized using a hybrid of wave-pipelining and asynchronous co-flow sequencing as in the 32-bit multiplier described in Section 4.2.2.2 and is only briefly described here to highlight the differences between the two multipliers.

5.6.2.2 Tree Structure

As in the 32-bit integer multiplier, each column in the N-bit wide [4:2] compressor is used to reduce up to 4-bits belonging to the same bit slice. Therefore, the product generator output can be divided into six rows, each handled by a separate N-bit wide [4:2] compressor. The resulting compression tree is shown in Figure 5.13.

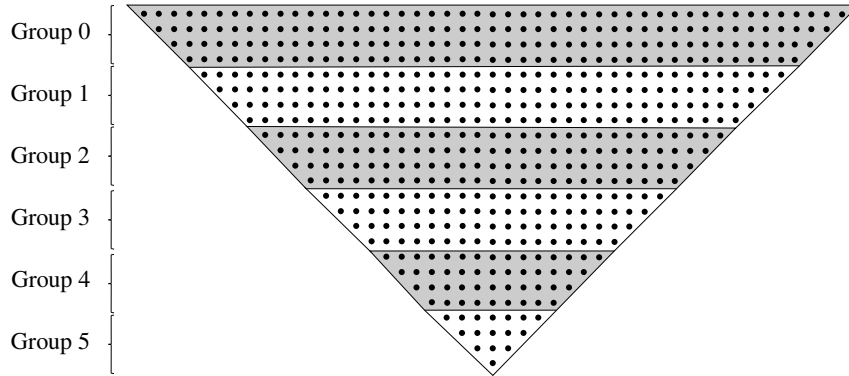


Figure 5.12: Partial products grouping for compression tree.

Since there are 47 columns and all are to be reduced, a 47-bit wide [4:2] compressor is used for group 0 of partial products, a 39-bit wide [4:2] compressor for group 1 and so on until a 7-bit wide [4:2] compressor for group 5.

The outputs from these N-bit wide [4:2] compressors are then grouped together and sent to another level of an N-bit wide [4:2] compressor as shown in Figure 5.13. The compression process continues until the final carry-sum pair is generated. This carry-sum pair is then completely reduced into a single product using a 39-bit adder.

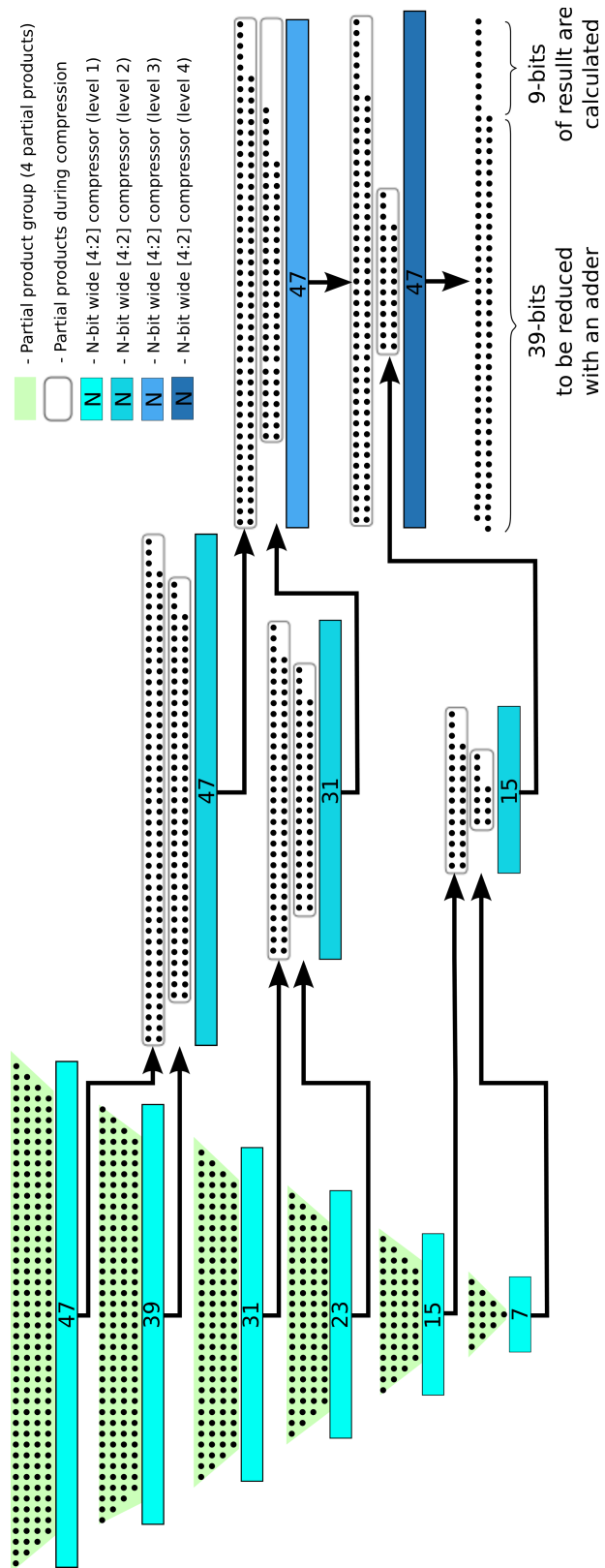


Figure 5.13: Compressor tree for the 32-bit RSFQ floating-point multiplier.

5.6.3 Final Summation Unit

As mentioned in Section 2.7 the CPA adder design is beyond the scope of this dissertation. Therefore, a high performance RSFQ 39-bit sparse tree adder developed by M. Dorojevets and C. Ayala [44] is used in this design.

5.6.4 Sticky Bit Calculation

The initial sticky bit is calculated during the compression and summation. It is then appropriately used in the rounding stage if necessary according to the rounding algorithm as described in Section 5.6.5.

5.6.5 Normalization and Rounding Units

Although, the Normalization and Rounding units were not designed specifically for this study, and these are taken from 32-bit floating-point adder described in [44], a slight modification had to be made to these blocks before they could be used in floating-point multiplier design. Hence, a short description of normalization and rounding steps is given below.

There is straightforward method of handling normalization and rounding in 32-bit floating-point multipliers. After both significands A and B are multiplied, a 48-bit product is obtained and stored in registers (P,A) forming a 48-bit register with most significant bits in register P . Let s represent the sticky bit, g (for guard) the most-significant bit of A , and r (for round) the second most-significant bit of A . There are two cases:

- case 1: The high-order bit of P is 0. Shift P left by 1 bit, shifting in the g bit from A . Shifting the rest of A is not necessary.
- case 2: The high-order bit of P is 1. Set $s := s \wedge r$ and $r := g$, and add 1 to the exponent.

Now if $r = 0$, P is correctly rounded product. If $r = 1$ and $s = 1$ then $P+1$ is the product. If $r = 1$, we are halfway and need to round up according to the least significant bit of P [51]. The precise rounding modes are listed in Table 5.2.

Table 5.2: Rules for implementation the IEEE rounding modes.

Rounding Mode	Sign ≥ 0	Sign < 0
$-\infty$		<i>if $r \vee s \Rightarrow +1$ to P</i>
$+\infty$	<i>if $r \vee s \Rightarrow +1$ to P</i>	
0		
Nearest	<i>if $r \wedge (p_0 \vee s) \Rightarrow +1$ to P</i>	<i>if $r \wedge (p_0 \vee s) \Rightarrow +1$ to P</i>

5.7 Floating-point Multiplier Design Summary

The cell-level design of the RSFQ 32-bit floating-point multiplier was verified using VHDL simulation with over 100,000 random operands at the processing rate of 11.1 GHz with a total latency of 1.772 ns.

The bias current distribution and complexity for each stage of the multiplier is given in Table 5.6.

Taking into account that the final design has around 89K JJs, we consider our design goals achieved for this 32-bit RSFQ single-precision floating-point multiplier.

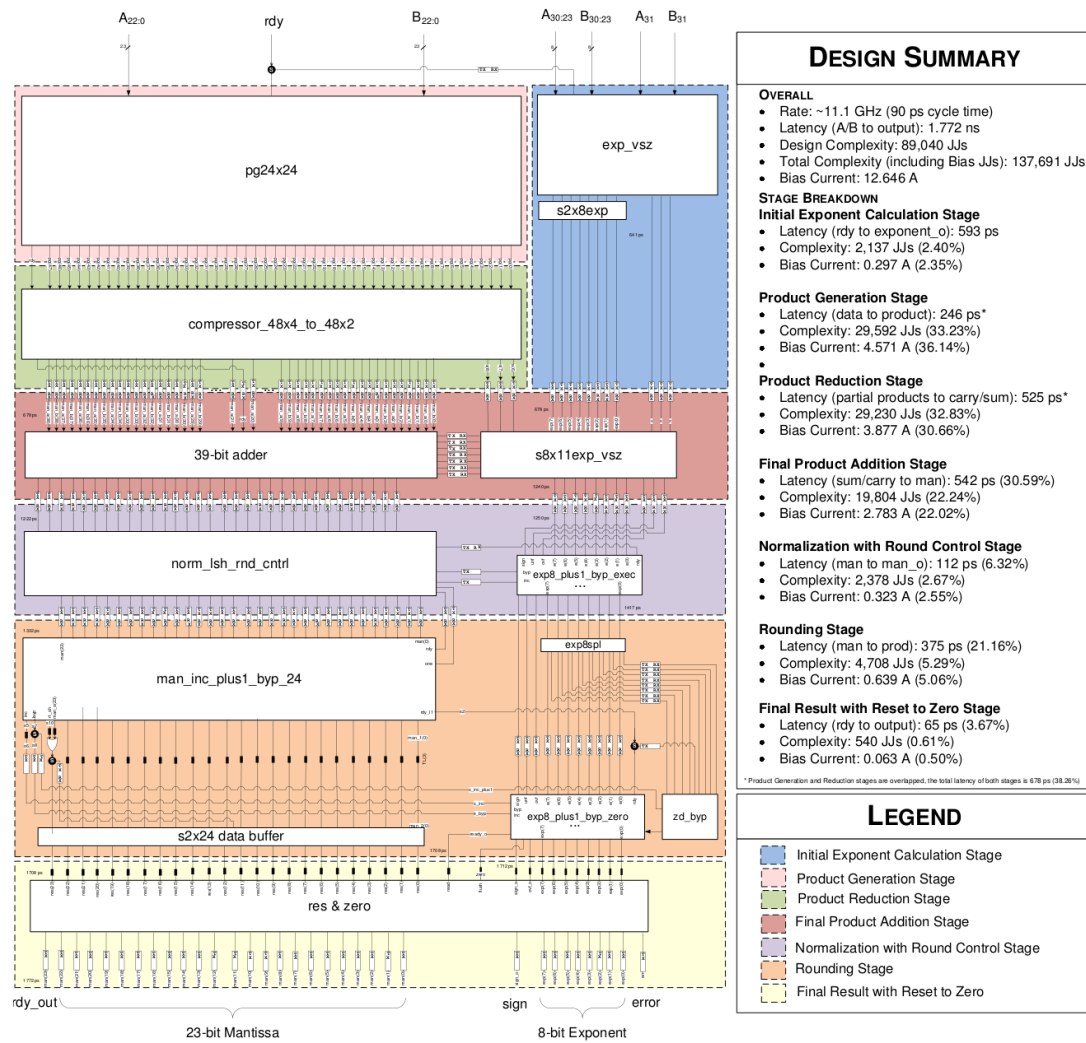


Figure 5.14: 11.1 GHz RSFQ 32-bit single-precision floating-point multiplier.

Table 5.3: RSFQ floating-point multiplier characteristics at $T = 4.2$ K.

Maximum frequency (GHz)	11.1
Total latency (ps)	1,772
Total complexity (JJs)	89,040
Total bias current (A)	12.646

Table 5.4: Latency breakdown for the RSFQ floating-point multiplier.

Stage	% Latency	Latency (ps)
Partial Product Generation	13.88	246
Partial Product Compression	29.63	525
Final Summation (STA)	30.59	542
Initial Exponent Calculation	N/A	641
Normalization	6.32	112
Rounding	21.16	375
Reset to Zero & Readout	3.67	65
Overlapped ¹	(5.25)	(93)
Total critical path latency:		1,772

Table 5.5: JJ and bias current breakdown per logic and interconnect for the RSFQ floating-point multiplier.

Category	JJ Count	% JJs	Total I_{bias}	% I_{bias}
Logic	31,369	35.23	2,612.73	20.72
Splitters	6,081	6.83	1,611.47	12.78
PTL	15,612	17.53	2,086.35	16.55
Other ²	35,978	40.41	6,296.15	49.94
Total	89,040	100.00	12,606.70	100.00

¹Partial product generation and compression steps are partially overlapped.

²Includes JJs used for cell separation, delay lines and clocking.

Table 5.6: JJ distribution per stage for the RSFQ floating-point multiplier.

Stage	Complexity (JJs)	I_{bias} (A)	I_{bias} (%)
Partial Product Generation	29,592	4.5705	2.35
Partial Product Compression	29,230	3.8771	36.14
Final Summation (STA)	19,804	2.7843	30.66
Initial Exponent Calculation	2,137	0.2972	22.02
Normalization	2,378	0.3229	2.55
Rounding	4,708	0.6394	5.06
Reset to Zero & Readout	540	0.0634	0.50
Other ³	651	0.0916	0.72
Total	89,040	12.6465	100.00

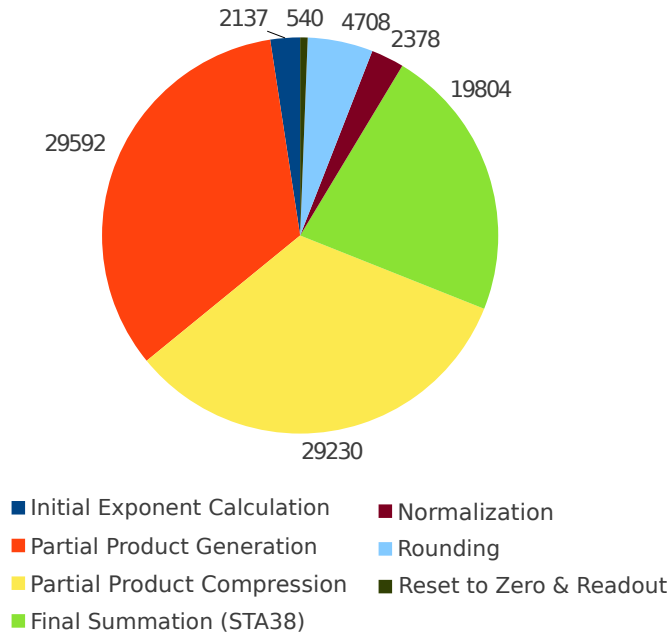


Figure 5.15: Complexity breakdown per stage for the RSFQ floating-point multiplier.

³Intermediate buffers and interconnects.

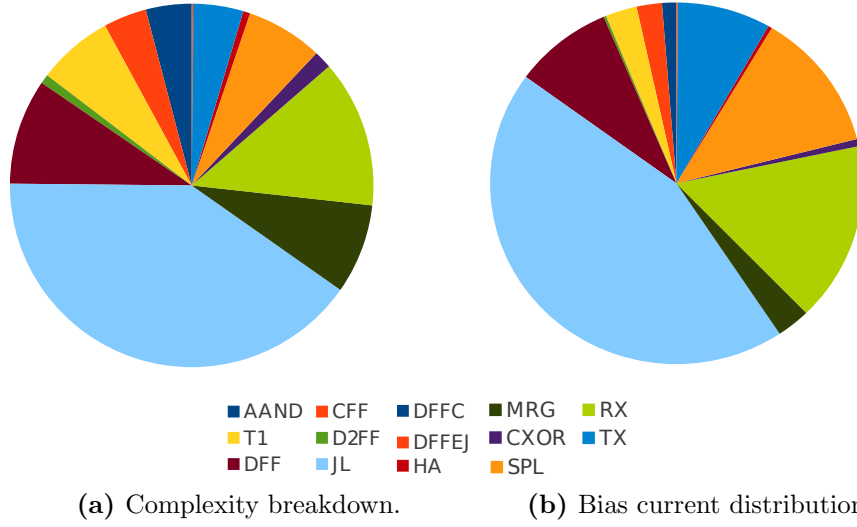


Figure 5.16: JJ and current breakdown per cell for the RSFQ floating-point multiplier.

Table 5.7: Cell use breakdown for the RSFQ floating-point multiplier.

Cell	Cell Count	JJs/Cell	Total JJs/Cell	% Cell	% JJs
AAND	604	6	3,624	1.09	4.07
CFF	214	16	3,424	0.39	3.85
T1	665	9	5,985	1.20	6.72
D2FF	104	7	728	0.19	0.82
DFF	2091	4	8,364	3.79	9.39
JL	35,978	1	35,978	65.19	40.41
MRG	1,423	5	7,115	2.58	7.99
RX	3,895	3	11,685	7.06	13.12
CXOR	158	9	1,422	0.29	1.60
SPL	6,081	1	6,081	11.02	6.83
HA	24	23	552	0.04	0.62
TX	3,927	1	3,927	7.12	4.41
DFFC	6	10	60	0.01	0.07
DFFEJ	19	5	95	0.03	0.11
Total	55,189	N/A	89,040	100.00	100.00

Table 5.8: Bias current distribution for the RSFQ floating-point multiplier.

Cell	Current/Cell (mA)	Total Current/Cell (mA)	Current (%)
AAND	0.85	513.40	4.06
CFF	1.14	244.75	1.94
T1	0.50	332.50	2.63
D2FF	0.21	22.10	0.17
DFF	0.35	731.85	5.79
JL	0.18	6,296.15	49.79
MRG	0.51	723.88	5.72
RX	0.31	1,202.78	9.51
CXOR	0.18	27.65	0.22
SPL	0.27	1,611.47	12.74
HA	1.84	44.25	0.35
TX	0.23	883.58	6.99
DFFC	0.91	5.48	0.04
DFFEJ	0.35	6.65	0.05
Total	N/A	12,646	100.00

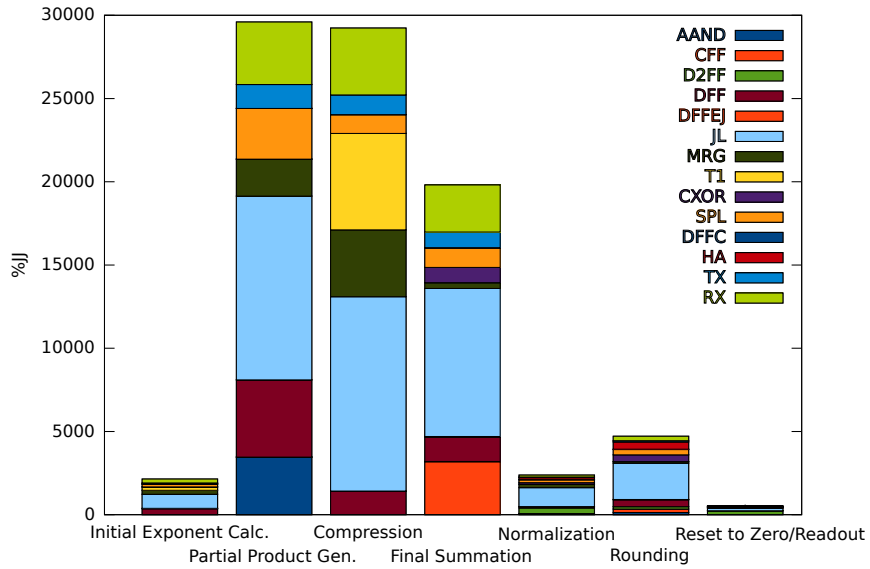


Figure 5.17: Cell breakdown per stage for the floating-point multiplier.

Physical Chip Design and Demonstration of 20 GHz 8-bit Integer Multiplier

Contents

6.1	Microarchitecture	86
6.2	Complexity and Power	86
6.3	Performance	88
6.4	Logical and Physical Layout Design	89
6.5	Experimental Test Results	92

In this chapter we will discuss the microarchitecture, design, and testing of the 20 GHz 8-bit (by modulo 256) parallel superconductor RSFQ multiplier which was implemented and fabricated using the ISTECH 1.0 μm 10 kA/cm² fabrication technology [52].

The goal of implementing this multiplier was to test and validate our compression tree based on [4:2] compressor cells. In fact, the 8-bit multiplier is similar to the 32-bit integer multiplier, but with a sparse tree adder used for the final summation replaced with a ripple-carry adder.

The complete logical and physical multiplier chip design has been done in this study using the CONNECT cell library and SFQ CAD tools developed at Nagoya University and Yokohama National University.

The multiplier core (without SFQ-to-DC and DC-to-SFQ converters) has 5,901 Josephson junctions occupying an area of 3.0 mm².

This multiplier was designed with the maximum operation frequency of 20 GHz, and the latency of 447 ps attainable with the bias voltage of 2.5 mV. Despite

some challenges related to process parameter variations and flux trapping, the multiplier chip was fabricated and successfully tested at low frequency for the vast majority of test vectors. The testing was carried out by the SBU team with the assistance of colleagues from National Yokohama University in Japan during February 2012.

6.1 Microarchitecture

Our 8-bit parallel integer multiplier with 8-bit output utilizes a high performance carry-save reduction tree similar to the one mentioned for both 32-bit multipliers, and a 3-bit ripple-carry adder is used for final summation. The block diagram for this multiplier is shown in Figure 6.1.

The partial product generation circuit is first divided into two main groups with four partial products per group. The first group is made of four top partial products rows and the second group of the four bottom partial products. Each group is processed in parallel to speed up the partial product generation process. The modules used to build the partial product generator are shown in Figures 6.1(b)-6.1(e).

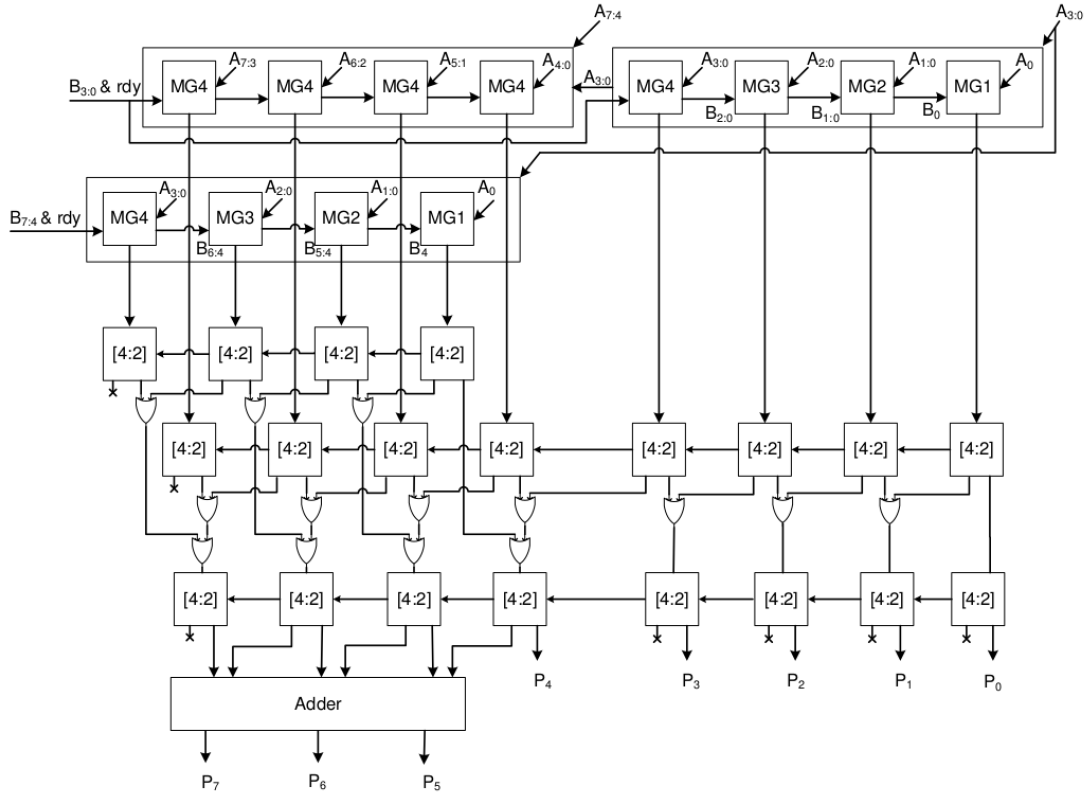
In each group, the partial products are generated sequentially where the top-most partial product corresponding to the least significant bit in the multiplier is generated first, then second, third, and finally the fourth one.

The compression tree is built with [4:2] compressors in a similar way the compression trees for 32-bit multipliers were built. The compression tree is shown in Figure 6.2.

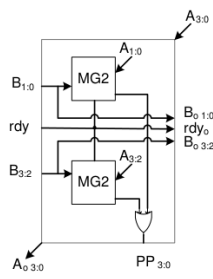
Using the two level [4:2] compression tree, the five least significant bits are reduced completely to one bit per column and need not to be reduced any further, so the final adder size is significantly reduced. The data in the remaining three columns are reduced to form a final product using a 3-bit ripple carry adder shown in Figure 6.1(g).

6.2 Complexity and Power

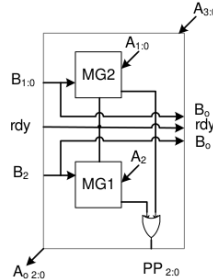
The 8-bit multiplier complexity, power dissipation, and area are summarized in Table 6.1.



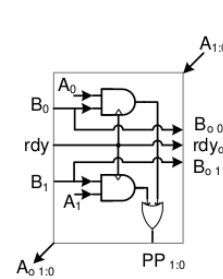
(a) Top level block diagram



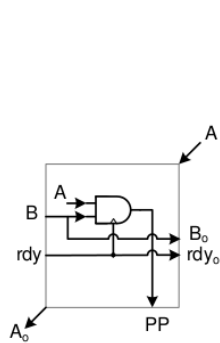
(b) MG4



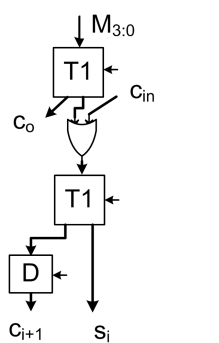
(c) MG3



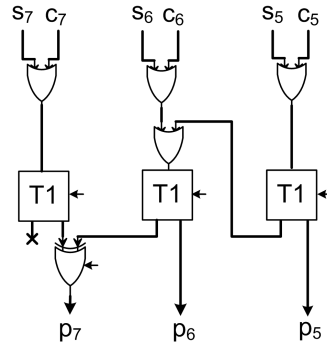
(d) MG2



(e) MG1



(f) [4:2]



(g) 3-bit Adder

Figure 6.1: 8-bit RSFQ integer multiplier block diagram.

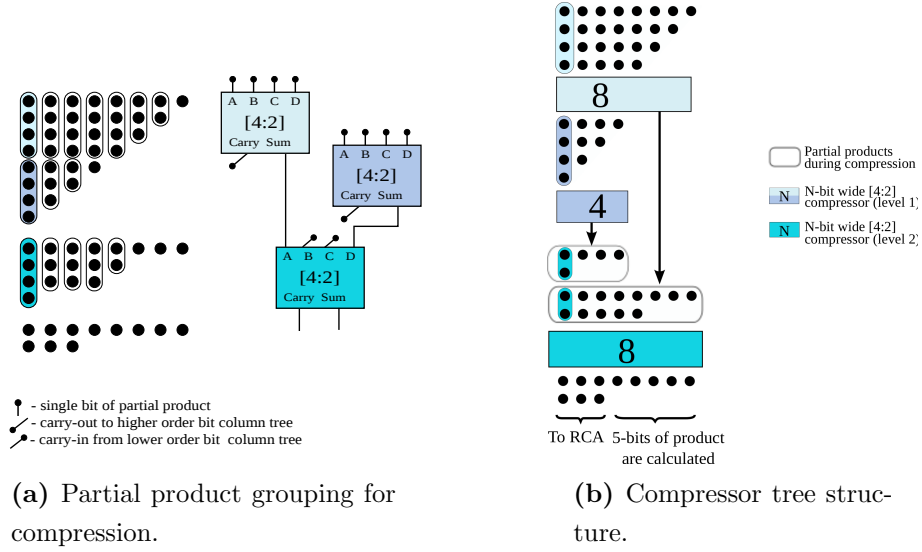


Figure 6.2: Compressor tree structure for the 8-bit RSFQ integer multiplier.

Table 6.1: Complexity, bias current, and area distribution for the 8-bit RSFQ integer multiplier.

Component	JJ Count	I_{bias}	Area (mm^2)
DC-to-SFQ converters	85	6.906	0.0816
Main circuit	5,901	675.717	3.0048
SFQ-to-DC converters	72	11.849	0.0384
Total	6,058	694.472	3.1248

6.3 Performance

The simulation for 8-bit multiplier showed correct operation at 20 GHz rate with $\pm 10\%$ bias margins. The bias simulation results are shown in Figure 6.3.

Table 6.2: 8-bit RSFQ integer multiplier chip summary.

Maximum frequency (GHz)	21.5
Total latency (ps)	447
Total complexity (JJs)	6,058
Total bias current (mA)	694.5
Total area (mm^2)	3.1

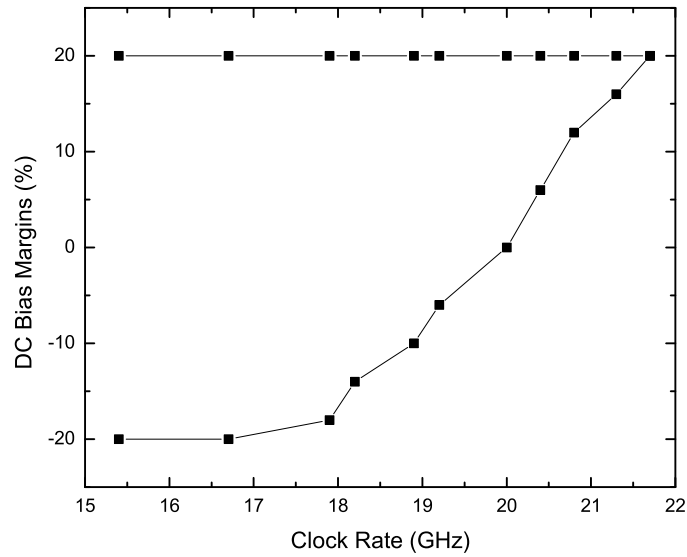


Figure 6.3: Operating margins of the 8-bit RSFQ integer multiplier.

6.4 Logical and Physical Layout Design

The layout for the 8-bit multiplier was done with Cadence using the latest CONNECT cell library. The multiplier layout with the main parts highlighted is shown in Figure 6.4. The multiplier was fabricated with the ISTECH 1.0 μm 10 kA/cm^2 process and the micro-photograph of a real chip is shown in Figure 6.5.

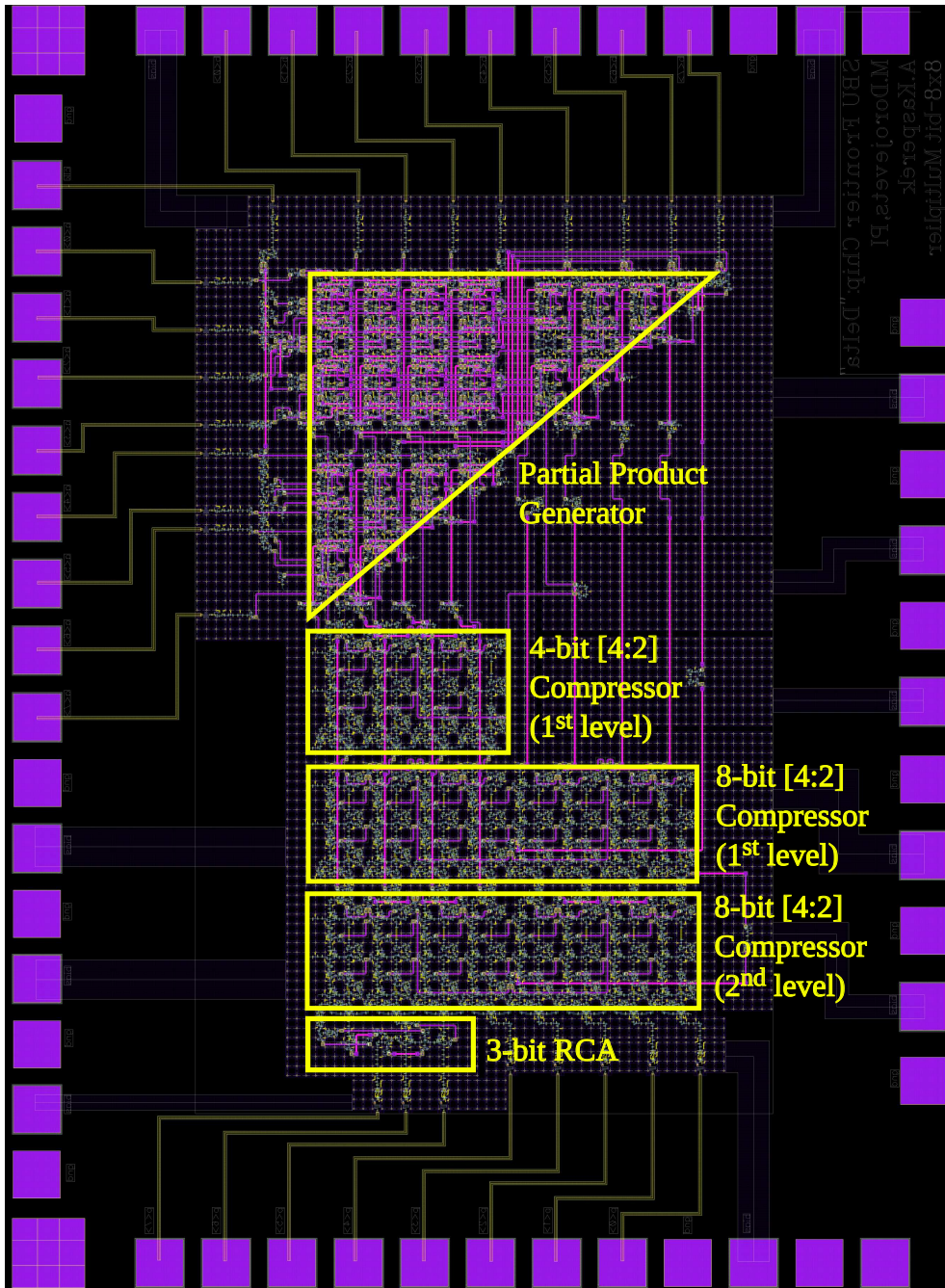


Figure 6.4: The 8-bit RSFQ integer multiplier layout implemented with the CONNECT cell library.

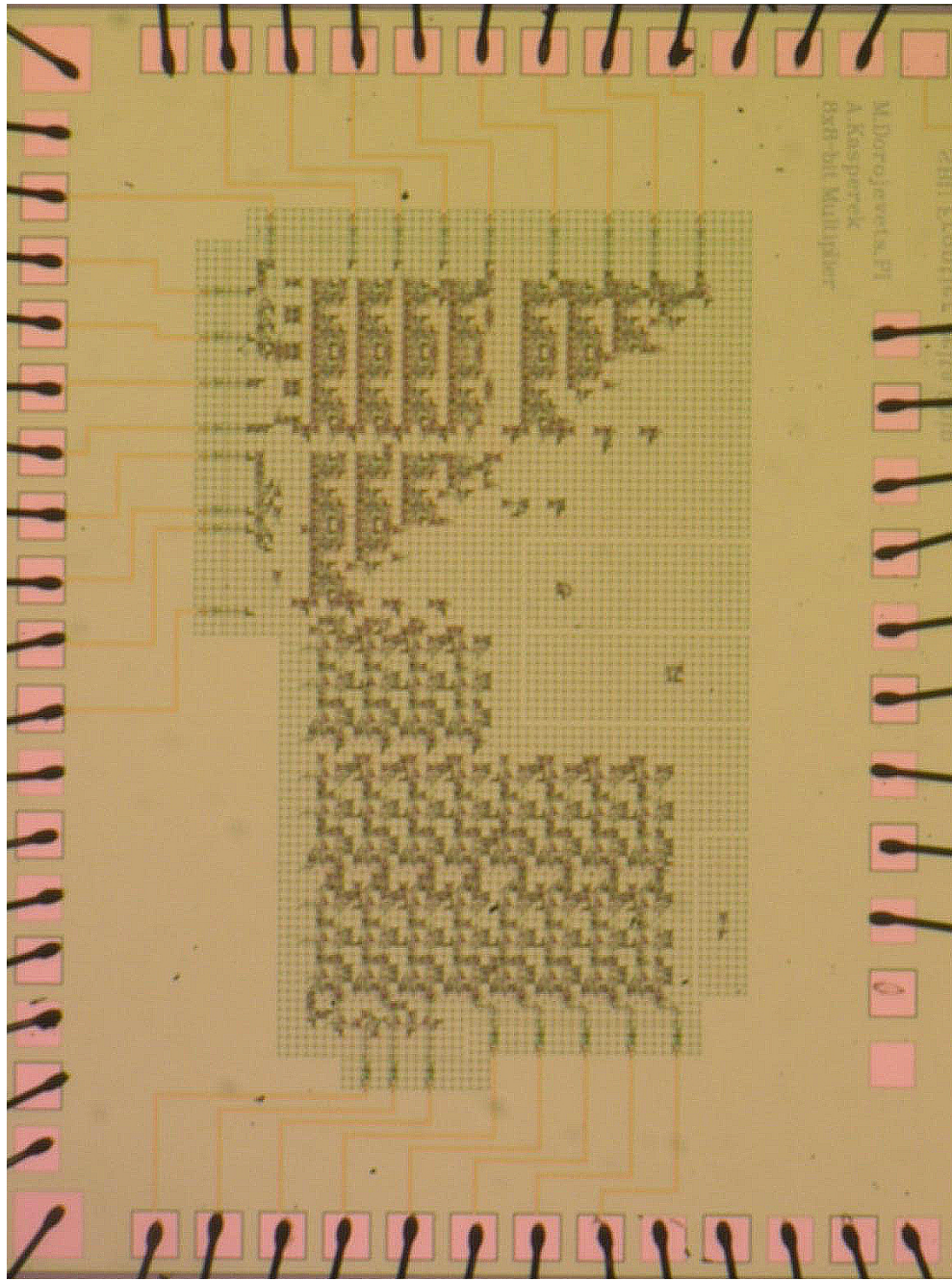


Figure 6.5: Micro-photograph of the 8-bit RSFQ integer multiplier chip fabricated with the ISTECH 1.0 μm 10 kA/cm^2 fabrication process.

6.5 Experimental Test Results

The 8-bit multiplier was tested at Yokohama National University in Japan in Yoshi Lab with a very high success rate in February 2012. The equipment used during the low frequency testing is listed in Table 6.3.

Table 6.3: Testing equipment for low frequency testing.

Equipment name	Model number	Purpose
Power Supply	PMR18-2.5DU	To supply bias current to the chip.
Step Attenuator	Tamagave Electronics VBA-641A	To convert the input signals from data generator to appropriate levels for superconductor circuit.
Data generator	Tektronix DG2020A	To generate input test pattern.
Variable Output Pod	Tektronix P3420	Input signal conditioning.
Differential Amplifier	Stanford Research Systems SR560	To amplify output data signals.
Bias filters	custom made	To provide filtered bias current.
Cryostat	Cryofab Inc. CMSH 88	To cool down the RSFQ chip to $T=4.2$ K.
Probe	custom made	To provide power and communication link between test equipment and the chip.

During testing, the chip is mounted inside a chip-holder designed to withstand cryogenic temperatures and covered with a magnetic shield. Then, the chip is pre-cooled to $T = 77$ K using liquid Nitrogen to avoid rapid temperature fluctuations inside the cryostat where lower temperatures are present. Next, wire connections to the chip are checked for continuity to make sure that all probe pads are connected to the pad frame. After this step, the probe is immersed in liquid helium to provide a low cryogenic temperature of $T = 4.2$ K sufficient for Niobium [53] to reach its superconductive stage. Once the chip temperature stabilizes, the required amount of bias current is applied and adjusted to reach stable operating point when needed.

At this point, the logical testing starts by applying the walking '1's pattern for multiplier while keeping each bit in other operand at logical '1' level. After this

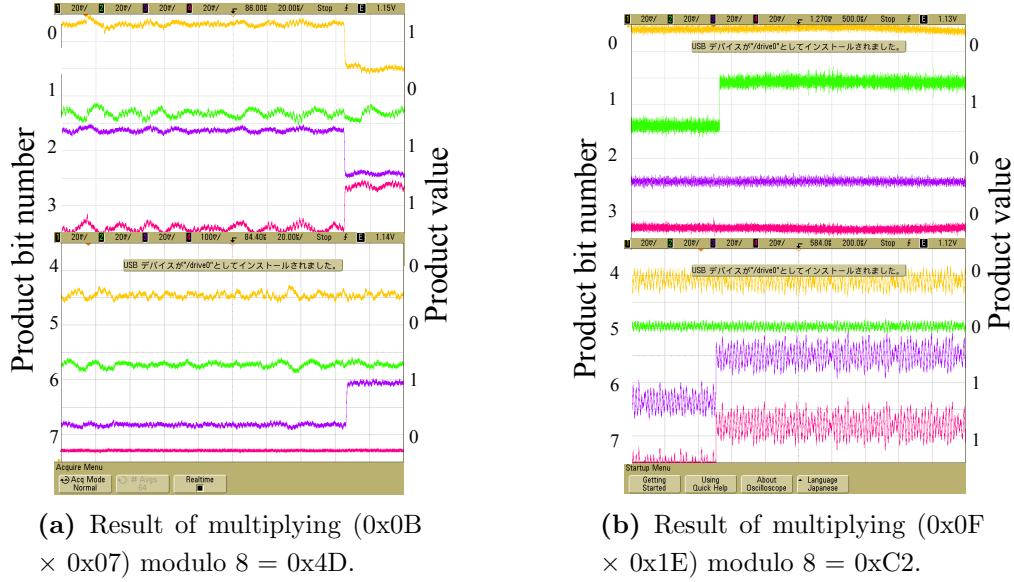


Figure 6.6: Oscilloscope waveforms for two test patterns¹.

test pattern is successfully tested for all vectors, a more complex testing process is performed. First, the operations that require T1 cell to work at 40 GHz are analyzed. This is followed by testing the T1 cell based compression at 80 GHz. A significant test pattern exercising all the output data bits is shown in Figure 6.6. The multiplier chip worked with voltage bias margins of $\pm 5\%$. We had incorrect results for some test vectors most likely due to flux trapping or/and significant variations of fabrication process parameters. Although not all test cases were successful, the vast majority of test vectors produced correct results.

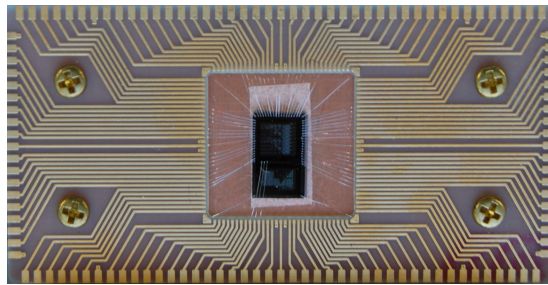


Figure 6.7: 8-bit RSFQ integer multiplier chip bonded to a chip holder.

¹Presence of a rising or falling edge represents a SFQ pulse and logical 1.



Figure 6.8: Pre-cooling of the 8-bit RSFQ multiplier chip in liquid nitrogen.

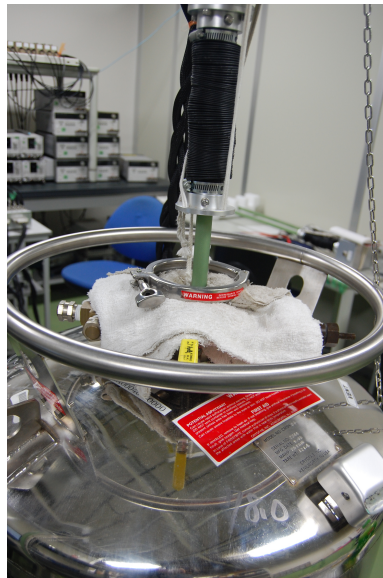


Figure 6.9: Testing probe inside a cryostat during testing.

Conclusions

The main focus of this research was to push current superconductor technology to the limits of what can be reached in terms of design complexity and processing rates.

Our objective of designing and evaluating an ultra-fast and energy-efficient 32-bit integer and single precision floating-point multipliers using superconductor technology has been achieved with the resulting designs operating at ultra-high performance with very low power consumption.

Various sequencing techniques such as synchronous pipelining, asynchronous co-flow, and wave-pipelining suitable for ultra-high speed superconductor RSFQ circuits were used to design both multipliers. Short execution time and low complexity was achieved using our novel logarithmic compression tree.

A 32-bit integer multiplier was designed, modeled in VHDL, elaborated, and successfully tested to work with the processing rate of 11.4 GHz with a total latency of 1.41 ns.

The single-precision floating-point multiplier was designed, modeled in VHDL, elaborated, and successfully tested to work with the processing rate of 11.1 GHz with a total latency of 1.78 ns.

Finally, an 8-bit (by modulo 256) parallel carry-save RSFQ multiplier was implemented using the ISTECH 1.0 μm 10 kA/cm² fabrication technology and tested with a very high success rate.

The fabricated 8-bit design multiplier worked with voltage bias margins of $\pm 5\%$ and the vast majority of test vectors produced correct results.

It will be worth-while to analyze the use of Booth encoding to reduce the number of partial products generated, especially for a single-precision floating-point multiplier as it will produce a more balanced compression tree structure.

In the future, zero-static-power superconductor logics should be looked upon as they can provide even better energy efficiency.

Bibliography

- [1] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2009. (Cited on page 1.)
- [2] *Superconducting Technology Assessment (STA)*. National Security Agency - Office of Corporate Assessments, Aug. 2005. [Online]. Available: <http://www.nitrd.gov/pubs/nsa/sta.pdf> (Cited on pages 2, 3, 5, 6 and 9.)
- [3] E. Strohmaier, “Japan reclaims top ranking on latest top500 list of worlds supercomputers,” Press Release, June 2011. [Online]. Available: <http://top500.org/lists/2011/06/press-release> (Cited on page 2.)
- [4] M. White and Y. Chen, “Scaled cmos technology reliability users guide,” NASA, Tech. Rep. WBS: 939904.01.11.10, November 2010. (Cited on page 2.)
- [5] O. A. Mukhanov, “Energy-Efficient Single Flux Quantum Technology,” *Applied Superconductivity, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011. (Cited on pages 2, 6 and 7.)
- [6] J. Evetts, *Concise Encyclopedia of Magnetic & Superconducting Materials*, ser. Advances in Materials Science and Engineering. Pergamon Press, 1992. [Online]. Available: <http://books.google.com/books?id=YGlhQgAACAAJ> (Cited on page 2.)
- [7] B. Josephson, “Possible new effects in superconductor tunneling,” *Physics Letters*, pp. 251–253, 1962. (Cited on page 3.)
- [8] K. Likharev and V. Semenov, “RSFQ logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems,” *Applied Superconductivity, IEEE Transactions on*, vol. 1, no. 1, pp. 3–28, Mar. 1991. (Cited on pages 3, 5, 6 and 34.)
- [9] M. Dorojevets, “An 8-bit flux-1 rsfq microprocessor built in 1.75- μ m technology,” *Physica C: Superconductivity*, vol. 378-381, no. Part 2, pp. 1446–1453, 2002. [Online]. Available: <http://www.sciencedirect.com/science/>

- article/B6TVJ-46RKYPM-2D/2/d3138f9a5126a86fa95f1ef97dfc3faf (Cited on pages 3, 7 and 8.)
- [10] A. Fujimaki, M. Tanaka, T. Yamada, Y. Yamanashi, H. Park, and N. Yoshikawa, “Bit-Serial Single Flux Quantum Microprocessor CORE,” *IEICE Transactions on Electronics*, vol. 91, pp. 342–349, 2010. (Cited on pages 3, 7 and 9.)
- [11] *Scientific American*, 1997. [Online]. Available: <http://www.scientificamerican.com/article.cfm?id=what-are-josephson-juncti> (Cited on page 4.)
- [12] J.-C. V. Emanuele Baggetta, Michel Maignan, “Development of fast nbn rsfq logic gates in sigmadelta converters for space telecommunications,” CEA-Grenoble, Laboratoire de Cryo-Physique and Alcatel SPACE, CEA-Grenoble, Laboratoire de Cryo-Physique, 17 rue des Martyrs, 38054 GRENOBLE Cedex-9, LCP, Tech. Rep., 2005. (Cited on page 4.)
- [13] D. E. Kirichenko, S. Sarwana, and A. F. Kirichenko, “Zero static power dissipation biasing of rsfq circuits,” *IEEE Trans. Appl. Supercond.*, no. 99, 2011, early Access. (Cited on pages 6 and 7.)
- [14] D. K. Brock, “Rsfq technology: Circuits and systems.” HYPRES INC., 175 Clearbrook Road, Elmsford, NY 10523, Tech. Rep. (Cited on pages 7 and 8.)
- [15] M. J. W. Rodwell, *High-speed integrated circuit technology: towards 100 GHz logic.*, M. J. W. Rodwell, Ed. World Scientific Publishing Co. Pte. Ltd., 2001. (Cited on page 7.)
- [16] P. Bunyk, M. Leung, J. Spargo, and M. Dorojevets, “Flux-1 RSFQ microprocessor: physical design and test results,” *Applied Superconductivity, IEEE Transactions on*, vol. 13, no. 2, pp. 433 – 436, Jun. 2003. (Cited on page 9.)
- [17] M. Dorojevets, D. Strukov, A. Silver, A. Kleinsasser, F. Bedard, P. Bunyk, Q. Herr, G. Kerber, and L. Abelson, in *Future Trends in Microelectronics: The Nano, the Giga, the Ultra, and the Bio*, S. Luryi, J. M. Xu, and A. Zaslavsky, Eds., ch. On the Road Towards Superconductor Computers: Twenty Years Later. (Cited on page 9.)

- [18] M. Dorojevets, C. L. Ayala, and A. K. Kasperek, “Data-Flow Microarchitecture for Wide Datapath RSFQ Processors: Design Study,” *Applied Superconductivity, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2010. (Cited on pages 10, 30, 33 and 35.)
- [19] T. Filippov, M. Dorojevets, A. Sahu, A. Kirichenko, C. Ayala, and O. Mukhanov, “8-bit asynchronous wave-pipelined rsfq arithmetic-logic unit,” *IEEE Trans. Appl. Supercond.*, vol. 21, no. 3, pp. 847–851, 2011. (Cited on pages 10 and 33.)
- [20] T. Kirichenko, A. and Filippov, A. Sahu, . Mukhanov, M. Dorojevets, and A. Kasperek, “Demonstration of rsfq 8-bit multi-port register file,” 2012, submitted. (Cited on page 10.)
- [21] S. F. Oberman and M. J. Flynn, “Design issues in division and other floating-point operations,” *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 154–161, 1997. (Cited on page 12.)
- [22] M. J. Flynn and S. F. Oberman, *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001. (Cited on pages 13 and 18.)
- [23] J. G. Earle, “Latched carry-save adder,” vol. 7, no. 10, pp. 909–910, Mar. 1965. (Cited on pages 15 and 20.)
- [24] O. Macsorley, “High-speed arithmetic in binary computers,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, jan. 1961. (Cited on page 16.)
- [25] S. Vassiliadis, E. Schwarz, and B. Sung, “Hard-wired multipliers with encoded partial products,” *Computers, IEEE Transactions on*, vol. 40, no. 11, pp. 1181–1197, nov 1991. (Cited on page 17.)
- [26] H. A. Al-Twaijry, “Area and performance optimized cmos multipliers.” Ph.D. dissertation, Stanford University, 1997. (Cited on pages 17, 22 and 23.)
- [27] L. Dadda, “Multiple addition of binary serial numbers,” in *Proc. IEEE 4th Symp. Computer Arithmetic (ARITH)*, 1978, pp. 140–148. (Cited on page 20.)

- [28] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, “A compact high-speed parallel multiplication scheme,” *IEEE Trans. Comput.*, no. 10, pp. 948–957, 1977. (Cited on page 20.)
- [29] C. S. Wallace, “A suggestion for a fast multiplier,” *IEEE Trans. Electron. Comput.*, no. 1, pp. 14–17, 1964. (Cited on page 23.)
- [30] M. Dorojevets and C. Ayala, “Logical design and analysis of a 32/64-bit wave-pipelined RSFQ adder,” in *Proceedings of 2nd Superconducting SFQ VLSI Workshop*, ser. O6, Fukuoka, Japan, Aug. 2009, pp. 15–16. (Cited on pages 25, 33, 42, 43 and 58.)
- [31] O. A. Mukhanov, S. V. Rylov, V. K. Semonov, and S. V. Vyshenskii, “Rsfq logic arithmetic,” *IEEE Trans. Magn.*, vol. 25, no. 2, pp. 857–860, 1989. (Cited on page 25.)
- [32] A. Akahori, M. Tanaka, A. Sekiya, A. Fujimaki, and H. Hayakawa, “Design and demonstration of sfq pipelined multiplier,” *IEEE Trans. Appl. Supercond.*, vol. 13, no. 2, pp. 559–562, 2003. (Cited on pages 25 and 26.)
- [33] M. Obata, M. Tanaka, Y. Tashiro, Y. Kamiya, N. Irie, K. Takagi, N. Takagi, A. Fujimaki, N. Yoshikawa, H. Terai, and S. Yorozu, “Single-flux-quantum integr multiplier with systolic array structure.” *Physica C: Superconductivity*, vol. C, pp. 445–448, 2006. (Cited on pages 25 and 26.)
- [34] Q. Herr, N. Vukovic, C. A. Mancini, K. Gaj, Q. Ke, E. G. Friedman, A. Krasniewski, M. F. Bocko, and M. J. Feldman, “Design and low speed testing of a four-bit rsfq multiplier-accumulator,” *MultiplierAccumulator, IEEE Trans. Appl. Supercond.*, vol. 7, 1997. (Cited on page 25.)
- [35] T. Onomi, K. Yanagisawa, M. Seki, and K. Nakajima, “Phase-mode pipelined parallel multiplier,” *IEEE Trans. Appl. Supercond.*, vol. 11, no. 1, pp. 541–544, 2001. (Cited on page 25.)
- [36] Y. Horima, T. Onomi, M. Kobori, I. Shimizu, and K. Nakajima, “Improved design for parallel multiplier based on phase-mode logic,” *IEEE Trans. Appl. Supercond.*, vol. 13, no. 2, pp. 527–530, 2003. (Cited on page 26.)

- [37] I. Kataeva, H. Engseth, and A. Kidiyarova-Shevchenko, “New design of an RSFQ parallel multiply accumulate unit,” *Superconductor Science Technology*, vol. 19, pp. 381–+, May 2006. (Cited on pages 26 and 27.)
- [38] R. Nakamoto, S. Sakuraba, T. Onomi, S. Sato, and K. Nakajima, “4-bit sfq multiplier based on booth encoder,” *IEEE Trans. Appl. Supercond.*, vol. 21, no. 3, pp. 852–855, 2011. (Cited on page 26.)
- [39] S. Yorozu, Y. Kameda, H. Terai, A. Fujimaki, T. Yamada, and S. Tahara, “A single flux quantum standard logic cell library,” *Physica C: Superconductivity*, vol. C, pp. 378–381, 2001. (Cited on pages 26 and 39.)
- [40] H., K. Hara, H. Obata, Y. Park, K. Yamanashi, N. Taketomi, M. Yoshikawa, A. Tanaka, N. Fujimaki, K. Takagi, S. Takagi, and Nagasawa, “Design, implementation and on-chip high-speed test of sfq half-precision floating-point multiplier.” 2008. (Cited on pages 26 and 28.)
- [41] A. Fujimaki, M. Tanaka, T. Yamada, Y. Yamanashi, H. Park, and N. Yoshikawa, “Bit-serial single flux quantum microprocessor core,” in *IEICE Transactions on Electronics*, vol. E91-C, no. 3, 2008, pp. 342–349. (Cited on page 32.)
- [42] L. Cotton, “Maximum rate pipelined systems,” ser. AFIPS Spring Joint Computer Conference, 1969, pp. 581–586. (Cited on page 32.)
- [43] M. Dorojevets, C. Ayala, and A. Kasperek, “Development and evaluation of design techniques for high-performance wave-pipelined wide datapath RSFQ processors,” in *Proceedings of 12th International Superconductive Electronics Conference*, Fukuoka, Japan, Aug. 2009, pp. SP–P46. (Cited on pages 33 and 34.)
- [44] M. Dorojevets, C. Ayala, S. Shah, and S. Venkatachalam, “Design study of a 13.9 ghz 32-bit ersfq single-precision floating-point adder.” Ultra High Speed Computing Laboratory, Department of Electrical and Computer Engineering, Stony Brook University, New York, Tech. Rep., 2011. (Cited on pages 33, 66, 69, 70 and 78.)
- [45] M. Dorojevets, P. Bunyk, and D. Zinoviev, “FLUX chip: design of a 20-GHz 16-bit ultrapipeled RSFQ processor prototype based on 1.75- μm ;

- LTS technology,” *Applied Superconductivity, IEEE Transactions on*, vol. 11, no. 1, pp. 326–332, Mar. 2001. (Cited on page 33.)
- [46] P. Patra, S. Polonsky, and F. D. S., “Delay insensitive logic for rsfq superconductor technology,” in *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ser. ASYNC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 42–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=523661.785251> (Cited on page 34.)
- [47] M. Maezawa, I. Kurosawa, M. Aoyagi, H. Nakagawa, Y. Kameda, and T. Nanya, “Rapid single-flux-quantum dual-rail logic for asynchronous circuits,” *IEEE Trans. Appl. Supercond.*, vol. 7, no. 2, pp. 2705–2708, 1997. (Cited on page 34.)
- [48] M. Radparvar, “Niobium integrated circuit fabrication process 03-10-45,” <http://hypres.accountsupport.com/wp-content/uploads/2010/11/DesignRules.pdf>, January 2008. (Cited on page 36.)
- [49] W. Chen, A. V. Rylyakov, V. Patel, J. E. Lukens, and K. K. Likharev, “Rapid single flux quantum t-flip flop operating up to 770 ghz,” *IEEE Trans. Appl. Supercond.*, vol. 9, no. 2, pp. 3212–3215, 1999. (Cited on pages 36 and 37.)
- [50] T. Satoh, K. Hinode, S. Nagasawa, Y. Kitagawa, M. Hidaka, N. Yoshikawa, H. Akaike, A. Fujimaki, K. Takagi, and N. Takagi, “Planarization process for fabricating multi-layer nb integrated circuits incorporating top active layer,” *IEEE Trans. Appl. Supercond.*, vol. 19, no. 3, pp. 167–170, 2009. (Cited on page 39.)
- [51] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann, 2008. (Cited on pages 65 and 78.)
- [52] M. Dorojevets, A. Kasperek, A. F. N. Yoshikawa, and M. Hidaka, “8x8-bit parallel carry-save RSFQ multiplier,” 2012, submitted. (Cited on page 85.)

- [53] M. Peiniger and H. Piel, “A superconducting nb3sn coated multicell accelerating cavity,” *Nuclear Science, IEEE Transactions on*, vol. 32, no. 5, pp. 3610–3612, oct. 1985. (Cited on page 92.)